

Preface

StrongForth 3.1 is a non-standard Forth system for 32-bit Windows console applications. In contrast to the Forth 2012 standard, it incorporates strong static typing, overloading and object orientation. It has an optimizing x86 native-code compiler that produces very efficient code, using general-purpose registers instead of a data stack to pass parameters.

The *StrongForth 3.1 Reference Manual* has been written for programmers who already have experience with Forth based on the Forth 2012 standard. Within this manual, the Forth 2012 standard is referenced rather frequently in order to highlight the differences. It is recommended to begin with reading the separate *Introduction to StrongForth 3.1*.

This manual not only presents all words that are predefined in StrongForth 3.1 or that can be added by including source library files. It also shows the definitions of many words and explains how everything fits together and how the system is implemented. Design decisions are justified.

StrongForth 3.1 comes with a comprehensive set of source library files. These sources may also serve as programming samples that demonstrate the typical StrongForth style.

As additional documentation means, a Forth 2012 vs. StrongForth cross reference and a complete glossary are provided. The glossary consists of four files, one for each of the vocabularies `forth`, `assembler`, `msvcrt` and `protected`.

Table of Contents

Preface	1
Table of Contents	2
1 Stack Manipulation	6
Handling Different Data Types	6
The Complete Set	7
More on Double Numbers	10
What's missing?	12
2 Memory Access	15
Variables	15
Fetch and Store	16
Memory Blocks	19
3 Arithmetic and Logic.....	21
Plus and Minus	21
Multiplication and Division.....	27
Other Arithmetic Operations	30
Logical Operations	35
4 Memory Spaces	39
What's in a Memory Space?	39
The Default Memory Space.....	41
Memory Alignment	41
Storing Data in Memory Spaces	42
5 Input and Output.....	44
Basic I/O	44
Other Output Words	45
Pictured Numeric Output.....	46
The Console Window	50
6 Character Strings	51
What's a String?	51
String Processing	51
String Literals	53
7 Data Types.....	56
What's in a Data Type?	56
Data Type Heaps	58
Operations on Lists of Data Types	60
Creating new Data Types.....	63
Stack Diagrams	64
8 Object Orientation.....	67
Class Members	67
Class Methods and the <code>this</code> Object.....	70
Encapsulation.....	71
Inheritance and Binding.....	75
The Virtual Method Table	83
Class memory-space.....	85

9 Data Types Revisited	87
Data Type Attributes	87
Class Attributes.....	89
The Data Type Hierarchy	90
Class stack-diagram	92
10 Definitions.....	95
Class definition.....	95
Attributes of a Definition.....	100
Classes code-definition and colon-definition	102
Literal definitions	106
Class created-definition	110
Defining Words with Variable Parameters.....	115
11 Vocabularies.....	116
Why not <i>Word Lists</i> ?	116
Class vocabulary.....	117
Searching Vocabularies	120
Matching Rules.....	122
A Search Criterion for Matching Rules	125
More Search Criteria.....	128
Class integer-vocabulary	130
12 Input And Output Streams	135
Input streams.....	135
Output Streams	139
13 Object Orientation Revisited	143
Creating Data Types and Classes	143
Creating and Deleting Objects.....	144
Unions of Members	146
Container Classes	148
14 Compilation.....	153
Compiling Literals.....	153
Compiling Definitions	155
The Interpreter	156
Evaluating and Postponing	158
15 Values And Locals	160
Values	160
Locals.....	161
Changing Values and Locals	164
An Implicit Local: <i>r@</i>	166
16 Conditionals And Loops.....	169
Saving and Restoring Data Type Information	169
Conditional Clauses.....	172
Loops	174
do Loops	176
Between case and endcase	181
17 Qualified Tokens.....	186
Saving the Data Type System.....	186
Creating Qualified Tokens.....	188
Selecting Overloaded Definitions.....	190
Example: A Simple Jump Table	192

18 Implementing Object Orientation	193
Creating the Virtual Method Table	193
The Class Definition	194
Friend Classes	195
Data Members	196
Virtual Methods	198
Early Binding	200
19 Deferred Definitions	203
20 Exception Handling	206
catch in StrongForth	206
Class exception-handler	207
The Exception Extension Word Set	210
Error Codes	214
Operating System Exceptions	214
21 C Runtime Library	216
The msvcrt Vocabulary	216
Error Handling	217
Memory Allocation	217
Zero-terminated Strings	218
File Access	219
Loading Source Files	223
Arguments and Environment	224
The System Clock	225
Keyboard Events	225
Unsupported Features	228
22 Programming-Tools Word Set	229
?	229
dump	230
see	231
[if], [else] and [then]	232
[defined] and [undefined]	232
SYNONYM vs. alias	233
23 The Search Order	235
Displaying Vocabularies	235
The Search-Order Word Set	235
marker	237
24 Floating-Point Numbers	240
The Floating-Point Stack	240
Floating-Point Arithmetic	241
Floating-Point Numbers In Memory	242
Floating-Point Literals	243
Floating-Point Representation	248
25 Blocks	252
The Block File	252
Block Structure	254
Blocks As Input Source	255
A Line Editor	256
26 String Extensions	259
27 Structures	263

28 The Assembler.....	267
First steps.....	267
Addressing modes.....	269
Instruction words.....	274
Register Usage.....	282
Conditionals.....	284
The Disassembler	287
29 ASCII Characters	288
Lower Case And Upper Case	288
Non-Graphic Characters	289
30 Bit Fields.....	291
31 Environmental Queries	295
32 Utilities.....	296
Permutation.....	296
Help from the Glossary.....	296
Qualified Token Literals.....	297
Replacements for ?dup.....	297
Long Branches.....	299
Command Line Input.....	300
Quicksort	301
Templates.....	302
The StrongForth Test Suite.....	304
execute-parsing	306
Executing catch.....	306
33 Inside StrongForth	308
Appendix A: Starting StrongForth	313
Appendix B: Complex Numbers.....	315

1 Stack Manipulation

Handling Different Data Types

Words in StrongForth always apply to specific stack patterns. If, for example, you want to duplicate the item on top of the stack, you would execute or compile the word `dup` to the contents of the stack:

```
56 .s unsigned ok
dup .s unsigned unsigned ok
. . 56 56 ok
```

Note that StrongForth's version of `.s` displays the data types of the items on the stack instead of their values. In this example, `dup` is applied to an item of data type `unsigned`. This means, StrongForth has to provide a version of `dup` that accepts items of this data type. Let's try the same operation with a different data type:

```
char G .s character ok
dup . . GG ok
```

From this example it can be concluded that StrongForth provides a version of `dup` that can be applied to items of data type `character`. Given the whole lot of predefined data types, does this mean StrongForth provides one overloaded version of `dup` for each data type? Even worse, does it mean that you have to define a new version of `dup` for each newly defined data type? No, certainly not. In the above examples, only one version of `dup` is used, a version that can be applied to both `unsigned` and `character` data types. Here's the stack diagram of this version of `dup`:

```
dup ( single -- 1st 1st )
```

`dup` expects on the stack an item of data type `single` or any of this data type's direct and indirect subtypes. Since `unsigned` and `character` both are children of data type `integer`, and `integer` is a child of `single`, this version of `dup` accepts both data types.

But what about the right side of `dup`'s stack diagram? `1st` is not a data type. It actually is a representative for a data type that is substituted by an actual data type depending on the usage of the word. In the case of `dup`, using `1st` two times on the right side of the stack diagram means that `dup` produces two stack items of the same data type as the first input parameter. Looking again at the two examples above, you can see that the same version of `dup` produces two items of data type `unsigned` when applied to an item of data type `unsigned`, and it produces two items of data type `character` when applied to an item of data type `character`. If `dup` were defined without using `1st`, strange things may happen:

```
: dup2 ( single -- single single ) dup ; ok
char H dup2 .s single single ok
. . 72 72 ok
```

Instead of producing two items of data type `character`, the newly defined word `dup2` leaves two items of data type `single` on the stack. The information that you're dealing with a character that has to be treated as such, is lost.

As stated above, `dup` expects an item of data type `single` or one of its direct or indirect subtypes on the stack. This includes a large number of predefined data types, but not all. What do you have to do to duplicate a double number? Let's try it the Forth 2012 way:

```
100000000000. 2dup . .  
100000000000. 2dup ? undefined word  
unsigned-double
```

Sorry, StrongForth does not know 2dup. Why? Because 2dup is not required. StrongForth is capable of overloading words, so it is much more natural to provide a second version of dup that applies to double numbers, or, more precisely, to items of data type double and it's direct and indirect subtypes:

```
dup ( double -- 1st 1st )
```

The above example can now be fixed:

```
100000000000. dup . . 100000000000 100000000000 ok
```

The replacement of 2dup with an overloaded version of dup is typical for StrongForth. Actually, most Forth 2012 words dealing with double numbers, like D+ and DABS, are replaced by overloaded versions of the respective words used for single numbers, because the semantics is supposed to be the same from a high-level point of view.

However, you may have noticed that the second version of dup is not a full replacement for 2dup, because dup can not be applied to two single-cell items. This restriction will be discussed later in in this chapter.

With a total of three versions of dup, including an overloaded version for floating-point numbers, all data types can be duplicated. In fact, no other versions of dup are required:

```
dup ( single -- 1st 1st )  
dup ( double -- 1st 1st )  
dup ( float -- 1st 1st )
```

The Complete Set

After having discussed the overloaded versions of dup, it's now time to have a look at the other stack manipulation words:

- drop
- swap
- over
- rot
- nip
- tuck

Like dup, drop expects one item on top of the stack. To be able to apply drop to all data types, three overloaded versions are required:

```
drop ( single -- 1st 1st )  
drop ( double -- 1st 1st )  
drop ( float -- 1st 1st )
```

The second version is StrongForth's replacement for the Forth 2012 word 2DROP, and the third version is the replacement for FDROP.

Now let's look at swap and over. It might seem obvious to provide again three versions for each of these words, one for single-cell items, one for double-cell items, and one for floating-point numbers:

```

swap ( single single -- 2nd 1st )
swap ( double double -- 2nd 1st )
swap ( float float -- 2nd 1st )

over ( single single -- 1st 2nd 1st )
over ( double double -- 1st 2nd 1st )
over ( float float -- 1st 2nd 1st )

```

Consider the data type representatives on the right side of the stack diagrams. In the case of `swap`, `2nd` will be replaced by the actual data type of the *second* parameter on the left side of the stack diagram, which is the data type of the item on top of the stack before `swap` is executed or compiled. `1st` means that the data type on top of the stack after executing or compiling `swap` will be identical to the data type of the *first* item on the left side of the stack diagram. This guarantees that `swap` doesn't change the data types of the swapped items, as can be seen in the following example:

```

-100 true .s signed flag ok
swap .s flag signed ok
. . -100 true ok

```

With `over`, things are similar. Here's an example for double-cell items:

```

200000000000. +7375550160281. .s unsigned-double signed-double ok
over .s unsigned-double signed-double unsigned-double ok
. . . 200000000000 7375550160281 200000000000 ok

```

Obviously, the double-cell versions of `swap` and `over` replace the Forth 2012 words `2SWAP` and `2OVER`, respectively. However, there's still something missing. What if you want to swap a single-cell and a double-cell item? Let's try it:

```

1234567890. char k .s unsigned-double character ok
swap .s character unsigned-double ok
. . 1234567890 k ok

```

No problem. But neither of the above three versions of `swap` is capable of doing that. For `swap` to be complete, six additional overloaded versions are required:

```

swap ( single double -- 2nd 1st )
swap ( single float -- 2nd 1st )
swap ( double single -- 2nd 1st )
swap ( double float -- 2nd 1st )
swap ( float single -- 2nd 1st )
swap ( float double -- 2nd 1st )

```

For these words, there seems to be no match in Forth 2012. Well, this is not quite true. `swap (single double -- 2nd 1st)` performs nothing else but the Forth 2012 word `ROT`. For `swap (double single -- 2nd 1st)`, you would have to write `ROT ROT` in Forth 2012, which is sometimes defined as `-ROT`. However, from a high-level point of view, calling these operations `swap` is much clearer, since using `ROT` implies that there are three items on the stack. Interpreting one double number like `112233445566778899.` as two single-cell items is definitely not intuitive. Think about a program change, where you have to replace a single number variable with a double number variable, because it has turned out that the numeric range of a single number is insufficient. You would have to check all occurrences of the variable to make sure it is handled correctly in all places. In StrongForth, this is not necessary. You just change the data type of the variable, and StrongForth's compiler does the rest.

In total, StrongForth provides nine overloaded versions each for `swap` and `over`. The same applies to `nip` and `tuck`:


```

swap ( single single -- 2nd 1st )
swap ( single double -- 2nd 1st )
swap ( single float -- 2nd 1st )
swap ( double single -- 2nd 1st )
swap ( double double -- 2nd 1st )
swap ( double float -- 2nd 1st )
swap ( float single -- 2nd 1st )
swap ( float double -- 2nd 1st )
swap ( float float -- 2nd 1st )

over ( single single -- 1st 2nd 1st )
over ( single double -- 1st 2nd 1st )
over ( single float -- 1st 2nd 1st )
over ( double single -- 1st 2nd 1st )
over ( double double -- 1st 2nd 1st )
over ( double float -- 1st 2nd 1st )
over ( float single -- 1st 2nd 1st )
over ( float double -- 1st 2nd 1st )
over ( float float -- 1st 2nd 1st )

nip ( single single -- 1st )
nip ( single double -- 1st )
nip ( single float -- 1st )
nip ( double single -- 1st )
nip ( double double -- 1st )
nip ( double float -- 1st )
nip ( float single -- 1st )
nip ( float double -- 1st )
nip ( float float -- 1st )

tuck ( single single - 2nd 1st 2nd )
tuck ( single double - 2nd 1st 2nd )
tuck ( single float - 2nd 1st 2nd )
tuck ( double single -- 2nd 1st 2nd )
tuck ( double double - 2nd 1st 2nd )
tuck ( double float - 2nd 1st 2nd )
tuck ( float single - 2nd 1st 2nd )
tuck ( float double -- 2nd 1st 2nd )
tuck ( float float -- 2nd 1st 2nd )

```

What about `rot`? Since `rot` expects three items on the stack, $3^3 = 27$ combinations of single-cell items, double-cell items and floating-point numbers have to be taken care of:

```

rot ( single single single - 2nd 3rd 1st )
rot ( single single double - 2nd 3rd 1st )
rot ( single single float - 2nd 3rd 1st )
rot ( single double single -- 2nd 3rd 1st )
rot ( single double double - 2nd 3rd 1st )
rot ( single double float - 2nd 3rd 1st )
rot ( single float single - 2nd 3rd 1st )
rot ( single float double -- 2nd 3rd 1st )
rot ( single float float -- 2nd 3rd 1st )
rot ( double single single - 2nd 3rd 1st )
rot ( double single double - 2nd 3rd 1st )
rot ( double single float - 2nd 3rd 1st )
rot ( double double single -- 2nd 3rd 1st )
rot ( double double double - 2nd 3rd 1st )

```

```

rot ( double double float - 2nd 3rd 1st )
rot ( double float single - 2nd 3rd 1st )
rot ( double float double -- 2nd 3rd 1st )
rot ( double float float -- 2nd 3rd 1st )
rot ( float single single - 2nd 3rd 1st )
rot ( float single double - 2nd 3rd 1st )
rot ( float single float - 2nd 3rd 1st )
rot ( float double single -- 2nd 3rd 1st )
rot ( float double double - 2nd 3rd 1st )
rot ( float double float - 2nd 3rd 1st )
rot ( float float single - 2nd 3rd 1st )
rot ( float float double -- 2nd 3rd 1st )
rot ( float float float -- 2nd 3rd 1st )

```

That's quite a lot. Fortunately, there are no stack manipulation words that expect more than three items on the stack. However, the availability of a complete set of overloaded versions makes `rot` much more versatile than its Forth 2012 equivalents. Only three versions of `rot` actually have matches in Forth 2012, namely `ROT`, `2ROT` and `FROT`. For the other 24 overloaded versions, more or less tricky and error-prone constructions have to be used in Forth 2012.

More on Double Numbers

In Forth 2012, the stack image of a double number is clearly defined. A double number on top of the stack is stored in two cells, and the cell on top of the stack contains the higher part of the double number. This means, entering `123456.` in Forth 2012 is equivalent to entering `123456 0.`

You can convert a double number to a single number by simply `DROPPING` the cell containing its most significant part, and you can convert an unsigned single number to a double number by pushing `0` onto the stack. To convert a signed single number into a double number, you have to use `S>D` to make sure the sign is extended.

In StrongForth, the stack image of double numbers is implementation dependent. If the hardware already supports operations that load and store double numbers, using these operations may result in a performance gain. As a consequence, making any assumption on the order of the two cells a double number consists of is dangerous. The resulting code is not portable.

Since a double number may not be interpreted as two single numbers, StrongForth provides a set of low-level words that perform the necessary conversions. This is a direct consequence of strong static data typing. Similarly to Forth 2012, StrongForth uses the word `s>d` to convert signed single numbers to double numbers. However, `s>d` must also be used to convert unsigned single numbers into unsigned double numbers. Just pushing `0` onto the stack will not make a double number out of a single number:

```

123456 0 .s unsigned unsigned  ok
. . 0 123456  ok
123456 s>d .s unsigned-double  ok
. 123456  ok

```

Always using `s>d` for converting a single number into a double number makes the intention of the programmer much clearer than pushing `0` onto the stack. Actually, StrongForth provides four overloaded versions of `s>d`:

```
s>d ( single -- double )
s>d ( integer -- integer-double )
s>d ( unsigned -- unsigned-double )
s>d ( signed -- signed-double )
```

Only the last version performs sign extension. The other three are semantically identical, performing zero extension. They only differ in the data types of their output parameter. For example, the third overloaded version converts items of data type `unsigned` and all its subtypes to items of data type `unsigned-double`.

An interesting aspect regarding these four overloaded versions of `s>d` is the order in which they are arranged in the dictionary. It is important that the version for input parameters of data type `single` is defined first, then the version for `integer`, and finally the two versions für `signed` and `unsigned` numbers. The four equally named words are found in the dictionary in reverse order. Do you see what would happen if the order was not like this? Since `signed` and `unsigned` numbers are integers as well, and data type `integer` is itself a subtype of `single`, `s>d` for data type `integer` could hide the versions for data types `signed` and `unsigned`, and `s>d` for data type `single` could hide all other versions. That's why the order of definitions is crucial in this case.

Now let's focus on the other direction. In Forth 2012, converting a double number into a single number can be performed by `DROPPing` the cell containing its most significant part. In StrongForth, this does not work, because `drop` applied to a double number will drop the complete double number, not only its most significant part. However, even Forth 2012 provides the word `D>S` to make the conversion more obvious. `d>s` also exists in StrongForth. Since the semantics are identical for `signed` and `unsigned` numbers, only the stack diagrams differ:

```
d>s ( double -- single )
d>s ( integer-double -- integer )
d>s ( unsigned-double -- unsigned )
d>s ( signed-double -- signed )
```

Here's an example of how to use it:

```
-123456. .s signed-double  ok
d>s .s signed  ok
. -123456  ok
```

Even though the stack image of a double number is not specified in StrongForth, there's still the fact that each double number is composed of two cells. To be able to access both parts of a double number, or more generally, to access both cells of a double-cell item, StrongForth provides four low-level words:

```
low ( double -- single )
high ( double -- single )
split ( double -- single single )
merge ( single single -- double )
```

`low` and `high` return the lower and the upper cell of a double-cell item as an unspecific single-cell item, respectively. `low` is actually an alias of the first version of `d>s`.

`split` splits a double-cell item into two single-cell items in such a way that in case of a double number the cell containing the most significant part is on top of the stack. Even on machines that usually store the least significant part at the lower address, this behaviour is guaranteed.

```
hex FEDCBA9876543210. split .s single single  ok
.. FEDCBA98 76543210  ok
FEDCBA9876543210. low .s . single 76543210  ok
FEDCBA9876543210. high .s . single FEDCBA98  ok
decimal  ok
```

Note that `d>s` and `low` only return the least significant part of a double number, while `split` returns both parts. Since interpreting the two cells that constitute a double-cell item is totally out of control of StrongForth's type system, `split` should be used with care.

`merge` performs the reverse operation. It expects two single-cell items on the stack and merges them into a double-cell item. The single-cell item on top of the stack is always the most significant part of the resulting double-cell item, if it is interpreted as a double number.

```
hex a5a5a5a5 12345678 merge .s double ok
. 12345678A5A5A5A5 ok
decimal ok
```

What's missing?

?DUP

StrongForth requires that each word has a unique stack diagram. The stack diagram describes what a word expects to be on the stack before it is executed, and what it leaves on stack after execution. For example, the word `d>s` (`double -- single`) expects an item of data type `double` on top of the stack, which it replaces with an item of data type `single`. Although Forth 2012 is an untyped language, its glossary specifies stack comments for all words. Most of these stack comments are definite in the sense that the stack effect is already known at compile time. However, for some Forth 2012 words, the stack effect is not known at compile time. It rather depends on the values of one or more parameters at runtime. For example, the stack diagram of `?DUP` is (`x -- 0 | x x`). So, the stack effect of `?DUP` is either (`x -- 0`) or (`x -- x x`), depending on the value of `x` at the time `?DUP` is executed. How can an ambiguous stack diagram like that be implemented in StrongForth? It can not. Since StrongForth adheres to strong static typing, the compiler requires that the stack effect of each word is known at compile time.

Is this a problem? Let's have a look at a typical application of `?DUP`. In many cases, `?DUP` is used immediately preceding `IF`, as in the following example. Note that this definition is Forth 2012 code; it will not work in StrongForth.

```
: SPACES ( n -- )
  0 MAX ?DUP IF 0 DO SPACE LOOP THEN ;
```

Here, `?DUP` is used as a shortcut to avoid the `ELSE` branch. Alternatively, the above example could be written as

```
: SPACES ( n -- )
  0 MAX DUP IF 0 DO SPACE LOOP ELSE DROP THEN ;
```

Since `?DUP` is avoided here, this definition can, with minor changes and conversion to lower-case letters, be compiled in StrongForth. Actually, `?DUP` can in most cases be avoided by either spending an additional `IF` clause or by adding an `ELSE` branch to an existing `IF` clause. In chapter 32 you'll see that it is possible to implement `?DUP` in combination with conditional jump instructions in StrongForth.

PICK and ROLL

Here are two other Forth 2012 words with ambiguous stack diagrams:

```
PICK ( xu ... x1 x0 u -- xu ... x1 x0 xu )
ROLL ( xu xu-1 ... x0 u -- xu-1 ... x0 xu )
```

Can you see why these two words have ambiguous stack diagrams? Yes, the problem is that the compiler is not able to find out the data type of `xu`. The data type of `xu` depends on `u`, and the

actual value of `u` is not known at compile time. Of course, in most cases `PICK` and `ROLL` are preceded by numeric constants as in the following Forth 2012 example:

```
: EXAMPLE ( c-addr u d -- char flag ) 3 ROLL ( u d c-addr ) ... ;
```

Although the stack effect is obvious to the programmer, it is not obvious to the compiler. In general, the index `u` is not a numeric constant, but a value calculated at runtime, which may even depend on which keys the user presses. In those cases, even a very clever compiler will have to surrender. To cover at least those cases where the index is a numeric constant, it might be tempting to define a set of words like `2pick`, `3pick`, `3roll` etc. Note that `1 PICK` is semantically the same as `OVER`, and `2 ROLL` has the same semantics as `ROT`. However, you have seen that the number of overloaded versions of `rot` is already rather high. `2pick` would take three parameters, so we'd need 27 overloaded versions as well. For `3pick` and `3roll`, already 81 overloaded versions would be required to apply these words to all combinations of single, double and float parameters. This can't be a reasonable solution.

On the other hand, if you ever come into a situation where you think you need `pick` or `roll`, you should consider changing your code anyway. Since accessing the fourth or fifth item on the stack usually makes the code rather obscure, using `pick` and `roll` is discouraged. In StrongForth, this recommendation can be somewhat relaxed. If you have a second look at the above example, you'll probably find out that `3 ROLL` is not required if the code is ported to StrongForth. Although `EXAMPLE` uses four stack cells, these four cells constitute only three data items. This means, an overloaded version of `rot` will do:

```
: example ( caddress -> character unsigned signed-double --
  character flag )
  rot ( unsigned signed-double caddress -> character )
  ... ;
```

Thus, the availability of 27 overloaded versions of `rot` turns out to be handy. Whenever you want to bring the third item on the stack to the top, `rot` does the job, no matter whether you have to skip two, three or four cells. This makes the necessity for `ROLL` in your code unlikely. Another example demonstrates that `2 PICK` can be replaced by `over` in some cases. The Forth 2012 definition

```
: EXAMPLE2 ( c-addr u d -- char flag )
  2 PICK ( c-addr u d u ) ... ;
```

can in StrongForth be written as

```
: example2 ( caddress -> character unsigned signed-double --
  character flag )
  over ( caddress -> character unsigned signed-double unsigned )
  ... ;
```

Let's summarize: StrongForth does not provide `pick` and `roll`. In some cases, the overloaded versions of `rot` and `over` will do the job. In all other cases, you have to revise your code to avoid deep or random stack access.

2DUP, 2DROP, 2SWAP, 2OVER and 2ROT

It was already mentioned that overloading `dup`, `drop`, `swap`, `over` and `rot` for double-cell items makes the Forth 2012 words `2DUP`, `2DROP`, `2SWAP`, `2OVER` and `2ROT` obsolete. This is only partly true, because these words all have a double semantics in Forth 2012. Consider the word `2DUP`. When the top of the stack contains a double-cell item, its semantic would be to duplicate this item:

```
2DUP ( d -- d d )
```

On the other hand, if there are two single-cell items on top of the stack, 2DUP will duplicate this pair of single-cell items:

```
2DUP ( x1 x2 -- x1 x2 x1 x2 )
```

In Forth 2012, these two interpretations of 2DUP are identical, because each double number is a pair of single numbers. In StrongForth, the two interpretations are completely different. The first one is just an overloaded version of dup, while the second one is a new word, which expects two items on the stack instead of only one. Of course, it is possible to define 2DUP in StrongForth:

```
: 2dup ( single single -- 1st 2nd 1st 2nd )  
  over over ;
```

But what if you want to duplicate a pair consisting of one single-cell and one double-cell item? Consequently, StrongForth would have to provide a complete set of nine overloaded versions of 2dup, just as it does for other stack manipulation words.

No problem so far. Now, what about the other words? 2drop would be similar to 2dup: Nine overloaded versions are required. 2SWAP and 2OVER expect four items on the stack, 2ROT expects six items:

```
2SWAP ( x1 x2 x3 x4 -- x3 x4 x1 x2 )  
2OVER ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 )  
2ROT ( x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2 )
```

Providing the complete set of overloaded versions for these words would result in 81 or even 729 different versions, respectively! As already stated before, handling four or more stack items simultaneously is not considered being good programming style in Forth, and this is certainly one of the reasons why 2SWAP, 2OVER and 2ROT are usually only applied to pairs or triples of double-cell items. For these applications, StrongForth's overloaded versions of swap, over and rot will do nicely. Therefore, 2swap, 2over and 2rot for single-cell items are not provided in StrongForth.

2DUP and 2DROP are used more frequently. However, to avoid confusion with the double-cell versions of dup and drop, StrongForth does not even provide these two words. You can always write over over for 2DUP, or DROP DROP for 2DROP, so not having these two words shouldn't be a real drawback. If you still think it is, don't hesitate to define 2dup and 2drop and several overloaded versions according to the sample definition above.

An interesting detail is the fact that the source code of all overloaded versions of 2dup and 2drop looks identical except for their stack diagrams. At least, they *look* identically. For each version of 2dup, different overloaded versions of over are being compiled, and for each version of 2drop, different overloaded versions of drop are being compiled, because the data types involved have different sizes. Of course, StrongForth's compiler always ensures that the compiled words fit to the items on the stack.

If you think it's annoying to write nine times almost identical definitions, here are alternative solutions that perform the overloading at compile time:

```
: 2dup ( -- ) postpone over postpone over ; immediate  
: 2drop ( -- ) postpone drop postpone drop ; immediate
```

Please note that this alternative, though it looks more elegant, does not behave identically to the first alternative. Using the first alternative compiles only one word, while using the second alternative compiles two words. The second alternative is faster at runtime, because it saves the overhead of entering and exiting a word by inlining the two words it compiles. And compiling these two words (over or drop) does not necessarily produce any object code at all.

2 Memory Access

Variables

For doing things with items on the stack, you normally don't need to bother about the memory locations of these items. Even if the stack is located in memory, accessing items is completely handled by the system. However, as soon as you want to write one or more items to any other place in memory than the stack, and later want to read it from the same location, you need the address of this memory location.

How can you get a memory address? The easiest way is to define a variable:

```
variable longitude  
variable ? undefined word
```

This is supposed to work in Forth 2012. Doesn't StrongForth provide variables? Of course it does. But since StrongForth is a strongly typed language, you need to specify the data type of the variable. And this is easily done by supplying an initialization value for each variable:

```
+49 variable longitude ok
```

`variable` expects an item of any data type on the stack, which explains the error message we got at the first try, where `variable` was applied to an empty stack. Note that StrongForth always issues `? undefined word` if it doesn't find a word that matches the parsed name *and* whose stack diagram fits to the items on the stack. If a word with the correct name exists, but it can not be applied, because its expected parameters are not on the stack, the word is ignored.

The parameter to `variable`, `+49`, has two meanings. First, it specifies the data type of the variable, which is signed in this example. Second, it initialises the variable with the given value:

```
longitude @ .s signed ok  
. 49 ok
```

But what is the data type of the address of the variable? Let's see:

```
longitude .s . address -> signed 9840248 ok
```

Oops, is this really a data type? It is. `longitude` is an address of data type `address` that points to an item of data type `signed`. Whenever two or more *basic data types* like `address` and `signed` are combined by the symbol `->`, the result is called a *compound data type*. Compound data types are mostly used for addresses, because an address is somehow incomplete if the information about what kind of item it points to is not available. If `longitude` would just push something of data type `address` onto the stack, `@` wouldn't know the data type of whatever it should fetch from the address. In the next section you will see that several overloaded versions of `@` exist, and that each of these versions expect an address on the stack that has a compound data type similar to `address -> signed`.

There are actually three overloaded versions of `variable`:

```
variable ( single -- )  
variable ( double -- )  
variable ( float -- )
```

As you might have guessed, the version for double-cell items is StrongForth's equivalent of the Forth 2012 word `2VARIABLE`, and the version for floating-point numbers maps to `FVARIABLE` in Forth 2012. Note that the double-cell version is not suited for creating two single-cell variables at succeeding addresses. In order to accomplish this, you have to write

```
<single> 2 variables <name>
```

in StrongForth, with `<single>` being any single-cell item. Of course, `variables` accepts other values than 2, so it is a convenient means to define arbitrary-length arrays of elements with a specific data type. The array is initially filled with the value of `<single>`.

There's actually a fourth version of `variable` with a different name:

```
cvariable ( single -- )
```

This version creates a variable that occupies an item with the size of a character in the data space. When the variable is invoked, it returns a character address:

```
char K cvariable ch ok
ch .s @ . caddress -> character K ok
```

Data type `caddress` is a child of data type `address`. It does not need to point to a character. Integer numbers, flags and other single-cell data types can also be stored in and retrieved from character-size memory locations, provided their value can be represented by 8 bits. This condition is normally not fulfilled for addresses. `base` and `state` are system variables that fit into a character-size memory location:

```
base .s @ . caddress -> unsigned 10 ok
state .s @ . caddress -> flag false ok
```

However, the amount of memory space that can be saved by using single character-size variables is rather limited. The advantage can be considerably greater in connection with arrays, as in this example:

```
bl 80 cvariables pad2 ok
```

Fetch and Store

You certainly have noticed that in the above examples with `base` and `state`, we used `@` instead of `C@`, as in Forth 2012. This is correct, because StrongForth provides overloaded versions of `@` that fetch character-size items from character addresses. These versions even perform the proper sign extension, when the content of the memory location is a signed number or a flag.

This is an important point. The information about what is stored at a memory location, like the data type and the size of the item (character, single cell, double cell, floating-point number), is an attribute of the memory location itself. You need not take care about whether to use `2@`, `C@`, `F@`, `SF@` and `DF@` instead of `@`, and `2!`, `C!`, `F!`, `SF!` and `DF!` instead of `!`, because the compiler will always select the semantically correct word for you. Consequently, `@` and `!` are overloaded words:

```
@ ( address -> single -- 2nd )
@ ( address -> double -- 2nd )
@ ( address -> float -- 2nd )
@ ( caddress -> single -- 2nd )
@ ( caddress -> signed -- 2nd )
@ ( caddress -> flag -- 2nd )
@ ( sfaddress -> float -- 2nd )
@ ( dfaddress -> float -- 2nd )

! ( single address -> 1st -- )
! ( double address -> 1st -- )
! ( float address -> 1st -- )
```



```
! ( single caddress -> 1st -- )
! ( float sfaddress -> 1st -- )
! ( float dfaddress -> 1st -- )
```

However, @ and ! for double-cell items cannot replace 2@ and 2! for pairs of single-cell items. If you want to implement these two words yourself, you have to take into account that type-save versions require that the two single-cell items need to have exactly the same data type:

```
: 2! ( single 1st address -> 1st -- )
  tuck ! 1+ ! ;

: 2@ ( address -> single -- 2nd 2nd )
  cast address -> double @ split swap ;
```

The output parameter 2nd of the eight overloaded versions of @ references the tail of the compound data type of the input parameter. For example, @ applied to an address of data type address -> character returns an item of data type character fetched from a cell-size memory location. If the character is stored at a character-size location represented by caddress -> character, the fourth overloaded version of @ will be applied, which performs zero extension to cell size. The first four versions of @ actually have the semantics of the Forth 2012 words @, 2@, F@ and C@.

Zero extension means that the high-order bits of the value are filled with zero bits. This is the behaviour that Forth 2012 specifies for C@, and it is correct when fetching a character or an unsigned number from a character size memory location. However, zero extension is not desirable when you want to fetch a signed number or a flag from a character size memory location, as the following example demonstrates:

```
-119 pad cast caddress -> signed !   ok
pad cast caddress -> single @ . 137   ok
```

What happened? If the bit pattern of decimal -119 is stored in a character size memory location, it is binary 10001001, assuming character size is 8 bits. Fetching this value with zero-extension is decimal 137. The correct way to handle this case would have been to use sign extension instead of zero extension. Sign extension fills the high-order bits of the single cell with copies of the most significant bit of the contents of the character size memory location, which is 1 in the above example. Fetching binary 10001001 with sign extension correctly yields decimal -119:

```
pad cast caddress -> signed @ . -119   ok
```

That's the reason why StrongForth provides more than one overloaded version for fetching data from character size memory locations. Sign extension has to be applied to items of data type signed and flag. Items of all other data types have to be zero extended. Since data types signed and flag are not subtypes of the same data type, it is necessary to provide separate overloaded versions of @ for both of them. Note that the dictionary order is important. The special versions

```
@ ( caddress -> flag -- 2nd )
@ ( caddress -> signed -- 2nd )
```

with sign extension are found before the general version

```
@ ( caddress -> single -- 2nd )
```

with zero extension. This is a general rule. When implementing overloaded words, you always have to define the general version before defining the special versions, because interpreter and compiler search the dictionary starting with the most recently defined words. If a general version is defined after the special versions, interpreter and compiler would never find the special versions, because the general version would hide the special versions of the overloaded word. In Forth 2012, a word *always* hides previously defined words with the same name.

The last two overloaded versions of @ introduce two new data types: `sfaddress` and `dfaddress` are, like `caddress`, children of `address`. They point to single-precision and double-precision floating-point numbers, respectively. Since the format and memory footprint differ from that of an 80-bit long floating-point number, they are required to tell StrongForth's data type system which kind of floating-point number is stored at a given address.

Here's a final example regarding @:

```
longitude variable pointer ok
pointer .s address -> address -> signed ok
@ .s address -> signed ok
@ .s signed ok
. 49 ok
```

`pointer` is a variable that contains the address of another variable, `longitude`. Though the data type of `pointer` looks strange, it should be rather obvious what happens here. Since `pointer` is an address of an address of a signed number, applying @ results in an item of data type `address -> signed`, and applying @ again finally produces a signed number. Items of compound data types like `address -> signed` can certainly be stored in variables just like any items of basic data types.

Let's now have a closer look at the overloaded versions of !. The stack diagrams reveal that each version expects two items on the stack, the first one being the data to be stored and the second one the address. The second input parameter has always a compound data type with a reference to the first parameter. This means, that the first parameter can only be stored at an address pointing to exactly the same data type. Any attempt to store something at a location where it does not belong will fail, because there doesn't exist a matching version of ! for it:

```
char x base !
char x base ! ? undefined word
character address -> unsigned
36 longitude !
36 longitude ! ? undefined word
unsigned address -> signed
```

From the stack dump on the line below the error message, you can easily see what went wrong. You can't store a character into a location for unsigned numbers. Even if `character` were a subtype of `unsigned`, the attempt would fail. References in the input parameter list mean that the actual data types must exactly match. However, the following code works:

```
36 base .s ! unsigned caddress -> unsigned ok
decimal ok
+36 longitude .s ! signed address -> signed ok
```

The third overloaded version of ! stores an item of data type `single` or any of its direct or indirect subtypes into a character size memory location. Since the size of a character in bits is less than the size of a cell, only the lower bits that fill up a character are stored in the memory location. Actually, the semantic of this version of ! is identical to that of the Forth 2012 word `C!`.

Character size memory locations are normally used for packed arrays of data. A typical example of a packed array of data is a character string, and a predefined transient area for character strings is `pad`:

```
char P pad ! ok
pad .s caddress -> character ok
@ . P ok
```

Items of data type `double` and their direct and indirect subtypes cannot be stored in character-size memory locations.

Memory Blocks

Forth 2012 specifies a set of words that perform operations on memory blocks: `MOVE`, `CMOVE`, `CMOVE>`, `FILL` and `ERASE`. `MOVE` and `ERASE` apply to address units, while the other three words apply to characters and may thus be interpreted as character string operations. As usual, StrongForth provides overloaded versions of these words, which can be applied to all available data types. Let's start with `fill`.

The StrongForth equivalent to the Forth 2012 word `FILL` is

```
fill ( caddress -> single unsigned 2nd -- )
```

Note that `caddress -> single` is not exactly an address of a character. It's an address of a character size item. The last parameter `2nd` ensures that the item to be filled in has the same data type as the one the address points to.

However, StrongForth provides quite a number of overloaded versions of `fill`:

```
fill ( address -> single unsigned 2nd -- )
fill ( address -> double unsigned 2nd -- )
fill ( address -> float unsigned 2nd -- )
fill ( caddress -> single unsigned 2nd -- )
fill ( sfaddress -> float unsigned 2nd -- )
fill ( dfaddress -> float unsigned 2nd -- )
```

Obviously, all of these versions can be used to fill a block of memory not only with character-size items, but also with single-cell or double-cell items, and with floating-point numbers of different formats. These versions prove to be useful for initialising arrays that do not consist of characters. The second parameter `unsigned` directly specifies the number of items to be filled into the memory block. If, for example, single-cell items occupy four address units in memory, the phrase

```
here cast address -> unsigned 5 1000 fill
```

affects a total of twenty address units by five times storing the unsigned number 1000 into consecutive memory cells. The phrase can be shortened to

```
here -> unsigned 5 1000 fill
```

because `here` already returns an item of data type `address`.

Directly derived from `fill` is `erase`. Other than in Forth 2012, `erase` does not apply to address units. Instead, `erase` uses `fill` to fill the specified number of single cells, double cells, character size items or floating-point number in three different sizes and formats with binary zero. There are actually seven overloaded versions of `erase`:

```
erase ( address -> single unsigned -- )
erase ( address -> double unsigned -- )
erase ( address -> float unsigned -- )
erase ( caddress -> single unsigned -- )
erase ( sfaddress -> float unsigned -- )
erase ( dfaddress -> float unsigned -- )
erase ( object -- )
```

The last overloaded version erases the members of an object. It will be discussed chapter 8 in connection with classes and objects.

The other six versions are actually defined like this:

```
: erase ( address -> single unsigned -- )
  null single fill ;
```

`null` is an immediate word that pushes an item with a given data type (`single` in this definition) and a bit pattern of zero onto the stack. It is used in a similar way like `cast`, parsing the data type it is supposed to produce. Here's its definition:

```
: null ( -- )
  postpone false postpone cast ; immediate
```

Similarly as for `fill` and `erase`, StrongForth provides six overloaded versions of `move`:

```
move ( address -> single 1st unsigned -- )
move ( address -> double 1st unsigned -- )
move ( address -> float 1st unsigned -- )
move ( caddress 1st unsigned -- )
move ( sfaddress 1st unsigned -- )
move ( dfaddress 1st unsigned -- )
```

Again, you can see that `move` can be applied to areas of memory that contain either single-cell items, double-cell items, character size items and three kinds of floating-point numbers. Therefore, a version that operates on address units is superfluous. Using `chars` and `cells` for calculating the number of address units to be moved, like in Forth 2012, is not necessary. Please note that the first two input parameters of each of these versions need to have exactly the same data type. For example, it is not possible to move a memory block of character-size unsigned numbers to a memory block of characters:

```
here cast caddress -> unsigned pad 8 move
here cast caddress -> unsigned pad 8 move ? undefined word
caddress -> unsigned caddress -> character unsigned
```

Since there's no replacement for the propagation feature of `CMOVE` and `CMOVE>`, these two words are also provided by StrongForth:

```
cmove ( caddress 1st unsigned -- )
cmove> ( caddress 1st unsigned -- )
```

If you just want to move characters, the overloaded version of `move` for data types `ccaddress` can safely be used. Only if you explicitly want to take advantage of the propagation feature, `cmove` and `cmove>` should be considered. Using `move`, you don't need to consider whether source and destination memory areas overlap. `move` is always performed in the direction that avoids propagation, because propagation is unintended in most cases. If you follow this rule, you can later easily see at which places your code relies on propagation.

3 Arithmetic and Logic

Plus and Minus

Integer and Floating-Point Numbers

A whole section about + and -? What can be so special about these two words? Again, it's the fact that StrongForth provides a whole set of overloaded versions of + and -, thus distinguishing between integer arithmetic and address arithmetic and single-cell, double-cell, floating-point and mixed operations.

Let's begin with the most obvious versions of + and -:

```
+ ( integer integer -- 1st )  
- ( integer integer -- 1st )
```

Both words work as expected. But have a closer look at the stack diagrams. The two operands are both of data type `integer`, which means you can add or subtract two arbitrary, different items of data type `integer` and its subtypes, which are `unsigned`, `signed` and `character`. For example, you can subtract an item of data type `unsigned` from an item of data type `signed`, as in the following example:

```
-35 7 - .s signed ok  
.- -42 ok
```

The data type of the result is identical to the data type of the first operand. This is a general principle for all of StrongForth's arithmetic and logic words. It is usually pretty handy:

```
char C 3 + . F ok  
3 char C + . 70 ok
```

The result of adding 3 to the character `C` is still a character, while the reverse operation yields an item of data type `unsigned`, which is the ASCII value of the character `C`, incremented by 3.

As expected, StrongForth has overloaded versions of + and - for double precision arithmetic, which replace the Forth 2012 words `D+` and `D-`. Even for mixed-mode operations, StrongForth provides overloaded versions of + and -:

```
+ ( integer-double integer-double -- 1st )  
+ ( integer-double integer -- 1st )  
+ ( integer-double signed -- 1st )  
- ( integer-double integer-double -- 1st )  
- ( integer-double integer -- 1st )  
- ( integer-double signed -- 1st )
```

Regarding mixed-mode operations, two things are interesting. First, there are two different versions for adding signed and unsigned numbers to a double-precision integer. This is because an unsigned single number has to be zero extended before adding it to a double number, while a signed single number has to be sign extended prior to the addition. Items of data type `integer` are assumed to be unsigned numbers in this context.

```
125000000000. 2500000000 + . 127500000000 ok  
366666666600. -18 + . 3666666666582 ok
```

If, for example, sign extension would be applied in the first example, the result would be 123205032704, which is obviously not the expected result.

The second interesting thing about the mixed-mode versions of + and - is the fact that you can not add a double-precision number to a single-precision number.

```
45 838383830000. + .  
45 838383830000. + ? undefined word  
unsigned unsigned-double
```

Actually, this operation makes no sense if the data type of the result should be identical to the data type of the first operand, because the result usually does not fit into a single-cell number. In cases like this, you have to use d>s or an explicit type cast to convert the double number into a single number:

```
45 838383830000. d>s + . 865207325 ok  
45 838383830000. cast integer + . 865207325 ok
```

This is not very satisfying, but what else did you expect? Cutting off the most significant part of a double number can't be safe. Generally, the presence of a type cast is an indicator for a potentially dangerous or tricky operation.

StrongForth also overloads + and - for adding and subtracting floating-point numbers:

```
+ ( float float -- 1st )  
- ( float float -- 1st )
```

So you don't have to write F+ and F-, as in Forth 2012. StrongForth chooses the correct versions of + and - if floating-point numbers are on top of the stack. Operator overloading allows StrongForth to generally get rid of special names for floating-point arithmetic operations. F+ becomes +, F- becomes -, FABS becomes abs and so on. This is a common standard in most other high-level programming languages.

Like in Forth 2012, + and - are accompanied by a set of shortcut words: 1+, 1- and +!. Even -! is available in StrongForth:

```
1+ ( integer -- 1st )  
1+ ( integer-double -- 1st )  
  
1- ( integer -- 1st )  
1- ( integer-double -- 1st )  
  
+! ( integer address -> integer -- )  
+! ( integer-double address -> integer-double -- )  
+! ( signed address -> integer-double -- )  
+! ( integer address -> integer-double -- )  
+! ( integer caddress -> integer -- )  
+! ( float address -> float -- )  
+! ( float sfaddress -> float -- )  
+! ( float dfaddress -> float -- )  
  
-! ( integer address -> integer -- )  
-! ( integer-double address -> integer-double -- )  
-! ( signed address -> integer-double -- )  
-! ( integer address -> integer-double -- )  
-! ( integer caddress -> integer -- )  
-! ( float address -> float -- )  
-! ( float sfaddress -> float -- )  
-! ( float dfaddress -> float -- )
```

Note that StrongForth also has overloaded versions of 1+ and 1- for double numbers. It is a general philosophy of StrongForth to provide complete sets of overloaded versions, so that each word works with as many different data types as seems reasonable.

More overloaded versions are required for `+` and `-`. The first version is identical to the Forth 2012 word `+`. Next, you have an overloaded version for double numbers, which would probably be called `D+`, if it had been specified in Forth 2012. The third and fourth version perform mixed-mode operations by adding an unsigned or signed single number to a double number in memory, respectively. Zero and sign extensions are properly applied. The next overloaded version adds a single number to a character size number in memory. Here's a simple example of how it might be used:

```
char A pad ! ok
7 pad .s +! unsigned caddress -> character ok
pad @ . H ok
```

Finally, three overloaded versions of `+` deal with floating-point numbers. Note that Forth 2012 does not specify anything like `C+`, `F+`, `SF+` or `DF+`.

Address Arithmetic

In the preceding section, you have seen how `+`, `-`, `1+`, `1-`, `+` and `-` are applied to integer and floating-point numbers. But what about addresses? In StrongForth, addresses are not numbers. And since address arithmetic has some special rules, StrongForth provides separate overloaded versions of these words for addresses.

What are the operations, a programmer usually wants to perform on addresses? Well, he or she generates addresses by defining variables and store data items in memory and fetch them from memory. Working with arrays, it further makes sense to add an index to an address, to subtract an index from an address and to calculate the difference between two addresses. On the other hand, things like multiplying an address with a constant factor, or even adding two addresses do not make sense at all. Arithmetic operations on addresses are limited to a rather small subset of what can be done with integers. That's the reason why data type `address` is not a subtype of data type `integer`. In fact, StrongForth provides overloaded versions of `+`, `-`, `1+`, `1-`, `+` and `-` for exclusive use on addresses: Here are the overloaded versions of `+` that can be applied to addresses:

```
+ ( address integer -- 1st )
+ ( address -> single integer -- 1st )
+ ( address -> double integer -- 1st )
+ ( address -> float integer -- 1st )
+ ( caddress integer -- 1st )
+ ( sfaddress integer -- 1st )
+ ( dfaddress integer -- 1st )
```

The first of these overloaded versions, `+ (address integer -- 1st)`, just adds an integer offset to any address. It has the same semantics as the version of `+` for two single-cell integers. The result of the addition has the same data type as the first operand, i. e., it is still an address. Note that the reverse operation, adding an address to an integer, is not supported. Adding integers to addresses is not a commutative operation. This means that you sometimes have to explicitly swap the operands to make sure the address is the first operand of an addition. Because of the general rule for all arithmetic words to return a result of the same data type as the first operand, adding an address to an integer would result in an integer instead of an address. This is in most cases not desired. If you consider the missing commutativity as a deficiency, you can easily define new overloaded versions of `+` like the this:

```
: + ( integer address -- 2nd )
  swap + ;
```

But what about the second overloaded version, `+ (address -> single integer -- 1st)`? Isn't this operation already covered by the first one? It is not. In StrongForth, adding an integer offset to an address of a single-cell item is semantically different from adding an integer to

an unspecific address. Imagine you have an array of 10 single-cell items, and you want to fetch the one with index 5. In Forth 2012, you'd write

```
ARRAY 5 CELLS + @
```

To calculate the address, you have to multiply the index by the number of address units per single cell using the word `CELLS`. In StrongForth, this explicit multiplication is not required. Because StrongForth's compiler knows that `ARRAY` returns the address of a single-cell item, it selects the version of `+` for data types `address -> single`, which automatically multiplies the address by the number of address units per cell before the addition is performed. If, for example, `array` returns an address of data type `address -> unsigned`, you can simply write

```
array 5 + @
```

to fetch the array item with index 5. The same mechanism works with addresses of other data types. For example, `+` (`address -> double integer -- 1st`) multiplies the integer offset with two times the number of address units per single cell before it is added to the address. Here are some demonstrations:

```
here dup . 5 + . 9053828 9053833 ok
here -> single dup . 5 + . 9053828 9053848 ok
here -> double dup . 5 + . 9053828 9053868 ok
here -> float dup . 5 + . 9053828 9053878 ok
here cast caddress -> single dup . 5 + . 9053828 9053833 ok
here cast sfaddress -> float dup . 5 + . 9053828 9053848 ok
here cast dfaddress -> float dup . 5 + . 9053828 9053868 ok
```

In this sample code, `here` is just used to generate a random address. Adding 5 to an unspecific address really increments the address by 5, while adding 5 to an address of a single-cell item, a double-cell item or a floating-point number increments the address by 20, 40 or 50, respectively. This means, on the system in use the number of address units per single cell is 4 and a floating-point number requires 10 address units. The integer offset is interpreted as an index. The resulting address is the original address plus the size of the specified number of items both addresses point to.

But what about the last three cases, where 5 is added to the address of character-size items and to addresses of single- and double-precision floating-point numbers? Although `+` (`address -> single integer -- 1st`) matches `caddress -> single unsigned`, the address is only incremented by 5. And this is correct, because the number of address units per character is 1 on the system in use. `+` (`caddress integer -- 1st`) takes care of the special case of adding integer offsets to character addresses. From the last two lines of the example you can conclude that a single-precision floating-point number takes 4 bytes in memory, and a double-precision floating-point number takes 8 bytes.

The order in which the different versions of `+` are being defined is important. `+` (`caddress integer -- 1st`) is found before `+` (`address -> single integer -- 1st`), because it handles the more special case. Adding an integer offset to a character address is independent of what the address points to, because the items always have character size. The most general overloaded version, `+` (`address integer -- 1st`) is the last one that matches the data types of the items on the stack. It takes care of the cases, where the system has no information at all about the size of the items the address points to. Therefore, the integer offset is just interpreted as address units.

Similarly, the versions of `+` for single-precision and double-precision floating-point numbers have to be found before `+` (`address -> float integer -- 1st`). Since `sfaddress` and `dfaddress` are subtypes of data type `address`, this version would also match if the data type of the item on the stack is `sfaddress -> float` or `dfaddress -> float`.

Since the compiler automatically selects the correct overloaded version of `+`, it relieves the programmer from thinking about proper address arithmetic. You usually don't have to apply `cells` or `chars` to adapt the integer offset to the size of the items stored at the address.

Note that there is no overloaded version of `+` for adding two addresses, because this operation makes no sense. It even does not make sense when calculating the mean of two addresses:

```
: mean ( address 1st -- 1st ) + 2/ ;  
: mean ( address 1st -- 1st ) + ? undefined word  
address address
```

This operation is required by some array sorting algorithms, to calculate the address of the element that is in the middle between the first and the last item. But bear in mind that the simple mean is not what you really want in this case, because the arithmetic mean does not necessarily point to an array element, if the address difference is not a multiple of the element size in address units. In order to ensure that the mean of two addresses always points to an element, `mean` has to be defined this way:

```
: mean ( address -> single 1st -- 1st ) over - 2/ + ; ok
```

We first calculate the total number of array elements between the two addresses using StrongForth's address arithmetic. After this value has been divided by two, we add it to the first address, taking again advantage of address arithmetic. The result is guaranteed to be a valid pointer to the array element in the middle of the two addresses, no matter whether the second address is lower or higher than the first one. Of course, we'd have to define overloaded versions of `mean` for all possible element sizes, i. e., single-cell, double-cell, character-size, floating-point and single- and double-precision floating point.

The overloaded versions of `-` resemble those of `+`, so we don't need to discuss them in detail:

```
- ( address integer -- 1st )  
- ( address -> single integer -- 1st )  
- ( address -> double integer -- 1st )  
- ( address -> float integer -- 1st )  
- ( caddress integer -- 1st )  
- ( sfaddress integer -- 1st )  
- ( dfaddress integer -- 1st )
```

But that's not all. You certainly have noticed that the above definition of `mean` contains one version of `-` that has no `+`-counterpart. While the sum of two addresses cannot be interpreted in a meaningful way, it makes a lot of sense to calculate the difference of two addresses. The result is a signed number, not an address. It tells you how many address units lie between the two addresses, and if we add address arithmetic and assume the two addresses point to elements of the same array, it can even tell you how many elements of a given size fit between the two addresses. That's the calculation that is performed in `mean`. Actually, StrongForth provides a complete set of overloaded versions for calculating address differences:

```
- ( address 1st -- signed )  
- ( address -> single 1st -- signed )  
- ( address -> double 1st -- signed )  
- ( address -> float 1st -- signed )  
- ( caddress 1st -- signed )  
- ( sfaddress 1st -- signed )  
- ( dfaddress 1st -- signed )
```

You've already seen a sample application in the definition of `mean`. Here's another example:

```
here -> unsigned 123 , 456 , 789 , here -> unsigned over - . 3 ok  
cast address here - . -12 ok
```

3 single-cell unsigned numbers have been added to the data space, occupying a total of 3 cells address units. The minus sign is caused by the fact that the two addresses are swapped in the second calculation. Note that both address parameters need to have exactly the same data types, because it is assumed they point to elements within the same array.

Of course, + and - are not the only operations performing address arithmetic. 1+, 1-, +! and -!, and even (loop), (+loop) and (-loop), the runtime parts of loop, +loop and -loop, do it as well. Again, seven overloaded versions for each of these words are required:

```

1+ ( address -- 1st )
1+ ( address -> single -- 1st )
1+ ( address -> double -- 1st )
1+ ( address -> float -- 1st )
1+ ( caddress -- 1st )
1+ ( sfaddress -- 1st )
1+ ( dfaddress -- 1st )

1- ( address -- 1st )
1- ( address -> single -- 1st )
1- ( address -> double -- 1st )
1- ( address -> float -- 1st )
1- ( caddress -- 1st )
1- ( sfaddress -- 1st )
1- ( dfaddress -- 1st )

+! ( integer address -> address -- )
+! ( integer address -> address -> single -- )
+! ( integer address -> address -> double -- )
+! ( integer address -> address -> float -- )
+! ( integer address -> caddress -- )
+! ( integer address -> sfaddress -- )
+! ( integer address -> dfaddress -- )

-! ( integer address -> address -- )
-! ( integer address -> address -> single -- )
-! ( integer address -> address -> double -- )
-! ( integer address -> address -> float -- )
-! ( integer address -> caddress -- )
-! ( integer address -> sfaddress -- )
-! ( integer address -> dfaddress -- )

(loop) ( address -- flag )
(loop) ( address -> single -- flag )
(loop) ( address -> double -- flag )
(loop) ( address -> float -- flag )
(loop) ( caddress -- flag )
(loop) ( sfaddress -- flag )
(loop) ( dfaddress -- flag )

(+loop) ( integer address -- flag )
(+loop) ( integer address -> single -- flag )
(+loop) ( integer address -> double -- flag )
(+loop) ( integer address -> float -- flag )
(+loop) ( integer caddress -- flag )
(+loop) ( integer sfaddress -- flag )
(+loop) ( integer dfaddress -- flag )

```

```
(-loop) ( integer address -- flag )
(-loop) ( integer address -> single -- flag )
(-loop) ( integer address -> double -- flag )
(-loop) ( integer address -> float -- flag )
(-loop) ( integer caddress -- flag )
(-loop) ( integer sfaddress -- flag )
(-loop) ( integer dfaddress -- flag )
```

Here's an example of how address arithmetic is performed within a do loop:

```
: total ( address -> integer 1st -- 2nd )
  null integer rot rot swap ?do i @ + loop ; ok
here -> unsigned 12 , 55 , 136 , 7 , ok
here -> unsigned total . 210 ok
```

The + operation within the body of the do loop is pure integer arithmetic. However, address arithmetic is hidden in loop, which compiles (loop) to increment the loop index by cell size, because the loop index is of data type address -> integer.

To emphasise the difference, this is how total would be implemented in Forth 2012:

```
: TOTAL ( addr1 addr2 -- n )
  0 ROT ROT SWAP ?DO I @ + 1 CELLS +LOOP ;
```

Apart from the type cast required in the StrongForth version, address arithmetic makes the difference. In Forth 2012, address arithmetic is explicitly performed by using CELLS to ensure the address is incremented by cell size. Since address arithmetic is implicit in StrongForth, the Forth 2012 words CELLS, CELL+, CHARS, CHAR+, FLOATS and FLOAT+ are less often used. Anyway, cells, chars and floats do exist in StrongForth, because they are still required in connection with low-level words like allot and allocate. cell+, char+ and float+, on the other hand, are fully replaced by overloaded versions of 1+ with address arithmetic.

Multiplication and Division

Forth 2012, and also StrongForth, provides a variety of words for multiplication and division. Let's begin with a simple multiplication. This is what StrongForth has to offer:

```
* ( integer unsigned -- 1st )
* ( signed signed -- 1st )
* ( integer-double unsigned -- 1st )
* ( signed-double signed -- 1st )
* ( float float -- 1st )
```

That's not as much as for addition and subtraction, but of course there's no address arithmetic for multiplication. It makes no sense at all to multiply or divide two addresses, or even an address and an integer number.

Multiplying two unsigned or two signed numbers yields an unsigned or signed number, respectively. If you multiply a signed number with an unsigned number, the result is a signed number. But, StrongForth does not allow multiplying an unsigned number with a signed number. The result would have to be a signed number, which violates the general rule for all arithmetic operations in StrongForth, which says that the data type of the result is always identical to the data type of the first operand. You have to swap the operands in order to perform this operation.

Multiplying a signed or unsigned double-precision number by a signed or unsigned single-precision number is overloaded as well. Again, the data type of the result is identical to the data type of the first operand. There's no Forth 2012 equivalent to these operations. The last overloaded version of

* implements the Forth 2012 word `F*` for floating-point numbers. As with `F+` and `F-`, in StrongForth there's no need to keep the prefix `F` for a floating-point operation.

Sometimes you might want the result of multiplying two single-precision numbers to be a double-precision number. The result of this operation is violating the general rule, because the product does not have the same data type as the multiplicand. StrongForth provides two additional multiplication words, which implement the Forth 2012 words `M*` and `UM*`:

```
m* ( signed signed -- signed-double )
m* ( unsigned unsigned -- unsigned-double )
```

These two words can not be overloaded versions of `*`, because the compiler would not be able to distinguish them from normal single-cell multiplications by the input parameters.

An interesting detail is the fact that multiplication with an unsigned single-precision number allows the first operand to be either signed or unsigned, while this is not possible for `m*`. The reason is that only the most significant part of the product depends on whether the multiplicand is signed or unsigned, while the least significant word is identical in both cases.

Division is a similar case. The result depends on whether the operands are interpreted as signed or unsigned numbers. Therefore, signed and unsigned division are completely different operations. However, all plain division words are overloaded:

```
/ ( unsigned unsigned -- 1st )
/ ( signed signed -- 1st )
/ ( unsigned-double unsigned -- 1st )
/ ( signed-double signed -- 1st )
/ ( float float -- 1st )
```

Since `/` has to be more specific on the data types of its arguments, it can not be applied to items of data types `integer`, `integer-double` or `character`. The signed/unsigned attribute of these three data types is supposed to be undefined. But, does it really make sense to perform a division on an ASCII character or any other number without signed/unsigned attribute? If it's still required in one of your StrongForth applications, you can use a type cast:

```
char x cast unsigned 3 / cast character . ( ok
```

In addition to `/`, StrongForth provides `m/` for situations, where you need a single-cell quotient from a double-cell dividend:

```
m/ ( unsigned-double unsigned -- unsigned )
m/ ( signed-double signed -- signed )
```

Instead of using these two words, you could also use the plain version of `/` for double-cell dividends, and then cast the double-cell quotient to a single-cell data type, cutting off the most significant cell from the result.

`mod` and `/mod` are close relatives to `/`. They perform divisions with the same operand combinations, except for floating-point numbers:

```
mod ( unsigned unsigned -- 2nd )
mod ( signed signed -- 2nd )
mod ( unsigned-double unsigned -- 2nd )
mod ( signed-double signed -- 2nd )

/mod ( unsigned unsigned -- 2nd 1st )
/mod ( signed signed -- 2nd 1st )
/mod ( unsigned-double unsigned -- 2nd 1st )
/mod ( signed-double signed -- 2nd 1st )
```

Against the rule, the result of `mod` as well as the remainder returned by `/mod` do not have the same data type as the first operand. The reason is that the remainder is related to the divisor instead of to the dividend. Its absolute value is always less than that of the divisor. This is most obvious when considering the versions with a double-cell dividend. Although the data type of the dividend is unsigned-double or signed-double, the result will always fit into a single-precision number of the same data type as the divisor.

Next, there's `*/` and `*/mod`:

```
*/ ( unsigned unsigned unsigned -- 1st )
*/ ( signed signed signed -- 1st )
*/ ( unsigned-double unsigned unsigned -- 1st )
*/ ( signed-double signed signed -- 1st )

*/mod ( unsigned unsigned unsigned -- 3rd 1st )
*/mod ( signed signed signed -- 3rd 1st )
*/mod ( unsigned-double unsigned unsigned -- 3rd 1st )
*/mod ( signed-double signed signed -- 3rd 1st )
```

Again, StrongForth provides overloaded versions for signed and unsigned, single-cell and double-cell numbers. The versions for double-cell numbers use triple-precision intermediate results, i. e. they multiply a double-precision number with a single-precision number, giving a triple-precision intermediate result. The intermediate result is then divided by a single-precision number, giving a double-precision quotient and, in the case of `*/mod`, a single-precision remainder. `*/` for signed double-cell numbers is equivalent to the Forth 2012 word `M*`, and it allows negative divisors. It is, like `*/mod`, defined in StrongForth source code:

```
: */mod ( signed-double signed signed -- 3rd 1st )
  cast unsigned rot dup 0<
  if negate cast unsigned-double rot dup 0<
    if negate cast unsigned rot */mod swap
    else cast unsigned rot */mod negate swap negate
    then
  else cast unsigned-double rot dup 0<
    if negate cast unsigned rot */mod negate swap negate
    else cast unsigned rot */mod swap
    then
  then cast signed swap cast signed-double ;

: */ ( signed-double signed signed -- 1st )
  */mod nip ;
```

The Forth 2012 words `FM/MOD` and `SM/REM` are also available in StrongForth. and `UM/MOD` has been renamed to `m/mod`. Data type prefixes are superfluous, because Strongforth keeps track of the data types itself. To complete the set of words, a signed version of `m/mod` has been added. Its semantic is identical to that of `sm/rem`.

```
sm/rem ( signed-double signed -- 2nd signed )
fm/mod ( signed-double signed -- 2nd signed )
m/mod ( unsigned-double unsigned -- 2nd unsigned )
m/mod ( signed-double signed -- 2nd signed )
```

Note that the data type of the remainder is identical to the data type of the divisor, but the single-cell quotient does not necessarily have the same data type as the dividend, because the dividend has to be a double-cell number.

Finally, let's see how `2*` and `2/` look like in StrongForth:

```
2* ( integer -- 1st )
2* ( integer-double -- 1st )

2/ ( integer -- 1st )
2/ ( integer-double -- 1st )
2/ ( signed -- 1st )
2/ ( signed-double -- 1st )
```

Because the semantics of `2*` is the same for signed and unsigned numbers, only two overloaded versions suffice. `unsigned` and `signed` are both children of data type `integer`, so `2*` for single numbers applies to both signed and unsigned numbers. `2*` for double numbers replaces the Forth 2012 word `D2*`.

In contrast to that, the semantics of `2/` depends on whether the operand is signed or unsigned, because the most significant bit has to be preserved for signed numbers and not for unsigned numbers. This means StrongForth has to provide a total of four separate overloaded versions for unsigned single numbers, signed single numbers, unsigned double numbers, and signed double numbers. The Forth 2012 word `D2/` is replaced by the version of `2/` for signed double numbers.

Other Arithmetic Operations

abs

The Forth 2012 words `ABS`, `DABS` and `FABS` fit nicely into the concept of StrongForth. As usual, StrongForth provides the double number version and the floating-point version by overloading the single number version:

```
abs ( integer -- 1st )
abs ( integer-double -- 1st )
abs ( float -- 1st )
```

Following the general rule, the data type of the result is identical to the data type of the (first) operand. Although calculating the absolute value of an unsigned number makes no sense, `abs` is not restricted to explicit signed values only, but accepts data types `integer` and `integer-double`. The reason is that in some cases, an item of data type `integer` or `integer-double` can represent a signed number. However, be careful when you apply `abs` to an unsigned number. The result can be strange if the operand is greater than the maximum positive signed number, as demonstrated in this example:

```
2000000000 abs . 2000000000 ok
3000000000 abs . 1294967296 ok
```

Quite often, the absolute value is calculated with the intention to make a signed number unsigned. In those cases, the result of `abs` has to be explicitly casted to the desired unsigned data type.

negate

`negate` is closely related to `abs`, because both words change the sign of their respective operands. The three overloaded versions of `negate` implement the semantics of the Forth 2012 words `NEGATE`, `DNEGATE` and `FNEGATE`:

```
negate ( integer -- 1st )
negate ( integer-double -- 1st )
negate ( float -- 1st )
```

Because applying `negate` to a single-cell or double-cell unsigned number also makes sense, it allows operands of data type `integer` or `integer-double` as well. Typical applications for negating unsigned integer numbers arise in connection with `allot`. If the (unsigned) number of allotted address units is on the stack and you want to de-allot them, you could simply write `negate allot`, without casting the unsigned number to data type `signed` before negating it. Just bear in mind that you cannot negate an unsigned number whose value is larger than the maximum positive signed number.

However, if you do not immediately hand over the result of a negated unsigned number to a word that expects a signed number of data type `integer`, make sure to add a type cast making it signed. Otherwise, something like in the second line of this example could happen:

```
+41 .s negate . signed -41 ok
41 .s negate . unsigned 4294967255 ok
```

A typical application of `NEGATE` in Forth 2012 is subtracting an unsigned number from the contents of a memory cell using `+`!. In StrongForth, the phrase `negate +!` can be shortened to `-!`.

min and max

Let's see what StrongForth has to offer for calculating the minimum and the maximum of two values:

```
min ( integer 1st -- 1st )
min ( address 1st -- 1st )
min ( signed 1st -- 1st )
min ( integer-double 1st -- 1st )
min ( signed-double 1st -- 1st )
min ( float 1st -- 1st )

max ( integer 1st -- 1st )
max ( address 1st -- 1st )
max ( signed 1st -- 1st )
max ( integer-double 1st -- 1st )
max ( signed-double 1st -- 1st )
max ( float 1st -- 1st )
```

All of these versions have in common, that the two operands and the result have exactly the same data type. It makes no sense to calculate, let's say, the maximum of an address and a character.

Now, what are the operands `min` and `max` can be applied to? The versions for items of data type `integer` can also be applied to items of data type `unsigned` and `character`, because these are children of `integer`. But isn't `signed` another child of `integer`, and `signed-double` a child of `integer-double`? Since there are special versions of `min` and `max` for items of data types `signed` and `signed-double`, which will be found in the dictionary before the versions for items of data type `integer` and `integer-double`, the latter ones have no chance to get applied to signed integer numbers. Actually, the signed versions of `min` and `max` implement the semantics of the respective Forth 2012 words `MIN`, `MAX`, `DMIN` and `DMAX`. The versions for items of data type `integer` and `integer-double` have an unsigned semantics and might have been named `UMIN` etc. in Forth 2012, if they had been specified.

In addition to the versions for integer numbers, StrongForth provides overloaded versions of `min` and `max` for data types `address` and `float`. The versions for data type `address` have the same semantics as the respective versions for unsigned numbers. The versions for floating-point numbers are equivalent to `FMIN` and `FMAX` in Forth 2012.

Comparison Operators

Comparison operators are `<`, `>`, `=`, `<>`, `<=` and `>=` as well as `0<`, `0>`, `0=`, `0<>`, `0<=` and `0>=`. `<=`, `=>`, `0<=` and `0>=` have been added with respect to the Forth 2012 standard in order to provide a complete set of comparison operators like in other programming languages, and reduce the necessity of using `invert`.

The stack diagrams of `<`, `>`, `<=` and `>=` resemble those of `min` and `max`, with the obvious difference that they leave a flag on the stack instead of an item of the same data type as the operands:

```
< ( integer 1st -- flag )
< ( address 1st -- flag )
< ( signed 1st -- flag )
< ( integer-double 1st -- flag )
< ( signed-double 1st -- flag )
< ( float 1st -- flag )

> ( integer 1st -- flag )
> ( address 1st -- flag )
> ( signed 1st -- flag )
> ( integer-double 1st -- flag )
> ( signed-double 1st -- flag )
> ( float 1st -- flag )

<= ( integer 1st -- flag )
<= ( address 1st -- flag )
<= ( signed 1st -- flag )
<= ( integer-double 1st -- flag )
<= ( signed-double 1st -- flag )
<= ( float 1st -- flag )

>= ( integer 1st -- flag )
>= ( address 1st -- flag )
>= ( signed 1st -- flag )
>= ( integer-double 1st -- flag )
>= ( signed-double 1st -- flag )
>= ( float 1st -- flag )
```

`=` and `<>` are different. These words can be applied not only to pairs of numbers, but to pairs of any data type, as long as the data types of both operands are exactly the same. Being applicable to any data type means in StrongForth, that three overloaded versions are available: one for `single`, one for `double`, and one for `float`:

```
= ( single 1st -- flag )
= ( double 1st -- flag )
= ( float 1st -- flag )

<> ( single 1st -- flag )
<> ( double 1st -- flag )
<> ( float 1st -- flag )
```

Unsigned numbers and addresses can never be negative. Therefore, `0<`, `0>`, `0<=` and `0>=` do only exist for signed single-cell and double-cell numbers and for floating-point numbers:

```
0< ( signed -- flag )
0< ( signed-double -- flag )
0< ( float -- flag )
```



```

0> ( signed -- flag )
0> ( signed-double -- flag )
0> ( float -- flag )

0<= ( signed -- flag )
0<= ( signed-double -- flag )
0<= ( float -- flag )

0>= ( signed -- flag )
0>= ( signed-double -- flag )
0>= ( float -- flag )

```

Like = and <>, 0= and 0<> can be applied to items of any data type:

```

0= ( single -- flag )
0= ( double -- flag )
0= ( float -- flag )

0<> ( single -- flag )
0<> ( double -- flag )
0<> ( float -- flag )

```

One comparison word is still missing: `within`. Because the semantics of `within` does not differ for signed and unsigned numbers, one version for items of data type `integer` and one for items of data type `address` suffices:

```

: within ( integer 1st 1st -- flag )
  over - rot rot - swap < ;

: within ( address 1st 1st -- flag )
  over - cast integer rot rot - cast integer swap < ;

```

Remember that `address` is not a subtype of `integer`. The data types of the three operands must be identical, because it normally makes no sense to compare items of different data types. If it's still necessary for whatever reasons, use type casts. An overloaded version for double-cell numbers is not provided. It can easily be added to StrongForth's dictionary:

```

: within ( integer-double 1st 1st -- flag )
  over - rot rot - swap < ;

```

Since all words used in these definitions are overloaded for data types `integer` and `integer-double`, the source code looks identical in both cases. In Forth 2012, one would write

```

: WITHIN ( n lo-limit hi-limit+1 -- f )
  OVER - ROT ROT - SWAP U< ;

```

to define `WITHIN` for single-precision numbers. Note that StrongForth uses the overloaded versions of `<` for unsigned numbers instead of `U<`. The two type casts in the definition of `within` for addresses are necessary, because subtracting two addresses results in a signed number. Without the type casts, StrongForth would compile the signed version of `<` instead of the unsigned version.

In addition to `within`, StrongForth provides words that perform range checks:

```

single? ( integer-double -- flag )
single? ( signed-double -- flag )
byte? ( integer -- flag )
byte? ( signed -- flag )

```

`single?` returns `true` if its double-cell numeric parameter can be represented as a single-cell number with the same value. Otherwise it returns `false`. Two overloaded versions for unsigned and signed numbers are available. `byte?` checks whether a single-cell number fits into a byte, i.e.,

if the value is between 0 and 255 (unsigned version) or -128 and +127 (signed version). These four words could be implemented with `within`, but the machine code versions are more efficient. Here are some examples:

```
max-unsigned . 4294967295 ok
4000000000. single? . true ok
5000000000. single? . false ok
-130 byte? . false ok
+125 byte? . true ok
```

Based on `single?` and `byte?`, StrongForth provides similar words that do not return a flag. Instead, each of them returns its unchanged input parameter and throws an exception if the range check is not passed:

```
: ?range ( flag -- )
  invert if -289 throw then ;

: ?single ( integer-double -- 1st )
  dup single? ?range ;

: ?single ( signed-double -- 1st )
  dup single? ?range ;

: ?byte ( integer -- 1st )
  dup byte? ?range ;

: ?byte ( signed -- 1st )
  dup byte? ?range ;
```

Processor flags

Other than Forth 2012, Strongforth provides access to the carry, overflow and parity flags in the processor's `eflags` register. After an arithmetic operation, you can examine the carry and overflow flags with words that return a respective flag:

```
carry? ( -- flag )
-carry? ( -- flag )
overflow? ( -- flag )
-overflow? ( -- flag )
```

The words with the leading minus character return the inverse flags. With these four words, it is possible to detect numeric overflows directly, as in these examples:

```
max-unsigned . 4294967295 ok
2000000000 2000000000 + carry? . false ok
2000000000 3000000000 + carry? . true ok
max-signed . 2147483647 ok
-1000000000 +1000000000 - overflow? . false ok
-1000000000 +2000000000 - overflow? . true ok
```

You can use this feature to implement arithmetic operations on integers that are longer than a double number. However, it is certainly more efficient to implement arithmetic operations on very large numbers using assembly code.

If you just need to assert that no carry or overflow occurred as a result of the preceeding arithmetic operation, you can use the corresponding version of `?overflow`:

```
?overflow ( integer -- 1st )
?overflow ( signed -- 1st )
?overflow ( integer-double -- 1st )
?overflow ( signed-double -- 1st )
?overflow ( address -- 1st )
```

These five overloaded words return their input parameter instead of a flag. If an unsigned arithmetic operation caused a carry, or a signed operation caused a numeric overflow, an exception is being thrown:

```
null address 2 + ?overflow . 2 ok
null address 2 - ?overflow
null address 2 - ?overflow ? numeric overflow
address
+2147483640 7 + ?overflow . 2147483647 ok
+2147483640 8 + ?overflow
+2147483640 8 + ?overflow ? numeric overflow
signed
```

The parity of characters can be obtained with the words `even-parity?` and `odd-parity?`:

```
even-parity? ( character -- flag )
odd-parity? ( character -- flag )
```

Here are some examples:

```
binary ok
char A dup cast single . even-parity? . 1000001 true ok
char a dup cast single . even-parity? . 1100001 false ok
char C dup cast single . odd-parity? . 1000011 true ok
char c dup cast single . odd-parity? . 1100011 false ok
decimal ok
```

Logical Operations

Bitwise Logical Operations

Like Forth 2012, StrongForth provides the usual binary and unary bitwise logical operations:

```
and ( single logical -- 1st )
or ( single logical -- 1st )
xor ( single logical -- 1st )
invert ( logical -- 1st )
```

Apart from the stack diagrams, these operations bear no surprise. As usual, the data type of the result is identical to the data type of the first operand. But maybe you expected all parameters to be of data type `single`, so it would be possible to apply logical operations to all kinds of operands. Why are the second operand and the only operand of `invert` restricted to items of data type `logical`?

If arbitrary data types were allowed, even obviously meaningless operations, like a logical `and` of an address with a character, would be possible. To understand the restriction, consider the typical applications of logical operations.

First of all, logical operations are used to calculate a condition. In this case, both operands are of data type `flag`, which is a child of `logical`. This causes no problem:

```
6 value x ok
x 4 > state @ and . false ok
```

Second, logical operations are frequently used to set, clear and test specific bits in any kind of operand. Since data type `logical` is a bit vector by definition, it's perfectly suited for this purpose. StrongForth even provides a word `bit` that delivers an item of data type `logical` with one bit set:

```
: bit ( unsigned -- logical )
  1 cast logical swap lshift ;
```

With this word and the logical operations, it's easy to set, clear and test individual bits in items of any data type. Here are some examples:

```
5 bit .s . logical 32 ok
11 bit . 2048 ok
char A 2 bit or . E ok
hex ABCD decimal 12 bit and 0= . true ok
```

Logical Constants

The logical constants `true` and `false` are both of data type `flag`:

```
null flag constant false
false invert constant true
```

Since Forth 2012 does not support data types, there is no difference between 0 and `false`, or -1 and `true`. But this is different in StrongForth. Consider a situation where you want to define a flag variable:

```
0 variable running
```

defines `running` as a constant of data type `unsigned`. This means, it can not be used in logical expressions like this one:

```
>in @ 6 > running @ and .
>in @ 6 > running @ and ? undefined word
flag unsigned
```

The interpreter complains, because `and` does not accept an item of data type `unsigned` as the second parameter. If `running` were defined as a variable of data type `flag`, the previous example would work. Note that only items of data type `flag` can now be stored in `running`:

```
false variable running ok
>in @ 6 > running @ and . false ok
true running ! ok
-1 running !
-1 running ! ? undefined word
signed address -> flag
```

Shift Operations

Forth 2012 specifies two shift operations, `LSHIFT` and `RSHIFT`, which are defined in StrongForth as well:

```
lshift ( logical unsigned -- 1st )
rshift ( logical unsigned -- 1st )
```

The first operand of these two words is of data type `logical`, and the result has the same data type. Items of data type `integer` cannot be shifted, because shifting is a logical operation, while operations on integers are arithmetic by definition. StrongForth strongly distinguishes between arithmetical and logical operations. To *shift* an integer value, you have to use either a multiplication or a type cast:

```
15 3 lshift .
15 3 lshift ? undefined word
unsigned unsigned
15 8 * . 120 ok
15 cast logical 3 lshift . 120 ok
```

Instead of multiplying by 8, you can also apply three times `2*` to the operand. `2*` and `2/` are in turn arithmetic operations that don't apply to logical values:

```
15 2* 2* 2* . 120 ok
6 bit 2* .
6 bit 2* ? undefined word
logical
```

An obvious solution is to do a logical left shift by one:

```
6 bit 1 lshift . 128 ok
```

Actually, StrongForth provides overloaded versions of `lshift` and `rshift` that just shift by one bit:

```
lshift ( logical -- 1st )
rshift ( logical -- 1st )
```

The unary versions of `lshift` and `rshift` are replacements of `2*` and `2/` for items of data type `logical`. Since `unsigned` is not a subtype of `logical` and vice versa, the decision on which version to compile or interpret will never be ambiguous.

In addition to `lshift` and `rshift`, StrongForth provides `lrotate` and `rrotate`, which rotate the bits of a cell instead of shifting them. No equivalent words are specified in Forth 2012.

However, these words might be useful when dealing with bit fields in some situations. Again, overloaded versions for rotating by a single bit are available:

```
lrotate ( logical unsigned -- 1st )
lrotate ( logical -- 1st )
rrotate ( logical unsigned -- 1st )
rrotate ( logical -- 1st )
```

Here's an example of how to use these words:

```
hex 12345678 cast logical 4 rrotate . 81234567 ok
binary 11110101101000001011110110101110 cast logical ok
lrotate . 11101011010000010111101101011101 ok
decimal ok
```

Bit Queries

StrongForth provides two additional words for logical operations that are not specified by Forth 2012:

```
: set? ( single logical -- flag )
  and 0<> ;

: clear? ( single logical -- flag )
  and 0= ;
```

`set?` returns `true` if *any* of the bits set in the second parameter is also set in the first operand, and `false` otherwise. `clear?` is the complementary operation. It returns `true` if and only if *all* bits that are set in the second operand are cleared in the first operand.

These two operations can be used to investigate the state of one or more bits in the first operand. The advantage of using `set?` and `clear?` instead of `and` plus `0<>` or `0=` is that they produce very efficient machine code. Here are a few simple examples:

```
hex   ok
12345678 4 bit set? . true  ok
12345678 0 bit 1 bit or set? . false  ok
12345678 7 bit clear? . true  ok
decimal  ok
```

4 Memory Spaces

What's in a Memory Space?

The Forth 2012 specification mentions three different memory spaces: *name space*, *code space* and *data space*. In order to allow various existing Forth systems to be Forth 2012 compliant, the semantics of these memory spaces are supposed to be implementation dependent.

StrongForth has a code space and a data space as well. The code space contains only executable machine code, as it is compiled and assembled following high-level words like `:`, `:noname` and `code`. An execution token with data type `token` is actually a pointer to a location in the code space. Like in Forth 2012, the data space is used to store variables, values, data for `created` words and other data. `>body`, when applied to the definition of a `created` word, returns a pointer to a location in the data space.

However, there is no explicit name space in StrongForth. New definitions, including their names and stack diagrams, are allocated in dynamic memory. Generally, all newly created objects obtain memory space for their members from dynamic memory. The advantage of using dynamic memory is that objects can be deleted without leaving holes in the dictionary.

As a third predefined memory space, StrongForth has the stack space. This memory space contains, as you might have guessed, the return stack. Additionally, it can be used to store process-owned variables, which are called *user variables* in some other Forth systems. Each independent process has its own stack space. The code and data space, on the other hand, are shared by all processes.

In addition to those three predefined memory spaces, you can define your own application-specific memory spaces. Actually, a memory space is an object of class `memory-space`. The three constants

```
data-space ( -- memory-space )
code-space ( -- memory-space )
stack-space ( -- memory-space )
```

return the predefined memory spaces.

```
10000 new memory-space constant my-space
```

creates a new memory space with a size of 10000 address units, and makes it available as a constant with the given name.

The internal data structure of class `memory-space` is hidden in its private members. But it has a number of methods you can use to manipulate its objects and to obtain information about them:

```
memory-space ( address unsigned memory-space -- 3rd )
memory-space ( unsigned memory-space - 2nd )
here ( memory-space -- address )
chere ( memory-space -- caddress )
sfhere ( memory-space -- sfaddress )
dfhere ( memory-space -- dfaddress )
unused ( memory-space -- unsigned )
allot ( integer memory-space -- )
shrink ( integer memory-space -- )
```

The first two methods are the so-called constructors of class `memory-space`. Constructors always have the same name as the class they belong to. Their last input parameter and their one and only output parameter must be objects of the class. They perform the initialization of the new

object by assigning initial values to its members and doing other initialization stuff after the space for the members has been allocated in dynamic memory. The first constructor creates a memory space with a size of `unsigned address units` at a given address `address`. The second one does also create a memory space with `unsigned address units`, but it automatically allocates it from dynamic memory.

Class `memory-space` has actually one more method, which it inherits from class `object`, the common ancestor of all classes. The virtual method `delete (object --)` is the destructor of all objects. Whenever a destructor of a specific object is executed, it will destruct the object by cleaning up and deallocating all memory that has been allocated for the object. Since these tasks will naturally vary depending on the kind of the actual object, a dedicated version of `delete` exists for most classes. Virtual methods resolve their data type at runtime, which means that even something like

```
my-class cast object delete
```

calls the destructor of class `my-space`, not the one of class `object`.

`here` returns the first unused address of a memory space. The data type of this address is just `address`, because it is generally not known at compile time which data type this address points to. `chere`, `sfhere` and `dfhere` do exactly the same as `here`, but return a different address data type. Using them can avoid explicit type casts in many cases.

`unused` returns the number of address units still remaining in a memory space. This is an `unsigned number`, because the amount of free memory cannot be negative.

`allot (integer memory-space --)` reserves `integer` addressing units in `memory-space`, starting at `here`. Since `integer` is interpreted as a signed number, the method can also be used to free a given number of address units. Its your responsibility has to take care about the size of the items you want to reserve space for. The preferred way to do this is by with `chars`, `cells`, `floats`, `sfloats` and `dfloats`:

```
5 2* cells data-space allot
```

allocates space for 5 double-cell items in the data space.

```
18 chars my-space allot
```

allocates space for 18 character-size items in `my-space`. Note that `my-space` might be unaligned after allocating space for characters.

Finally, `shrink` can be used to reduce the size of a memory space by `integer` address units. `integer` must be a positive number. An exception is thrown if the number of unused address units in `memory-space` is less than `integer`.

Here's a list of the above mentioned words that calculate sizes in address units:

```
cells ( integer -- 1st )
chars ( integer -- 1st )
floats ( integer -- 1st )
sfloats ( integer -- 1st )
dfloats ( integer -- 1st )
```

In StrongForth, these words are only needed for low-level address arithmetic with unspecified addresses, because address arithmetic with explicit addresses of cells, characters or floating-point numbers automatically considers the size of those items in memory. Examples of words that produce unspecified addresses are `here` and `allocate`:


```

here .s . address 8791632 ok
here 2 + . 8791634 ok
here 2 cells + . 8791640 ok
here -> single 2 + . 8791640 ok

```

The Default Memory Space

Three of the methods of class `memory-space` have the same names and similar semantics as the respective Forth 2012 words `HERE`, `ALLOT` and `UNUSED`. The difference is that they expect an object of data type `memory-space` as an input parameter, whereas the Forth 2012 words always refer to the data space. Of course, it would be possible to overload the three methods like this:

```
: here ( -- address ) data-space here ; 1 retreat
```

The phrase `1 retreat` at the end of the definition ensures that this overloaded version of `here` will be found in the dictionary *after* the method `here` that was presented in the previous section, although it was defined later. Otherwise, it would hide the method.

Anyway, StrongForth provides a more versatile solution:

```
data-space variable default-memory-space
```

```

: default ( memory-space -- )
  default-memory-space ! ;

: here ( -- address )
  default-memory-space @ here ; 1 retreat

: chere ( -- caddress )
  here cast caddress ; 1 retreat

: sfhere ( -- sfaddress )
  here cast sfaddress ; 1 retreat

: dfhere ( -- dfaddress )
  here cast dfaddress ; 1 retreat

: unused ( -- unsigned )
  default-memory-space @ unused ; 1 retreat

: allot ( integer -- )
  default-memory-space @ allot ; 1 retreat

: shrink ( integer -- )
  default-memory-space @ shrink ; 1 retreat

```

Using these definitions you can declare any memory space being the default and then use `here` and the other words to operate on this memory space:

```

data-space default ok
unused . 5 cells allot unused . 55756 55736 ok

```

Memory Alignment

Although the x86 processor architecture allows reading and writing cells from non-aligned memory addresses, it is more efficient accessing cells in memory at aligned addresses. Therefore, StrongForth aligns addresses to multiples of 1 cells. The basic alignment word is defined for two unsigned numbers:

```
: aligned ( unsigned unsigned -- 1st )
  over over mod dup if - + else drop drop then ;
```

If the first unsigned is a multiple of the second unsigned, 1st is equal to the first unsigned. Otherwise, 1st equal to the lowest value greater than the first unsigned, that is a multiple of the second unsigned. With this word, an unsigned number can be aligned to multiples of any operand size. An application is the StrongForth equivalent of the Forth 2012 word `ALIGNED`, which aligns an item of data type address to cell size:

```
: aligned ( address -- 1st )
  cast unsigned [ 1 cells ] literal aligned cast address ;
```

The two words can share the same name, because they apply to different data types, none of which is a direct or indirect subtype of the other one. With this word in turn, we can define an alignment operator for a given memory space:

```
: align ( memory-space -- )
  dup here dup aligned swap - swap allot ;
```

Finally, we're ready to define StrongForth's equivalent of the Forth 2012 word `ALIGN`, which aligns the default memory space:

```
: align ( -- )
  default-memory-space @ align ; 1 retreat
```

Storing Data in Memory Spaces

Forth 2012 specifies two words that reserve memory in the data space and store data in it: `,` and `C,`. If it becomes necessary to store a pair of cells or a double-cell value in the data space, one could just write `,` `,`. Floating-point numbers are not considered. StrongForth, on the other hand, provides a complete set of partly overloaded reserve-and-store words for data types of all sizes: single cells, double cells, floating-point numbers, character-size items, and single- and double-precision floating-point numbers. In addition to that, there's a word for character strings given by the address of the first character and the length of the string. Here are the words that operate on a specific memory space:

```
: , ( single memory-space -- )
  dup here -> single [ 1 cells ] literal rot allot ! ;

: , ( double memory-space -- )
  dup here -> double [ 2 cells ] literal rot allot ! ;

: , ( float memory-space -- )
  dup here -> float [ 1 floats ] literal rot allot ! ;

: c, ( single memory-space -- )
  dup here -> single [ 1 chars ] literal rot allot ! ;

: sf, ( float memory-space -- )
  dup sfhere -> float [ 1 cells ] literal rot allot ! ;

: df, ( float memory-space -- )
  dup dfhere -> float [ 2 cells ] literal rot allot ! ;

: ", ( caddress -> character unsigned memory-space -- )
  dup >r chere -> character over chars r> allot swap move ;
```

Note that `, ,` stores two different items. This phrase cannot be used to store a double-cell item, because in StrongForth, a double-cell item is a single entity. That's why double-cell items need their own overloaded version of `, .`

And these are the words that apply to the default memory space. Two of them, the first and fourth, implement the Forth 2012 words `,` and `C,`:

```
: , ( single -- )
  default-memory-space @ , ; 3 retreat

: , ( double -- )
  default-memory-space @ , ; 3 retreat

: , ( float -- )
  default-memory-space @ , ; 3 retreat

: c, ( single -- )
  default-memory-space @ c, ; 1 retreat

: sf, ( float -- )
  default-memory-space @ sf, ; 1 retreat

: df, ( float -- )
  default-memory-space @ df, ; 1 retreat

: ", ( caddress -> character unsigned -- )
  default-memory-space @ ", ; 1 retreat
```

5 Input and Output

Basic I/O

What is basic I/O? On a typical system, console input and output are based on services provided by the operating system or by some library functions. These functions usually don't handle any formatting. They just read and write characters and strings from the console. Forth 2012 specifies four words for these purposes:

	Input	Output
Character	KEY	EMIT
String	ACCEPT	TYPE

These basic I/O words including the accompanying device-ready query words are also available in StrongForth:

```
key ( -- character )
key? ( -- flag )
accept ( caddress -> character integer -- 3rd )
emit ( integer -- )
emit? ( -- flag )
type ( caddress -> character unsigned -- )
```

Note that the second input parameter of `accept`, the maximum length of the character string to be received, is of data type `integer`, although you might have expected this to be `unsigned`. Forth 2012 specifies that this parameter is a signed positive integer, because this allows certain Forth implementations playing tricks when zero or a negative number is provided. Using `integer` instead of `unsigned` is just a matter of compatibility. Of course, it's still possible to provide an unsigned number as parameter for `accept`, because data type `unsigned` is a child of `integer`. In any case, the output parameter has the same data type as the actual input parameter.

`emit` is rarely used in StrongForth. This is because StrongForth provides an overloaded version of `.` for displaying items of data type `character`:

```
: . ( character -- )
  emit ;
```

With an input parameter of data type `integer`, `emit` accepts also signed and unsigned numbers as well as characters and all other direct or indirect subtypes of `integer`:

```
42 emit * ok
+65 emit A ok
char % emit % ok
```

In Forth 2012, `.` can only be used for pictured numeric output. Overloading makes it possible to provide individual versions of `.` for each data type. StrongForth has indeed a rather comprehensive set of output words named `.` for all kinds of data types. If `.` is applied to an item of data type `character`, it simply performs the semantics of `emit`. When applied to a number, another overloaded version of `.` performs pictured numeric output instead.

However, `.` cannot be applied to character strings, because character strings are constituted by two items, an address and a character count:

```
" Forth" .s type caddress -> character unsigned Forth ok
```

If `.` were used instead of `type`, it would just look at the item on top of the stack and display an unsigned number, leaving the character address on the stack. StrongForth's overloaded versions of `.` are not able to distinguish two items that constitute a character string from two separate single-cell items, one being a character address and the other being an unsigned number.

`key` and `accept` work exactly like `KEY` and `ACCEPT` as specified by Forth 2012, respectively. They receive a single character or a character string from the console and make it available to your program. `emit` and `type` are somewhat different. In Forth 2012, `EMIT` and `TYPE` are specified to always display a single character or a character string on the console. In StrongForth, these two words are more versatile. They actually send a character or a string to an object of a class named `output-stream`. This output stream can be the console, but it can also be a string, a file, or anything else that can take a stream of characters. Since the default output stream is the console, you will not notice the difference, unless you intentionally change the default output stream. You'll learn more about output streams in chapter 12.

Other Output Words

Based on the basic output words `emit` and `type`, some higher-level output words are defined:

```
: space ( -- )
  bl emit ;

: zero ( -- )
  [char] 0 emit ;

: cr ( -- )
  13 emit 10 emit ;
```

`bl` is a character constant:

```
32 cast character constant bl
```

The ASCII value of the space character, 32, needs to be casted to an item of data type `character`. If this type cast is omitted, `bl` would leave an unsigned number instead of a character on the stack. With `emit`, this makes no difference, but with the overloaded word `.`, it does:

```
32 constant no-bl   ok
no-bl emit         ok
no-bl . 32         ok
```

`spaces` expects an item of data type `integer` on the stack, because this data type covers both signed and unsigned numbers. The value of the parameter is interpreted as a signed number:

```
: spaces ( integer -- )
  cast signed
  begin dup 0>
  while space 1-
  repeat drop ;
```

Finally, here are the definitions of the two words `."` and `.(`:

```
: ." ( -- )
  [char] " parse type ; execute-only

: . ( -- )
  [char] " parse postpone sliteral postpone type ; compile-only
```

```

: .( ( -- )
  [char] ) parse type ; execute-only
: .( ( -- )
  [char] ) parse postpone sliteral postpone type ; compile-only

```

Defining state-smart immediate words is discouraged in StrongForth. The recommended method is taking advantage of overloading, and thus define different words for interpretation and compilation. Here, the versions to be interpreted are marked as `execute-only` and the versions to be compiled are marked as `compile-only`. `execute-only` and `compile-only` each set a specific flag in the attributes of the latest definition. In interpretation mode, the `compile-only` version is invisible, and in compilation mode, the `execute-only` version is invisible. In compilation mode, words marked `compile-only` are handled the same way as immediate words, i. e., they are being immediately executed.

Here's an overview over the effects of the three attributes:

Attribute	Interpretation	Compilation
(none)	execute	compile
execute-only	execute	(not found)
compile-only	(not found)	execute
immediate	execute	execute

Pictured Numeric Output

The Transient Area

For pictured numeric output, StrongForth provides a transient area. It starts at `line` and is `/hold` characters long:

```

address-unit-bits cells 4 * 2 + constant /hold
bl /hold cvariables line

```

A number to be printed is converted digit by digit into a character string, which is located in the transient area. Finally, this string is sent to an output stream. The transient area is used for other purposes as well, for example as an intermediate storage for error messages or as a buffer for converting character strings with embedded non-graphic characters. It's usage as a string buffer within applications is discouraged, because parts of the area will be overwritten during pictured numeric output conversion.

While `line` is used by StrongForth itself. Another transient area is reserved for usage by applications:

```

84 constant /pad
bl /pad cvariables pad

```

Let's get back to pictured numeric output.

```

/hold cvariable #hold

```

is a character-size variable that contains the index of a character within the transient area. At the beginning of a pictured numeric output conversion, this index is initialised to `/hold`, the size of the transient area. Subsequently, it is decremented character by character towards zero, until number conversion is done. Since the longest pictured numeric output is a 64 characters long binary number plus sign character, there's normally no risk that the index decrements below zero. Nevertheless, StrongForth's version of `hold` checks for an index underflow:

```

: hold ( character -- )
  #hold @
  if -1 #hold +! line #hold @ + !
  else drop -17 throw
  then ;

```

An interesting detail is the usage of `drop` to remove the `character` item at the beginning of the `else` branch. Isn't `throw` supposed to clear the stack contents anyway? Sure, but from the compiler's point of view, `throw` is just an ordinary word that consumes an item of data type `signed`. If `drop` were not present, the compiler would complain that the two branches of the conditional clause do not have the same stack effect, which prevents joining the two branches after `then`.

`hold` is overloaded by a version that adds a character string to the transient area. This is the equivalent of the Forth 2012 word `HOLDS`, changing the name. StrongForth is capable of distinguishing the two words.

```

: hold ( caddress -> character unsigned -- )
  #hold @ over >=
  if dup #hold -! line #hold @ + swap move
  else drop drop -17 throw
  then ;

```

The Number-Conversion Radix

Before we get engaged with pictured numeric output itself, let's have a quick glance at an important system variable. As specified in Forth 2012, the number-conversion radix is kept in the variable `BASE`:

```
10 cvariable base
```

Because the number-conversion radix is always less than 255, it can be stored in a character-size memory location.

StrongForth provides five words which directly set the number-conversion radix. Two of them, `DECIMAL` and `HEX`, are specified in Forth 2012. Their definitions bear nothing unexpected:

```

: binary ( -- ) 2 base ! ;
: octal ( -- ) 8 base ! ;
: decimal ( -- ) 10 base ! ;
: hex ( -- ) 16 base ! ;
: alpha-numeric ( -- ) 36 base ! ;

```

Painting a Picture

Pictured numeric output conversion always starts with `<#` and ends with `#>`. In StrongForth, `<#` expects any item of data type `double` on the stack, while `#>` leaves a character string, consisting of an address and a character count:

```

: <# ( double -- number-double )
  /hold #hold ! cast number-double ;

: #> ( number-double -- caddress -> character unsigned )
  drop line /hold #hold @ /string ;

```

During the conversion, a special subtype of `unsigned-double`, called `number-double`, remains on the stack. Items of data type `number-double` can only be produced by `<#` and can only be consumed by `#>`, which ensures `<#` and `#>` are always used in pairs. It is a common technique in StrongForth to introduce a special data type with the sole purpose of forcing the

programmer to stick to a given syntax. Any violation of the syntax rules requires using a type cast. Forth 2012, on the other hand, allows using any word in any place as long as the stack does not run empty. Thus, syntax violations are not detected during compilation. They usually lead to runtime errors or crashes.

The next step is to convert the double number of data type `number-double` into a sequence of digits. Like in Forth 2012, this is done by `#` and `#s`:

```
: # ( number-double -- 1st )
  base @ /mod over 10 <
  if [char] 0
  else [ char A 10 - ] literal
  then rot + hold ;

: #s ( number-double -- 1st )
  begin # dup 0= until ;
```

Remember that `number-double` is a subtype of `unsigned-double`. This means that all words expecting an item of data type `unsigned-double` also accept an item of data type `number-double`. Based on these words, pictured numeric output conversion is done by two overloaded words, one for unsigned and one for signed double-cell numbers, which both return character strings:

```
: picture ( double -- caddress -> character unsigned )
  <# #s #> ;

: picture ( signed-double -- caddress -> character unsigned )
  dup abs <# #s swap sign #> ;
```

`sign` adds a minus sign to the transient area if its input parameter is negative. Note that the input parameter, other than Forth 2012 specifies for the word `SIGN`, is a double-cell number:

```
: sign ( signed-double -- )
  0< if [char] - hold then ;
```

Forth 2012 specifies the three words `.`, `U.` and `D.` for displaying signed single-precision numbers, unsigned single-precision numbers and signed double-precision numbers in free field format, respectively. StrongForth overloads `.` to provide all of the above words plus one additional word for unsigned double-precision numbers:

```
: . ( double -- )
  picture type space ;

: . ( signed-double -- )
  picture type space ;

: . ( single -- )
  s>d . ;

: . ( signed -- )
  s>d . ;
```

Because the versions for unsigned numbers are defined first, they will be found in the dictionary only if the specialized versions of `.` for subtypes of `single` and `double`, including the ones for characters and signed numbers, do not match. Note that the conversion from a single number to a double number cannot be accomplished by just padding a single-cell zero, because this would result in two single numbers instead of one double number on the stack. Different overloaded versions of `s>d` do the job nicely in both cases.

Another overloaded version of `.` expects items of data type `flag`:


```
: . ( flag -- )
  if " true " else " false " then type ;
```

Here are some examples that demonstrate the various overloaded versions of .:

```
716 30 + . 746 ok
char E 3 - . B ok
-4000000000000. +1000000000000. - . -5000000000000 ok
6 8 > . false ok
base .s . caddress -> unsigned 4300813
```

Because there is no specialised version of . for addresses, applying . to addresses like `base` defaults to converting an unsigned single-precision number. Of course, you can easily define one that fits your specific needs.

Now, let's have a quick glance at the four overloaded versions of `.r`. These definitions look similar to the definitions of . for single and double numbers. They implement the Forth 2012 words `.R`, `U.R` and `D.R` plus an additional word that would be named something like `UD.R` in Forth 2012:

```
: .r ( double integer -- )
  swap picture rot over - spaces type ;

: .r ( signed-double integer -- )
  swap picture rot over - spaces type ;

: .r ( single integer -- )
  swap s>d swap .r ;

: .r ( signed integer -- )
  swap s>d swap .r ;
```

Note that the second parameter of all four overloaded versions of `.r` has data type `integer`. You can provide a signed or an unsigned number as the field width:

```
536009 713828 m* 15 .r \ instead of ... +15 .r      382618232452 ok
```

StrongForth additionally provides a set of four words similar to `.r` that display leading zeros instead of leading spaces, if required. A word analogous to `spaces` displays a given number of leading zeros:

```
: zeros ( integer -- )
  cast signed
  begin dup 0>
  while zero 1-
  repeat drop ;

: 0.r ( double integer -- )
  swap picture rot over - zeros type ;

: 0.r ( signed-double integer -- )
  over 0<
  if 1 cast integer max 1- swap abs swap [char] - .
  then 0.r ;

: 0.r ( single integer -- )
  swap s>d swap 0.r ;

: 0.r ( signed integer -- )
  swap s>d swap 0.r ;
```

The Console Window

Forth 2012 specifies two words that manipulate the console window. These are available in StrongForth as well. `at-xy` uses an operating system function to position the text cursor relative to the upper left corner of the window:

```
at-xy ( unsigned unsigned -- )
```

A function equivalent to `PAGE` is not provided by the operating system. Assuming that the console has 25 lines of 80 columns each, StrongForth's version of `page` overwrites the console window with spaces and positions the cursor at the upper left corner:

```
: home ( -- )  
  0 0 at-xy ;  
  
: page ( -- )  
  home 25 0 do 80 spaces cr loop home ;
```

6 Character Strings

What's a String?

A string is a sequence of characters with a well-defined length. The characters are stored in consecutive character-size memory locations. According to the Forth 2012 specification, a character string *is specified by a cell pair (c-addr u) representing its starting address and length in characters*. In StrongForth, a string is usually represented by two items with data types `caddress -> character unsigned`. Among the predefined storage locations for strings is the `scratchpad` and the `transient area` line.

Forth 2012 actually specifies a second kind of string storage. A so-called *counted string* is a sequence of characters in memory, which is preceded by a *length character*. The length character determines the length of the string as an unsigned number. Here's an example of the memory image of a counted string:

11	S	t	r	o	n	g	F	o	r	t	h
----	---	---	---	---	---	---	---	---	---	---	---

A counted string in memory is identified by the address of its length character. Forth 2012 even specifies the word `COUNT`, that converts a counted string into the `caddress -> character unsigned` representation. This word is available in StrongForth as well:

```
: count ( caddress -> character -- 1st unsigned )  
  dup cast caddress -> unsigned @ swap 1+ swap ;
```

Anyway, since Forth 2012 explicitly discourages using counted strings, they are abandoned in StrongForth. The Forth 2012 words `C"`, `FIND` and `WORD` do not exist in StrongForth. Since `C"` is obsolete, `S"` was renamed to `"` in StrongForth. `FIND` has been replaced by `search-context`, which (among other differences) expects a string in the `caddress -> character unsigned` representation (see chapter 8). Instead of `WORD`, `parse-name` can be used. Getting rid of these words is not considered being a deficiency, because strings in the `caddress -> character unsigned` representation can easily replace counted strings, and the advantage of having to deal with only one kind of representation is pretty obvious.

String Processing

A group of string processing words were already presented in of chapter 2:

```
fill ( caddress -> single unsigned 2nd -- )  
erase ( caddress -> single unsigned -- )  
move ( caddress 1st unsigned -- )  
cmove ( caddress 1st unsigned -- )  
cmove> ( caddress 1st unsigned -- )
```

Since character strings are stored in memory blocks, these words can be applied to character strings as well. `fill` initializes a string with a sequence of any character:

```
pad 5 char A fill   ok  
pad 5 type AAAAAA ok
```

`erase` is a specialized version of `fill` that initializes a memory block with zero. For strings, it is far more common to get initialized with space characters. This is what `blank` does:

```
: blank ( caddress -> character unsigned -- )
  bl fill ;
```

Note that `blank` can only be applied to strings, but not to memory blocks in general.

The overloaded version of `move` for character-size items is not identical with the Forth 2012 word `MOVE`, because it works with character-size memory locations instead of address units. `cmove` and `cmove>`, on the other hand, work explicitly with characters. This means, these two words implement exactly the Forth 2012 words `CMOVE` and `CMOVE>`, respectively.

Now, let's continue with some more string processing words. `/string` adjusts a string by the number of characters given as the last input parameter. This input parameter is of data type integer in order to allow signed positive, signed negative and unsigned numbers:

```
: /string ( caddress -> character unsigned integer -- 1st 3rd )
  rot over chars + rot rot - ;
```

Forth 2012 only specifies this version as `/STRING`. StrongForth provides an additional, overloaded version of `/string` without the last parameter:

```
: /string ( caddress -> character unsigned -- 1st 3rd )
  1- swap 1+ swap ;
```

The second version of `/string` adjusts a string by always removing the first character, i. e., it assumes a default adjustment value of 1. Similar to `lshift` and `rshift` (see chapter 3), StrongForth takes advantage of its overloading capability by providing a special version for the most common usage of a word. Here's a simple example:

```
pad 16 blank      ok
pad 16 type                ok
parse-name StrongForth pad swap move  ok
pad 16 type StrongForth    ok
pad 16 5 /string over over type gForth      ok
/string over over type Forth      ok
-trailing type Forth ok
```

This example leads to the next string processing word, which is overloaded as well:

```
: -trailing ( caddress -> character unsigned 2nd -- 1st 3rd )
  locals( c )
  begin dup
  while over over + 1- @ c =
    while 1-
      repeat
    then ;
: -trailing ( caddress -> character unsigned -- 1st 3rd )
  bl -trailing ;
```

The semantics of the second version is as specified by Forth 2012. It removes trailing spaces from a string, whereas the first version removes any trailing characters that are equal to 2nd. The implementation of the first version contains a loop with two exit conditions, one for encountering a non-matching character and one for the string being empty.

Note the usage of `locals(...)` instead of `locals| ... |`. The semantics are the same but for the fact that `locals(` expects the locals in the same order as in a stack diagram, i. e., the rightmost item is the one on top of the stack. `locals|`, as specified by Forth 2012, expects the locals in reverse order, which might be confusing. Although StrongForth provides both variants, using `locals|` is discouraged. The implementation of `locals(`, which will be presented in chapter 15, is based on a recursive algorithm.

StrongForth provides an additional set of overloaded words for removing characters from the beginning of a string. These words are not included in the Forth 2012 specification. Again, there is a version for spaces and a version for arbitrary characters:

```
: -leading ( caddress -> character unsigned 2nd -- 1st 3rd )
  locals( c )
  begin dup
  while over @ c =
    while /string
      repeat
    then ;
: -leading ( caddress -> character unsigned -- 1st 3rd )
  bl -leading ;
```

The Forth 2012 words COMPARE and SEARCH have equivalent versions in StrongForth:

```
compare ( caddress -> character unsigned 1st 3rd -- signed )
search ( caddress -> character unsigned 1st 3rd -- 1st 3rd flag )
```

These words apply exclusively to character strings and not to arrays of character-size items, as fill and move do (see chapter 2).

String Literals

Forth 2012 specifies S" as a means to compile character string literals. If the *File-Access* word set is present, this word assumes additional interpretation semantics, so that string literals can also be interpreted.

Because StrongForth does not support counted strings, it does not need to distinguish between a version for counted strings and a version for character strings in caddress -> character unsigned representation. However, there are still two overloaded versions of ", one for the interpretation semantics and one for the compilation semantics:

```
: " ( -- caddress -> character unsigned )
  [char] " parse ;
: " ( -- )
  " postpone sliteral ; compile-only
```

These definitions are fairly simple. The version that implements the interpretation semantics returns an address within the input buffer. Since the input buffer will typically be overwritten by the next line of input, the character string returned by " is valid only within the same input line. If you want it to remain, you have to copy the character string to a more permanent storage:

```
" abcdefghij" .s caddress -> character unsigned ok
\ do something \ type do somethi ok
" abcdefghij" pad swap cmove ok
pad 10 type abcdefghij ok
```

As expected, the implementation of \", the StrongForth version of the Forth 2012 word S\", is considerably more complex. To make \" available, you have to include the source file escape.sf.

```

: ?parse-char ( -- character )
  source >in @ >
  if >in @ + @ 1 >in +!
  else -295 throw drop null character
  then ;

: ?parse-byte ( -- character )
  base @ hex ?parse-char digit?
  if 16 * ?parse-char digit?
    if +
    else + -24 throw
    then
  else -24 throw
  then cast character swap base ! ;

: escaped-char ( character -- 1st )
  case [char] a of 7 cast character endof
    [char] b of 8 cast character endof
    [char] e of 27 cast character endof
    [char] f of 12 cast character endof
    [char] l of 10 cast character endof
    [char] m of 13 cast character hold> 10 cast character endof
    [char] n of 13 cast character hold> 10 cast character endof
    [char] q of [char] " endof
    [char] r of 13 cast character endof
    [char] t of 9 cast character endof
    [char] v of 11 cast character endof
    [char] z of null character endof
    [char] x of ?parse-byte endof
  endcase ;

: \" ( -- caddress -> character unsigned )
  0 #hold !
  begin ?parse-char dup [char] " <>
  while dup [char] \ =
    if drop ?parse-char escaped-char
    then hold>
  repeat drop line #hold @ ;

: \" ( -- )
  \" postpone sliteral ; compile-only

```

This code contains two words that are not included in the Forth 2012 specification.

`digit? (character -- unsigned flag)`

expects an item of data type character on the stack. If this character is a valid digit according to the current number-conversion radix, `digit?` returns the numeric value of the digit as an unsigned number together with a true flag. Otherwise, the numeric value is invalid and the flag is false.

`hold>` is a variant of `hold` that constructs a character string in the transient area from the bottom up to the top, instead of the opposite direction, as `hold` does. Here's the definition:

```

: hold> ( character -- )
  #hold @ /hold <
  if line #hold @ + ! 1 #hold +!
  else drop -18 throw
  then ;

```

As with `hold`, there's an overloaded version that adds a whole character string to the transient area instead of only one character. However, this version is not required by the implementation of `\`:

```
: hold> ( caddress -> character unsigned -- )
  #hold @ over + /hold <=
  if tuck line #hold @ + swap move #hold +!
  else drop drop -18 throw
  then ;
```

If you studied the implementation of `\` carefully, you might have noticed a kind of *bug* within `escaped-char`. If this was Forth 2012 source code, this bug would show if the character supplied to `escaped-char` was not one of the characters selected by the `case ... endcase` clause. In this case, `endcase` would drop the character from the stack.

StrongForth would complain about the data type incongruence and refuse to compile `escaped-char`. But it doesn't. The reason is that StrongForth has changed the semantics of `endcase` in such a way, that it does *not* drop the selector. This behavior has turned out to be more useful, because in most cases, the code between the last `endof` and `endcase` does something with the left-over selector, so it doesn't need to be dropped by `endcase`. Dropping the selector has to be done manually in StrongForth.

7 Data Types

What's in a Data Type?

In order to support StrongForth's data type system, the interpreter and the compiler have to store information about the data types of the items on the stack. The stack diagrams of all words have to be included in the dictionary. Interpreter and compiler use these data to find out which version of an overloaded word is to be interpreted or compiled next.

StrongForth's data type system is build up on a data type called `data-type`, which is a direct subtype of `double`. For example, the data structure that describes the data types of the items that are currently on the stack, is a list of items of this data type. Stack diagrams also consist of lists of items of data type `data-type`. So, the first question to answer is: What's in a data type?

Items of data type `data-type` are composed of two cells. The lower cell contains an identifier or an offset, while the upper cell is a collection of attributes. Most attributes are used internally by the compiler. But two of them you should know: The *reference attribute* and the *prefix attribute*.

If the reference attribute is clear, the identifier of the data type is an object of the class `data-type-attributes`, which is located in the body of created words that compose a stack diagram, like `unsigned` or `caddress`. The members of an object of data type `data-type-attributes` specify the size and the parent of its associated data type, its virtual method table (if the data type is a class) as well as some internal information.

If the reference attribute is set, the data type references another data type within the same stack diagram. Data type references are created by the words `1st`, `2nd`, `3rd` and `th`. In these cases, the lower cell contains the offset of the referenced data type, starting at 1 (for `1st`), instead of an identifier.

As an example, let's view the stack diagram that is stored in the definition of a word:

```
@ ( address -> single - 2nd )
```

Its stack diagram is a list of three data types:

upper cell (attributes)	lower cell (identifier or offset)
prefix attribute	address
(none)	single
reference attribute	2

Setting the prefix attribute of the first basic data type connects it with the second basic data type to compose a compound data type. The reference attribute of the third basic data type indicates that it is a reference to the second basic data type. Whether a parameter is an input or an output parameter, is not an attribute of data types. The numbers of input and output parameters are kept as internal information of the stack diagram as a whole.

Obviously, the usual arithmetic and logic operations cannot be applied to items of data type `data-type`. So how can you set and query the identifier and the attributes of a data type? First of all, you need a way to push an item of data type `data-type` with a specific identifier onto the stack. For this purpose, StrongForth provides the words `dt` and `[dt]`, which are applied similarly like the Forth 2012 words `'` and `[']`. Here's how to use `dt` in interpretation mode:

```
dt logical .s . data-type logical ok
```


dt actually parses the input stream for the name of data type and then pushes an item of data type `data-type` onto the stack with the appropriate identifier and no attributes. Similarly, `[dt]` can be used during compilation to compile a literal data type.

Now what about the attributes? StrongForth provides two constants of data type `data-type` that are bit masks for the attributes:

```
dt-prefix ( -- data-type )
dt-reference ( -- data-type )
```

`dt-prefix` is a data type with no identifier and only the prefix attribute. `dt-reference` has only the reference attribute. To be able to set and clear the attributes of a data type, overloaded versions of the logical operators `and`, `or`, `xor` and `invert` are available in StrongForth:

```
and ( data-type data-type -- 1st )
or ( data-type data-type -- 1st )
xor ( data-type data-type -- 1st )
invert ( data-type -- 1st )
```

As can be seen from the stack diagrams, these words can only be applied to items of data type `data-type` and its subtypes. Following the general rule for arithmetical and logical operations, the data type of the output parameter is identical to the data type of the first input parameter. For example,

```
dt-prefix dt-reference or
```

is a data type with both the prefix attribute and the reference attribute. To clear an attribute, you can apply a logical `and` of the inverted bit mask:

```
dt-reference invert and
```

The logical operations on data types only apply to the attributes, not to the identifier. This means, the identifier of the resulting data type is always identical to the identifier of the first parameter of a logical operation. For example, the result of

```
dt caddress
dt flag dt-prefix or
or
```

is a data type with the identifier `caddress` combined with the prefix attribute. Creating a reference data type can be accomplished like this:

```
2 cast data-type dt-reference or
```

In addition to the standard logical operators, StrongForth provides four words that query the attributes of a data type:

```
: attributes? ( data-type data-type -- flag )
  and high 0<> ;

: prefix? ( data-type -- flag )
  dt-prefix attributes? ;

: reference? ( data-type -- flag )
  dt-reference attributes? ;

: offset ( data-type -- unsigned )
  dup reference? if cast unsigned else drop 0 then ;
```

`attribute?` returns `true` if and only if the first data type has at least one of the attributes of the second data type. This word is used by `prefix?` and `reference?` to test a data type for the presence of the prefix or reference attribute, respectively. `offset` returns the offset of the referenced data type if the input parameter has the reference attribute. Otherwise, it returns zero.

A good demonstration of how to use these words are the definitions of the overloaded version of `.` for items of data type `data-type`, and the word `.params`:

```
: .name ( definition -- )
  name dup 0<> if type space else drop drop then ;

: . ( data-type -- )
  dup offset
  case 0 of >definition cast definition endof
    1 of drop ['] 1st endof
    2 of drop ['] 2nd endof
    3 of drop ['] 3rd endof
    \ default \ . drop ['] th
  endcase .name ;

: .prefix ( data-type -- )
  prefix? if ['] -> .name then ;

: .params ( address -> data-type unsigned -- )
  0 ?do dup @ dup . .prefix 1+ loop drop ;
```

`.name` displays a definition's name and a space character, if the definition has a name. `name` is a method of the `definition` class that returns the name of a definition as a character string. If the definition has no name, the length of the string is zero.

`.` for `data-type` actually uses `.name`. For data type references, it displays the offset of the referenced data type instead. Remember that in StrongForth, `endcase` does not drop the case selector. `>definition` returns the definition that adds a given data type to a stack diagram. The name of this definition the name of the data type.

`.prefix` also uses `.name`. It displays "`->` " if the data type prefixes another data type.

`.params` displays a list of data types including all prefix and reference attributes. The data type list starts at `address -> data-type` and consists of unsigned basic data types.

Data Type Heaps

StrongForth keeps the data type information on two separate data structures, which are called data type heaps. In contrast to a stack, which typically grows from high to low addresses, a heap grows from low to high addresses. Using a heap instead of a stack brings some advantages for the implementation, because the order of the basic data types on a heap matches the order of the basic data types within a stack diagram.

But why does StrongForth need two data type heaps? Is there one for the data stack and one for the return stack? Good guess, but this is wrong. There's no data type heap for the return stack. Instead, StrongForth has two data type heaps for the items on the data stack, which are called *interpreter data type heap* and *compiler data type heap*.

The interpreter data type heap contains the data types of the items that are currently on the data stack. For each single or double cell item on the data stack, and for each floating-point number on the hardware floating-point stack, the interpreter data type heap contains one compound data type. Although this is an exact one-to-one relation, the memory requirements might look different. An

item on the data stack takes one or two cells, while a compound data type on the data type heap consists of one or more basic data types, that each have the size of an item of data type `data-type`. For example, an item of data type `signed-double` obtains one double cell on both the interpreter data type heap and the data stack, whereas an item of data type `caddress -> character` requires two double cells on the interpreter data type heap and only one cell on the data stack.

The compiler data type heap is used during compilation to specify the data types of the items which will be on the stack at runtime. Let's see what happens on both the interpreter and the compiler data type heap during compilation of `.prefix`:

```
: .prefix ( data-type -- )
  prefix? if ['] -> .name then ;
```

: parses the definition's name `.prefix` and creates an object of data type `colon-definition` with this name and an empty stack diagram. It then enters compilation state. Data type `colon-definition` is added to the interpreter data type heap. (is an immediate word, which creates and initializes a new empty stack-diagram and switches back to interpretation state, because the stack diagram is constructed at compile-time. We now have `colon-definition` and `stack-diagram` on the interpreter data type heap. In this example, the stack diagram is rather simple. `data-type` adds an input parameter, `--` switches to the output parameter field, which is empty, and) finally consumes `stack-diagram`. Consuming the stack diagram means that its contents is made the stack diagram of the colon definition.) copies the input parameter list of the stack diagram to the compiler data type heap, because compilation is expected to start with these parameters. Finally,) turns back to compilation state. Now, there's again only `colon-definition` on the interpreter data type heap, while the compiler data type heap contains `data-type`.

When compiling `prefix?`, the compiler replaces `data-type` on the compiler data type heap with `flag`, because that's the stack effect of `prefix?`. The data stack and the interpreter data type heap do not change.

Now it's getting interesting, because `if` is again an immediate word. Being executed immediately, it can affect the compiler data type heap *and* the interpreter data type heap. `if` compiles (0branch), which consumes `flag` from the compiler data type heap. To mark the beginning of a conditional clause, `if` it pushes an object of data type `origin` onto the data stack, so that data type `origin` must be added to the interpreter data type heap.

['] is again an immediate word. Because its stack diagram is empty, the interpreter data type heap remains unchanged. ['] parses `->`, finds it in the dictionary and compiles a literal of this definition, adding `definition` to the compiler data type heap. `.name` consumes `definition`, so the compiler data type heap is now empty.

The immediate word `then` consumes `origin` and checks whether the contents of the compiler data type is now exactly the same as it was after compilation of (0branch). Otherwise, the jump would lead to an unbalanced data type heap.

Now we have again only `colon-definition` on the interpreter data type heap. This is consumed by the immediate word `;`, which ends the definition and makes it available in the dictionary. `;` further checks whether the compiler data type heap at the end of the definition matches its output parameter list. In the case of `.prefix`, the output parameter list is empty, just like the compiler data type heap. Finally, `;` returns to interpretation state.

The immediate word `.s` may be used to visualise the contents of the data type heaps at any time. During interpretation, it prints the contents of the interpreter data type heap:

```
17 -5 .s unsigned signed ok
```

During compilation, `.s` prints the contents of the compiler data type heap. If you want to see what's on the interpreter data type heap during compilation, you have to temporarily switch to interpretation state and then execute `.s`. Here's again the definition of `.prefix`:

```
: .prefix .s
  ( data-type -- .s ) colon-definition stack-diagram
  prefix? .s flag
  if [ .s ] colon-definition origin
  [' ] -> .s .name definition
  then [ .s ] ; colon-definition ok
```

Operations on Lists of Data Types

Being a strongly typed language, the implementation of StrongForth is based to a large degree on operations on lists of data types, especially stack diagrams and the two data type heaps. Most of the operations that implicitly deal with a data type heap refer to the one that is selected by `state`. I.e., if `state` is `false`, they deal with the interpreter data type heap, and if `state` is `true`, they deal with the compiler data type heap.

Forth 2012 specifies the system variable `STATE` to be zero (`false`) in interpretation state, and non-zero (`true`) in compilation state. In StrongForth, `state` is a character-size variable of data type `flag`:

```
false cvariable state
```

Messing around with `STATE` is discouraged by the Forth 2012 specification. Only few words are supposed to change its value. Among these words are `[` and `]`, which are typically used during compilation to temporarily turn to interpretation state. Their definition in StrongForth is straightforward:

```
: [ ( -- )
  false state ! ; immediate

: ] ( -- )
  true state ! ;
```

Let's get back to the data type heaps. The most basic operations on the data type heaps are these:

```
dt-init ( -- )
dt-bottom ( -- address -> data-type )
dt-here ( -- address -> data-type )
dt-lock ( -- )
dt-allot ( integer -- )
```

`dt-init` initializes the selected data type heap, which means that it contains zero data types. This word is executed at the beginning of a new word's definition, before a stack diagram is specified.

`dt-bottom` returns the address where the first basic data type of the data type heap is stored. `dt-here` returns the address of the first unused position.

`dt-lock` can only be applied to the compiler data type heap. After `dt-lock` has been executed, `dt-here` returns zero to indicate that the compiler data type heap is invalid and can no longer be used. This situation happens after executing `exit`, `ahead`, `again` or `endof`, because these words break the control flow during compilation. Keeping data type information after compiling one of these words makes no sense, because the data flow is broken as well. If any word is compiled after `exit`, `ahead`, `again` or `endof`, it has to start with new data type information, which may come from a different control flow that happens to touch down here. `dt-lock` ensures

that the data flow can not be continued after the control flow has been broken. Only `dt-init` can unlock the compiler data type heap.

`dt-allot` increments or decrements `dt-here` by the number of basic data types given by its input parameter, which is interpreted as a signed number. If the resulting `dt-here` is an address outside of the memory area reserved for the data type heap, `dt-allot` throws an exception. An overloaded version of `dt-allot` allocates space on the data type heap for exactly one basic data type:

```
: dt-allot ( -- )
  1 dt-allot ; 1 retreat
```

Using these words, we can define some more operations on the selected data type heap that turn out to be useful. For example, the number of basic data types on the data type heap can easily be calculated:

```
: dt-depth ( -- unsigned )
  dt-here dt-bottom - cast unsigned ;
```

Note that `-` in this definition performs address arithmetic, which means that the result of data type signed is not the address difference, but the number of basic data types stored on the data type heap. Note also that `dt-depth` is not identical with the Forth 2012 word `DEPTH`. An equivalent word to `DEPTH` makes no sense in StrongForth, because StrongForth does not have a physical data stack. The two words `sp@` and `sp!` apply to the *return* stack and should be used with care:

```
sp@ ( -- address )
sp! ( address -- )
```

With `dt-bottom` and `dt-depth`, it is now pretty easy to define the word `.s`, which has been used in many of the examples so far:

```
: .s ( -- )
  dt-bottom dt-depth .params ; immediate
```

Note that `.s` is an immediate word. It immediately displays the interpreter data type heap in interpretation mode, and the compiler data type heap in compilation mode.

`>dt` is an overloaded word, which adds either a basic data type to the selected data type heap, or the compound data type starting at address `->` data-type.

```
: >dt ( data-type -- )
  dt-here dt-allot ! ;

: >dt ( address -> data-type -- )
  begin dup @ dup >dt prefix?
  while 1+
  repeat drop ;
```

With all these definitions, it is now possible to define one of the two overloaded versions of `->`. This is an immediate word that links a basic data type to a basic data type, making it the second one a compound data type, or that links another basic data type to an existing compound data type. It is the version used in phrases like

```
here -> unsigned
and
cast address -> logical
```

Because it's an immediate word, it works in interpretation mode and in compilation mode. Here's the definition:

```

: -> ( -- )
  -1 dt-allot dt-here @ dt-prefix or dt-here ! dt-allot
  dt >dt ; immediate

```

The other version of `->` is the one to be used inside stack diagrams. Its definition will be presented at the end of chapter 9.

Now, here are a few definitions that are applied to lists of data types in general instead of implicitly to one of the two data type heaps:

```

: dt-next ( address -> data-type -- 1st )
  begin dup 1+ swap @ prefix? invert
  until ;

: dt-length ( address -> data-type -- unsigned )
  dup dt-next swap - cast unsigned ;

: dt-stripped ( address -> data-type unsigned -- 1st 3rd )
  dup
  if begin 1- dup
    while over over 1- + @ prefix? invert
    until
  then
  then ;

```

`dt-next` advances `address -> data-type` to the end of the compound data type it points to. If `address -> data-type` points to a basic data type, it is only incremented by the size of one data-type.

`dt-length` returns the size in basic data types of the compound data type `address -> data-type` points to, or one if `address -> data-type` points to a basic data type.

`dt-stripped` expects a data type list on the stack, represented by the address of its first basic data type and its length given by the total number of basic data types. From this list, it removes the last compound data type, provided `unsigned` is greater than zero. `1st` is equal to `address -> data type`, and `3rd` is zero or a value less than `unsigned`.

Here is an example of how all these words can be used:

```

here -> data-type constant dt-list ok
dt address dt-prefix or , dt character , dt unsigned , ok
dt address dt-prefix or dup , , dt logical , ok
here -> data-type dt-list - cast unsigned constant dt-len ok
dt-len . 6 ok
dt-list dt-len cr .params
caddress -> character unsigned address -> address -> logical ok
dt-list dt-next dt-list - . 2 ok
dt-list dt-length . 2 ok
dt-list dt-next dt-length . 1 ok
dt-list dt-next dt-next dt-length . 3 ok
dt-list dt-len dt-stripped cr .params
caddress -> character unsigned ok

```

`dt-compare` is a word that compares two data type lists. It returns true if the two lists are identical with respect to identifier/offset and the prefix and reference attributes. Other data type attributes are not considered. `flag` is true if the two data type lists are identical in this sense, otherwise `flag` is false:

```

: dt-compare ( address -> data-type unsigned 1st 3rd -- flag )
  rot over =
  if 0
    ?do over i + @ [ dt-prefix dt-reference or ] literal and
      over i + @ [ dt-prefix dt-reference or ] literal and
      <> if drop drop false exit then
    loop true
  else drop false
  then nip nip ;

```

The next two words deal again with the two data type heaps:

```

: dt-drop ( -- )
  dt-depth dup
  if dt-bottom over dt-stripped rot - dt-allot drop
  else drop -257 throw
  then ;

: dt-restore ( -- )
  dt-here dt-length dt-allot ;

```

`dt-drop` removes the last compound data type from the selected data type heap. It throws an exception if the data type heap is empty. `dt-restore` can be used to restore a compound data type that has been removed with `dt-drop`, provided it has not been overwritten in the meantime.

Creating new Data Types

Data types are added to a stack diagram by simply executing a word with the name of the data type. Whenever words like `unsigned`, `character`, or even `data-type` are executed, the corresponding data type is compiled as an input or output parameter into a stack diagram.

StrongForth provides a large collection of predefined data types, which are arranged in a hierarchical structure. The generic data types `single`, `double` and `float` are at the top level. They are called ancestors. The only property these data types have is their size. All other data types are derived from the ancestor data types and thus inherit their size.

Since the semantics of all words representing data types within a stack diagram are identical, it's obvious to define new data types by using a defining word. This defining word is `procreates`. It creates a new child of an existing data type, which is expected as an input parameter to `procreates`. For example, a new data type `temperature` could be defined as a child of `signed` to ensure that all operations on `signed single-cell` numbers can be applied to it:

```
dt signed procreates temperature
```

Does this mean that you always have to create a new data type as a subtype of an already existing data type? No, it doesn't. You can create a data type that has no parent with an overloaded version of `procreates` that additionally expects the size of the new data type in address units as an unsigned number. The size parameter is required because the size cannot be inherited from the parent. The parent data type has to be `null`:

```
null data-type 1 cells procreates adam
null data-type 3 cells procreates eve
```

Bear in mind that new ancestor data types will not match any input parameter of the words in StrongForth's dictionary. Even words like `dup` and `drop` can not be applied, because they are only defined for items of data types `single`, `double` and `float`, and their respective subtypes:

```

25 cast adam .s adam ok
drop
drop ? undefined word
adam

```

Nevertheless, a new data type with no parent might be useful if its usage shall be restricted. You need to manually define or redefine all words that can be applied to items of these data types:

```

: drop ( adam -- )
  cast single drop ;

```

Because the parent of an ancestor data type is null, we can easily walk up the line of parent, grand parent and so on to obtain the ancestor of a given data type:

```

: ancestor ( data-type -- 1st )
  begin dup parent d>s
  while parent
  repeat ;

```

Note that `ancestor` is not a predefined word in StrongForth.

Stack Diagrams

A stack diagram is a sequence of basic and compound data types or data type references enclosed in parenthesis, with a double-dash inbetween to separate input and output parameters:

```

stack diagram      := ( <parameter-list> -- <parameter-list> )
<parameter-list>   := <parameter>*
<parameter>        := <reference> | <compound-data-type>
<compound-data-type> := <basic-data-type> [ -> <parameter> ]
<reference>         := { 1st | 2nd | 3rd | <offset> th }

```

Other than in Forth 2012, `(` is not starting a comment. It doesn't even parse upto `)`, building a stack diagram from the parsed names. The stack diagram rather builds itself while interpreting its components.

It is good Forth programming style to specify the stack diagram of each word in the source code. Most Forth programmers go conform with this rule, so changing the semantics of `(` with respect to Forth 2012 shouldn't be a big deal. But it means that parenthesis are no longer available for usage as comments. A comment in StrongForth is always delimited by a backslashes (`\`).

`(` creates an object of data type `stack-diagram`, which is consumed by `)`. Basic data types are actually StrongForth words, which all expect an object of data type `stack-diagram` on the stack, modify it, and return it as an output parameter.

```
single ( stack-diagram -- 1st )
```

is an example. The other components of a stack diagram have execution semantics as well:

```

-- ( stack-diagram -- 1st )
-> ( stack-diagram -- 1st )
th ( stack-diagram unsigned -- 1st )
1st ( stack-diagram -- 1st )
2nd ( stack-diagram -- 1st )
3rd ( stack-diagram -- 1st )

```

Each of these words performs a syntax check, as can be seen in the following examples:


```

: bad1 ( unsigned -- 1st -- flag )
: bad1 ( unsigned -- 1st -- ? invalid stack diagram
colon-definition stack-diagram

: bad2 ( address -> -- character )
: bad2 ( address -> -- ? invalid stack diagram
colon-definition stack-diagram

: bad3 ( caddress -- 1st -> character )
: bad3 ( caddress -- 1st -> ? invalid stack diagram
colon-definition stack-diagram

: bad4 ( logical -- 1st 2nd )
: bad4 ( logical -- 1st 2nd ? invalid reference
colon-definition stack-diagram

: bad5 ( integer 0 th -- 1st )
: bad5 ( integer 0 th ? invalid reference
colon-definition stack-diagram

: bad6 ( token -- -> 1st )
: bad6 ( token -- -> ? invalid stack diagram
colon-definition stack-diagram

: bad7 ( address -> -> signed -- )
: bad7 ( address -> -> ? invalid stack diagram
colon-definition stack-diagram

```

th always creates a reference to data types in the input parameter list of the same stack diagram. The parameter of data type unsigned selects which basic data type in the input parameter list it refers to, starting with one. 1st, 2nd and 3rd can easily be derived from th:

```

: 1st ( stack-diagram -- 1st )
  1 th ;

: 2nd ( stack-diagram -- 1st )
  2 th ;

: 3rd ( stack-diagram -- 1st )
  3 th ;

```

Since data type references belong to StrongForth's most basic concepts, let's get into some more detail. Assume we have

```
logical address -> logical
```

on the data type heap and ! is the name of the word to be interpreted or compiled.

```
! ( single address -> 1st -- )
```

is the matching version of !. logical is a subtype of single. 1st refers to the first data type in the input parameter list, which is single in the stack diagram, but logical on the actual data type heap. References always have to be exact matches. Something like

```
single address -> logical
```

or

```
logical address -> single
```

on the data type heap will not match any overloaded version of !. In this case, this is easy to understand. It usually doesn't make sense to store, for example, a signed or an unsigned number in a memory location whose content is supposed to be a logical value:

```

15 bit variable bitfield ok
4096 bitfield !
4096 bitfield ! ? undefined word
unsigned address -> logical

```

Because logical and unsigned are not identical, the interpreter does not find any version of ! whose stack diagram matches the contents of the interpreter data type heap. In this particular case, it prevents storing an unsigned number into a logical variable.

This mechanism also works if the referenced parameters are compound data types or tails of compound data types. In the following stack diagram, 2nd refers to address -> integer:

```
dummy ( address -> address -> integer 2nd -- unsigned )
```

Whether dummy is found in the dictionary depends on the data types of the two items on top the stack. Here are a few examples:

next of stack	top of stack	match
address -> address -> unsigned	address -> unsigned	yes
address -> address -> unsigned	address -> character	no
address -> address -> unsigned	address -> logical	no
address -> address -> unsigned	caddress -> unsigned	no
address -> caddress -> unsigned	caddress -> unsigned	yes
address -> caddress	caddress	no
address -> caddress -> signed	caddress -> signed	yes
address -> caddress -> signed	caddress	no
address -> address -> integer	address -> integer	yes

1st, 2nd, 3rd and th can also be used as output parameters, but the reference must always be to an input parameter. Whenever a word containing a reference as an output parameter is executed or compiled, the output parameter assumes the data type of the actually referenced input parameter. The referenced input parameter might be a compound data type or the tail of a compound data type. A typical application is

```
@ ( caddress -> single -- 2nd )
```

as used in the following example:

```

base .s caddress -> unsigned ok
@ .s . unsigned 10 ok

```

The compound data type caddress -> unsigned on the top of the data type heap matches caddress -> single, because unsigned is a grandchild of single. 2nd in the output parameter list of @ references to single, and since single corresponds to the actual data type unsigned, the item returned by @ is also of data type unsigned. In

```
here -> caddress -> character @ .s drop caddress -> character ok
```

the actual data type referenced is a compound data type. single corresponds to caddress -> character.

There's more to say about data types, data type attributes and stack diagrams. Because data-type-attributes and stack-diagram are classes, details about their implementation have to be postponed until StrongForth's concept of object orientation has been presented.

8 Object Orientation

StrongForth's concept of object orientation is mainly based on the C++ model, providing similar features and using the same terminology. It fully supports polymorphism, encapsulation and inheritance. Classes are actually special data types. Their objects can be passed on the stack and are produced and consumed by the methods of the class. Other important features and properties are:

- Early and late binding
- Single inheritance for code and data
- Static, dynamic and class specific memory allocation
- Explicit constructors
- Full support for bit fields

Multiple class inheritance is not supported. Furthermore, all kinds of implicit data type conversions and default actions have been omitted. Implicit data type conversions are dangerous anyway, because they are sometimes ambiguous and can lead to unexpected results. Anyway, StrongForth's hierarchical data type system often makes type conversions obsolete, because words that expect items of a specific data type can generally be applied to all subtypes of it.

Class Members

Object orientation is a common feature modern of high-level programming languages. A class specifies a set of data members that can be accessed in predefined ways. For example, consider a class that describes a rectangle on the screen. A very simple version might consist of its width and its height, plus the coordinates of its lower left corner.

There are numerous ways to implement classes and objects in Forth 2012. In StrongForth, the data type system makes the implementation more challenging, because each class and each of its members has to have a data type. As a reward, StrongForth's classes are type-safe. You can only store data of the specified type in a data member, and you cannot access members and methods that do not exist in a class. Furthermore, there are no name conflicts between members and methods of different classes. You can overload members and methods as often as you like without any risk of accessing a member or a method of the wrong structure.

Here's a first example of a simple class definition and how it can be used:

```
dt object procreates rectangle

class rectangle
  null signed member px
  null signed member py
  null unsigned member width
  null unsigned member height
  : rectangle ( rectangle -- 1st ) dup erase ;
endclass

new rectangle constant rect1
+100 rect1 px !
+150 rect1 py !
40 rect1 width !
25 rect1 height !
```

Data type `object` is a direct subtype of data type `single`. Each class has its own data type that has to be directly or indirectly derived from `object`. The fact that each class is a unique data type

enables the interpreter and the compiler to perform the necessary type checks and to ensure that only members and methods that are actually included in a given class can be accessed.

Once a new data type for a class has been created, its members and methods can be defined in a class definition between `class` and `endclass`. `class` checks whether the given data type was really derived from data type `object`. It leaves an item of data type `object-size` on the stack, which counts the total size of class members in bits. This bit counter is incremented each time a new member is added to the class. At the end of the class definition, `endclass` stores the accumulated size of the class members as an attribute of the class's data type. The attributes of the data type of the currently defined class are stored in the global variable `this-attributes` to be readily available within the class definition wherever they are needed.

`member` is a defining word that defines the data members of the class. In the above example, the data members have the following stack diagrams:

```
px ( rectangle -- address -> signed )
py ( rectangle -- address -> signed )
width ( rectangle -- address -> unsigned )
height ( rectangle -- address -> unsigned )
```

`member` expects the current size of the class in bits, which is provided by `class`, plus a dummy parameter that delivers the data type of the new member. A member definition looks like a variable definition. But since the data member is not being initialized, a null item can be provided as a sample for the data type. The execution semantic of a data member is to return its address within a specific object of the class. StrongForth does not provide defining words for data members that have the semantics of values.

`new` allocates dynamic memory for an object of a given class, without initializing it. Additionally, it invokes the so-called *constructor* of the class for the new object. A constructor is a method of a class that has the same name as the class itself. Constructors always expect an object of the class as the last input parameter, and return this object as their only output parameter. They usually perform initialization tasks, including assigning all data members predefined values. In the case of `rectangle`, the constructor sets all data members to zero using an overloaded version of `erase` for items of data type `object`. The new object may be stored as a constant, as in this example. `px`, `py`, `width` and `height` can from now on be used to access the data members of the object.

Note that `new` requires the existence of a constructor. Without a constructor, it is not possible to create an object of a class. Defining a class without a constructor can make sense, if the only purpose of this class is to derive child classes from it. Class `object` is an example:

```
dt single procreates object
class object
  forth definitions
  virtual delete ( object -- )
  :noname ( object -- )
    cast address free ; is delete
endclass
```

To free the dynamic memory space occupied by an object, you should use `delete`. `delete` expects an object on the stack. It can thus be applied to any kind of object, including objects of class `rectangle`. Apart from freeing the dynamic memory space allocated for an object, many classes require additional cleanup work to be done when one of their objects is to be deleted. To accomplish this cleanup work, you can provide a class with an individual version of `delete`. But a simple overloaded version of `delete` will not do the job correctly in all situations. In fact,

`delete` is a so-called *virtual method* of class `object`. Virtual methods will be presented later in this chapter. Class `rectangle` does not need an additional cleanup.

If you prefer to allocate static memory space or to use any other specific memory location for an object instead of allocating dynamic memory, you simply provide `new` with the address as an additional parameter:

```
here -> rectangle 5 cells allot
new rectangle constant rect2
```

In this sample code, a new object of class `rectangle` is created at a pre-allocated memory area in the default memory space. But why do we need to allocate 5 cells, although objects of the `rectangle` class only have four cell-size members? The fifth cell has to be reserved for a pointer to the so-called *virtual method table*. For each class, this table contains the size of objects of the class and the addresses of all its virtual methods. The pointer can thus be used to unambiguously identify the class the object belongs to. Details about the virtual method table will be explained later in this chapter.

An alternative to the above creation of `rect2` is to supply `new` just with the memory space in which a sufficiently large memory area for an object of class `rectangle` shall be allotted:

```
data-space new rectangle constant rect3
```

If the default memory space actually was the data space, the two code samples would have exactly the same effect. Of course, objects that are not allocated in the dynamic memory space may not be deleted. Applying `delete` to a statically allocated object will cause an ambiguous condition and can lead to a system crash.

Data members need not be single-cell items. It is also possible to define double-cell items, floating-point numbers with three different levels of precision, and character size items as members:

```
member ( object-size single -- 1st )
member ( object-size double -- 1st )
member ( object-size float -- 1st )
cmember ( object-size single -- 1st )
sfmember ( object-size float -- 1st )
dfmember ( object-size float -- 1st )
```

`cmember`, for example, defines a character size data member, that returns an address of data type `caddress` -> Of course, defining character size members can lead to the following members getting unaligned. `aligned` can be used to re-align the offset. Because `object-size` is a child of `unsigned`, the overloaded version of `aligned` for data type address would apply only in connection with type casts:

```
cast address aligned cast object-size
```

That's not really elegant. A better solution is using the overloaded version of `aligned` for two items of data type `unsigned`:

```
1 cells aligned
```

That'll do nicely. If you need this phrase rather often, you might consider defining a new word for it:

```
: aligned ( object-size - 1st )
  1 cells aligned ;
```

Arrays of data members with the same data type can be defined with `members`, `cmembers` etc. These defining words expect an additional size parameter of data type `unsigned` on the stack. Their runtime semantics is to return the address of the first array element. Here's a list of all defining words for data members:

```

members ( object-size single unsigned -- 1st )
members ( object-size double unsigned -- 1st )
members ( object-size float unsigned -- 1st )
cmembers ( object-size single unsigned -- 1st )
sfmembers ( object-size float unsigned -- 1st )
dfmembers ( object-size float unsigned -- 1st )

```

Note that the creation of a new data type for the class is decoupled from the class definition. I. e., you first have to create a new data type that is directly or indirectly derived from data type `object`, and then you can specify the members and methods of the class. This decoupling makes it possible to implement cross references between two or more classes, like in this example:

```

dt object procreates class-a
dt object procreates class-b

class class-a
  null class-b member cb
  \ ...
endclass

class class-b
  null class-a member ca
  \ ...
endclass

```

When dealing with classes, overloading once more becomes a useful feature, because the names of members and methods can be reused for different classes. Since the input parameters of member definitions like `px` have the data type of the class they belong to, data members from different classes do never interfere, even if they have the same name. The interpreter and the compiler are always able to choose the correct version.

Class Methods and the `this` Object

Up to now, the constructor of the `rectangle` class is its only method. Usually, a method of a class is an ordinary word whose last input parameter is an object of the class's data type. To make the class object available throughout the definition of the method, it is generally assigned to a local named `this`. Let's add two more methods to the `rectangle` class:

```

dt object procreates rectangle
class rectangle
  +0 member px
  +0 member py
  0 member width
  0 member height
  : rectangle ( rectangle -- 1st )
    locals( this ) this erase this ;
  : set-position ( signed signed rectangle -- )
    locals( this ) this py ! this px ! ;
  : get-position ( rectangle -- signed signed )
    locals( this ) this px @ this py @ ;
endclass

```

It is also possible to provide multiple overloaded constructors with different sets of parameters, for example:

```

: rectangle ( signed signed rectangle -- 3rd )
  locals( this ) this erase this set-position this ;

```

Now, two different kinds of initializations are possible:

```
new rectangle dup get-position . . delete 0 0 ok  
+12 +40 new rectangle dup get-position . . delete 40 12 ok
```

The second constructor can be defined outside of the scope of the class definition, although it is good practice to define all constructors between `class` and `endclass`. When members are being encapsulated, there's usually no other choice. You'll learn more about encapsulation in the next section of this chapter.

The two other methods of class `rectangle` are `set-position` and `get-position`. Of course, they both have a parameter of data type `rectangle` at the end of their input parameter list.

All members and methods have a specific object of their class as the last input parameter. As the definition of the `rectangle` class in the previous section demonstrates, it is often convenient to have this object available as a local called `this` within the definition of a method. Whenever a data member or another method of the same class is used, it just has to be preceded by `this`. So, wouldn't it be nice if the compiler automatically inserted `this` whenever a data member or a method of the same class is used? Of course, it should only do so if an object of the class is not already on the stack, because in some cases the member or method belongs to a different object of the same class.

Such a feature really exists. And here's how it works. If the compiler does not find a match for the word to be compiled, given the word's name and stack diagram plus the contents of the compiler data type heap, it looks for a local named `this`, compiles it, and then tries again to find a match for the word. This means, the compiler simply assumes that the missing object on the data stack is the one returned by the local `this`. With this feature, the definitions of the methods of class `rectangle` can be simplified:

```
: rectangle ( rectangle -- 1st )  
  locals( this ) erase this ;  
: set-position ( signed signed rectangle -- )  
  locals( this ) py ! px ! ;  
: get-position ( rectangle -- signed signed )  
  locals( this ) px @ py @ ;
```

However, you have to be careful whenever you deal with two different class objects within the definition of a method. In those cases, it is good practice not to rely on the automatic `this` feature and instead compile `this` explicitly for each member and method compiled into the definition.

Encapsulation

Encapsulation is one of the major properties of object oriented programming. It means that classes hide the details of their internal data representation by only providing access to a number of interface methods that restrict the manipulation of internal data. The internal representation of the `rectangle` class is not encapsulated, because you can freely access its data members:

```
new rectangle constant rect4 ok  
-20 rect4 px ! ok  
rect4 get-position . . 0 -20 ok
```

Like C++, StrongForth has three levels of information hiding:

- *Private* members can be accessed only within the same class definition. For example, if `px` were a private data member, it could be used by `set-position`, `get-position` and `rectangle`, but not by any word that is defined after `endclass`.

- *Protected* members can be accessed like private members, and additionally within the class definitions of all derived classes.
- *Public* members don't have any access restrictions. So far, all members of the `rectangle` class are public, because they are not explicitly declared *private* or *protected*.

Note that the three levels can be applied to both members and methods. Usually, all data members are either private or protected, but it is often useful to also restrict access to methods that are supposed to be used only internally by the class.

Access to members and methods can be restricted by defining them in the `private` or `protected` vocabularies. Each class has its own instances of these two vocabularies, and it is generally not possible to access them from outside the class definition. For example, we can make the `px` and `py` data members of the `rectangle` class private:

```
dt object procreates rectangle
class rectangle
  private definitions
  +0 member px
  +0 member py
  forth definitions
  0 member width
  0 member height
  : rectangle ( rectangle -- 1st )
    locals( this ) erase this ;
  : set-position ( signed signed rectangle -- )
    locals( this ) py ! px ! ;
  : get-position ( rectangle -- signed signed )
    locals( this ) px @ py @ ;
endclass
```

To be able to access the four data members within the class definition, the `private` vocabulary has to be in the search order. `class` saves the current compilation vocabulary at the beginning of the class definition, and `endclass` restores it. After `endclass`, the `private` and the `protected` vocabularies of the `rectangle` class are inaccessible, which means that all access to the data members is restricted to using the public members and methods:

```
new rectangle constant rect5   ok
+20 rect5 px !
+20 rect5 px  ? undefined word
signed rectangle
rect5 get-position . . 0 0   ok
+20 +0 rect5 set-position   ok
rect5 get-position . . 0 20  ok
45 rect5 width !   ok
```

In some cases, it is necessary for one class to access the data members of another class. Does this mean you have to make the data members public? No, not necessarily. StrongForth supports the same mechanism as C++. A class that is a *friend* of another class can access its private and protected members, just as if these members were defined in the friend class itself. Within the definition of a class `class-A`, a list of classes `class-B1 class-B2 ... class-Bn` can be granted access to the private and protected members of class `A` with the phrase:

```
friends( class-B1 class-B2 ... class-Bn )
```

Within each of the classes `class-B1 class-B2 ... class-Bn`, you can then add the combined `private` and `protected` vocabularies of class `class-A` to the search order with the phrase

access class-A

access throws an exception if class-A has not been granted access to its private and protected vocabularies with an appropriate friends (phrase. Note that only classes can be declared friends of a class. It is not possible, like in C++, to declare a single method a friend to a class.

A typical application that takes advantage of the friendship mechanism is an iterator. An iterator is a class that provides iterative access to items that belong to another class. The iterator contains a member word that returns the *next* item of the other class each time it is executed. In the following example, an iterator is defined for a simplified string class. The method next of the iterator returns one character of the string after the other, starting with the first one:

```
dt object procreates string
dt object procreates string-iterator

class string
  friends( string-iterator )
  protected definitions
  null unsigned member len
  null caddress -> character member buf
  forth definitions
  : string ( caddress -> character unsigned string -- 4 th )
    locals( this ) dup len !
    callocate -> character buf ! buf @ len @ move
    this ;
  : . ( string -- )
    locals( this ) this
    if len @
      if buf @ len @ type
        else ." <empty>"
        then
      else ." <null>"
      then space ;
  : length ( string -- unsigned )
    len @ ;
endclass

class string-iterator
  access string
  protected definitions
  null string member stri
  null unsigned member indx
  forth definitions
  : string-iterator ( string string-iterator -- 2nd )
    locals( this ) stri ! 0 indx ! this ;
  : next ( string-iterator -- character )
    locals( this ) indx @ stri @ length <
    if stri @ buf @ indx @ + @ 1 indx +!
    else null character
    then ;
endclass
```

Class string-iterator is a friend of class string, because it needs access to string's data members. Here's an example of how the two classes may be used together:

```

" abc" new string constant string1 ok
" " new string constant string2 ok
string1 . abc ok
string2 . <empty> ok
null string . <null> ok
string1 length . 3 ok
string2 length . 0 ok
string1 new string-iterator constant sil ok
sil next . a ok
sil next . b ok
sil next . c ok
sil next cast integer . 0 ok
sil delete ok

```

After the iterator has exceeded the end of the string, it is useless and can be deleted. Of course, it is possible to define an additional method that restarts the iteration.

StrongForth does not allow class definitions to be nested. In the above example, the class definition of `string-iterator` cannot be enclosed within the class definition of `string`, although this might look like a reasonable approach.

The class definitions of `string` and `string-iterator` contain, just like the latest version of the class definition of `rectangle`, only data members defined by `member`, and methods defined by `:`. Now, what happens if you define words with `variable`, `constant` and `value`? These words obviously define variables, constants and values that are *not* data members, and that are not even related to the class they are defined in. Nevertheless, it can make sense to use these defining words within a class definition, if they are added to the `private` or `protected` vocabularies of the class. Private variables, constants and values are only available to the methods of the class and to friends of the class. Their names can be reused in other classes. The following extension of the `rectangle` class contains both a private variable and a private constant:

```

dt object procreates rectangle
class rectangle
  private definitions
  +0 member px
  +0 member py
  0 member rindex
  char R constant id
  0 variable count
  forth definitions
  0 member width
  0 member height
  : set-position ( signed signed rectangle -- )
    locals( this ) py ! px ! ;
  : get-position ( rectangle -- signed signed )
    locals( this ) px @ py @ ;
  : rectangle ( rectangle -- 1st )
    locals( this ) +0 +0 set-position
    1 count +! count @ rindex ! this ;
  : .name ( rectangle -- )
    id . rindex @ . ;
endclass

```

Each object of class `rectangle` is assigned an index `rindex`. In order to ensure that the index is unique, the index of a new `rectangle` object is taken from the non-member variable `count`, that is incremented by the constructor. `.name` is a public method that displays the index together

with a one-letter identifier that is specific for objects of class `rectangle`. The identifier is defined as a constant in the private vocabulary. Here's an example of how this version of `rectangle` can be used:

```
new rectangle constant first ok
+3 -8 first set-position ok
new rectangle constant second ok
first .name first get-position swap . . R1 3 -8 ok
second .name R2 ok
```

Colon definitions can in some cases be included in a class definition as so-called *static* methods, if they are not bound to a specific object. The necessity to include those words in the class definition might arise from the fact that they access private or protected non-member variables, constants and values. If you need to overload static methods, you can add an object of the respective class type as a dummy input parameter. However, please note that static methods within StrongForth class definitions are actually not the same as static members in C++ classes.

Similarly, private constants in StrongForth must not be confused with *const* members in C++. It is not possible to reference a constant through an object of the class the constant is defined in. Furthermore, there is nothing like *const* classes or *const* members that are not allowed to change the data members. Of course, you can define a class whose methods (except for the constructors) do not change the data members, but there are no means for the compiler to enforce that a class definition really does not contain non-constant member words.

Inheritance and Binding

Inheritance is another fundamental concept of object oriented programming. Generally, a class whose data type is a direct or indirect subtype of another class, inherits its members and methods. It's a similar mechanism as for ordinary data types. For example, `dup` is a word that expects an item of data type `single` on the stack, but it can also be applied to all direct or indirect subtypes of `single`. So, let's design a simple class hierarchy that will serve as an example:

```
dt object procreates medium
dt medium procreates paper-medium
dt medium procreates electronic-medium
dt paper-medium procreates book
dt paper-medium procreates journal
dt electronic-medium procreates analog-medium
dt electronic-medium procreates digital-medium
dt digital-medium procreates CD
dt digital-medium procreates DVD
\ ...
```

Class `medium` is at the top of the class hierarchy. Each medium is supposed to have a title and a price. Here's the class definition:

```
class medium
  protected definitions
  null string member title
  null unsigned member price \ in cent
  forth definitions
  : medium ( caddress -> character unsigned medium -- 4 th )
    locals( this ) new string title ! 0 price ! this ;
  : set-price ( unsigned medium -- )
    price ! ;
```

```

: get-price ( medium -- unsigned )
  price @ ;
: sale ( medium -- )
  locals( this ) price @ 8 10 */ price ! ;
: .price ( medium -- )
  price @ 100 /mod 0 .r [char] . . s>d <# # # #> type ;
: . ( medium -- )
  locals( this ) title @ . ." ($" .price ." )" ;
endclass

```

The data members are protected, while the member words including the constructors are public definitions that can be accessed from outside of the class definition. Remember that protected members can still be accessed within the class definitions of derived classes.

Since class `paper-medium` is a child of class `medium`, it inherits all of `medium`'s members and methods. Additionally, it defines the number of pages as a member and a method that returns the number of pages. Its constructor reuses the constructor of `medium` and additionally initializes the number of pages:

```

class paper-medium
  protected definitions
  null unsigned member #pages
  forth definitions
  : paper-medium ( caddress -> character unsigned unsigned
    paper-medium -- 5 th )
    locals( this ) #pages ! medium ;
  : pages ( paper-medium -- unsigned )
    #pages @ ;
  : . ( paper-medium -- )
    locals( this ) . cr pages . ." pages" ;
endclass

```

Finally, here are the definitions of the `book` and `journal` classes, which are both derived from `paper-medium`. The other classes (`electronic-medium` and its derived classes) are not required for this example.

```

class book
  protected definitions
  null string member author
  forth definitions
  : book ( caddress -> character unsigned unsigned book -- 5 th )
    locals( this ) null string author ! paper-medium ;
  : set-author ( caddress -> character unsigned book -- )
    locals( this ) new string author ! ;
  : get-author ( book -- string )
    author @ ;
  : . ( book -- )
    dup get-author . ." - " . ;
endclass

class journal
  protected definitions
  null unsigned member year
  null unsigned member month
  forth definitions
  : journal ( caddress -> character unsigned unsigned unsigned
    unsigned journal -- 7 th )

```

```

    locals( this ) year ! month ! paper-medium ;
: sale ( journal -- )
  100 swap price ! ;
: .edition ( journal -- )
  dup month @ 0 .r [char] / . year @ . ;
: . ( journal -- )
  dup . cr ." edition " .edition ;
endclass

```

Let's now create objects of each of these four classes. The two classes at the bottom of the class hierarchy (book and journal) inherit all public and protected members from their common parent paper-medium and their grandparent medium.

```

" No name" new medium constant medium1 ok
medium1 . No name ($0.00) ok
" White paper" 1 new paper-medium constant paper1 ok
paper1 . White paper ($0.00)
1 pages ok
" Starting Forth" 348 new book constant book1 ok
book1 . <null> - Starting Forth ($0.00)
348 pages ok
" Scientific American" 114 3 2008 new journal constant journ1 ok
journ1 . Scientific American ($0.00)
114 pages
edition 3/2008 ok
" Leo Brodie" book1 set-author ok
1999 book1 set-price ok
book1 . Leo Brodie - Starting Forth ($19.99)
348 pages ok
book1 pages . 348 ok
book1 get-author . Leo Brodie ok
book1 sale ok
book1 get-price . 1599 ok
book1 .price 15.99 ok
799 journ1 set-price ok
journ1 pages . 114 ok
journ1 get-price . 799 ok
journ1 sale ok
journ1 . Scientific American ($1.00)
114 pages
edition 3/2008 ok

```

The polymorphism of the words `.` and `sale` is simply implemented by overloading. Since this is one of StrongForth's basic features, no special treatment for object oriented programming is required. A different version of `.` is included in each of the four classes. However, `.` for a derived class generally uses the version defined for the respective parent class. For example, the method `.` of class `book` first displays the author followed by a dash and then executes `..`. Which version of `.` is it? The compiler sees an item of data type `book` on the compiler data type heap. Searching the dictionary, the first match it finds is the method `.` of `book`'s parent class `paper-medium`. As a result, `.` for objects of class `book` displays the author, the title, the price and the number of pages. `sale`, on the other hand, is defined in `medium` and redefined in `journal`. This means that executing `sale` for objects of the classes `medium`, `paper-medium` and `book` causes a 20% price reduction, while `sale` for objects of the class `journal` causes the price to be reduced to \$1.00, no matter what the previous price was.

Based on the class hierarchy defined in the previous section, let's now try to implement a word that advertises the sellout of a given medium:

```
: sellout ( medium -- )
  ." Save money now!" cr dup . cr dup sale
  ." Now for only $" .price ." !" ; ok
1999 book1 set-price ok
book1 . Leo Brodie - Starting Forth ($19.99)
348 pages ok
799 journ1 set-price ok
journ1 . Scientific American ($7.99)
114 pages
edition 3/2008 ok
book1 sellout Save money now!
Starting Forth ($19.99)
Now for only $15.99! ok
journ1 sellout Save money now!
Scientific American ($7.99)
Now for only $6.39! ok
```

Well, this did not work as intended. A 20% price reduction is granted for the book, but `sellout` does not display the author and the number of pages. From the display of the journal, the number of pages and the edition is missing. Even worse, the reduced price is not correct. `sellout` grants 20% for the journal, even though class `journal` has its own version of `sale` that reduces the price to \$1.00. What happened? When `.` and `sale` are compiled into `sellout`, the compiler sees an item of data type `medium` on the compiler data type heap and thus selects `medium`'s versions of the two methods. `.` and `sale` are statically bound to `medium` by the compiler. To resolve this problem, they have to be made virtual methods of the class hierarchy. Virtual methods are bound dynamically at runtime, i. e., the versions of `.` and `sale` to be executed depend on the actual class of the object.

The data type of an object can be determined at runtime, because each object contains a pointer to the virtual method table of its class. The memory image of a virtual method table contains the size in address units of the class objects in the first cell, plus the execution tokens of the virtual methods of the class. The size of the virtual method table itself is an attribute of the class. It cannot directly be obtained from an object, because there's no link from the virtual method table to the class it belongs to.

When a virtual method is executed for an object, its execution token is taken from the virtual method table of the class, requiring an index operation and an additional level of indirection. Executing a virtual method is thus similar to executing a deferred word.

```
virtual <name> ( ... <class> -- ... )
```

within the class definition defines a virtual method for the actual class and for all inherited classes. Its semantics can be assigned later in the class definition and reassigned within the class definitions of derived classes by

```
:noname ( ... <class> -- ... ) ... ; is <name>
```

The complete virtual method table of a class is inherited by the class's children. This means, as long as the semantics of a virtual method are not reassigned with `is <name>`, within the child's class definition, the semantics assigned by the parent remains unchanged. It is even possible to define a virtual method and postpone assigning the semantics to one or more of the derived classes. Such a virtual method is called a *pure virtual method*. An attempt to execute a pure virtual method results in an exception being thrown, because the virtual method table is initialized with the tokens of the word `unassigned`. This word does nothing else but throw this exception:

```
: unassigned ( object -- )
  drop -267 throw ;
```

The last input parameter of a virtual method always needs to be an object of the respective class. This object is required to locate the virtual method table and to get the token of the virtual method that has been assigned to the class the object belongs to. Invoking a virtual method with a null object on the stack will almost certainly cause a crash. On the other hand, a non-virtual method may be invoked with a null object on the stack, as long as no data members are being accessed based on the null object. It is even possible to define non-virtual methods that do not expect an object of its class as the last input parameter.

With this knowledge about virtual methods and dynamic binding, let's redefine the `medium` class hierarchy. Both `.` and `sale` become virtual methods:

```
class medium
  protected definitions
  null string member title
  null unsigned member price \ in cent
  forth definitions
  virtual . ( medium -- )
  virtual sale ( medium -- )
  : medium ( caddress -> character unsigned medium -- 4 th )
    locals( this ) new string title ! 0 price ! this ;
  : set-price ( unsigned medium -- )
    price ! ;
  : get-price ( medium -- unsigned )
    price @ ;
  :noname ( medium -- )
    locals( this ) price @ 8 10 */ price ! ; is sale
  : .price ( medium -- )
    price @ 100 /mod 0 .r [char] . . s>d <# # # #> type ;
  :noname ( medium -- )
    locals( this ( title @ . ." ($" .price ." )" ; is .
endclass
```

```
class paper-medium
  protected definitions
  null unsigned member #pages
  forth definitions
  : paper-medium ( caddress -> character unsigned unsigned
    paper-medium -- 5 th )
    locals( this ) #pages ! medium ;
  : pages ( paper-medium -- unsigned )
    #pages @ ;
  :noname ( paper-medium -- )
    locals( this ) [parent] . cr pages . ." pages" ; is .
endclass
```

```
class book
  protected definitions
  null string member author
  forth definitions
  : book ( caddress -> character unsigned unsigned book -- 5 th )
    locals( this ) null string author ! paper-medium ;
  : set-author ( caddress -> character unsigned book -- )
    locals( this ) new string author ! ;
  : get-author ( book -- string )
```

```

    author @ ;
:noname ( book -- )
    locals( this ) get-author . ." - " [parent] . ; is .
endclass

class journal
    protected definitions
    null unsigned member year
    null unsigned member month
    forth definitions
: journal ( caddress -> character unsigned unsigned unsigned
    unsigned journal -- 7 th )
    locals( this ) year ! month ! paper-medium ;
:noname ( journal -- )
    100 swap price ! ; is sale
: .edition ( journal -- )
    dup month @ 0 .r [char] / . year @ . ;
:noname ( journal -- )
    dup [parent] . cr ." edition " .edition ; is .
endclass

```

Every class has a virtual method table. Class object, which is the ancestor of all classes, has one virtual method, which is thus inherited by all other classes:

```
virtual delete ( object -- )
```

When an object of class object is deleted, it just has to free the dynamic memory that has been allocated for its data members. Additional cleanup work needs to be accomplished for more complex classes. If, for example, the constructor of a class allocates a buffer or opens a file, it is the task of the class-specific instance of delete to deallocate the buffer or to close the file.

Let's try sellout again:

```

: sellout ( medium -- )
  ." Save money now!" cr dup . cr dup sale
  ." Now for only $" .price ." !" ; ok
" Starting Forth 348 new book constant book1 ok
" Leo Brodie book1 set-author ok
1999 book1 set-price ok
book1 . Leo Brodie - Starting Forth ($19.99)
348 pages ok
book1 sellout Save money now!
Leo Brodie - Starting Forth ($19.99)
348 pages
Now for only $15.99! ok
" Scientific American 114 3 2008 new journal constant journ1 ok
799 journ1 set-price ok
journ1 . Scientific American ($7.99)
114 pages
edition 3/2008 ok
journ1 sellout Save money now!
Scientific American ($7.99)
114 pages
edition 3/2008
Now for only $1.00! ok

```


Now it works as expected. Both `.` and `sale` used in `sellout` are bound dynamically to their actual objects. Instead of statically being bound by the compiler to an object of class `medium`, the two words are bound to the object `sellout` receives from the stack at runtime.

But you also have to change the definitions of `.` for classes `paper-medium`, `book` and `journal`. Instead of just writing `.` to compile the version of `.` from the respective parent class, you now have to write `[parent] .` in order to accomplish the same thing as before. `[parent]` is an immediate word that compiles a virtual method bound *statically* to the version of the parent class. If you just write `.`, the compiler uses dynamic binding, which results in `.` executing itself and thus causing an endless recursion. But in this case, you want to explicitly compile the token of the parent's version of `.`. Generally, `[parent]` is useful whenever the semantics of a virtual method that is defined in the parent class is to be somehow extended in the child class. This definitely applies to `.`. On the other hand, the original version of `sale`, which is defined in class `medium`, is not being extended. The version of `sale` in class `journal` has its own semantics. If `journal`'s version were an extended version of `medium`'s version, `[parent]` could be applied as well, for example like this:

```
:noname ( journal -- )
  dup [parent] sale dup price @ 100 max swap price ! ; is sale
```

This definition works as expected even though the grandparent and not the parent of class `journal` defines the original version of `sale`. The parent `paper-medium` has inherited the version from `journal`'s grandparent `medium`. If for whatever reason you want to compile a version that is different from that of the parent class, you can use `[bind]` instead of `[parent]`:

```
[bind] medium sale
```

`[bind]` allows you to specify the direct parent class as well as any indirect parent class whose version of the virtual method word shall be compiled with static binding. Specifying any other than direct or indirect parent classes is possible, but will cause an ambiguous condition in most cases. `[bind]` is actually a generalized version of `[parent]`. Note that the usage of `[parent]` and `[bind]` is not restricted to extending the semantics of virtual methods within a class hierarchy. You can even chose to statically bind a virtual method in the definition of a non-member word like `sellout`. For example, If you want to display only the title and the original price of a medium, you can statically bind `.` to class `medium` in the definition of `sellout`:

```
: sellout ( medium -- )
  ." Save money now!" cr dup [bind] medium . cr dup sale
  ." Now for only $" .price ." !" ; ok
1999 book1 set-price ok
book1 sellout Save money now!
Starting Forth ($19.99)
Now for only $15.99! ok
799 journ1 set-price ok
journ1 sellout Save money now!
Scientific American ($7.99)
Now for only $1.00! ok
```

Members that have been compiled into the protected vocabulary of a class are passed on to child classes. However, private members can only be directly referred to within the respective class definition. Accessing private members or private methods within child class definitions or outside of a class definition can only happen indirectly through public or protected methods. That's because the private vocabulary of a class is not passed on to its children. The protected vocabulary of a class, on the other hand, is passed on to all its children. Each child class actually starts with the protected vocabulary of its parent class and extends it with its own protected members, whereas a child's private vocabulary is initially empty. Given the parent class definition

```

dt object procreates parent-class
dt parent-class procreates child-class

class parent-class
  null unsigned member public-data-member
  : public-method ( parent-class -- )
    locals( this ) ;
  private definitions
  null unsigned member private-member
  : private-method ( parent-class -- )
    locals( this ) ;
  protected definitions
  null unsigned member protected-member
  : protected-method ( parent-class -- )
    locals( this ) ;
endclass

```

here's what happens in the child class definition:

```

class child-class ok
  private words ok
  protected words
  protected-method ( parent-class -- )
  protected-member ( parent-class -- address -> unsigned ) ok

```

Note that the assignment of a virtual method word does not affect the vocabulary in which the virtual method had been originally defined with `virtual`.

Now that virtual methods have been introduced, we have to revisit the `string` class from the previous section. Although the constructor of this class allocates dynamic memory for the string buffer, it provides no means to deallocate this buffer. Each time an object of class `string` is deleted, the available dynamic memory space shrinks by the size of the character buffer. Thus, class `string` has to be extended by a dedicated version of `delete`:

```

:noname ( string --)
  locals( this ) buf @ free [parent] delete ; is delete

```

This definition extends the semantics of the virtual method `delete` of class `object` by freeing the dynamic memory allocated for the character buffer before deleting the object itself.

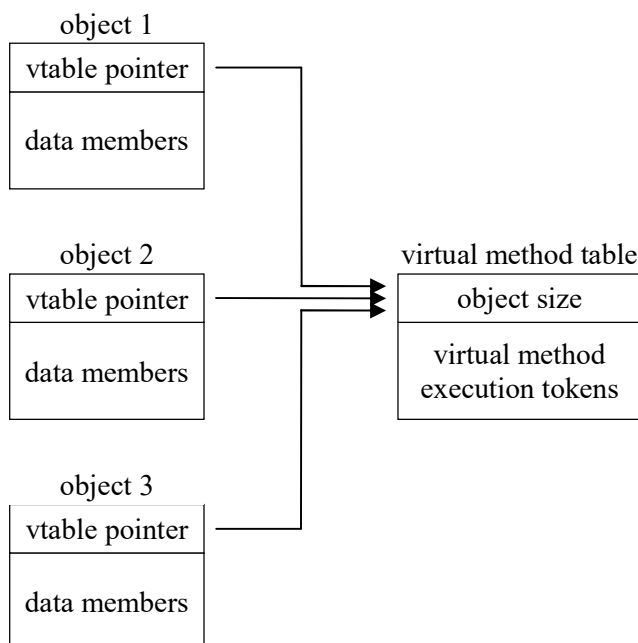
You might have noticed that the concept of multiple inheritance has not been mentioned so far. Some other object oriented languages support this concept. It means that a class can inherit members and methods from more than one parent class. However, the benefits of multiple inheritance in relation to the complexity of the implementation and the danger of ambiguities are being discussed controversially. In order to keep StrongForth's design clear and simple, multiple inheritance is not supported.

The Virtual Method Table

The memory image of an object is composed of its data members, preceeded by a pointer to the virtual method table of the class it belongs to. This is the memory image of an object of the `rectangle` class, that has been defined earlier in this chapter:

virtual method table pointer
px
py
width
height

All objects belonging to a class point to the same virtual method table:



The virtual method table pointer has its own data type `vtable`, which is a direct subtype of `single`. Thus, a limited set of operations, excluding e. g. arithmetical and logical operations, can be executed on it. To obtain the virtual method table of the class a given object belongs to, you can use this word:

```
: vtable ( object -- vtable )
  cast address -> vtable @ ;
```

The first cell of the virtual method table always contains the total size of the class's data members in address units. So it's easy to query the size of an object:

```
: size ( vtable -- unsigned )
  cast address -> unsigned @ [ address-unit-bits cells ] literal
  aligned address-unit-bits / ;

: size ( object -- unsigned )
  vtable size [ 1 cells ] literal - ;
```

We can apply `size` to an object of class `rectangle`:

```
rect1 size . 16 ok
```

That's the size in address units of four cells, as expected. The word `size`, of which several overloaded versions exist in StrongForth, generally returns the size in address units of its input parameter. Note that the object size stored in the virtual method table does not include the `vtable` pointer in the first cell of an object. In order to reserve memory space for an object, the size of this cell has to be added to the value returned by `size (object - unsigned)`.

Based on `size`, two very useful words can be applied to all objects:

```
: copy ( object 1st -- )
  over size over size min [ 1 cells ] literal /
  rot cast address -> single 1+
  rot cast address -> single 1+
  rot move ;

: erase ( object -- )
  dup cast address -> single 1+ swap size [ 1 cells ] literal /
  erase ;
```

`copy` copies the data members of an object to the data members of another object of the class. This is what is called in C++ *memberwise initialization*. However, note that something like *default memberwise initialization* does not exist in StrongForth. All initialization has to be done explicitly. For specific objects, `copy` can be overloaded if memberwise initialization is not desired. Since in most cases these overloaded versions require direct access to the data members, they usually have to be made methods within the class definition.

The phrase *another object of the same class* at the end of the first sentence of the previous paragraph has to be interpreted statically. Because, two objects with the same (static) data type do not necessarily belong to the same class. You can, for example, copy an object of class `paper-medium` to an object of class `book` or vice versa, if both objects have the same (static) data type, e. g., `medium`. That's why `copy` has to calculate the minimum of the sizes of both objects. Only those data members that both objects have in common are actually copied.

`erase` initializes all data members of an object with zero. This overloaded version of `erase` has already been used in the constructor of the `rectangle` class.

In this chapter, you've seen three virtual methods so far:

```
virtual delete ( object -- )
virtual . ( medium -- )
virtual sale ( medium -- )
```

`delete` is the only virtual method of class `object`. Because this class has no data members, its virtual method table looks like this:

0
token of delete

Since `object` is the ancestor of all classes, all other classes inherit the virtual method `delete` from it. However, they can replace it with their own versions, and they can add additional virtual methods by extending their virtual method table. `medium`, for example, adds two virtual methods:

8
token of delete
token of sale
token of .

Note that `medium` has two cell-size data members, so that the size of its objects is 8 address units. Class `paper-medium`, which is directly derived from class `medium`, adds another cell-size data member and provides its own version of the virtual method `.`:

12
token of delete
token of sale
token of <code>.</code> (modified)

On the next level in the class hierarchy, we have `book` and `journal`. Again, data members are added and virtual methods are being modified. These classes or their children or grandchildren, if they existed, could add more data members, replace the execution tokens of virtual methods or define additional virtual methods. Thus, the length of the virtual method tables as well as the size of the objects may grow from level to level in the class hierarchy.

Virtual method tables are hidden in the implementation of `StrongForth`, so you usually do not have to mess around with them. Nevertheless, two words are available that return the addresses of components of virtual method tables:

```
: 'object-size ( vtable -- address -> object-size )
  cast address -> object-size ;

: 'virtual ( unsigned vtable -- address -> token )
  cast address -> token swap + ;
```

The semantic of `'object-size` is nothing more than a type cast. `'virtual` returns the address of the virtual method with a given index within a virtual method table. Remember that `+` applies address arithmetic.

Class memory-space

The definition of class `memory-space` is a good example of how to implement a class in `StrongForth`.

```
dt object procreates memory-space
class memory-space
  private definitions
    null address member 'bottom
    null address member 'ptr
    null address member 'top
  forth definitions
    : memory-space ( address unsigned memory-space -- 3rd )
      locals( this ) over 'bottom ! over 'ptr ! + 'top ! this ;
    : memory-space ( unsigned memory-space -- 2nd )
      locals( this ) dup allocate swap memory-space ; 1 retreat
    : here ( memory-space -- address )
      'ptr @ ;
    : chere ( memory-space -- address )
      here cast address ;
```

```

: sfhere ( memory-space -- sfaddress )
  here cast sfaddress ;

: dfhere ( memory-space -- dfaddress )
  here cast dfaddress ;

: unused ( memory-space -- unsigned )
  locals( this ) 'top @ here - cast unsigned ;

: allot ( integer memory-space -- )
  locals( this ) here swap + dup 'bottom @ 'top @ within
  if 'ptr ! else drop -8 throw then ;

: shrink ( integer memory-space -- )
  locals( this ) dup unused <=
  if 'top -! else drop -8 throw then ;

' nodelete is delete
endclass

```

The internal structure of the class is hidden in the three private members 'bottom, 'ptr and 'top. All operations on objects of the class have to be done using its public methods. This is the preferred technique in object oriented programming. More complex classes typically have a number of private methods as well, but class `memory-space` does not. As can be seen from the source code, it is not possible to change the value of 'bottom, the starting address of the memory space, once it has been initialized by one of the constructors. 'top may only be changed by `shrink`, and only in one direction, provided the amount of unused memory space is still large enough. You already know the public methods of class `memory-space` from chapter 4.

Assigning `nodelete` to the virtual method `delete` prevents objects of class `memory-space` from being deleted. Deleting a memory space is potentially dangerous, because it is difficult to ensure that none of the data stored in it is referenced after freeing the dynamic memory. `nodelete` simply throws an exception instead of deleting an object:

```

: nodelete ( object -- )
  drop -275 throw ;

```

9 Data Types Revisited

Data Type Attributes

Data types are identified by their name and by a set of attributes, primarily by their parent and their size. These attributes are actually the data members of a class:

```
dt object procreates data-type-attributes
class data-type-attributes
  forth definitions

  null data-type-attributes member 'parent
  null vtable member 'vtable
  null logical cmember 'register
  null unsigned cmember 'size
  null unsigned cmember #vtable

  : data-type-attributes ( data-type unsigned data-type-attributes
    -- 3rd )
    locals( this ) erase 'size !
    >attributes dup this 'parent ! 'register @ 'register !
    this ;

  ' nodelete is delete
endclass
```

Note that the members of this class are public and that the class does not have any methods apart from its constructor. The members are not encapsulated; they can be accessed arbitrarily. Note also that objects of this class may not be deleted, because it's difficult to ensure that no stack diagrams referencing these objects remain.

Before we continue, we have to introduce two words that handle the relationship between objects of data type `data-type-attributes` and items of data type `data-type`. In chapter 7, you've learned that a data type, as it is stored in stack diagrams and in the two data type heaps, is usually composed of an identifier plus additional attributes like `dt-prefix` and `dt-reference`. If the data type is not a reference, the identifier is an object of data type `data-type-attribute`. It can be extracted with `>attributes`:

```
: >attributes ( data-type -- data-type-attributes )
  low cast data-type-attributes ;
```

In the opposite direction, you can construct a data type with attributes from its identifier:

```
: >data-type ( data-type-attributes -- data-type )
  null logical merge cast data-type ;
```

The first data member of class `data-type-attributes`, `'parent`, contains the identifier of the parent data type. Another one stores the size in address units:

```
dt unsigned >attributes 'parent @ >data-type . integer   ok
dt unsigned >attributes 'size @ . 4                       ok
```

This looks pretty cumbersome, doesn't it? Okay, StrongForth provides two words that make the access to `'parent` and `'size` more convenient:

```

: parent ( data-type -- 1st )
  >attributes 'parent @ >data-type ;

: size ( data-type -- unsigned )
  >attributes 'size @ ;

```

Using these words, querying the parent and the size in address units of a data type is indeed easier:

```

dt data-type parent . double ok
dt data-type size . 8 ok

```

If `parent` is applied repeatedly, starting with any data type, it will sooner or later end up with either `single`, `double` or `float`, because all predefined data types in StrongForth are direct or indirect subtypes of these three data types:

```

dt unsigned parent dup . integer ok
parent dup . single ok
parent dup .
parent dup . ? is not a data type
data-type

```

Data type `single` has no parent. The content of `'parent` is null.

`'vtable` keeps a pointer to the virtual method table, and `#vtable` contains the number of execution tokens in the virtual method table, if the data type is a class. Otherwise, the contents of `'vtable` and `#vtable` are zero. There's a word that fetches the contents of `'vtable` for a given data type:

```

: vtable ( data-type -- vtable )
  >attributes 'vtable @ ;

```

`'register` indicates the preferred processor register or registers in which items of the associated data type shall be stored. This is a hint for the compiler you normally won't have to take care about.

The constructor of the `data-type-attributes` class initializes `'parent` and `'size` from its input parameters. `'register` is copied from the parent data type and `'vtable` and `#vtable` remain zero. An interesting detail in this otherwise pretty straightforward definition is the explicit usage of the `this` object as the input parameter for `'parent`. Is that really required? Yes, it is. `'parent` expects an item of data type `data-type-attributes` on the stack. If this was missing, `'parent` would assume the output parameter of `>attributes` as its input parameter, which is actually the data type attribute of the parent data type. This is not what we want to do here. So be careful whenever you deal with multiple objects of the class type within a method definition.

But what about the name of the data type? As you can see, it is not contained as a data member of class `data-type-attributes`. Finding the name of a data type actually requires a dictionary search that is included in words like `.` for items of data type `data-type`. We'll get to this in chapter 11.

parent can be applied iteratively to find out whether a data type is an ancestor of another data type:

```
: ancestor? ( data-type data-type -- flag )
  swap
  begin over over <>
  while dup >attributes
    while parent
      repeat false
    else true
  then nip nip ;
```

The first input parameter of `ancestor?` is the data type to be analysed, the second one is the ancestor data type. If, for example, we need to find out whether a data type is a class, we just have to check whether data type object is its ancestor:

```
: object? ( data-type -- flag )
  [dt] object ancestor? ;
```

[dt] is the compile-time version of dt. It compiles a data type as a literal.

Just querying the content of the 'vtable member to find out whether a data type is a class does not always deliver the correct result, because it is initialized with null. 'vtable does not contain a valid virtual method table before the end of the class definition. Here are some examples:

```
dt single dt integer ancestor? . false ok
dt integer dt integer ancestor? . true ok
dt unsigned dt integer ancestor? . true ok
dt object-size dt integer ancestor? . true ok
dt memory-space object? . true ok
dt integer object? . false ok
```

Class Attributes

Data type object and its direct and indirect subtypes are classes. The pointer to its virtual method table is stored in the 'vtable member of class data-type-attributes. However, a class requires some more attributes. That's why class data-type-attributes has a child class:

```
dt data-type-attributes procreates class-attributes
class class-attributes
  forth definitions
  null unsigned cmember #friends
  null address -> class-attributes member 'friends
  null definition member 'last
  null vocabulary member 'vocabulary

  : class-attributes ( data-type unsigned class-attributes
    -- 3rd )
    data-type-attributes ;

endclass
```

In fact, classes are being interpreted as a kind of extended data types with additional members.

'friends keeps a pointer to a list of the class attributes of all classes that have been declared friends. #friends holds the length of this list, or zero if the class has no friends.

The pointer to the latest definition in the `protected` vocabulary of the class is stored in `'last`. The `protected` vocabulary of all children of the class starts with this definition. As a consequence, all `protected` definitions of the class and its ancestors are made available to the children, who may in turn extend the `protected` vocabulary and hand it over to their own children.

`'vocabulary` has a similar semantics. However, this member contains an independent vocabulary that includes both the private and `protected` definitions of the class, in order to make them available to its friends. In chapter 13 you'll see how this mechanism works in detail.

In analogy to `>attributes`, `StrongForth` provides a word that extracts the class attributes from an item of data type `data-type`. If the data type is not a class, `>class-attributes` throws an exception:

```
: >class-attributes ( data-type -- class-attributes )
  dup object?
  if >attributes cast class-attributes
  else drop -271 throw null class-attributes
  then ;
```

Note that the phrase `null class-attributes` after `throw` cannot be omitted. The compiler would complain that the two branches of the `if` clause lead to different data type heaps and can thus not be merged by `then`.

The Data Type Hierarchy

All predefined data types are directly or indirectly derived from the three ancestors `single`, `double` and `float`. In total, `StrongForth` provides 75 different data types. You can define additional data types suitable for your applications, and it is even possible to define new ancestor data types. The latter ones become handy if you want to restrict manipulation of private data types to a handful of selected words you provide. Here's the complete predefined data type hierarchy in the `forth` vocabulary:

```
single
  integer
    unsigned
      object-size
    signed
  character
  keyboard-event
  address
    caddress
    sfaddress
    dfaddress
  logical
    flag
  token
    search-criterion
      (--)
      (unsigned--)
      (--string)
  file
  vtable
  fam
  r-index
  search-order
```

```

structure
object
  stack-diagram
  input-stream
    terminal-input-stream
    string-input-stream
    file-input-stream
    block-input-stream
  output-stream
    terminal-output-stream
    string-output-stream
    file-output-stream
  control-flow
    origin
      of-origin
      endof-origin
    destination
      do-destination
  exception-frame
  compiler-workspace
  memory-space
  data-type-attributes
    class-attributes
    structure-attributes
  definition
    code-definition
      colon-definition
    created-definition
    single-definition
      value-definition
      to-value-definition
    double-definition
    float-definition
    local-definition
      to-local-definition
    deferred-definition
    virtual-definition
    member-definition
      bmember-definition
  vocabulary
    integer-vocabulary
    float-vocabulary
  marker-class
  line-editor
double
  integer-double
    unsigned-double
    number-double
    signed-double
  data-type
  baddress
float

```

Additional data types are defined in some of the StrongForth utilities and examples, which are considered to be applications, and in the `assembler` and `msvcrt` vocabularies.

Class stack-diagram

In StrongForth, a stack diagram is implemented as an object of class `stack-diagram`. Usually, this object is deleted once the stack diagram is completed, verified and processed, e. g., by storing it as part of a definition. This is the class definition:

```
dt object procreates stack-diagram

class stack-diagram
  private definitions
    null unsigned cmember #params
    null unsigned cmember #input-params
    null flag cmember 'output
    null flag cmember 'saved-state
    null logical 2 cmembers 'registers
    null data-type /params members 'params

    : next-param ( stack-diagram -- address -> data-type )
      locals( this ) 'params #params @ + ;

    : latest-param ( stack-diagram -- address -> data-type )
      next-param 1- ;

    : attributes? ( data-type stack-diagram -- flag )
      locals( this ) #params @
      if latest-param @ and high if true exit then
      else drop
      then false ;

    : prefix? ( stack-diagram -- flag )
      dt-prefix swap attributes? ;

    : reference? ( stack-diagram -- flag )
      dt-reference swap attributes? ;

    : invalid ( stack-diagram -- )
      drop -262 throw ;

  forth definitions

    : stack-diagram ( flag stack-diagram -- 2nd )
      locals( this ) erase 'saved-state ! this ;

    : input-params ( stack-diagram --
      address -> data-type unsigned )
      locals( this ) 'params #input-params @ ;

    : output-params ( stack-diagram --
      address -> data-type unsigned )
      locals( this ) input-params + #params @ #input-params @ - ;

    : param, ( data-type stack-diagram -- )
      locals( this )
      #params @ /params <
      if next-param ! 1 #params +!
        'output @ invert if 1 #input-params +! then
      else drop -259 throw
      then ;
```

```

: input-param, ( data-type stack-diagram -- )
  locals( this )
  output-params null data-type param, over dup 1+ rot move !
  1 #input-params +! ;

: -> ( stack-diagram -- 1st )
  locals( this )
  #params @ #input-params @ 'output @ and - 0=
  prefix? or reference? or
  if invalid
  else latest-param dup @ dt-prefix or swap !
  then this ;

: -- ( stack-diagram -- 1st )
  locals( this ) prefix? 'output @ or
  if invalid
  else true 'output !
  then this ;

: ?stack-diagram ( stack-diagram -- )
  locals( this ) prefix? 'output @ invert or
  if invalid
  then ;

:noname ( stack-diagram -- )
  dup 'saved-state @ state ! [parent] delete ; is delete
endclass

```

All data members and a number of internally used methods are private. The basic data types that constitute the stack diagram are stored in the given order, first the input parameters and then the output parameters, in the private array 'params. /params is the maximum number of basic data types in a stack diagram. It is defined as a constant outside the class definition:

```
32 constant /params
```

#params is the total number of basic data types in the array. #input-params is the number of basic data types used as input parameters, or the index of the first basic data type used as output parameter. The content of 'output tells whether input or output parameters are being added to the stack diagram. Because stack diagrams are always interpreted code, the state at the beginning of the stack diagram needs to be saved in 'saved-state and restored at the end of the stack diagram. The two character-size members called 'registers contain internal information for the compiler.

Class stack-diagram has some private methods that can only be used within the class definition itself. The definition of publicly visible methods begins after the phrase forth definitions with the constructor. input-params and output-params return the lists of basic data types that have been added so far as input and output parameters, respectively. param, adds one basic data type to the input or output parameters, depending on 'output. If it becomes necessary to add an input parameter although adding output parameters has already been started or even completed, input-param, will do the job.

-> and -- have already been presented in chapter 7 as part of the class definition of class stack-diagram. These words check for syntax violations. For example, -> ensures that it is preceded by at least one input or output parameter that does not have the reference or prefix attribute.

?stack-diagram checks whether a stack diagram is complete, i. e., it doesn't have open ends like an unfinished prefixed parameter of a missing output parameter list.

Class `stack-diagram` extends the semantics of the virtual method `delete`, which it inherited from class `object`. It restores the compilation state to what it was when the stack diagram was created, and then executes the semantics of the version of `delete` that came with class `object`. This is quite typical for virtual methods. They first do some extra stuff, and then do the same things their parent class does.

The word `th`, which is another important component of stack diagrams, is not included in the class definition, because it can be defined outside. It does not need access to the private members and methods of class `stack-diagram`:

```
: th ( stack-diagram unsigned -- 1st )
  locals( index )
  dup input-params index >=
  if index + 1- @ reference?
  else drop true
  then
  if -261 throw
  else index [ dt-reference high ] literal merge cast data-type
    over param,
  then ;
```

The first half of the definition of `th` is dedicated to syntax checks. The `index` has to reference a basic data type within the input parameter list, which is not a reference itself.

Stack diagrams are syntactically initiated by a left parenthesis:

```
: ( ( -- stack-diagram )
  state @ new stack-diagram [compile] [ ; immediate
```

`(` is an immediate word, because it can be used during compilation. It saves the current compilation state and enters interpretation state in order to interpret the components of the stack diagram.

At the end of a stack diagram, `)` stores the stack diagram in the most recent or current definition and then deletes it, thereby restoring the compilation state. Assigning a stack diagram to a definition is a method of the class `definition`, which will be presented in the next chapter.

10 Definitions

Class definition

Just like in Forth 2012, the StrongForth dictionary is a list of words. Each word consists of several components:

- its name (unless the word was defined by `:noname`),
- a link to the previous word in the dictionary,
- a parameter and/or data field,
- attributes (like the immediate flag).

In StrongForth, a word additionally contains its stack diagram. Otherwise, the interpreter and the compiler wouldn't be able to perform type checking and to distinguish overloaded words.

The parameter and/or data field may contain things like an execution token, a pointer to executable machine code, or other data that specify the semantics of the word.

In StrongForth, each word is an object of class `definition` or a derived class. New words are being allocated in dynamic memory. StrongForth has no name space. The advantage of allocating words in dynamic memory is that individual words can be deleted without resulting memory leaks.

```
dt object procreates definition
```

```
class definition
```

```
    forth definitions
```

```
    virtual >body ( definition -- address )
    virtual token ( definition -- token )
    virtual (compile) ( compiler-workspace definition -- )
    virtual see ( definition -- )
```

```
    private definitions
```

```
    : unlink ( vocabulary definition -- 1st )
      locals( this )
      begin dup
      while dup last this =
        if prev over last! exit
        then next
      repeat ;
```

```
    protected definitions
```

```
    null caddress -> character member 'name
    null definition member 'link
    null address -> data-type member 'params
    null logical member 'attributes
    null unsigned cmember #name
    null unsigned cmember #params
    null unsigned cmember #input-params

    : name! ( caddress -> character unsigned definition -- )
      locals( this ) dup
      if dup callocate -> character
        over #name ! dup 'name ! swap move
```

```

    else drop drop
    then ;
: link! ( definition -- )
  to latest current @ last latest 'link ! ;
forth definitions
: definition ( definition -- 1st )
  dup erase dup link! ;
: definition ( caddress -> character unsigned
  definition -- 4 th )
  locals( this ) definition name! this ;
: definition ( caddress -> character unsigned definition
  definition -- 5 th )
  locals( this ) this copy link! name!
  0 #params ! 0 #input-params !
  null address -> data-type 'params ! this ;
: params! ( stack-diagram definition -- )
  locals( sd this ) sd ?stack-diagram
  sd input-params dup #input-params !
  sd output-params nip + dup #params !
  dup cells 2* allocate -> data-type dup 'params ! swap move
  sd delete ;
: input-params ( definition -- address -> data-type unsigned )
  locals( this ) 'params @ #input-params @ ;
: output-params ( definition -- address -> data-type unsigned )
  locals( this ) input-params + #params @ #input-params @ - ;
: name ( definition -- caddress -> character unsigned )
  locals( this ) 'name @ #name @ ;
: prev ( definition -- definition )
  'link @ ;
: link ( definition definition -- )
  'link ! ;
: attributes! ( logical definition -- )
  locals( this ) 'attributes @ or 'attributes ! ;
: attributes? ( logical definition -- flag )
  'attributes @ and 0<> ;
code ?congruent ( definition -- )
  \ ...
  endcode
: enddef ( definition -- )
  current @ last! ;
:noname ( definition -- address )
  drop null address ; is >body
:noname ( definition -- token )
  drop null token ; is token
:noname ( compiler-workspace definition -- )
  drop drop ; is (compile)

```



```

:noname ( definition -- )
  drop ; is see

:noname ( definition -- )
  locals( this ) context @ unlink 0=
  if hidden @ unlink 0=
    if null caddress -> character 0 this
      link-criterion search-all
      if this 'link swap link
      else drop
      then
    then
  then 'name @ free 'params @ free
  [parent] delete ; is delete

endclass

```

The seven data members, three of them character-size, contain the name, a link to the previous word, the stack diagram and the attributes. Class `definition` doesn't have a parameter field or a data field, because it describes a generic word without compilation or interpretation semantics. More specific words, e. g. for colon definitions or constants, are objects of derived classes, which extend class `definition` with additional data like execution tokens or constant values. Because all data members are `protected`, they can be accessed in the class definitions of derived classes, but not outside class definitions.

Two methods of class `definition` are also included in the `protected` vocabulary. `name!` allocates space in dynamic memory for the name string of the definition, and stores the address in `'name` and the character count in `#name`. `link!` establishes a link to the previous word by storing it in `'link`. The phrase `current @ last` returns the previously defined word within the same word list as an object of data type `definition`. Furthermore, `link!` stores the new word in a value called `latest`. At any time, `latest` contains the word that is currently being compiled or that has most recently been compiled:

```

null definition value latest

```

Class `definition` has three publicly available constructors. The first one is for words with no name. It just erases all its data members and establishes a link to the previous word. The second version extends the first version by assigning the word a name given by the input parameters `caddress -> character unsigned`. The third version of the constructor creates a copy of an existing object of the same class with a new name, a new link and a blank stack diagram. Erasing the stack diagram is crucial. Otherwise, the original definition and the copy would share the same stack diagram. If one of the definitions gets deleted, the stack diagram of the other could no longer be safely accessed.

The majority of the remaining methods grant access to the data members by either querying or changing them. `params!` assigns a word a stack diagram by copying the input and output parameters stored in an object of data type `stack-diagram` to an array of basic data types, which is allocated from dynamic memory. At the end of `params!`, the original stack diagram is being deleted, because it is no longer required. Once a word has been assigned a stack diagram, its input and output parameters can be queried with `input-params` and `output-params`. The definitions of these two methods look identical to the corresponding methods of class `stack-diagram`.

As you might have guessed, `params!` is executed at the end of each stack diagram specification. Actually, it is included in the definition of `)`:

```
: ) ( stack-diagram -- )
  latest params! ;
```

This means, a stack diagram is always assigned the most recently created definition.

`name` returns the name of the word as a character string `caddress -> character unsigned`. The previous word in the dictionary can be obtained by `prev`. `prev` is used when it comes to iterate through the words in a vocabulary. In some cases, `prev` can also be applied to get access to a specific overloaded version of a word that is hidden below another overloaded version. This is because `'` always returns the most recently defined overloaded version of a word with a given name:

```
' rshift . rshift ( logical unsigned -- 1st ) ok
' rshift prev . rshift ( logical -- 1st ) ok
```

A more versatile means to get access to different overloaded versions will be presented in chapter 17.

`'name` and `'link` are initialized by the constructors of class definition. Nevertheless, `link` provides a means to update the link. Other than in Forth 2012, in StrongForth it is sometimes necessary to change the order of words in the dictionary. You've already seen that the order can be crucial when names are overloaded. Consider the example of the word `here`:

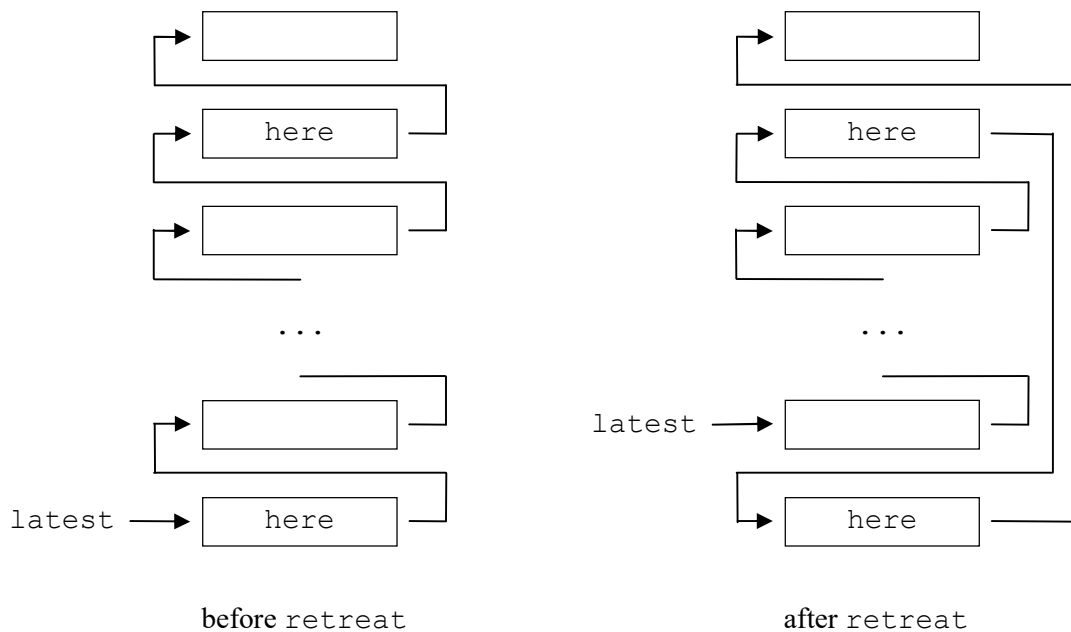
```
: here ( memory-space -- address )
  'ptr @ ;

: here ( -- address )
  default-memory-space @ here ; 1 retreat
```

The first version, with `memory-space` as the input parameter, is a method of class `memory-space`. Because the second version uses the first version, it has to be defined after the first version. However, if the two versions remain in this order, the first version will never be found in the dictionary. Even if an object of data type `memory-space` were on top of the stack, the second version will always match first. Therefore, the order of the two words has to be changed after they both have been defined. That's what `retreat` does:

```
: retreat ( unsigned -- )
  latest swap 0
  ?do
    begin prev dup 0= if drop -13 throw exit then
      dup name latest name compare 0=
    until
  loop
  dup prev latest rot link
  latest prev swap latest link
enddef ;
```

Starting with the latest definition, `retreat` traverses the dictionary, counting overloaded versions of the latest definition. Finding the `n`-th overloaded version, where `n` is the value of `unsigned`, it stops. The link of this overloaded definition and the link of the latest definition are now being changed in such a way, that the latest definition is inserted as the predecessor of the overloaded version within the word list.



Now let's get back to the methods of class definition. `attributes!` sets bits within the `'attributes` data member. Note that attributes can only be set, not cleared. The constructors of class definition always turn all attributes off. `attributes?` queries the state of the attributes. It returns `true` if one or more of the bits set in `logical` are also set in `'attributes`. Details about the attributes of a definition will be presented in the next section of this chapter.

`?congruent` is a method used by `exit` and `;`. It checks whether the contents of the compiler data type heap exactly matches the output parameters of the stack diagram of definition. A word can not be compiled if this match fails. `?congruent` compares the output parameters of the definition one by one to the data types on the data type heap. Data type references in the output parameter list are resolved. For example, given the definition

```
: example ( caddress -> character unsigned -- 2nd 3rd 1st flag )
```

the data type heap must contain the following data types in order for `?congruent` not to throw an exception:

```
character unsigned caddress -> character flag
```

Remember that data type references only occur in the stack diagrams of definitions, but never on the data type heaps. `?congruent` works both in interpretation and in compilation state, because it automatically selects the proper data type heap.

`enddef`, as the name suggests, is used at the end of a definition. It adds the definition to the top of the current word list. You've already seen this word at the end of the definition of `retreat`. It has to be used there, because after the latest definition has been relocated, its predecessor moves again to the top of the word list.

Class definition has five virtual methods, four new ones and one inherited from class `object`. Because class definition is a generic parent class, whose objects do not have any semantics, `>body` returns a null pointer as the data field. For the same reason, execution tokens returned by `token` are null. `(compile)` is a low-level word that compiles the machine code of a definition. Having no semantics at all, objects of class definition do not compile any machine code. Finally, it is not possible to display any objects of class definition with `see`. All these virtual methods will be assigned specific semantics in the children derived from class definition, e. g., colon definitions, code definitions, constants and variables.

Note that a definition in StrongForth has only one execution token. This is sufficient, because different interpretation and compilation semantics can easily be implemented by overloading two different definitions, one of them marked `execute-only` and the other one marked `compile-only`. token actually replaces both of the Forth 2012 words `NAME>INTERPRET` and `NAME>COMPILE`.

The fifth virtual method, `delete`, is inherited from class `object`. Deleting an object of class `definition` requires more work than deleting objects of class `object`. The dynamic memory spaces allocated by the constructors for the name string and the stack diagram have to be freed before the memory occupied by the data members is being returned to the dynamic memory pool. The most complex task of `delete` is re-establishing a consistently linked word list. This is done with the help of the private word `unlink`. If the definition is on top of its word list, removing the link is pretty simple. But if it has a successor, the task is more complicated, because there's no link to it. A dictionary search has to be performed to find the definition whose predecessor is the definition to be deleted.

Something you should keep in mind is that words that belong to StrongForth's kernel cannot be deleted, because they reside in static memory. These are the words whose definitions are shown in `model.sf`. All other definitions have to be explicitly compiled by including their source code files.

Like all objects, new definitions are being allocated in dynamic memory. This makes it possible to delete single definitions from the dictionary without affecting other definitions. In Forth 2012, definitions can only be deleted by either `FORGET` or `MARKER`. Based on `delete`, `forget` is implemented like this:

```
: (forget) ( definition definition -- )
  begin dup prev swap delete
    over over = over 0= or
  until drop drop ;
: forget ( -- )
  ' prev current @ last (forget) ;
```

`(forget)` deletes the definition specified by the second input parameter, plus all previous definitions in the same vocabulary up to and excluding the definition specified by the first input parameter.

StrongForth's definition of `marker` is far more complex. It will be presented in chapter 23.

Attributes of a Definition

The attributes stored in the `'attributes` member of objects of class `definition` mainly contain information about

- the registers that are changed by the code the definition compiles,
- whether the word is found by `search` in interpretation and compilation state and
- whether the word is to be executed or compiled in compilation state.

We will focus on the last two properties. With bits 16 and 17 of a definition's attributes, four types of definitions can be distinguished:

Bit 17	Bit 16	Type	Interpretation state	Compilation state
0	0	normal	execute	compile
0	1	immediate	execute	execute
1	0	execute-only	execute	not found
1	1	compile-only	not found	execute

By default, a definition can be found by `search` in either interpretation and compilation state. It is executed in interpretation state and compiled in compilation state. These definitions are called *normal*.

An *immediate* word is one that is also found in both states, but in contrast to normal words, it is being executed in compilation state as well. Examples for these types of definitions are `cast` and `ignore`, as well as all words that add a vocabulary to the search order, like `forth` and `assembler`.

execute-only words are being executed in interpretation state. In compilation state, they are invisible to `search`. Definitions of this type are usually overloaded versions of definitions marked as *compile-only*. In StrongForth, state-smart words can be avoided by defining separate overloaded versions for interpretation state and compilations state, and assigning one of them the type *execute-only* and the other one the type *compile-only*. *compile-only* words are executed in compilation state and are invisible in interpretation state. Examples of words with separate versions for each state are `."`, `new` and `to`.

Words that have a compilation semantics, but no interpretation semantics, like `if` or `;`, are *compile-only*. They are not *immediate*, because it makes no sense executing them in interpretation state.

To make a word immediate, execute-only or compile-only, one of the following words can be used after the definition is complete:

```
: immediate ( -- )
  [ 16 bit ] literal latest attributes! ;

: execute-only ( -- )
  [ 17 bit ] literal latest attributes! ;

: compile-only ( -- )
  [ 16 bit 17 bit or ] literal latest attributes! ;
```

Additionally, bit 16 of the attributes can be queried with `immediate?`. `immediate?` returns true for *immediate* and *compile-only* words, and false for *normal* and *execute-only* words:

```
: immediate? ( definition -- flag )
  [ 16 bit ] literal swap attributes? ;
```

The type of a definition can be displayed with `.attributes`:

```
: .attributes ( definition -- )
  [ 17 bit ] literal over attributes?
  if immediate?
    if [' ] compile-only
    else [' ] execute-only
    then .name
  else immediate?
    if [' ] immediate .name
    then
  then ;
```

Using `.name`, `.params` and `.attributes`, a version of `.` for objects of data type definition can be defined:

```
: . ( definition -- )
  dup
  if dup .name ." ( "
    dup input-params .params ." -- "
    dup output-params .params ." ) "
    .attributes
  else drop
  then ;
```

Classes code-definition and colon-definition

In StrongForth, a code definition is a definition whose execution semantics is specified by a sequence of machine code instructions. Code definitions are compiled either as a subroutine call or by inlining the machine code instructions.

Class code-definition is a child of the generic class definition:

```
dt definition procreates code-definition
class code-definition
  protected definitions
  null unsigned cmember #inline
  null token member 'token

  : token! ( code-definition -- )
    code-space here cast token swap 'token ! ;

  forth definitions

  : code-definition ( code-definition -- 1st )
    definition dup token! ;

  : code-definition ( caddress -> character unsigned
    code-definition -- 4 th )
    definition dup token! ;

  : code-definition ( virtual-definition token
    code-definition -- 3rd )
    locals( this ) erase
    this 'token ! 'attributes @ 'attributes ! this ;

  : inline! ( code-definition -- )
    code-space here over 'token @ cast address - cast unsigned
    swap #inline ! ;

  : inline? ( code-definition -- flag )
    #inline @ 0<> ;

  code ?alias ( code-definition -- )
    \ ...
    endcode

  : enddef ( code-definition -- )
    enddef ;
```

```

:noname ( code-definition -- token )
  'token @ ; is token

:noname ( compiler-workspace code-definition -- )
  locals( this ) drop assembler 'token @ #inline @
  if cast caddress -> single dup #inline @ + swap
    ?do i @ db,
      loop
  else cast address call,
  then ignore ; is (compile)
endclass

```

The class has two data members in addition to those of its parent class. 'token contains the execution token of the code definition, which is actually the starting address of its sequence of machine code instructions. #inline is a character-size member. If it is zero, the code definition will be compiled as a subroutine call. Otherwise, its machine code will be inlined. In this case, the value of #inline is the number of bytes of machine code that will be compiled.

The first two constructors of class code-definition extend the semantics of the respective constructors of class definition. In addition, they initialize the execution token with the current code space pointer, using the protected method token!. Machine code instructions will subsequently be compiled starting at this address.

The third constructor is a special version to be used in the definition of [bind]. It creates a temporary code definition with no name that is not linked within a dictionary.

#inline is initialized with zero. To instruct the compiler to inline the machine code instructions of a code definition, inline! stores the number of machine code instructions in #inline. It is recommended not to use inline! directly, but rather use the word inline at the end of the latest definition. inline calls inline!, and additionally ensures that the latest definition is indeed a code definition:

```

: inline ( -- )
  latest token
  if latest cast code-definition inline!
  else -292 throw
  then ;

```

inline? can be used to find out whether a code definition is to be inlined or not by the compiler.

?alias helps the compiler performing a simple optimization. If the machine code of a code definition contains nothing else but a subroutine call and a return from subroutine instruction, ?alias discards this machine code and replaces the execution token of the code definition with the one of the called subroutine. Consider the definition of . for objects of data type character:

```

: . ( character -- )
  emit ;

```

Since ?alias is automatically executed as a part of ., you don't have to bother about this optimization.

But why does class code-definition need a specific overloaded version of enddef, which has the same semantics as the version for objects of its parent class? The answer is that the new version performs some additional, hidden optimizations for code definitions. Actually, the definition of enddef within the class definition is more complex than it looks.

Two of the virtual methods of class definition need to be extended for class `code-definition`. `token` returns the execution token instead of just null. `(compile)` either compiles a subroutine call or it inlines all bytes of the machine code, depending on the content of `#inline`. `db`, and `call`, are words from the assembler vocabulary.

Using the constructors of class `code-definition`, new code definitions can easily be defined. Here's the definition of `code`:

```
: code ( -- code-definition )
  parse-name new code-definition [compile] assembler ;
```

With `assembler`, which is an immediate word, `code` adds the assembler vocabulary to the search order, so that subsequent assembly code can be compiled.

Now, what about ordinary colon definitions? Although class `colon-definition` is directly derived from class `code-definition`, it does not have a separate class definition. This means, all members and methods, including virtual methods, are identical to those belonging to class `code-definition`.

`dt code-definition` procreates `colon-definition`

Having a separate class for colon definitions helps keeping syntax rules. Words like `;`, `code` and `does>`, and also `locals` (`,` should only be used within a colon definition. The definitions of `:` and `:noname` look pretty simple, because most of the work is done in the constructors.

```
: begin-compilation ( -- )
  ] dt-init 0 #locals ! ;

: : ( -- colon-definition )
  parse-name new code-definition cast colon-definition
  begin-compilation ;

: :noname ( -- colon-definition colon-definition )
  new code-definition cast colon-definition dup
  begin-compilation ;
```

`begin-compilation` is used at the end of both words. It switches to compilation state, clears the compiler data type heap and resets the number of locals.

Now, what happens if a colon definition has a stack diagram with at least one input parameter? Starting compilation with an empty compiler data type heap is not correct in those cases. Because the stack diagram is usually provided before compilation begins, `)` is the right place to initialize the compiler data type heap with the input parameters of the definition. However, this applies only to colon definitions. Therefore, `)` has to be overloaded with a special version for colon definitions:

```
: ) ( colon-definition stack-diagram -- 1st )
  ) dup params>dt ;
```

Unfortunately, the task of copying the input parameters to the compiler data type heap is not as simple as it appears. That's because a stack diagram might contain data type references that need to be resolved. Even worse, data type reference may be used recursively, like in this example:

```
( unsigned address -> address -> 1st 3rd - 2nd )
```

Yes, this stack diagram complies to all syntactic rules. It might even make sense to specify such a stack diagram. The compiler data type heap at the beginning of compilation would look like this:

```
unsigned address -> address -> unsigned address -> unsigned
```

Naturally, it requires a recursive algorithm to resolve the stack diagram:


```

: (params>dt) ( data-type address -> data-type -- )
  locals( params ) dup reference?
  if offset 1- params swap +
    begin dup @ dup params recurse prefix?
    while 1+
      repeat drop
    else >dt
  then ;

: params>dt ( definition -- )
  input-params 0
  ?do dup i + @ over (params>dt)
  loop drop ;

```

`params>dt` itself is a simple loop that iteratively deals with all basic data types in the input parameter list. The recursion is hidden in `(params>dt)`, whose second parameter is the address of the input parameter list. If a basic data type is a data type reference, it calculates the address of the reference and then handles all or the tail of the compound data type this address points to. The recursion is necessary because `(params>dt)` might encounter another data type reference.

At the end of a definition, we usually have `;`:

```

: exit ( -- )
  latest ?congruent
  #locals @ assembler exit, ignore dt-lock ; compile-only

: end-compilation ( -- )
  [compile] [ forget-locals ;

: ; ( colon-definition -- )
  dt-here if [compile] exit then
  end-compilation dup ?alias enddef ; compile-only

```

Let's begin with the definition of `exit`. This word can be used anywhere within a colon definition, even within a conditional clause or a loop. Since control-flow words leave data objects on the stack, `exit` cannot rely on `colon-definition` being on top of the stack. That's why `exit` does not have input or output parameters. But with `latest`, it still can get access to the definition that is presently being compiled. `exit` uses `?congruent` to check whether the contents of the compiler data type heap exactly matches the output parameters of the stack diagram of this definition. The machine code that returns to the calling definition is being compiled by `exit`,. This code depends on whether the definition has locals or not. Finally `exit` locks the compiler data type heap, because the machine code following the return to the calling definition is inaccessible, unless it is the destination of a conditional or unconditional jump instruction. With the compiler data type heap locked, no further data processing is allowed. As a consequence, you cannot compile any words with input and/or output parameters immediately after compiling `exit`. This makes no sense anyway, because these words would never be executed. Compilation can only continue after the compiler data type heap has been unlocked again.

`;` executes `exit` only if the compiler data type heap is not locked. Using `end-compilation`, it then returns to interpretation state and deletes all locals that have been used within the definition. the semantics of `?alias`, which is a member of class `code-definition`, has already been explained. `enddef`, at the end of `;`, adds the newly compiled definition to the current compilation vocabulary.

Literal definitions

Words that return single-cell, double-cell or floating-point literals could have been defined as created definitions, as described in the next section of this chapter. However, it is more efficient to implement these words based on classes with specialized versions of `(compile)` that allow a number of code optimizations. Let's start with the class for definitions that compile single-cell literals:

```
dt definition procreates single-definition
class single-definition
  protected definitions
  null single member 'value
  forth definitions
  : single-definition ( single-definition -- 1st )
    definition ;
  : single-definition ( caddress -> character unsigned
    single-definition -- 4 th )
    definition ;
  : assign ( single single-definition -- )
    'value ! ;
  :noname ( compiler-workspace single-definition -- )
    locals( this ) drop
    'value @ output-params drop @ (literal) ; is (compile)
endclass
```

Class `single-definition` contains a protected member called `'value`, which stores the value of the single-cell literal. The constructors are identical to those of the parent class. `assign` can be used to assign a value to an object of class `single-definition`.

`(compile)` directly uses `(literal)` to compile the single-cell literal. The machine code generally consists of a single instruction. In cases where the literal can be embedded in a succeeding load or store machine code instruction, even this single instruction can be saved. `(literal)` is similar to `(compile)`, but instead of compiling the semantics of a definition, it compiles a literal. It has three overloaded versions:

```
(literal) ( single data-type -- )
(literal) ( double data-type -- )
(literal) ( float data-type -- )
```

The data type of the literal has to be specified in order to allow the compiler to perform code optimizations.

To define words like `constant` and `variable` based on class `single-definition`, we need a number of words that help to implicitly construct the stack diagrams. You already know `param,`, a member of class `stack-diagram` which adds a basic data type to a stack diagram. `params,` adds a compound data type to a stack diagram, which is located at a given address in memory, clearing all attributes except for `dt-prefix` and `dt-reference`:

```
: params, ( address -> data-type stack-diagram -- )
  begin over @ [ dt-prefix dt-reference or ] literal and
    over param, over @ prefix?
  while swap 1+ swap
  repeat drop drop ;
```

Next, an overloaded version of `params!` constructs a stack diagram whose output parameter is a given compound data type. This stack diagram is then assigned to a definition:

```
: params! ( address -> data-type definition -- )
  swap state @ new stack-diagram --
  tuck params, swap params! ;
```

Another overloaded version of `params!` does the same, but adds a basic data type, typically an address, as the head of the compound data type:

```
: params! ( data-type address -> data-type definition -- )
  swap rot dt-prefix or state @ new stack-diagram --
  tuck param, tuck params, swap params! ;
```

Now we can define constant for single-cell values:

```
: constant ( single -- )
  parse-name new single-definition tuck assign
  dt-here over params! enddef ;
```

The most interesting thing about this definition is how the new constant is assigned its stack diagram. To understand how this is done, let's view a small example:

```
42 constant answer ok
latest . answer ( -- unsigned ) ok
```

After interpreting 42, the topmost data type on the interpreter data type heap is unsigned. Next, `constant` is about to be interpreted. The interpreter finds `constant` in the dictionary, updates the interpreter data type heap according to its stack diagram and then executes `constant`. The stack diagram indicates that this word expects an item of data type `single` or any derived data type on the stack. It does not have any output parameters. Data type `unsigned` in this example is removed from the interpreter data type heap, immediately before `constant` is being executed. Removing an item from a data type heap means that the heap pointer, which always points to the next free cell on the heap, is decremented. The interpreter data type heap pointer now points to `unsigned`. This is where `constant` gets the data type from.

Note that this mechanism does not work when `constant` is being compiled. You as the programmer have to make sure that `dt-here` returns the address of the compound data type of the constant. A good example is the word `label`, which creates an assembler label, like this:

```
label main-loop
```

An assembler label is actually a constant of data type `address` with the value of the current code space pointer. Here's the definition:

```
: label ( -- )
  code-space here [ ' here output-params drop ] literal >dt
  dt-drop constant ;
```

The address of the data type is first added to the interpreter data type heap, and then immediately dropped. But during execution of `label`, it still remains at the location where `dt-here` points to.

A single-cell variable is nothing else but a constant of data type `address` pointing to a single-cell value. With `data-type` being the basic data type at the head of its compound data type, `(variable)` creates such an address constant. Its value is the current data space pointer, because

this is the location where the variable is stored. Remember that in StrongForth, every variable is being initialized. The data type of the initialization value becomes the tail of the variable's compound data type.

```
: (variable) ( data-type -- single-definition )
  parse-name new single-definition local def
  dt-here def params! data-space here def assign def ;
```

The remaining tasks are done by variables. This word creates an array of *n* values instead of a single variable. *n* is provided as the second input parameter of data type unsigned:

```
: variables ( single unsigned -- )
  data-space align [dt] address (variable) rot rot
  data-space here -> single over cells data-space allot
  swap rot fill enddef ;
```

According to the Forth 2012 specification, variables have to be cell aligned. After the variable has been created using (variable), all that remains to be done is allotting the memory area in the data space and filling it with the initialization value. To create a single variable instead of an array, variables just has to be executed with 1 as *n*:

```
: variable ( single -- )
  1 variables ;
```

Variables for double-cell values and for floating-point numbers can be created in a very similar way. variables and variable are overloaded in StrongForth. Name prefixes like in Forth 2012, which specifies 2VARIABLE and FVARIABLE, are not required:

```
: variables ( double unsigned -- )
  data-space align [dt] address (variable) rot rot
  data-space here -> double over cells 2* data-space allot
  swap rot fill enddef ;

: variable ( double -- )
  1 variables ;

: variables ( float unsigned -- )
  data-space align [dt] address (variable) rot rot
  data-space here -> float over floats data-space allot
  swap rot fill enddef ;

: variable ( float -- )
  1 variables ;
```

Note that the version of variable for double-cell items does not cover the semantics of 2VARIABLE if applied to pairs of single-cell items. For this purpose, you can use the phrase 2 variables in StrongForth. If the two single-cell variables have different data types, like character strings in the format caddress -> character unsigned, you should consider defining two separate variables instead.

StrongForth can even create arrays of character-size items. These arrays need not be cell aligned in memory.

```
: cvariables ( single unsigned -- )
  [dt] caddress (variable) rot rot
  data-space chere -> single over chars data-space allot
  swap rot fill enddef ;

: cvariable ( single -- )
  1 cvariables ;
```

And finally, StrongForth provides means to create variables and arrays of single-precision and double-precision floating-point numbers:

```
: sfvariables ( float unsigned -- )
  data-space align [dt] sfaddress (variable) rot rot
  data-space sfhere -> float
  over cells data-space allot swap rot fill
enddef ;

: sfvariable ( float -- )
  1 sfvariables ;

: dfvariables ( float unsigned -- )
  data-space align [dt] dfaddress (variable) rot rot
  data-space dfhere -> float
  over cells 2* data-space allot swap rot fill
enddef ;

: dfvariable ( float -- )
  1 dfvariables ;
```

Variables are always addresses that point to items of specific data types. And addresses fit into a single cell. Constants, on the other hand, can be double-cell items as well as floating-point numbers. To be able to handle non-single-cell constants, two more classes have to be derived from class definition:

```
dt definition procreates double-definition
class double-definition
  protected definitions
  null double member 'value
  forth definitions

  : double-definition ( double-definition -- 1st )
    definition ;

  : double-definition ( caddress -> character unsigned
    double-definition -- 4 th )
    definition ;

  : assign ( double double-definition -- )
    'value ! ;

  :noname ( compiler-workspace double-definition -- )
    locals( this ) drop
    'value @ output-params drop @ (literal) ; is (compile)
endclass

dt definition procreates float-definition
class float-definition
  protected definitions
  null float member 'value
  forth definitions

  : float-definition ( float-definition -- 1st )
    definition ;
```

```

: float-definition ( caddress -> character unsigned
  float-definition -- 4 th )
  definition ;

: assign ( float float-definition -- )
  'value ! ;

:noname ( compiler-workspace float-definition -- )
  locals( this ) drop
  'value @ output-params drop @ (literal) ; is (compile)

endclass

```

Based on these two classes, non-single-cell constants can be defined:

```

: constant ( double -- )
  parse-name new double-definition tuck assign
  dt-here over params! enddef ;

: constant ( float -- )
  parse-name new float-definition tuck assign
  dt-here over params! enddef ;

```

These two overloaded versions cover the semantics of the Forth 2012 words `2CONSTANT` and `FCONSTANT`. However, in Forth 2012, `2CONSTANT` can be alternatively used for defining a double-cell constant or a couple of single-cell constants. A word with the latter semantic is not predefined in StrongForth. The overloaded double-cell version of `constant` can only be applied to double-cell items, but not to couples of single-cell items. If you consider this as a drawback, you can define `2constant` like this:

```

: 2params! ( address -> data-type definition -- )
  swap state @ new stack-diagram --
  over over params, swap dt-next over params, swap params! ;

: 2constant ( single single -- )
  merge parse-name new double-definition
  dt-here over 2params! tuck assign enddef ;

```

Class created-definition

Another kind of definitions are those created by `create` or by defining words that execute `create`. These words have a data field, whose address they return as their execution semantics. These definitions are objects of another child of class definition:

```

dt definition procreates created-definition

class created-definition
  protected definitions

  null address member 'body
  null code-definition member 'runtime

  forth definitions

  : created-definition ( created-definition -- 1st )
    definition ;

  : created-definition ( caddress -> character unsigned
    created-definition -- 4 th )
    definition ;

```

```

: body! ( address created-definition -- )
  'body ! ;

: runtime ( created-definition -- code-definition )
  'runtime @ ;

: runtime! ( code-definition created-definition -- )
  'runtime ! ;

:noname ( created-definition -- address )
  'body @ ; is >body

:noname ( compiler-workspace created-definition -- )
  locals( this ) >body runtime 0<>
  if runtime input-params dt-stripped + @ (literal)
    runtime (compile)
  else output-params drop @ (literal) drop
  then ; is (compile)

endclass

```

Class `created-definition` has two additional protected members with respect to its parent class. `'body` contains the address of the data field. `'runtime` contains the code definition to be executed at runtime, or null if the `created` definition shall just put its data field address onto the stack at runtime.

The two constructors are identical to the constructors of class `definition`, which means that the two additional members are being initialized with null. `body!` assigns the data field address to a `created definition`. `runtime` and `runtime!` allow read and write access to `'runtime`.

Two virtual members have to be updated. `>body` returns the data field address. Because objects of class `created-definition` are the only definitions that have data fields, these definitions can easily be identified at runtime:

```

: created-definition? ( definition -- created-definition flag )
  cast created-definition dup >body 0<> ;

```

`created-definition?` returns true if and only if `>body` returns a data field address that is not null. Based on `created-definition?`, a word can be defined that throws an exception if a given definition has no data field, i. e., if it is not an object of class `created-definition`:

```

: ?created-definition ( definition -- created-definition )
  created-definition? invert if -266 throw then ;

```

These two words turn out to be useful when dealing with `created definitions` that have a non-standard semantics specified by `does>` or `;code`.

`(compile)`, the other virtual member of class `created-definition` that needs to be updated with respect to class `definition`, compiles the data field address as a literal. If the `created definition` has a specific runtime semantics, i. e., if `'runtime` contains a non-zero value, `(compile)` also compiles the runtime semantics.

Using the constructors of class `created-definition`, the definition of `create` can be kept pretty simple. `(create)` is a word that is sometimes missed in Forth 2012. It's a non-parsing version of `create` that expects the name of the word to be created as a character string on the stack:

```

: (create) ( caddress -> character unsigned -- )
  new created-definition align here over body! enddef ;

```

```
: create ( -- )
  parse-name (create) ;
```

Note that according to the Forth 2012 standard, the data field of a `created` word has to be cell aligned. Depending on which is the default memory space, the data field will be located in the data space, the code space, the stack space or any user-defined memory space.

Using `create`, `buffer:` can be defined in the same way as in Forth 2012:

```
: buffer: ( unsigned -- )
  create allot ;
```

Bear in mind that `create` does not assign a stack diagram to the `created` word. It is always necessary to specify a stack diagram, either manually or implicitly by the defining word that uses `create`. Failing to provide the proper stack diagram will result in StrongForth's data type system becoming corrupted the first time the new word is being executed:

```
c/l chars buffer: pad2 ok
latest . pad2 ( -- ) ok
( -- caddress -> character ) ok
latest . pad2 ( -- caddress -> character ) ok
```

A more elegant way to define a character buffer is to use `cvariables`:

```
b1 c/l cvariables pad2 ok
```

You've already seen a similar example in chapter 4 about memory access. `cvariables` implicitly assigns the correct stack diagram to `pad2`, because it knows `pad2` has to return an item of data type `caddress` that points to an item of the same data type as `b1`. Manually specifying a stack diagram is not required.

Finally, here's an example of how `create` can be used to define an array of constant numbers:

```
create days-per-month ( -- caddress -> unsigned ) ok
31 c, 28 c, 31 c, 30 c, 31 c, 30 c, ok
31 c, 31 c, 30 c, 31 c, 30 c, 31 c, ok
days-per-month 1+ @ . \ February 28 ok
days-per-month 8 + @ . \ September 30 ok
```

Modifying the semantics of `created` words with `does>` and `;` code requires some more effort, especially for implicitly generating the stack diagrams. First of all, we need a number of new words, some of them overloaded, that add input and/or output parameters to objects of class `stack-diagram`:

```
: params, ( address -> data-type unsigned stack-diagram -- )
  locals( sd ) 0
  ?do dup @ [ dt-prefix dt-reference or ] literal and sd param, 1+
  loop drop ;

: params, ( definition stack-diagram -- )
  over over swap input-params rot params,
  -- swap output-params rot params, ;

: params-alias, ( address -> data-type unsigned stack-diagram -- )
  locals( sd ) 0
  ?do dup @ sd param, 1+
  loop drop ;

: params-alias, ( definition stack-diagram -- )
  over over swap input-params rot params-alias,
  -- swap output-params rot params-alias, ;
```



```
: params-stripped, ( definition stack-diagram -- )
  over over swap input-params dt-stripped rot params-alias,
  -- swap output-params rot params-alias, ;
```

The first version of `params,` is almost the same as the version presented in the previous section about literal definitions. Instead of adding one compound data type to an object of class `stack-diagram`, it adds a given number of basic data types. This word is used in the next version of `params,,` which adds the complete stack diagram of a definition to an object of class `stack-diagram`.

Both new instances of `params,` hide attributes other than `dt-prefix` and `dt-reference`. If you need to copy all attributes of all data types, you have to use the respective version of `params-alias` instead.

`params-stripped,` does the same as `params-alias,,` but it omits the last input parameter. You'll see in an instance, that this is the word required for constructing the stack diagram of created definitions.

Up to now, we already have three versions of `params!`, which are used to construct stack diagrams for generic definitions as well as for constants and variables. This one for `created` definitions takes the stack diagram of a code or colon definition as a sample, omitting the last input parameter:

```
: params! ( code-definition created-definition -- )
  state @ new stack-diagram rot over params-stripped,
  swap params! ;
```

Why is it necessary to omit the last input parameter? In order to answer this question, let's have a look at a simple defining word:

```
: :n+ ( integer -- )
  create , does> ( integer address -> integer -- 1st ) @ + ; ok
3 :n+ 3+ ok
5 3+ . 8 ok
latest . 3+ ( integer -- 1st ) ok
latest prev . ( integer address -> integer -- 1st ) ok
latest prev prev . :n+ ( integer -- ) ok
```

Here we have three words. The first one, `:n+`, is the defining word. Its parameter is being compiled into the data space of the `created` word. The immediate word `does>` ends the defining word and starts the runtime code shared by all words created by `:n+`. The runtime code has no name, but it is added to the vocabulary. Now, the most interesting thing about this example is the stack diagram of the runtime code. The last input parameter, `address -> integer`, is the data field address of the `created` word. The stack diagram of the `created` word is in turn is derived from the one of the runtime code by omitting this parameter.

Here's the definition of `does>`:

```
: does> ( colon-definition -- 1st )
  [ dt colon-definition vtable size ] literal allocate
  tuck cast colon-definition [literal]
  [''] (does) compile, [compile] ;
-> code-definition new code-definition cast colon-definition
begin-compilation ; compile-only
```

The code compiled by the `compile-only` word `[literal]` compiles a literal with its compile-time data type, in this case an item of data type `colon-definition`. `[literal]` will be presented in chapter 14.

You can see that `does>` is marked as `compile-only`, not as `immediate`. This means, it actually behaves like an immediate word in compilation mode, but the interpreter will not find it in interpretation mode. It begins allocating an object of class `colon-definition` for the runtime code and then compiles this object as a literal. At runtime of the defining word, this literal is passed to `(does)`, before the defining word ends. The last two lines of `does>` look similar to the code of `:noname.`, because it starts the definition of the runtime code. The main difference is that `new` is provided with the address of a `code-definition` object, which causes `new` to use this object instead of allocating a new one from dynamic memory.

Now, what is `(does)` doing? Its main task is assigning the `created` definition, which is the latest definition, a runtime code. If the `created` definition does not yet have a stack diagram, it constructs one from the runtime code using the appropriate version of `params!`. But before it is allowed to do so, the stack diagram of the runtime code has to pass two checks. It needs to have at least one input parameter, and none of its output parameters may contain a reference to the last input parameter. Otherwise, the stack diagram of the `created` definition would be invalid. For example, something like

```
... does> ( logical caddress -> signed -- 3rd ) ...
```

is not allowed, because the resulting stack diagram of the defined words would be invalid:

```
( logical -- 3rd ) \ invalid!
```

These are the definitions of `(does)` and the two checks:

```
: ?input-params ( definition -- )
  input-params 0= if -262 throw then drop ;

: ?references ( definition -- )
  dup output-params rot input-params dt-stripped nip rot rot 0
  ?do over over @ offset < if -261 throw then 1+
  loop drop drop ;

: (does) ( code-definition -- )
  latest ?created-definition dup input-params drop 0=
  if over ?input-params
    over ?references
    over over params!
  then runtime! ;

' ?input-params delete
' ?references delete
```

The two checks are so special that they cannot be used for other purposes than within `(does)`. That's why they are deleted after being used once. The deletion applies to their definitions, not to their machine code instructions.

The definition of `;code` looks quite similar to the definition of `does>`:

```
: ;code ( colon-definition -- code-definition )
  [ dt code-definition vtable size ] literal allocate
  tuck cast code-definition [literal]
  ['] (does) compile, [compile] ;
  -> code-definition new code-definition
  [compile] assembler ; compile-only
```

Instead of starting compilation, `;code` adds the assembler vocabulary to the list of context vocabularies.

Defining Words with Variable Parameters

We're not yet done with `(does)`. A conditional clause within the definition of `(does)` prevents the creation of the stack diagram of the created definition if it already has a stack diagram. This feature makes it possible to specify stack diagrams to the new definition, that deviate from the stack diagram of the runtime part of the defining word. What's that good for? Well, think about a word like `constant`:

```
700 constant int    ok
latest . int ( -- unsigned )    ok
char B constant letter    ok
latest . letter ( -- character )    ok
```

We could define `constant` with `create ... does>` like this:

```
: constant ( single -- )
  create , does> ( address -> single -- single ) @ ;    ok
700 constant int    ok
latest . int ( -- single )    ok
char b constant letter    ok
latest . letter ( -- single )    ok
```

You certainly see the problem. This version of `constant` always defines words with an output parameter of data type `single`. We'd need separate versions of `constant` for each data type it will be used for, which is definitely not a good solution. So, let's give it another try:

```
: constant ( single -- )
  create , dt-here latest params!
  does> ( address -> single -- single ) @ ;    ok
700 constant int    ok
latest . int ( -- unsigned )    ok
char B constant letter    ok
latest . letter ( -- character )    ok
```

, That's the way we want to have it: one version of `constant` for all single-cell data types. The stack diagram of the defined word is compiled between `create` and `does>`. `(does)` detects that the stack diagram is already present and skips the compilation of the stack diagram from the sample given by the stack diagram of the runtime part of the defining word.

However, in StrongForth `constant` is not implemented as a defining word. It rather creates definitions of class `single-definition`, which produce more efficient code than the alternative definition above.

Class definition has quite a number of other child classes. In addition to those presented in this chapter, StrongForth provides children of class `definition` for

- values,
- locals,
- virtual methods,
- class members and
- deferred words.

They will be presented later, in connection with the respective defining words.

11 Vocabularies

Why not *Word Lists*?

In Forth 2012, lists of words are called *word lists*. Why does StrongForth use a different term for it? Because a *vocabulary* is more than a simple word list. It is a set of rules that specify how to build names that can be found by the interpreter and the compiler, together with a definition for each valid name. A vocabulary can simply consist of one rule: *A valid name is one that is included in a given list*. This list could also be contained in a file. Or, a set of rules could specify how integer numbers look like. Or, floating-point numbers.

So, a vocabulary is actually a generalization of a word list. In StrongForth, integer and floating-point numbers are indeed vocabularies. It is thus not necessary to handle numbers in a special way. You can even define your own kind of syntax for numbers if you're not satisfied with what the predefined vocabularies provide. For example, you could make StrongForth correctly interpret roman numbers, if that was useful for whatever reason. What about fixed-point decimal numbers? No problem. Just invent a suitable format for representing those kinds of numbers and then define a vocabulary that can handle them.

StrongForth has eight predefined vocabularies:

- `forth`
- `assembler`
- `msvcrt`
- `protected`
- `private`
- `locals`
- `integer-lit`
- `float-lit`

The first six in this list are word lists. The `forth` vocabulary contains most of the commonly used words. `assembler` is a vocabulary for words that compile machine code instructions and their addressing modes. It is added to the search order by `code` and removed from the search order by `endcode`. `msvcrt` is a vocabulary for the functions of the *Microsoft Visual C Runtime Library*, which handles the operating system interface. These words are considered being low-level and will typically not be used directly in StrongForth. However, several higher-level words are based on them. Vocabularies `protected` and `private` are only used within class definitions for words that shall not be visible outside the class definition. The `locals` vocabulary contains locals, loop indexes and other temporary words that behave like locals and whose scope is limited to the definition of a word.

`integer-lit` and `float-lit` are not simple word lists. They provide search algorithms that interpret an integer or floating-point number according to rules given by the Forth 2012 standard and, if successful, return a definition that can compile these numbers as literals. You'll see at the end of this chapter how it works in detail.

Class vocabulary

In StrongForth, a vocabulary is represented by an object of class vocabulary:

```
dt object procreates vocabulary
class vocabulary
  forth definitions
  virtual search ( caddress -> character unsigned
    single search-criterion vocabulary -- definition flag )
  private definitions
  null definition member 'last-definition
  null vocabulary member 'next-vocabulary
  forth definitions
  : last! ( definition vocabulary -- )
    'last-definition ! ;
  : next! ( vocabulary vocabulary -- )
    'next-vocabulary ! ;
  : last ( vocabulary -- definition )
    'last-definition @ ;
  : next ( vocabulary -- 1st )
    'next-vocabulary @ ;
  private definitions
  : (ignore) ( vocabulary address -> vocabulary -- flag )
    over over @ =
    if swap next swap ! true
    else @
      begin dup 0<>
      while over over next dup rot <>
        while nip
          repeat next swap next! drop true
        else drop drop false
      then
    then ;
  forth definitions
  : vocabulary ( vocabulary -- 1st )
    locals( this ) hidden @ next! this hidden !
    null definition last! this ;
  : >context ( vocabulary -- )
    dup context (ignore) over hidden (ignore) or
    if context @ over next! context !
    else drop
    then ;
```

```

:noname ( caddress -> character unsigned single
  search-criterion vocabulary -- definition flag )
  locals( c-addr u param criterion this ) last
  begin dup 0<>
  while u
    if dup name c-addr u compare 0=
    else true
    then
    if dup param criterion execute if true exit then
    then prev
  repeat false ; is search
  ' nodelete is delete
endclass

```

Class vocabulary has two private data members. 'last-definition contains the definition that has most recently been added to the vocabulary. Starting with this definition, you can step by step traverse downwards to the vocabulary's first definition by repeatedly applying prev, a method of class definition.

'next-vocabulary, the second private member of class vocabulary, builds a linked list of vocabularies. There are actually two such vocabulary lists: The context vocabulary list and the hidden vocabulary list. The vocabularies at the head of each of these lists are stored in the system variables context and hidden, respectively. A third system variable contains the current compilation vocabulary, which can be changed with definitions:

```

null vocabulary variable context
null vocabulary variable hidden
null vocabulary variable current

: definitions ( -- )
  context @ current ! ;

```

Each vocabulary is included in either the context vocabulary list or the hidden vocabulary list, but not in both. The context vocabulary list specifies the search order. The interpreter and the compiler search for words only in those vocabularies that are included in the context vocabulary list.

To access 'last-definition and 'next-vocabulary outside the scope of the class definition, you have to use last!, next!, last and next.

The implementation of words demonstrates the method of traversing the definitions in a vocabulary:

```

: words ( -- )
  context @ last parse-name locals( addr count )
  begin dup 0<>
  while count
    if dup name addr count compare 0= else true then
    if dup cr . then prev
  repeat drop ;

```

StrongForth's version of words displays each word on a separate line, including their stack diagrams. To limit the length of the output, words optionally parses the name of the words to be displayed. This feature is pretty useful if you want to see all overloaded versions of a word and their order within the vocabulary:

```

words 2/
2/ ( signed-double -- 1st )
2/ ( signed -- 1st )
2/ ( integer-double -- 1st )
2/ ( integer -- 1st ) ok

```

The constructor of class `vocabulary` creates an empty vocabulary and adds it to the hidden vocabulary list. `>context` moves a vocabulary from either vocabulary list to the head of the context vocabulary list. To remove a vocabulary from a vocabulary list, `>context` uses the private method `(ignore)`, which expects the vocabulary to be removed and a vocabulary list on the stack. It returns `true`, if the vocabulary was in this vocabulary list and has been successfully removed, otherwise `false`. Because `(ignore)` is private, it cannot be used outside the scope of the class definition. Use `ignore` to remove the head of the context vocabulary list and add it to the hidden vocabulary list:

```

: ignore ( -- )
  context @ 0<>
  if context @ dup next context !
    hidden @ over next! hidden !
  then ; immediate

```

`ignore` is the replacement for the Forth 2012 word `PREVIOUS`. Note that `ignore` is an immediate definition. If it is used during compilation, the context vocabulary list will be immediately changed, just like during interpretation.

`forth`, `assembler` etc. are also immediate words. These words are actually created by the defining word `wordlist`. Their data field is the associated vocabulary, and their semantics is `>context`, i. e. they add their vocabulary to the head of the context vocabulary list:

```

wordlist forth immediate
wordlist assembler immediate
wordlist msvcrt immediate
wordlist protected immediate
wordlist private immediate
wordlist locals immediate

```

Here's the definition of `wordlist`:

```

: does-vocabulary ( -- )
  does> ( vocabulary -- ) >context ;

: wordlist ( -- )
  create new vocabulary cast address
  latest ?created-definition body! does-vocabulary ;

```

It is important to understand that the words created by `wordlist` do not return the associated vocabulary. To get access to the vocabularies, you have to obtain the data field. Only a few vocabularies are predefined constants:

```

' locals >body cast vocabulary constant locals-vocabulary
' private >body cast vocabulary constant private-vocabulary
' protected >body cast vocabulary constant protected-vocabulary

```

Objects of class `vocabulary` cannot be deleted.

Searching Vocabularies

The most interesting method of class vocabulary is the virtual method `search`. It expects the name of the word to be searched in the vocabulary as a character string `caddress -> character unsigned`. A null string as the name means that any name is considered being a match. It is important to note that `search` is case sensitive. Because the names of StrongForth words are usually lower case, words written in upper case will not be found by `search`. If you want StrongForth to accept words written in upper case, you have to assign an appropriate version of `search` to the virtual method.

Because a simple match on the name of a word is insufficient in most cases, StrongForth needs a functionality to apply additional search criteria. For example, the interpreter and the compiler have to find a word whose name matches the parsed name *and* whose stack diagram fits to the contents of the data type heap. This need is satisfied by adding two parameters, `single` and `search-criterion`. `search-criterion` is the execution token of a callback word that tells `search` whether a particular word in the vocabulary satisfies the additional criterion. `single` is an optional parameter to this callback word. Specifying the very general data type `single` ensures that `search` accepts different kinds of parameters to be used, as long as they fit into a single-cell item. `search` returns the definition that matches the name and the additional search criterion, and a flag indicating whether a definition has been found. If the flag is `false`, the definition returned is the null definition. It is even possible to search a vocabulary for a word that only matches the additional search criterion, i. e., without considering the name of the word. To initiate such a search, you simply have to specify an empty string (a string with zero length) for the name of the word.

Data type `search-criterion` is a child of data type `token`. It is called a *qualified token*, because it has its own version of `execute` that can be applied to execution tokens that belong to a word with a specific stack diagram:

```
execute ( definition single search-criterion -- flag )
```

If you just want to find the first occurrence of a word with a given name, you should specify `no-criterion` as the parameter of data type `search-criterion`, and an arbitrary value, e. g. zero, as the optional parameter of data type `single`:

```
:noname ( definition single -- flag )
  drop drop true ;
token cast search-criterion constant no-criterion
```

Executing `no-criterion` returns a `true` flag, which means that the additional search criterion is always met. All callback words for `search` need to have the same stack diagram. A detailed description of the search criterion the interpreter and the compiler use to find the word matching the contents of the data type heap will be presented in the next section.

Forth 2012 specifies the word `FIND` for searching the dictionary for the latest occurrence of a word with a given name:

```
FIND ( c-addr -- c-addr 0 | xt 1 | xt -1 )
```

This word does not fit well into StrongForth. First, StrongForth requires that stack diagrams are unique at compilation time. Alternative stack diagrams are not allowed. In Forth 2012, the stack diagram of `FIND` depends on the search result, which is unknown at compile time. Second, StrongForth has abandoned counted strings, i. e. strings with a leading length byte in memory. Strings should rather be passed as a pointer to the first character and a separate parameter for the character count: `caddress -> character unsigned`

However, in the *Search-Order* word set Forth 2012 offers an alternative to FIND:

```
SEARCH-WORDLIST ( c-addr u wid -- 0 | xt 1 | xt -1 )
```

This word does not expect a counted string. The ambiguity of the stack diagram could be fixed by adding a dummy `xt` as an additional output parameter in the case of an unsuccessful search.

This is still not what we need in StrongForth. But it's pretty near. Because bare execution tokens are not useful in StrongForth, it makes more sense to return an item of data type `definition` instead of one of data type `token`. The execution token of a code or colon definition can be obtained with the virtual method `token`. By applying `immediate?` to an object of class `definition`, you can query whether the definition is immediate or not. Therefore, it is sufficient to return a flag instead of 0, 1 or -1. And finally, by adding the two input parameters for additional search criteria, we're at the virtual method `search` of class `vocabulary`:

```
virtual search ( caddress -> character unsigned
  single search-criterion vocabulary -- definition flag )
```

Note that `search` is overloaded with the Forth 2012 word for searching substrings within strings. The suffix in `SEARCH-WORDLIST` has been omitted. Because of the different stack diagrams interpreter and compiler will never fail to cleanly distinguish between searching a vocabulary and searching a string.

Other than the Forth 2012 word `FIND`, StrongForth's `search` only searches one specific vocabulary, just like `SEARCH-WORDLIST`. To search all vocabularies of the context vocabulary list in the given order, you have to use `search-context`:

```
: search-list ( caddress -> character unsigned
  single search-criterion vocabulary -- definition flag )
  locals( c-addr u param criterion voc ) voc
  begin dup 0<>
  while dup c-addr u rot param criterion rot search invert
    while drop next
    repeat nip true
  else drop null definition false
  then ;

: search-context ( caddress -> character unsigned
  single search-criterion -- definition flag )
  context @ search-list ;
```

In some cases it is necessary to search *all* available vocabularies instead of only those in the search order. `search-all` searches first the context vocabulary list and then the hidden vocabulary list:

```
: search-all ( caddress -> character unsigned
  single search-criterion -- definition flag )
  locals( c-addr u param criterion )
  c-addr u param criterion search-context
  if true
  else drop c-addr u param criterion hidden @ search-list
  then ;
```

A simple application of `search-context` is `'`. This word requires no additional search criterion, because it is supposed to return the first definition with the parsed name it finds in the context vocabulary list:

```
: ' ( -- definition )
  parse-name null single no-criterion search-context
  invert if -13 throw then ;
```

`search` can also be viewed as special version of the Forth 2012 word `TRAVERSE-WORDLIST`. Like `search` with the parameter of data type `search-criterion`, `TRAVERSE-WORDLIST` expects the execution token of a word to be executed for each word in the wordlist. This word expects a *name token*, which is actually an object of class `definition` in `StrongForth`, and returns a flag to determine whether the search shall be continued. `TRAVERSE-WORDLIST` can not be implemented one-to-one in `StrongForth`, because it has an ambiguous stack diagram.

Matching Rules

When trying to find a word in the dictionary, the interpreter and the compiler do not only consider the name of the word. Since the concept of operator overloading allows to distinguish equally named words by the data types they are applied to, the stack diagrams become part of the matching rules. Note that Forth 2012 allows redefining words as well, but once a new version has been defined, the old version can no longer be found in the dictionary. In `StrongForth`, previously defined versions stay alive, as long as they can be distinguished by their stack diagrams from versions defined later.

The search algorithm that is applied when interpreting or compiling words needs to incorporate pretty complex matching rules. These matching rules implement `StrongForth`'s data type mechanism. Generally, interpreter and compiler try to find a word with the given name that can be applied to the items on the data stack. If, for example, the item on top of the stack has data type `unsigned`, a word whose stack diagram has an item of data type `flag` as it's last input parameter won't match. But what about a word whose stack diagram has only one input parameter of data type `integer`? This word matches, because it can be applied to data type `integer` as well as all of its subtypes.

So what are the exact matching rules? First of all, the data stack has to contain at least as many items as the word has input parameters. Next, the data type of each input parameter is compared with the data type of the corresponding item on the data type heap. Four different cases have to be considered:

Case	Input parameter	Data type heap
1	Basic data type	Basic data type
2	Basic data type	Compound data type
3	Compound data type	Basic data type
4	Compound data type	Compound data type

Case 1

If both the input parameter and the corresponding item on the data type heap are basic data types, they match if their data types are identical or if the item on the data type heap is a subtype of the input parameter. For example,

```
aligned ( address -- 1st )
```

matches if the item on top of the data type heap has data type `address`, or any of its direct or indirect subtypes, like `caddress`.

Case 2

Now, let's assume that the item on the data type heap is a compound data type, while the corresponding input parameter is a basic data type. A compound data type is always more specific than a basic data type. The input parameter matches the item on the data type heap, if the head of the compound data type, which is the leftmost of the basic data types that constitute the compound

data type, is identical to or a direct or indirect subtype of the basic data type of the input parameter. In the previous example, aligned matches even if the item on top of the data type heap is `caddress -> character`.

Case 3

On the other hand, if the input parameter has a more specific data type than the corresponding item on the data type heap, they don't match. A word that requires a compound data type as an input parameter can't be satisfied by a basic data type. For example,

```
@ ( address -> single -- 2nd )
```

won't match if the item on top of the data type heap has the basic data type `address`.

Case 4

What if both data types are compound data types? The matching rule for this case can be described by a recursion. Let

```
head1 -> tail1
```

be the compound data type of the input parameter and

```
head2 -> tail2
```

be the compound data type of the corresponding item on the data type heap. `head1` and `head2` are basic data types, while `tail1` and `tail2` may be either basic or compound data types. The input parameter matches the item on the data type heap if both of the following two conditions are met:

- `head2` is identical to `head1`, or `head2` is a direct or indirect subtype of `head1`
- `tail1` and `tail2` match according to the rules described by cases 1 to 4.

In the above example, `@` matches if the item on the data type heap is, for example, `caddress -> signed` or `address -> sfaddress -> float`. The same version of `@` doesn't match if the data type of the item on top of the data type heap is `address -> unsigned-double`.

Data Type References

But still, these four cases do not cover everything. What happens if an input parameter contains a data type reference? The reference can be either a basic data type or the last basic data type of a compound data type. This means, a basic data type with a data type reference does never have the prefix attribute:

- `2nd` \ okay
- `address -> 1st` \ okay
- `3rd -> unsigned` \ wrong!

If a reference is part of a compound data type, all basic data types up to the reference are processed according to the matching rules described in cases 1 to 4. The basic data type containing the reference is then substituted by the referenced basic or compound data type on the data type heap. Any of the basic data types in the input parameter list, whose index is lower than the index of the basic data type containing the reference, can be referenced:

- `(unsigned character 1st --)` \ okay
- `(caddress -> character 1st --)` \ okay
- `(caddress -> character 2nd --)` \ okay
- `(caddress -> character 3rd --)` \ wrong!

Note that the referenced data type can be basic data type, the head of a compound data type or even the tail of a compound data type.

Although the referenced data type is part of the input parameter list, the actual match is performed against the data types of the items on the data type heap. While matching the input parameters one by one with the data types of the items on the data type heap, each basic data type in the input parameter list is assigned to a unique basic data type on the data type heap. In order for the match to succeed, the basic or compound data type on the data type heap, which is assigned to the data type reference in the input parameter list, has to be *identical* to the basic or compound data type on the data type heap, which is assigned to the referenced basic or compound data type in the input parameter list. So, subtype and prefix relationships are not considered when matching references. Again: Referenced data types have to be *identical*.

The matching rules for data type references are best explained by a number of examples.

```
! ( single address -> 1st -- )
```

matches

```
flag caddress -> flag
```

and

```
caddress -> unsigned address -> caddress -> unsigned
```

1st is a reference to the first basic data type in the input parameter list, which is single.

However, the data type is substituted by the corresponding data type of the item on the data type heap, which is flag or caddress -> unsigned. According to this rule, the following data types will not match:

- integer address -> single
- caddress -> single address -> caddress
- caddress address -> caddress -> character

The rationale of using a reference in this special case is that an item of a specific data type can only be stored in a memory location, which is specified by an address of exactly the same data type.

```
- ( address -> single 1st -- integer )
```

matches every pair of identical data types, provided the first parameter has passed the matching rule for address -> single.

The last input parameter of

```
fill ( address -> single unsigned 2nd -- )
```

is a reference to the tail of the first parameter. It matches for

```
address -> token unsigned token
```

and

```
address -> caddress -> signed unsigned caddress -> signed
```

but not for

```
address -> logical unsigned flag
```

or

```
address -> single unsigned address -> single.
```

To make things even more complex, references are allowed to be recursive. Stack diagrams like in the following example are possible, and the matching rules for data type references are applied correctly:

```
( integer caddress -> 1st address -> 2nd -- )
```

A word with this stack diagram will be found in the dictionary if the data type heap contains

```
integer caddress -> integer address -> caddress -> integer
```

or something that is based on subtypes of data types integer, caddress and address, like

```
unsigned caddress -> unsigned address -> caddress -> unsigned.
```

A Search Criterion for Matching Rules

The matching rules explained in the previous section are implemented as an additional search criterion to be used in the search virtual method of class vocabulary:

```
:noname ( definition single -- flag )
  over [ 17 bit ] literal over attributes? invert
  swap immediate? state @ = or
  if cast flag over immediate? invert or state @ and
    state @ swap state ! swap input-params
    static-compiler-workspace compiler-workspace match?
    swap state !
  else drop drop false
  then ;
token cast search-criterion constant match-criterion
```

The stack diagram of each word with the proper name that is encountered during the vocabulary search is compared with the contents of the interpreter or compiler data type heap. A data type in the stack diagram matches any of its subtypes on the data type heap, and a basic data type in the stack diagram matches compound data types on the data type heap. Even references to other data types can be provided within stack diagrams and are considered by the interpreter and the compiler. In interpretation state, the interpreter data type heap is used. In compilation state, the selection of the data type heap depends on whether the word is immediate or not. The stack diagrams of immediate words are matched against the interpreter data type heap, while those of non-immediate words are matched against the compiler data type heap.

These rules work nicely for the interpreter and the compiler, but not for words like `[compile]` and `postpone`, which also use `match-criterion` as an additional search criterion. These two words can only be executed in compilation state, because they always *compile* a word. They require a special rule to always match a stack diagram against the compiler data type heap, no matter whether the word is immediate or not. This special rule is applied if `match` is combined with `true` as the value of `single`.

Executing `match-criterion` returns a false flag if a word does not fit to the current state, given its attributes. Otherwise, `match-criterion` continues by determining the data type heap the word's stack diagram shall be compared with. The selection depends on the word's attributes, the current state and the value of the second input parameter, which may force compilation:

single	state	attributes	data type heap
false	false		interpreter
false	false	immediate	interpreter
false	false	execute-only	interpreter
false	false	compile-only	(no match)
false	true		compiler
false	true	immediate	interpreter
false	true	execute-only	(no match)
false	true	compile-only	interpreter
true	false		interpreter
true	false	immediate	interpreter
true	false	execute-only	interpreter
true	false	compile-only	(no match)
true	true		compiler
true	true	immediate	compiler
true	true	execute-only	(no match)
true	true	compile-only	compiler

match-criterion temporarily changes the value of the system variable state in order to select the data type heap. The matching algorithm is hidden in the word match?, which is a public method of a class compiler-workspace. Here's the definition of this class:

```

dt object procreates compiler-workspace
class compiler-workspace
  private definitions
  null address -> data-type member 'original-here
  null address -> data-type member 'stripped-here
  null address -> data-type /params members 'references
  null unsigned cmember >references

  : dt-set ( address -> data-type -- )
    dt-here - dt-allot ;

  : save-reference ( address -> data-type compiler-workspace -- )
    locals( this )
    >references @ /params <
    if 'references >references @ + ! 1 >references +!
    else drop
    then ;

  : get-reference ( data-type compiler-workspace
    -- address -> data-type )
    'references swap offset 1- + @ ;

  : skip-params ( address -> data-type unsigned -- flag )
    0
    ?do dup @ prefix? invert
      if dt-depth
        if dt-drop
          else drop false exit
        then
      then 1+
    loop drop true ;

```

```

: direct-match ( data-type -- flag )
  dt-here @
  begin over null data-type and over null data-type and <>
  while dup parent swap or dup d>s 0=
    until drop drop false
  else over prefix?
    if nip prefix? 1
    else drop drop true dt-here dt-length
    then dt-allot
  then ;

: reference-match ( address -> data-type -- flag )
  begin dup @ null data-type and dt-here @ null data-type and =
  while dup @ prefix?
    while dt-here @ prefix?
      while 1+ 1 dt-allot
      repeat drop false
    else drop dt-here @ prefix? invert 1 dt-allot
    then
  else drop false
  then ;

: match-all ( address -> data-type unsigned compiler-workspace
  -- flag )
  locals( this ) 0
  ?do dt-here save-reference dup @ dup reference?
    if get-reference reference-match
    else direct-match
    then invert
    if drop false exit
    then 1+
  loop drop true ;

forth definitions

: compiler-workspace ( compiler-workspace -- 1st )
  locals( this ) 0 >references !
  dt-here dup 'original-here ! 'stripped-here ! this ;

: match? ( address -> data-type unsigned compiler-workspace
  -- flag )
  locals( this ) 'stripped-here @
  if over over skip-params
    if dt-here 'stripped-here ! match-all
    else drop drop false
    then dup invert
    if 'original-here @ dt-set
    then
  else drop drop false
  then ;

endclass

```

Because StrongForth never needs more than one object of class `compiler-workspace` at the same time, a static object can be created and assigned to a constant:

```
new compiler-workspace constant static-compiler-workspace
```

`match?` and the constructor are the only publicly available methods of class `compiler-workspace`. All other methods and all data members are `private`.

`match?` uses `skip-params` to check whether the selected data type heap contains at least as many items as the word has input parameters. These data types are actually dropped from the data type heap. While `'original-here` keeps a pointer to the original data type heap, `'stripped-here` is assigned a pointer to the data type on the heap that corresponds to the word's first input parameter. The private method `match-all` then compares each basic data type of the word's input parameter list with the corresponding data type on the data type heap. Finally, the data types that have been dropped from the data type heap are restored.

During the matching algorithm, objects of class `compiler-workspace` build a list of references to the data type heap. For each basic data type in the input parameter list, it stores a pointer to the corresponding data type on the data type heap. This is necessary, because each basic data type of the input parameter list may be referenced further down the list. This is what `save-reference` does. `get-reference` obtains the pointer from the list that belongs to a given index.

`match-all` handles basic data types that are no references with `direct-match`. The phrase `null data-type` and masks all data type attributes. The match fails, if the basic data type on the data type heap is not equal to or a subtype of the basic data in the input parameter list. Finally, the data type heap pointer is advanced. If the data type in the input parameter list has no prefix attribute, the data type heap pointer is just advanced by one, otherwise the complete tail of the compound data type on the data type heap is being skipped.

A data type reference is processed completely different. `reference-match` actually compares two compound data types on the data type heap, which need to be identical. Note that data type references do not occur on the data type heap.

Both `direct-match` and `reference-match` abort the match algorithm and return a false flag as soon as a data type mismatch is found. This flag is passed on to `match-all` and further to `match?` and finally to `match-criterion`. Executing `match-criterion` returns a true flag only if the match fully succeeds.

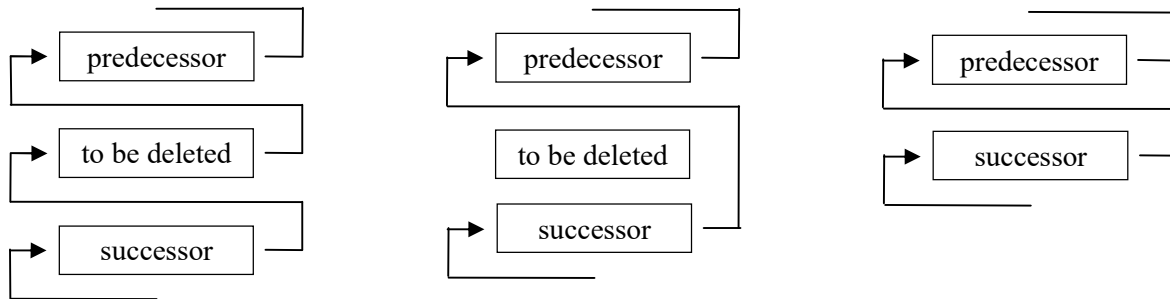
More Search Criteria

`match-criterion` is only one of several predefined search criteria, though it is definitively the most complex one. Another one is `no-criterion`, which was already presented at the beginning of this chapter. And we already did encounter another one when having a look at the `delete` virtual method of class definition:

```
:noname ( definition single -- flag )
  cast definition swap prev = ;
token cast search-criterion constant link-criterion
```

Each definition is linked to its predecessor in the vocabulary. It can be obtained with `prev`. But there is no link to the *successor* of a definition, i. e., to the definition whose predecessor in the same vocabulary is the first definition. To get the successor of a definition, we have to search the complete vocabulary. This is exactly what a vocabulary search with `link-criterion` does. The parameter of data type `single` is the definition whose successor we search for.

You already know an application for `link-criterion`. `delete` for class definition has the replace the link in the successor definition in such a way, that it points to the predecessor of the deleted definition.



Here's a second application of link-criterion:

```
: first ( vocabulary -- definition )
  null address -> character 0 rot
  null definition link-criterion rot search drop ;
```

first uses search with link-criterion to traverse a complete vocabulary down to the first definition. The first definition of a vocabulary is the only one that has a null definition stored in 'link.

token-criterion is a criterion that can be applied when searching a code definition or colon definition with a given execution token. The parameter of data type single is the execution token to be looked for:

```
:noname ( definition single -- flag )
  cast token swap token = ;
token cast search-criterion constant token-criterion
```

However, be aware that some definitions may share the same execution token, as in this example:

```
null address -> character 0 ' here token ok
token-criterion search-context . true ok
. dfhere ( memory-space -- dfaddress ) ok
```

Another search criterion is used to find a created definition with a specific runtime definition. The parameter of data type single is the runtime definition to be searched for:

```
:noname ( definition single -- flag )
  cast code-definition swap created-definition?
  if runtime =
  else drop drop false
  then ;
token cast search-criterion constant runtime-criterion
```

A typical example of how to use runtime-criterion is finding the data type with a name given by a character string address -> character unsigned:

```
: ?data-type ( caddress -> character unsigned -- data-type )
  [ ' single cast created-definition runtime ] literal
  runtime-criterion search-all
  if >body cast data-type-attributes >data-type
  else -260 throw drop null data-type
  then ;
```

?data-type searches all vocabularies, context as well as hidden, for a created definition with the given name and the same runtime code as single, which is a data type word as well. The data field of single and all other data type words contain the associated object of data type data-type-attributes. This object is finally converted into a data type.

With `?data-type`, we can now define the parsing word `dt`:

```
: dt ( -- data-type )
  parse-name ?data-type ;
```

Finally, let's have a look at `body-criterion`. This search criterion matches only with created definitions whose data field has the value specified by its parameter:

```
:noname ( definition single -- flag )
  cast address swap >body = ;
token cast search-criterion constant body-criterion
```

Definitions which have not been created do not match, because their version of `>body` always returns a null address. `body-criterion` is used by an overloaded version of `created-definition?`, which searches all vocabularies, even the hidden ones, for a created definition with a given data field:

```
: created-definition? ( address -- created-definition flag )
  dup
  if null address -> character 0 rot body-criterion
    search-all swap cast created-definition swap
  else drop null created-definition false
  then ;
```

If the requested data field is non-null and a matching definition was found, `created-definition?` returns it as an object of data type `created-definition`, together with a true flag. Otherwise, `created-definition?` returns a null definition and a false flag.

`created-definition?` is in turn used to convert data types and vocabularies to their associated definitions:

```
: >definition ( data-type -- created-definition )
  >attributes cast address created-definition? invert
  if -260 throw
  then ;

: >definition ( vocabulary -- created-definition )
  cast address created-definition? invert
  if -293 throw
  then ;
```

Both of these words throw exceptions if invalid objects of data types `data-type` or `vocabulary` are provided as input parameters, respectively. The first overloaded version of `>definition` was used in the definition of `.` for objects of data type `data-type`, which was already presented in chapter 7.

Class integer-vocabulary

Forth 2012 handles numbers in the input stream differently than words that are included in the dictionary. This makes the interpreter more complex. It first tries to find a word in one of the context vocabularies, and if it doesn't find it, it tries to interpret it as a number. StrongForth, on the other hand, provides a special kind of vocabulary for integer numbers, which may be included in the search order. Because the search algorithm for integer numbers does not simply try to find a given name in a dictionary, it needs to be embedded in a dedicated vocabulary class:

```

dt vocabulary procreates integer-vocabulary
class integer-vocabulary
  private definitions
  null character constant character-number
  null unsigned constant unsigned-number
  null signed constant signed-number
  null unsigned-double constant unsigned-double-number
  null signed-double constant signed-double-number

  forth definitions

  : integer-vocabulary ( integer-vocabulary -- 1st )
    vocabulary ;

  :noname ( caddress -> character unsigned single
    search-criterion integer-vocabulary -- definition flag )
    drop drop drop dup 3 =
    if over [char] ' over @ =
      [char] ' rot [ 2 chars ] literal + @ = and
      if drop 1+ @ [' ] character-number tuck
        cast single-definition assign true exit
      then
    then
    base @ rot rot
    null integer-double rot rot dup
    if over @
      case [char] # of /string decimal endof
        [char] $ of /string hex endof
        [char] % of /string binary endof
        \ default \ drop
      endcase
    then
    dup
    if over @ >sign dup
      if rot rot /string rot
        then
      else +0
      then
      locals( s ) dup
      if over @ digit? nip
        if >number true
        else false
        then
      else false
      then
      if dup
        if " ." compare
          if drop null definition false
          else s
            if s ?negate [' ] signed-double-number
            else [' ] unsigned-double-number
            then
            tuck cast double-definition assign true
          then
        else drop drop d>s s

```

```

        if s ?negate [''] signed-number
        else [''] unsigned-number
        then
            tuck cast single-definition assign true
        then
    else drop drop drop null definition false
    then rot base ! ; is search
endclass

```

Only one instance named `integer-lit` of class `integer-vocabulary` is required. The interpreted code to create it is similar to the definition of `wordlist`:

```

create integer-lit new integer-vocabulary
cast address latest ?created-definition body!
does-vocabulary immediate

```

Note that an attempt to apply words to `integer-lit` will display no words at all, because 'last-definition remains null:

```
integer-lit words ok
```

If the search in `integer-lit` is successful, i. e., if the string does represent a valid integer number, search returns one of the five private members of class `integer-vocabulary`, a constant of one of these data types:

- character
- unsigned
- signed
- unsigned-double
- signed-double

This is the rule for building an integer number:

```

<intnum>    := { <num> | <decnum> | <hexnum> | <binnum> | <cnum> }
<num>       := [<sign>]<digit><digit>*[.]
<decnum>    := # [<sign>]<decdigit><decdigit>*[.]
<hexnum>    := $ [<sign>]<hexdigit><hexdigit>*[.]
<binnum>    := % [<sign>]<bindigit><bindigit>*[.]
<cnum>       := '<char>'
<sign>      := { + | - }
<bindigit>  := { 0 | 1 }
<decdigit>  := { 0-9 }
<hexdigit>  := { { 0-9 } | { A-F } | { a-f } }

```

<digit> represents a digit according to the value of base. <char> represents any printable character. A number in the format <cnum> has data type character. The data type of numbers in all other formats depend on whether a leading sign and/or a trailing dot is present:

Leading sign	Trailing dot	Data type	Example
no	no	unsigned	453009462
no	yes	unsigned-double	9164150249955427.
yes	no	signed	+710054145
yes	yes	signed-double	-2091540334031891.

Now let's have a closer view at the implementation of `search`. The last three input parameters can be dropped, because they are not required. If the character string has the format '<char>', `search` returns `character-number` with the value of the character assigned to it. Otherwise,

it saves the current number-conversion radix base and prepares the number conversion by creating an item of data type integer-double. `search` then tries to detect one of the number prefixes #, \$ and %. If one of them is present, the character is skipped and the number-conversion radix is set accordingly.

>sign returns +1 if its input parameter is +, and -1 if it is -. Otherwise it returns zero:

```
: >sign ( character -- signed )
  case [char] + of +1 endof
    [char] - of -1 endof
    \ default \ drop +0
  endcase ;
```

This value is saved in a local for later usage.

An integer number must contain at least one valid digit. If the string is empty, or if it just contains a sign character, or if the first character is not a valid numerical digit, or if it violates the format of integer numbers in any other way, `search` returns a null definition and a false flag.

`digit?` expects a character as its input parameter, which might be a valid digit or not, given the current number-conversion radix base. If the character is a valid digit, `digit?` returns its numerical value and a true flag. Otherwise, an undefined numerical value and a false flag are returned:

```
: digit? ( character -- unsigned flag )
  dup [char] a >=
  if [ char a 10 - ] literal -
  else dup [char] A >=
    if [ char A 10 - ] literal -
    else [char] 0 - dup cast unsigned 9 >
      if cast unsigned false exit
    then
  then
  then cast unsigned dup base @ < ;
```

Here's a small demonstration of `digit?`:

```
char 6 digit? . . true 6 ok
char C digit? . . false 12 ok
char C hex digit? decimal . . true 12 ok
```

`digit?` actually divides the ASCII character set into 7 segments:

invalid	0 to 9	invalid	A to Z	invalid	a to z	invalid
---------	--------	---------	--------	---------	--------	---------

`digit?` is also used in the definition of `>number`:

```
: >number ( integer-double caddress -> character unsigned
  -- 1st 2nd 4 th )
  begin dup
  while rot rot tuck @ digit?
    while swap base @ * swap + rot rot swap /string
    repeat drop rot rot swap
  then ;
```

Based on the presence of a leading sign and a terminating dot, `search` decides which of its private data members shall be returned as a container for the integer value. Finally, the number-conversion radix base is restored.

?negate negates an integer number if its second input parameter is negative. There are three overloaded versions for single-cell and double-cell integer numbers and for floating-point numbers:

```
: ?negate ( integer signed -- 1st )  
  0< if negate then ;  
  
: ?negate ( integer-double signed -- 1st )  
  0< if negate then ;  
  
: ?negate ( float signed -- 1st )  
  0< if negate then ;
```

12 Input And Output Streams

Input streams

Forth 2012 specifies the concept of *input sources*. Input sources can be the terminal device, strings, files or blocks, identified by `SOURCE-ID` and system variable `BLK`.

StrongForth, on the other hand, follows a more flexible concept, which is based on so-called input streams. An input stream is a class whose objects contain individual input buffers, pointers to the parse area and other information. Each class provides virtual methods for refilling the input buffer and for saving and restoring its state. All these classes are derived from the `input-source` class:

```
dt object procreates input-stream
class input-stream
  forth definitions
  virtual refill ( input-stream -- flag )
  virtual save-input ( input-stream -- 1st )
  virtual restore-input ( input-stream 1st -- flag )
  protected definitions
  null address -> character member 'buffer
  null unsigned member #buffer
  null unsigned member /buffer
  forth definitions
  null unsigned member >in
  : input-stream ( unsigned input-stream -- 2nd )
    locals( this ) erase
    dup /buffer ! chars callocate -> character 'buffer ! this ;
  : input-stream ( input-stream 1st -- 1st )
    locals( this ) copy 0 /buffer ! this ;
  : source ( input-stream -- caddress -> character unsigned )
    locals( this ) 'buffer @ #buffer @ ;
  :noname ( input-stream -- flag )
    drop false ; is refill
  :noname ( input-stream -- 1st )
    new input-stream ; is save-input
  :noname ( input-stream 1st -- flag )
    locals( this ) dup 'buffer @ this 'buffer @ =
    if dup >in @ this >in ! dup #buffer @ this #buffer ! false
    else true
    then swap delete ; is restore-input
  :noname ( input-stream -- )
    locals( this ) /buffer @
    if 'buffer @ free
    then [parent] delete ; is delete
endclass
```

Class `input-stream` contains three protected data members: `'buffer` holds a pointer to the input buffer, `/buffer` its size and `#buffer` the number of characters actually stored in it. The fourth data member, `>in`, is public. It contains the offset from the start of the input buffer to the parse area.

The first constructor allocates the input buffer from dynamic memory and initializes the data members. But there's a second constructor, which creates a one-to-one copy of the input stream object. It sets `/buffer` to zero to indicate that the new object does not have its own input buffer. This constructor is required to implement `save-input`.

`source` simply returns the input buffer. Virtual method `refill` always returns false, because the input buffer of the generic `input-stream` class cannot be refilled. `save-input` creates a copy of the `input-stream` object to save its specification. The specification can be restored with `restore-input`, but only if the saved copy, which is the first input-parameter of method `restore-input`, shares the same input buffer. At the end of `restore-input`, the saved copy is deleted.

`delete` extends the semantics of the version for generic objects by freeing the dynamic memory that has been allocated for the input buffer. If the object of class `input-stream` has been created by the copy constructor, it does not have its own input buffer and the content of `/buffer` is zero. In this case, no additional dynamic memory has to be freed.

Input from the terminal is being handled by class `terminal-input-stream`, which is a child of class `input-stream`:

```
dt input-stream procreates terminal-input-stream
class terminal-input-stream
  forth definitions protected
  : terminal-input-stream ( unsigned terminal-input-stream
    -- 2nd )
    input-stream ;
  : terminal-input-stream ( terminal-input-stream 1st -- 1st )
    input-stream ;
  :noname ( terminal-input-stream -- flag )
    locals( this ) 'buffer @ /buffer @ accept #buffer !
    0 >in ! true ; is refill
endclass
```

The two constructors of class `terminal-input-stream` are identical to those of the parent class. Only `refill` has to be extended. This virtual method accepts a new command line from the terminal and resets the parse area.

Of course, there is only one terminal input stream. It is predefined as a constant, and its buffer has a size of 128 characters:

```
128 new terminal-input-stream constant user-input-device
```

If the input stream is a character string, this string becomes the input buffer. Because it is not necessary to allocate a new one, a new implementation of first constructor is required. Refilling the string is not possible.


```

dt input-stream procreates string-input-stream
class string-input-stream
  forth definitions protected
  : string-input-stream ( caddress -> character unsigned
    string-input-stream -- 4 th )
    locals( this ) this erase #buffer ! 'buffer ! this ;
  : string-input-stream ( string-input-stream 1st -- 1st )
    input-stream ;
endclass

```

Handling input from a file is somewhat more complicated. Class `file-input-stream` needs two additional data members with respect to its parent class. `'file` contains the file handle and `>line` contains the offset within the file to the beginning of the line that has been read into the input buffer:

```

dt input-stream procreates file-input-stream
class file-input-stream
  protected definitions
  null file member 'file
  null unsigned member >line
  forth definitions
  : file-input-stream ( file unsigned file-input-stream -- 3rd )
    input-stream tuck 'file ! ;
  : file-input-stream ( file-input-stream 1st -- 1st )
    input-stream ;
  :noname ( file-input-stream -- flag )
    locals( this )
    0 >in ! 'file @ msvcrt ftell -1?ferror ignore >line !
    'buffer @ /buffer @ 2 - 'file @ read-line
    swap #buffer ! ; is refill
  :noname ( input-stream file-input-stream -- flag )
    locals( this ) cast file-input-stream
    dup >line @ over 'file @
    rot this [parent] restore-input
    if drop drop true
    else over >line ! dup 'file ! null seek-origin
      rot cast signed rot msvcrt fseek ?ferror ignore
      >in @ refill swap >in ! invert
    then ; is restore-input
endclass

```

In addition to the semantics of the constructor of class `input-stream`, the one of class `file-input-stream` has to store the file handle. `refill` stores the current file position as the start of the next line in the `>line` member. It then reads the line into the input buffer and stores the line length without line terminators in the `#buffer` member. `restore-input` needs to be extended for handling `'file` and `>line`. Because the current line might have changed, it needs to be re-read into the input buffer. `fseek` is a runtime library function from the `msvcrt` vocabulary that returns the current file position as an unsigned double-cell number.

If the input shall not come from the terminal, from a string or from a file, a new child of class `input-stream` needs to be defined. An input stream for blocks will be presented in chapter 25.

The five public methods and members of class `input-stream` and its child classes can be used directly, with an object of data type `input-stream` as the last input parameter. However, there is also a default input stream:

```
user-input-device cast input-stream variable default-input-stream
```

In a certain sense, `default-input-stream` is StrongForth's replacement for the Forth 2012 word `SOURCE-ID`. The default input stream is used by overloaded versions of the five methods and members, if no specific input stream is provided:

```
: >in ( -- address -> unsigned )
  default-input-stream @ >in ; 1 retreat

: source ( -- caddress -> character unsigned )
  default-input-stream @ source ; 1 retreat

: refill ( -- flag )
  default-input-stream @ refill ; 1 retreat

: save-input ( -- input-stream )
  default-input-stream @ save-input ; 1 retreat

: restore-input ( input-stream -- flag )
  default-input-stream @ restore-input ; 1 retreat
```

Note the phrase `1 retreat` succeeding the definitions. It is required because otherwise these five words would hide their overloaded versions from the `input-stream` class.

Variable `default-input-stream` can be conveniently changed with an overloaded version of `default`:

```
: default ( input-stream -- )
  default-input-stream ! ;
```

The two words `parse` and `parse-name` receive their input from the default input stream. Additional versions with a specific input stream parameter are usually not required.

```
: parse ( character -- caddress -> character unsigned )
  source >in @ /string rot locals( end c ) 0
  begin dup end <
  while 1+ over over 1- + @ c =
    if dup >in +! 1- exit then
  repeat dup >in +! ;

: parse-name ( -- caddress -> character unsigned )
  source locals( end ) >in @
  begin dup end >=
    if dup >in ! + 0 exit
    then over over + swap 1+ swap @ b1 >
  until dup 1- rot rot
  begin dup end <
  while over over + swap 1+ swap @ b1 <=
    until dup >in ! 1-
  else >in ! end
  then rot /string ;
```

A simple application of `parse` is `\`. The backslash is StrongForth's only means to embed comments within the source code, because parentheses are already used up for stack diagrams. In

Forth 2012, parentheses enclose comments within a source line, while comments that extend to the end of the line start with the backslash:

```
... ( Forth 2012 comment ) ...
... \ another Forth 2012 comment
...
```

In this example, `...` indicates executable source code. To compensate for the missing parenthesis, StrongForth allows using another backslash to terminate the comment, just like the right parenthesis in Forth 2012:

```
... \ StrongForth comment \ ...
... \ another StrongForth comment
...
```

The definition of `\` is quite simple. It just parses until the next occurrence of a backslash or the end of the line, and then discards the parsed string. Because `\` works in both interpretation and compilation state, it has to be an immediate word:

```
: \ ( -- )
  [char] \ parse drop drop ; immediate
```

`char` is an application of `parse-name`. It parses the input source for a word delimited by spaces and returns its first character. Additional characters are discarded. If the parsed word is empty, which happens at the end of the input stream, `char` simply returns a space character. This is the definition of `char`:

```
: char ( -- character )
  parse-name if @ else drop bl then ;
```

Output Streams

You already know the words `emit` and `type` from the chapter about input and output. These two words are not just primitives that write characters and character strings to the terminal. Actually, the destination is an arbitrary output stream. In StrongForth, output with `emit` and `type` can easily be redirected to strings, files and other destinations. The basis of this feature is class `output-stream`:

```
dt object procreates output-stream
class output-stream
  forth definitions
  virtual emit ( integer output-stream -- )
  virtual type ( caddress -> character unsigned
    output-stream -- )
endclass
```

Class `output-stream` does not have any data members. It doesn't even have a constructor, because creating objects of this class makes no sense. It just serves as a parent class for more specific classes that assign semantics to the two virtual methods `emit` and `type`, for example this one:

```

dt output-stream procreates terminal-output-stream
class terminal-output-stream
  forth definitions
  : terminal-output-stream ( terminal-output-stream -- 1st ) ;
  :noname ( integer terminal-output-stream -- )
    ... ; is emit
  :noname ( caddress -> character unsigned
    terminal-output-stream -- )
    ... ; is type
endclass

```

Class `terminal-output-stream` does not have any data members as well. It has a constructor, because otherwise it wouldn't be possible to create objects. However, the constructor has no semantics. There is only one predefined user output device:

```
new terminal-output-stream constant user-output-device
```

This class assigns semantics to its virtual members, which are provided by the operating system. Object `user-output-device` is the initialization value for `default-output-stream`, a global variable used by the overloaded versions of `emit` and `type` that do not require an explicit input parameter of data type `output-stream`:

```

user-output-device cast output-stream
variable default-output-stream

: emit ( integer -- )
  default-output-stream @ emit ;

: type ( caddress -> character unsigned -- )
  default-output-stream @ type ;

```

Another child of class `output-stream` is class `string-output-stream`. It can be used to redirect the output of `emit` and `type` to a string instead of the terminal:

```

dt output-stream procreates string-output-stream
class string-output-stream
  protected definitions
  null caddress -> character member 'buffer
  null unsigned member /buffer
  null unsigned member >out

  forth definitions
  : reempty ( string-output-stream -- )
    0 swap >out ! ;

  : string-output-stream ( caddress -> character unsigned
    string-output-stream -- 4 th )
    tuck /buffer ! tuck 'buffer ! dup reempty ;

  : string ( string-output-stream --
    caddress -> character unsigned )
    dup 'buffer @ swap >out @ ;

```

```

:noname ( integer string-output-stream -- )
  locals( this ) >out @ /buffer @ <
  if cast character this 'buffer @ >out @ + ! 1 >out +!
  else drop -268 throw
  then ; is emit

:noname ( caddress -> character unsigned
  string-output-stream -- )
  locals( this ) >out @ over + /buffer @ <=
  if this 'buffer @ >out @ + swap dup >out +! move
  else drop -268 throw
  then ; is type

endclass

```

Class `string-output-stream` has three protected members. `'buffer` contains the address of a character string buffer, `/buffer` the buffer size and `>out` the index of the next character. All members are being initialized by the constructor. The public method `string` returns the character string that has been written to the output stream so far. In addition, `reempty` allows reusing the string output stream by starting again at the first index position.

The definitions of the virtual methods `emit` and `type` for class `string-output-stream` are straightforward. An exception is thrown if the character string buffer is not big enough to store the additional character or characters.

Class `file-output-stream` is the third child of class `output-stream`. It allows using `emit` and `type` to write characters and character strings to a file, respectively. The only data member is a file handle, which is stored by the constructor in `'file`. Note that `emit` uses `line` as an intermediate storage for the character that is to be written to the file. Here's the class definition:

```

dt output-stream procreates file-output-stream

class file-output-stream
  protected definitions
    null file member 'file

  forth definitions

    : file-output-stream ( file file-output-stream -- 2nd )
      tuck 'file ! ;

    :noname ( integer file-output-stream -- )
      'file @ swap cast character line ! line 1 rot write ; is emit

    :noname ( caddress -> character unsigned
      file-output-stream -- )
      'file @ write ; is type

endclass

```

Of course, you can derive additional output streams from class `output-stream`. For example, classes that redirect output to a text screen, to a block buffer or – in an embedded system – to a peripheral device.

The virtual methods `emit` and `type` can be used directly with an additional output stream parameter. More often, `emit` and `type` will be used with just a character or a character string as parameters. In those cases, the overloaded versions will apply, accessing the system variable `default-output-stream`. To redirect output, it is most convenient to use `default`, which

is an overloaded version of the ones for objects of class `memory-space` and class `input-stream`:

```
: default ( output-stream -- )
  default-output-stream ! ;
```

It is usually not a good idea to redirect output while working with the console. All system responses, including keystroke echoes and the `ok` prompt, will then be written to an invisible stream instead of to the terminal.

13 Object Orientation Revisited

Creating Data Types and Classes

Before a class is being defined, a data type has to be created for it, which has to be a direct or indirect child of data type `object`. This is why there's always a `procreates` phrase preceding the class definition:

```
dt <parent-class> procreates <new-class>
class <new-class>
...
endclass
```

The definition of `procreates` had to be postponed upto now, because it uses words and programming techniques that were not yet introduced upto this chapter. So here it is:

```
: does-data-type ( -- )
  [ dt single >definition runtime ] literal (does) ;

: procreates ( data-type unsigned -- )
  create over object?
  if new class-attributes cast address
  else new data-type-attributes cast address
  then latest ?created-definition body! does-data-type ;

: procreates ( data-type -- )
  dup size procreates ;
```

There are actually two overloaded version of `procreates`. The first one expects the parent data type and the size in address units of the new data type on the stack. The second one, which is used most commonly, expects only the parent data type, and takes over the size from the parent. The first version is only required if you want to create a new ancestor data type, like `single`, `double` or `float`. In those rare cases, the parent is usually a null data type.

`procreates` is a defining word. It creates a new definition with the name of the new data type, which can then be used in stack diagrams to represent parameters. If the parent of the new data type is `object` or a data type derived from `object`, `procreates` creates class attributes for it, otherwise data type attributes are sufficient. The class or data type attributes become the data field of the new definition.

The runtime semantics of the new data type is the same as that for the predefined data types, like `single` or `logical`. It converts the class or data type attributes into a data type using `>data-type` and then adds this data type to a stack diagram with the `param, method`. `does-data-type` would actually be defined like this:

```
: does-data-type ( -- )
  does> ( stack-diagram data-type-attributes -- 1st )
  >data-type over param, ;
```

However, if we chose to define it this way, newly created data types would have a different runtime code than the predefined data types, although the semantics is the same. To avoid this effect, the definition shown above has to be used. It simply copies the runtime code from one of the predefined data types.

The definition created by `procreates` can be converted into the data type it is associated with by using `data-type?` and `?data-type`. `data-type?` returns a data type and a flag telling

whether the definition was indeed created by `procreates`. If it was not, `data-type?` returns a null data type and a false flag:

```
: data-type? ( definition -- data-type flag )
  created-definition?
  if dup runtime
    [ ' single cast created-definition runtime ] literal =
    if >body cast data-type-attributes >data-type true exit
  then
  then drop 0. cast data-type false ;
```

`?data-type` is an overloaded version of the word that searches the dictionary for a data type by name. It returns the data type the definition is associated with, and throws an exception if the flag produced by `data-type?` is false:

```
: ?data-type ( definition -- data-type )
  data-type? invert if -260 throw then ;
```

Creating and Deleting Objects

New objects can be created with `new`. `new` assigns, allocates or allots memory for the new object and executes a constructor to initialize the data members and to perform other specific initialization tasks like allocating buffers, opening files, searching data structures and whatever more. Now, how does `new` work? Let's first see what happens in compilation mode:

```
: new ( -- )
  save-input dt dup vtable
  if swap restore-input drop
    [ ' (new) prev prev input-params drop @ ] literal >dt dup >dt
    vtable dt-drop dt-here literal, " (new)" evaluate
  else drop delete -302 throw
  then ; compile-only
```

As you already know, `compile-only` means that this word is only found in compilation mode and that its semantics are executed, just like that of an immediate word.

To obtain the virtual method table of the class of the new object, `new` parses the class name with `dt`. An exception is thrown if the data type returned by `dt` does not have a virtual method table, which means that it is not a class. By saving and restoring the class name, the compiler parses the class name again after `new` is done, in order to compile a suitable constructor. Remember that a constructor has the same name as the class it belongs to.

The actual creation of the object is performed by one of three overloaded versions of `(new)`:

```
: (new) ( address -> object vtable -> 2nd -- 2nd )
  over cast address -> vtable -> object ! cast object ;

: (new) ( vtable -> object -- 2nd )
  dup size allocate -> object swap (new) ; 1 retreat

: (new) ( memory-space vtable -> object -- 3rd )
  over here -> object rot rot dup size rot allot (new) ;
```

Before evaluating `(new)`, `new` compiles the address of the virtual method table of the class as a literal of data type `vtable -> <class>`, where `<class>` is the data type of the new object. The first version of `(new)` expects the address of the already allocated new object on the stack. It just stores the pointer to the virtual method table in the first cell of the new object and then returns the new object.

By default, new objects are allocated in dynamic memory. This case is handled by the second overloaded version of `(new)`, which does not require an additional input parameter. Note that this version is required to step back in the vocabulary, because the first version of `(new)` could otherwise not be found by the interpreter and the compiler.

Finally, if a memory space is provided as input parameter, the third version of `(new)` allots memory for the new object in the given memory space using `allot`. This method is typically applied for creating static objects that will never be deleted.

The interpreted version of `new` works similarly. For providing the input parameter of data type `vtable -> <class>` to `(new)`, the special low-level word `(vtable)` is required.

```
: (vtable) ( -- vtable )
  dt-here 1- dup @ dt-prefix or swap ! dt-here @ 1 dt-allot
  vtable ;

: new ( -- )
  save-input dt dup vtable
  if swap restore-input drop
    0. cast data-type >dt >dt dt-drop dt-drop
    " (vtable) (new)" evaluate
  else drop delete -302 throw
  then ; execute-only
```

You can even chose to implement your own versions of `(new)` for creating objects.

Objects that have been created by allocating dynamic memory can be deleted with the virtual method `delete`. The version of `delete` defined in class `object` does nothing more than freeing the dynamic memory space. It has to be included in the dedicated versions of all derived classes. But remember that only dynamically allocated objects may be deleted. Trying to delete objects that have been allotted in a memory space with `allot` or from any other pool of static memory will almost certainly cause trouble. Again: Static objects may not be deleted.

You can define dedicated overloaded versions of `(new)` that only apply to specific classes and to classes derived from them. For example, it might make sense to allocate objects of a specific class from a pre-allocated pool of memory chunks instead of from the general dynamic memory, because allocating and freeing dynamic memory is rather slow. To properly delete the objects of these classes, corresponding versions of `delete` have to be defined.

C++ allows using `new` to create arrays of objects. This is not possible in StrongForth. If you need to create multiple objects of the same class, you have to execute `new` within a loop, and store the resulting objects in an array. This restriction is a consequence of the way StrongForth invokes the constructor. Since a constructor consumes its input parameters, the parameters won't be available any more after the first object has been created.

Unions of Members

Sometimes it is useful to assign the same memory location to different members. Suppose you want to implement a class that can alternatively contain a single-cell item, a double-cell item, or a floating-point number. If you simply define three different members, plus a discriminator that tells the object which member to select, you always have at least three cells of unused memory space:

discriminator
single
double
float

A more efficient solution would be to use the same memory for all three items:

discriminator		
single	double	float

StrongForth supports this feature by enclosing the three members in a union:

```
union
  null single member s
and
  null double member d
and
  null float member f
endunion
```

Of course, the word `and` in this context is a special overloaded version of the versions of `and` that expect input parameters of data types `logical` or `data-type`. This example demonstrates that it might make sense to overload even words with completely different semantics. In Forth 2012, on the other hand, it is often necessary to assign a bad or misleading name to a word, just because all better names are already in use. In StrongForth, like in natural languages, you can reuse words as long as the interpreter and compiler can clearly distinguish the context, i. e., the set of input parameters.

`and` is required here in order to split the union into blocks. Each block may contain more than one member, and the members defined in a block share the reserved memory space with the members of all other blocks. The following example demonstrates how to use nested unions with blocks containing multiple members:

```
dt object procreates component
class component
  null unsigned member index
  null character cmember id
```

```

union
  null float member ohm
  null unsigned member max-milliwatt
and
  null flag cmember polar
  null float member farad
  null unsigned member max-volt
and
  union
    null flag cmember fet
    union
      null flag cmember pnp
      and
        null flag cmember p-channel
    endunion
  and
    null flag cmember zener
  and
    null unsigned cmember pins
  endunion
  null character 8 cmembers type#
endunion

: component ( component -- 1st )
  dup erase ;

endclass

```

This object stores the parameters of various electronic components. Each component has a character `id` and an index number `index` to identify it. `id` serves as the discriminator. Transistors, diodes and integrated circuits have a type number `type#`, while resistors and capacitors only have numerical and boolean parameters:

- Resistors: `index, id, ohm, max-milliwatt`
- Capacitors: `index, id, polar, farad, max-volt`
- Bipolar Transistors: `index, id, fet, pnp, type#`
- Field Effect Transistors: `index, id, fet, p-channel, type#`
- Diodes: `index, id, zener, type#`
- Integrated Circuits: `index, id, pins, type#`

The object can be used like this:

```

new component constant resistor1 ok
char r resistor1 id ! ok
12 resistor1 index ! ok
4.7e3 resistor1 ohm ! ok
125 resistor1 max-milliwatt ! ok

new component constant transistor1 ok
char t transistor1 id ! ok
5 transistor1 index ! ok
" 2N3055" transistor1 type# swap move ok
false transistor1 fet ! ok
false transistor1 pnp ! ok

```

The implementation of the three words `union`, `and` and `endunion` is surprisingly simple. `union` expects the current object size, which becomes the starting offset of the union on the stack, and creates two copies of this value. During the definition of the union, these three values stand for

- the starting offset of the union,
- the offset at the end of the largest block so far, and
- the offset within the current block.

These are the definitions:

```
: union ( object-size -- 1st 1st 1st )
  dup dup ;

: and ( object-size object-size object-size -- 1st 2nd 3rd )
  max over ;

: endunion ( object-size object-size object-size -- 1st )
  max nip ;
```

Container Classes

A container class is a class whose objects are collections of items. Typical container classes are arrays, lists, queues and stacks. An object of a container class usually has a compound data type like `stack -> signed` or `list -> string`. For example, a `stack` container will be used like this:

```
new stack -> signed constant st1   ok
-8156 st1 push                     ok
+601 st1 push                     ok
st1 pop .s . signed 601           ok
+20012 st1 push                   ok
st1 pop . 20012                   ok
st1 pop . -8156                   ok
st1 pop . empty stack
st1 delete                         ok
```

Trying to pop from an empty stack results in an exception being thrown, producing the error message *empty stack*. Let's now have a look at the implementation of the `stack` container class:

```
dt object procreates stack
dt object procreates stack-single

class stack-single
  friends( stack )

  private definitions
    null single member item
    null stack-single member next

    : stack-single ( single stack-single stack-single -- 3rd )
      tuck next ! tuck item ! ;

endclass
```

```

class stack
  access stack-single
  private definitions
  null stack-single member tos
  forth definitions private
  : stack ( stack -- 1st )
    null stack-single over tos ! ;
  : empty? ( stack -- flag )
    tos @ 0= ;
  : push ( single stack -> 1st -- )
    locals( this ) tos @ new stack-single tos ! ;
  : pop ( stack -> single -- 2nd )
    locals( this ) empty? abort" empty stack"
    tos @ dup item @ swap next @ tos @ delete tos ! ;
  :noname ( stack -- )
    begin dup tos @
    while dup -> single pop drop
    repeat drop ; is delete
endclass

```

stack-single is a class whose objects contain a single item of data type single, and a link to the next object of the same class. Since all of class stack-single's members are private, including the constructor, the class can only be used by its friends. That's why class stack is declared a friend of class stack-single. Class stack can create objects of class stack-single and access its members, because the phrase `access stack-single` adds stack-single's private word list to the search order. Class stack has a constructor, a version of `delete`, and the three public methods `empty?`, `push` and `pop`. Note that the last input parameters of `push` and `pop` are compound data type types. They ensure that access to a stack is restricted to items of the given data type.

`delete` empties the stack by deleting all objects of data type stack-single that belong to the stack. However, the items themselves are not deleted, which might become necessary if they are objects, as in

```
new stack -> string constant st2
```

You can consider defining an additional method called, e. g., `delete-items` that removes *and* deletes all items:

```

: delete-items ( stack -> object --)
  begin dup tos @
  while dup pop delete
  repeat drop ;

```

The items of the container class `stack` are always single-cell items. For double-cell items and floating-point numbers the class definitions have to be extended appropriately. First, we need to define variants of class `stack-single`, which contain items of data types `double` and `float`. The three classes can be derived from one parent called `stack-item`. In the definition of `stack` itself, we need overloaded versions of `push` and `pop` for the three data types. Finally, `delete` needs to be made independent from the data type of the item. This is the result:

```

dt object procreates stack
dt object procreates queue
dt object procreates linked-item
dt linked-item procreates linked-single
dt linked-item procreates linked-double
dt linked-item procreates linked-float

class linked-item
    friends( stack queue )
    protected definitions
    null linked-item member next
endclass

class linked-single
    friends( stack queue )
    private definitions protected
    null single member item

    : linked-single ( single linked-single -- 2nd )
      tuck item ! ;
endclass

class linked-double
    friends( stack queue )
    private definitions protected
    null double member item

    : linked-double ( double linked-double -- 2nd )
      tuck item ! ;
endclass

class linked-float
    friends( stack queue )
    private definitions protected
    null float member item

    : linked-float ( float linked-float -- 2nd )
      tuck item ! ;
endclass

class stack
    private definitions
    null linked-item member tos
    forth definitions

    : stack ( stack -- 1st )
      null linked-item over tos ! ;

    : empty? ( stack -- flag )
      tos @ 0= ;

    private definitions

```

```

: ?empty ( stack -- )
  empty? abort" empty stack" ;

access linked-item

: link ( linked-item stack -- )
  locals( this ) tos @ over next ! tos ! ;

: unlink ( stack -- )
  locals( this ) tos @ next @ tos @ delete tos ! ;

forth definitions private

:noname ( stack -- )
  locals( this )
  begin tos @
  while unlink
  repeat ; is delete

access linked-single

: push ( single stack -> 1st -- )
  locals( this ) new linked-single link ;

: pop ( stack -> single -- 2nd )
  locals( this ) this ?empty
  tos @ cast linked-single item @ unlink ;

access linked-double

: push ( double stack -> 1st -- )
  locals( this ) new linked-double link ;

: pop ( stack -> double -- 2nd )
  locals( this ) this ?empty
  tos @ cast linked-double item @ unlink ;

access linked-float

: push ( float stack -> 1st -- )
  locals( this ) new linked-float link ;

: pop ( stack -> float -- 2nd )
  locals( this ) this ?empty
  tos @ cast linked-float item @ unlink ;

endclass

```

Please note the word `protected` within the class definitions of `linked-single`, `linked-double` and `linked-float`. Without the `protected` vocabulary being included in the search order of the three class definitions it wouldn't be possible to access the protected data member `next` of their parent class `linked-item`. If `next` had been made a private data member by adding its definition to the private vocabulary of class `linked-item`, the three child classes had to be declared friends of `linked-item`, and additionally the phrase `access linked-item` had to be added to each of them.

The source code of the stack container class as presented above is included in the source file `stack.sf`. This file contains an additional container class `queue.`, which is based on the same classes for the three kinds of items.

We now have a universal stack container class for items of all data types. It's usage is easy and straight-forward:

```
new stack -> unsigned-double constant st1   ok
new stack -> definition constant st2        ok
new stack -> float constant st3             ok
7141107451109. st1 push                     ok
819015441284406556. st1 push                ok
st1 pop . 819015441284406556                 ok
911965606107645. st1 push                   ok
st1 pop . 911965606107645                   ok
st1 pop . 7141107451109                     ok
st1 pop . empty stack
st1 delete                                   ok
' emit st2 push                               ok
' space st2 push                               ok
st2 pop . space ( -- )                       ok
st2 delete                                   ok
st3 empty? . true                           ok
pi st3 push                                 ok
st3 empty? . false                           ok
st3 pop . 3.141592653589793                 ok
st3 delete                                   ok
```


14 Compilation

Compiling Literals

A literal is a constant with a specific value and a specific data type, which has no other semantics than being a constant. For example, `-6172` is a literal with data type `signed`.

The three overloaded versions of `(literal)`, which compile the machine code instructions for literals, were already presented in chapter 10. Their second input parameter of data type `data-type` is only required for optimization purposes, i. e., choosing the optimal register:

```
(literal) ( single data-type -- )
(literal) ( double data-type -- )
(literal) ( float data-type -- )
```

However, compiling a literal requires adding its data type to the data type heap as well. The three overloaded versions of `literal`, compile the machine code *and* update the data type heap. Because the data type of a literal generally is a compound data type, they all expect the address of a compound data type as the second input parameter:

```
literal, ( single address -> data-type -- )
literal, ( double address -> data-type -- )
literal, ( float address -> data-type -- )
```

The Forth 2012 words `LITERAL`, `2LITERAL` and `FLITERAL` are immediate words that compile literals whose values are being calculated at compile time. With `literal`, these words can easily be implemented in StrongForth. As usual, `literal` is overloaded for single-cell items, double-cell items and floating-point numbers:

```
: literal ( single -- )
  [compile] [ dt-here ] literal, ; compile-only

: literal ( double -- )
  [compile] [ dt-here ] literal, ; compile-only

: literal ( float -- )
  [compile] [ dt-here ] literal, ; compile-only
```

`literal` is typically used during compilation in the following way:

```
[ ... ] literal
```

The code enclosed in brackets is immediately executed in order to calculate the value of a literal at compile time. `literal` compiles the machine code instructions of the literal, and additionally adds its data type to the compiler data type heap. But where can we get the data type from? Obviously, the data type is stored at the top of the interpreter data type heap. At the point where `literal` is executed, `literal` has already been parsed and the interpreter data type heap has been updated. This means, the interpreter data type heap pointer now points directly to the data type of the literal. To get the interpreter data type heap pointer, `literal` temporarily switches to interpretation state. Otherwise, `dt-here` would return the compiler data type heap pointer, because `literal` is always executed in compilation state.

But these three versions not sufficient. The semantics of `2LITERAL` are not fully covered by the double-cell version of `literal`, because this version can only be applied to double-cell items. For compiling pairs of single-cell literals, another word is required:

```

: 2literal ( single single -- )
  swap
  [compile] [ dt-here ] literal,
  [compile] [ dt-here dt-next ] literal, ; compile-only

```

Another application of `literal`, is the Forth 2012 word `[']`, which calculates a value at compile time and then compiles it as a literal. As a first approach, it could be defined like this:

```

: ['] ( -- )
  ' [ ' ' output-params drop ] literal literal, ; compile-only

```

After the value has been calculated by `'`, its data type is obtained from the output parameters of `'`. Both parameters are then passed to `literal`,. This solution works fine. However, it leaves the bad taste of low-level programming, because the data type of the literal has to be specified manually instead of just using the data type of the calculated value, which is already present on the compiler data type heap during compilation of `[']`. The solution is `[literal]`:

```

: [literal] ( -- )
  data-space here -> data-type dt-here dt-drop dt-here
  do i @ data-space ,
  loop
  dt-restore [ ' >dt input-params drop ] literal literal,
  " literal," evaluate ; compile-only

```

`[literal]` is executed in compilation state. It assumes that the compound data type of the literal to be compiled is at the top of the compiler data type heap. After dropping this compound data type, `dt-here` returns its address. A `do` loop adds all the basic data types the compound data type is composed of to the data space. After the loop is done, the data type heap is restored and a pointer to the compound data type in the data space is compiled as a literal with a suitable data type. Finally, `literal`, is evaluated. Note that `[literal]` itself has no input parameters.

`[literal]` simplifies the definition of `[']`:

```

: ['] ( -- )
  ' [literal] ; compile-only

```

Two other words that take advantage of `[literal]`, are `[char]` and `[dt]`. They both compile data type literals:

```

: [char] ( -- )
  char [literal] ; compile-only

: [dt] ( -- )
  dt [literal] ; compile-only

```

Now, what about string literals? In StrongForth, a string is usually represented by the address of its first character and by its length:

```

caddress -> character unsigned

```

Of course, we could use `2literal` to compile these two values. But the result is not a string literal, because the characters themselves do not necessarily remain unchanged. This is the reason why Forth 2012 specifies the word `SLITERAL`, which copies a given string into a non-transient memory area and then compiles its address and length as literals. `sliteral` is available in StrongForth as well:

```

: sliteral ( caddress -> character unsigned -- )
  data-space chere -> character
  [ latest input-params drop ] literal literal,
  dup [ latest input-params + 1- ] literal literal,
  data-space ", data-space align ; compile-only

```

`sliteral` uses `"`, to copy the characters of the string to the data space. Other than the three versions of `literal`, `sliteral` does not take care of the actual data type of the literals it compiles. Instead, it provides `literal`, with the data types of its own input parameters.

Compiling Definitions

So far, you've seen how literals are being compiled. But what about compiling definitions? Again, the compilation consists of two tasks:

- Compile the word's machine code into the code space.
- Modify the compiler data type heap to reflect the stack effect of the word.

Analogous to `literal`, both tasks are performed by `compile,:`

```
compile, ( definition -- )
```

The data type heap is updated to carry out the data type transformations as determined by the stack diagram of the definition. If, for example, the data type heap contains

```
address flag unsigned
```

at it's top and

```
rot ( single single single -- 2nd 3rd 1st )
```

is the definition to be compiled, the three data types are replaced by

```
flag unsigned address
```

If the output parameters of the definition contain data type references, the data types to be referred to are those on the data type heap that correspond to the respective input parameters. Sounds confusing? With a couple examples, it might become obvious what is meant. Remember that the matching algorithm implemented in `search` builds a list of references that assign each basic data type in a definition's input parameter list a corresponding basic data type on the data type heap. The superscript numbers show which data types in the input parameter list correspond to which data types on the data type heap.

```
data type heap (before):  1address -> character 2unsigned
definition:               swap ( 1single 2single -- 2nd 1st )
data type heap (after):   2unsigned 1address -> character

data type heap (before):  1address -> 2address -> character
definition:               @ ( 1address -> 2single -- 2nd )
data type heap (after):   2address -> character
```

Let's have a look at two Forth 2012 words, that are actually applications of `compile,.`

`[compile]` has actually the same semantics as `compile,.`, but it additionally parses the input source for the name of the definition to be compiled, and looks it up in the context vocabularies. It is itself an immediate word that is being used during compilation to compile an immediate word, which would otherwise be interpreted:

```
: [compile] ( -- )
  parse-name true match-criterion search-context
  if state @ ] swap compile, state !
  else drop -13 throw
  then ; compile-only
```

Note that `search-context` uses the additional search criterion `match-criterion` in combination with the parameter `true`. As explained in chapter 11, this means that the search

performs an input parameter match against the *compiler* data type heap. This behaviour differs from the standard compilation behaviour, because immediate words are treated the same as non-immediate words. During normal compilation, the vocabulary search is executed with `false` as the parameter of `match-criterion`, in order to perform an input parameter match against the *interpreter* data type heap for immediate words, because these words are interpreted.

Another interesting application of `compile`, is `recurse`, which allows a word to execute itself recursively. Without `recurse`, this wouldn't be possible, because a word cannot be found in a vocabulary before its compilation has been completed. Because the constructors of class definition store the current definition in the value `latest`, it is available to `recurse`:

```
: recurse ( -- )
  latest compile, ; compile-only
```

The standard example for `recurse` is the calculation of the faculty of an integer number, although the faculty can be calculated by a simple iteration as well:

```
: fak ( signed -- 1st )
  dup +1 > if dup 1- fak * else drop +1 then ;
  dup +1 > if dup 1- fak ? undefined word
signed signed
: fak ( signed -- 1st )
  dup +1 > if dup 1- recurse * else drop +1 then ; ok
+6 fak . 720 ok
-3 fak . 1 ok
```

The fact that `recurse` compiles `latest` directly with `compile`, has an interesting consequence. Since the interpreter is not involved, it cannot try to compile `this` before `latest` if the attempt to compile `latest` fails. So, if you use `recurse` within the definition of a class method, you almost always have to explicitly compile `this` before `recurse`:

```
: recursion ( ... my-object -- ... )
  locals( this) ... this recurse ... ;
```

The Interpreter

Now we're getting into the very heart of StrongForth. Parsing the input source, looking up words in the dictionary and interpreting or compiling these words are actually the core functionalities of each Forth system. StrongForth's type system doesn't change this statement, but by considering data types, it adds useful features like type checking and operator overloading.

All these tasks are performed by a single word: `interpret`. Because Forth doesn't make a big difference between interpreting and compiling, `interpret` serves as both the interpreter and the compiler. Whether a word is to be interpreted or to be compiled depends on whether the system is in interpretation or compilation state. Immediate words are always interpreted, as long as they are not forced to be compiled by `[compile]` or `postpone`. So here's the definition of `interpret`:

```

: interpret ( -- )
  begin parse-name dup
  while over over search-local
    if nip nip compile,
    else drop over over false match-criterion search-context
      if nip nip state @ over immediate? invert and
        if compile,
        else <begin-code-pad> compile, <end-code-pad> (execute)
        then
      else drop " this" search-local
        if dup output-params drop >dt rot rot
          true match-criterion search-context dt-drop
          if swap compile, compile,
          else drop drop -13 throw
          then
        else drop drop drop -13 throw
        then
      then
    then
  repeat drop drop ;

```

`interpret` is a loop that parses the input source as long as there are words to parse. For example, if the input source is the user input device, `interpret` processes one complete line of text.

By passing the character string returned by `parse-name` to `search-local`, `interpret` first checks whether the word is a local. Locals only exist in compilation state. If `search-local` finds a match, the local is directly compiled by `compile`, . The implementation of locals and the definition of `search-local` will be presented in chapter 15.

Instead of the explicit search for the name of a local with `search-local`, an alternative solution would have been to make the locals vocabulary the head of the search order during compilation. This solution looks pretty elegant at a first glance. But it has a drawback. Locals would then generally remain visible in interpretation state between [and] inside a definition. That's why the solution with an explicit `search-local` was preferred.

If `search-local` does not succeed, `search-context` gets a chance to search all context vocabularies. With `match-criterion` and its parameter `false` being the additional search criterion, `search-context` performs an input parameter match on the interpreter or compiler data type heap, depending on whether the word is to be interpreted or compiled. Let's assume `search-context` really finds a suitable word. If the system is in compilation state and it is not an immediate word, the word is being compiled by `compile`, . Otherwise, i. e., either the system is in interpretation state or the word is immediate, the word is to be interpreted. Interpretation is performed by the mysterious phrase:

```
<begin-code-pad> compile, <end-code-pad> (execute)
```

`(execute)` is a low-level word that expects the execution token of a definition as the input parameter:

```
(execute) ( token -- )
```

`(execute)` should be used with care, because by not considering the stack effects of the words it executes, it can easily corrupt StrongForth's type system. Its usage in `interpret` is correct, because `compile`, applies the word's stack effect to the interpreter data type heap before the word is executed.

But why is the word compiled before it is executed? Actually, the word is compiled not into the code space, but into a dedicated memory area called the *code pad*. `<begin-code-pad>` saves the code pad pointer and then switches from the code space to the code pad. `<end-code-pad>` adds a special return statement to the code pad and then switches back from the code pad to the code space. It further returns the code pad pointer saved by `<begin-code-pad>` as the execution token for `(execute)`.

`<begin-code-pad>` and `<end-code-pad>` are not available in any vocabulary. Their semantics, which is actually more complex than described here, are just embedded in `interpret`. This means: Whenever you need to interpret a word, you have to use `interpret`.

Finally, let's see what happens if neither `search-local` nor `search-context` finds the word whose name was parsed in the input source. Other than in Forth 2012, `interpret` does not try to interpret the word as a number. Number conversion is performed by dedicated vocabularies like `integer-lit`, which are supposed to be included in the search order.

But there's still one option left, if the word is not found. It has to do with the `this` object. Methods of a class usually have an object of this class as the last input parameter, because they need to access data members and other methods of the same class. If this object is assigned to a local named `this`, data members and methods can be used without providing the object as a parameter. This works, because `interpret`, when a word is not found, compiles the local `this` and then gives the word another try. One of the examples was a method of the example class `rectangle` from chapter 8.

```
: get-position ( rectangle -- signed signed )
  locals( this ) px @ py @ ;
```

The private members `px` and `py` both expect an object of data type `rectangle` on the stack. This object is not present. However, after `interpret` has compiled `this`, `px` as well as `py` can be found. With this feature, implementing class methods becomes clearer and less tedious.

Note that before `interpret` compiles `this`, it first ensures that a local with this name exists and that the word to be compiled is really found in the context vocabularies with the `this` object being present. Of course, all of this does only work in compilation state, because in interpretation state, there are no locals.

Evaluating and Postponing

StrongForth implements the Forth 2012 word `EVALUATE` as an application of `interpret`:

```
: evaluate ( caddress -> character unsigned -- )
  default-input-stream @ locals( stream )
  new string-input-stream default
  interpret
  default-input-stream @ delete stream default ;
```

`evaluate` creates a new input stream from the character string that is to be evaluated, and makes it the default input stream. The character string is interpreted, and finally the saved input stream is restored. The present default input stream needs to be saved in a local. It cannot be kept on the data stack, because the data stack is generally affected by the evaluated words. Of course, `evaluate` can be applied recursively.

One of the most prominent applications of `evaluate` is the Forth 2012 word `postpone`. According to the Forth 2012 specification, it compiles *the compilation semantics* of a word *to the current definition*. What does this mean? The compilation semantics of an immediate word is to execute the word *immediately*. `postpone` compiles this compilation semantics by adding the

token of the immediate word into the current definition using `compile,`. The immediate word is then executed at runtime instead of at compile time. Its execution is *postponed*.

The compilation semantics of a non-immediate word is to compile the token of the word into the current definition. Therefore, `postpone` compiles code that compiles the word at runtime. It actually compiles the name of the word as a string and then compiles the token of `evaluate`. Here's the definition of `postpone`:

```
: postpone ( -- )
  parse-name over over true match-criterion search-context
  if dup immediate?
    if nip nip compile, exit
    then
  then
  drop [compile] sliteral ['] evaluate compile, ; compile-only
```

But why does `postpone` use `evaluate` instead of compiling `compile,` with the word's definition as a literal parameter? This is what a Forth 2012 system would do. In StrongForth, the integrity of the type system requires that a word is compiled within the context of the current definition. At compile time, when `postpone` is executed, it only knows the context of the word that compiles the postponed word, but not the context of the postponed word itself. Selecting the correct word is not possible if it is overloaded. Searching the word in the dictionary and updating the data type heap has to be postponed as well. This is exactly what `evaluate` does. For `postpone`, it does not even matter whether a postponed non-immediate word actually exists, because the context is invalid anyway. No matter whether `search-context` finds a non-immediate word or just fails, it simply compiles the name of the word as a string literal, plus `evaluate` to have the word compiled at runtime.

15 Values And Locals

Values

Constants and variables actually have the same interpretation and compilation semantics. They both return a literal constant. For a variable, this constant is the address of the variable's value.

Values, on the other hand, have different semantics. Instead of returning a literal constant, they return the contents of a memory location with a constant address. Therefore, it is necessary to derive a dedicated child of class `single-definition` for them:

```
dt single-definition procreates value-definition
class value-definition
  forth definitions protected
  : value-definition ( caddress -> character unsigned
    value-definition -- 4 th )
    definition ;
  : store ( single value-definition -- )
    'value @ cast address -> single ! ;
  : store ( double value-definition -- )
    'value @ cast address -> double ! ;
  : store ( float value-definition -- )
    'value @ cast address -> float ! ;
  : enddef ( value-definition -- )
    enddef ;
  :noname ( compiler-workspace value-definition -- )
    ... ; is (compile)
endclass
```

A value is used like a constant, although its value can be changed like that of a variable. Like a variable, the member `'value` contains the address of a data space location where its value is stored. Nevertheless, class `value-definition` has its own version of `(compile)` with an additional level of indirection with respect to that of class `single-definition`, and, for purposes of code optimization, a dedicated version of `enddef`.

Furthermore, class `value-definition` needs a means to change the value in the data space. This is what the three versions of `store` do. These methods are for low-level usage only, because they do not ensure data type integrity. In `StrongForth`, `store` is only used by `to`. Be aware of the difference between `store` and `assign`. `store` sets the value, while `assign` sets its location.

Like with `constant`, we need three different versions of `value` for single-cell and double-cell items and for floating-point numbers:

```
: value ( single -- )
  (value) swap data-space , enddef ;
: value ( double -- )
  (value) swap data-space , enddef ;
```



```
: value ( float -- )
  (value) swap data-space , enddef ;
```

Creating a new object of class value-definition, assigning it a location in the data space and constructing the stack diagram is done by (value). This word does not need to be overloaded, because it applies to all versions of value:

```
: (value) ( -- value-definition )
  parse-name new value-definition
  data-space here over assign dt-here over params! ;
```

Remember that dt-here returns the address of the data type of the input parameter of value, which the interpreter removed immediately before it executed value.

The second overloaded version of value covers the semantics of the Forth 2012 word 2VALUE for double-cell items. The third overloaded version implements the word FVALUE from the Forth 2012 specification.

The semantics of 2VALUE for pairs of single-cell items is not available in StrongForth. You have to use two separate values instead. Anyway, there is a technique to get around this deficiency. Let's demonstrate it with the example of character strings in the caddress -> character unsigned representation. We can merge address and count into one double-cell item of a new data type string and define two words that perform the conversion:

```
dt double procreates string
: >string ( caddress -> character unsigned -- string )
  merge cast string ;
: string> ( string -- caddress -> character unsigned )
  split swap cast caddress -> character swap cast unsigned ;
```

Now, we can convert a character string in the caddress -> character unsigned representation into a double-cell item of data type string and define a value with it. Each time the value is retrieved, it needs to be converted back to a character string:

```
' action-of name >string value word-name ok
word-name string> type action-of ok
' definition name >string to word-name ok
word-name string> type definition ok
```

This solution is pretty versatile. It can also be applied to variables and constants of character strings. For each combination of two single-cell data types other than caddress -> character unsigned, appropriate double-cell data types and conversion words have to be defined.

Locals

Locals have, other than constants, variables and values, a strictly limited lifetime. They are defined within the definition of word, and are abandoned when the definition of this word is finished. On the other hand, they can be used just like any word that is being compiled into a definition.

StrongForth keeps locals in a dedicated vocabulary named locals-vocabulary, which is just a standard wordlist. Besides locals in a narrow sense, this vocabulary also contains other temporary definitions like r@ and loop indexes. You've already seen glimpses of locals processing in the definitions of begin-compilation and end-compilation as shown in chapter 10.

begin-compilation initializes the system variable #locals with zero, end-compilation empties the locals vocabulary by executing forget-locals.

At runtime, the values of locals are kept in a stack frame, because stack frames are efficiently supported by the processor. At compile time, `#locals` contains the size of the stack frame in cells, i. e., the number of cells occupied by locals at runtime:

```
0 cvariable #locals
```

`forget-locals` deletes all definitions in the locals vocabulary:

```
forget-locals ( -- )
  begin locals-vocabulary last 0<>
  while locals-vocabulary last delete
  repeat ;
```

Class `local-definition` is derived from class `definition`:

```
dt definition procreates local-definition
class local-definition
  protected private definitions
  null unsigned cmember 'index
  null do-destination member 'enclosing-loop
  : link! ( local-definition -- )
    locals-vocabulary last swap 'link ! ;
  : index! ( local-definition -- )
    #locals @ swap 'index ! ;
  forth definitions
  : local-definition ( caddress -> character unsigned
    local-definition -- 4 th )
    locals( this ) this erase name! link! index! this ;
  : enclosing ( local-definition -- do-destination )
    'enclosing-loop @ ;
  : enclosing! ( do-destination local-definition -- )
    'enclosing-loop ! ;
  : enddef ( local-definition -- )
    locals-vocabulary last! ;
  :noname ( compiler-workspace local-definition -- )
    ... ; is (compile)
endclass
```

Private data member `'index` contains the index of the local within the stack frame. It is initialized by the private method `index!` with the present contents of `#locals`, which is incremented when defining a new local by its size in cells. Note that StrongForth allows multiple instances of `locals|` within the same definition.

`link!` establishes a link to the previous definition within the locals vocabulary. This version of `link!`, other than the one included in class `definition`, does not make an assignment to `latest`.

The constructor of class `locals-definition` initializes the locals name, establishes a link to the previous local and sets its index. A special version of `enddef` applies to the locals vocabulary instead of the current compilation vocabulary as with all other definitions.

But what about 'enclosing-loop? This private member is null unless the local is a loop index `i` or `j`. For loop indexes, 'enclosing-loop identifies the do loop the loop index belongs to. In the next chapter you'll learn why it makes sense to define loop indexes as locals and how everything fits together.

As specified in Forth 2012, new locals are to be created with `(local)`:

```
: (local) ( caddress -> character unsigned -- )
  dup
  if " (>r)" evaluate new local-definition
    dt-here over params! enddef
  else drop drop
  then ;
```

The message *last local* with a null string is not required in StrongForth, because locals can be declared repeatedly within the same definition. This makes it possible to define sets of locals at different locations within a definition.

`(>r)` increments `#locals` by the size in cells of the local, and then compiles code to store the local at the proper location within the stack frame. Because StrongForth also supports double-cell locals and floating-point locals, three overloaded versions of `(>r)` are required:

```
(>r) ( single -- )
(>r) ( double -- )
(>r) ( float -- )
```

The stack diagram of locals consists of a single output parameter, whose data type is obtained from the data type heap in the same way as by `(value)`. The data type of the item consumed by `(>r)` is still stored at the next free position of the compiler data type heap.

With `(local)`, the Forth 2012 word `LOCALS|` can be defined in StrongForth:

```
: locals| ( colon-definition -- 1st )
  begin parse-name over over " |" compare over 0<> and
  while (local)
  repeat drop drop ; compile-only
```

`colon-definition` is a dummy parameter that is returned unchanged as `1st`. It ensures `locals|` cannot be used inside loops and conditional clauses, or between `>r` and `r>`. Therefore,

```
: valid ( unsigned -- signed )
  locals| a | -45 locals| b | b a + ;
```

is a valid StrongForth definition and works as expected, whereas

```
: invalid ( unsigned flag -- )
  if locals| a | ... \ not allowed
```

will be rejected.

If you want to declare only one local, you can use the phrase `local <name>` in StrongForth, without the terminating `|`:

```
: local ( colon-definition -- 1st )
  parse-name (local) ; compile-only
```

One drawback of `locals|` is that the locals have to be declared in reverse order. If, for example, you have token `signed-double float` in this order on the stack and you want to assign these three items to locals, you need to write something like `locals| r d xt |`. To avoid this sometimes confusing syntax, StrongForth provides `locals (` as a replacement for `locals|`, with `)` being the suitable terminator. `locals (` accepts the locals in the same order as they are on the

data stack, in our example as `locals(xt d r)`. It takes advantage of a recursive algorithm to accomplish this task:

```
: locals( ( colon-definition -- 1st )
  dup parse-name over over " )" compare over 0<> and
  if rot recurse drop (local)
  else drop drop drop
  then ; compile-only
```

The locals vocabulary need not be included in the search order. In compilation state, `interpret` searches each word it parses in the locals vocabulary before it tries to find it elsewhere. It actually uses a dedicated search word for this purpose:

```
: search-local ( caddress -> character unsigned
  -- local-definition flag )
  state @
  if null single no-criterion locals-vocabulary search
    swap cast local-definition swap
  else drop drop null local-definition false
  then ;
```

No additional search criterion needs to be applied. If a local with the given name exists in the locals vocabulary, `search-local` returns it as an object of class `local-definition`, plus a `true` flag. Otherwise, it returns a null definition and a `false` flag.

What about the new Forth 2012 word `{ :? }`? In StrongForth, this word is *not* supported, because the notation conflicts with StrongForth's mandatory stack diagram. If you need to define locals within a definition, you have to use `local`, `locals|`, or `locals(`. However, using `locals|` is discouraged, since it expects the names of the locals in reverse order.

Changing Values and Locals

Interpretation of `to`

Because `to` works differently in interpretation and in compilation state, StrongForth provides overloaded versions with the attributes `execute-only` and `compile-only`. The versions for interpretation deal with values only, not with locals.

```
: to ( single -- )
  parse-name ?value-definition
  dt-restore dup ?to dt-drop store ; execute-only

: to ( double -- )
  parse-name ?value-definition
  dt-restore dup ?to dt-drop store ; execute-only

: to ( float -- )
  parse-name ?value-definition
  dt-restore dup ?to dt-drop store ; execute-only
```

All three versions of `to` parse a name and try to find a value with this name in the context vocabularies. The input parameter of `to` is then stored in the value, provided its data type fits to that of the value.

`?value-definition` performs a vocabulary search on a given name that only considers values. Because all values are objects of class `value-definition`, they share the same virtual method

table. `vtable-criterion` is a search criterion that matches only definitions whose virtual method table is equal to the one provided by the parameter of data type `single`:

```
:noname ( definition single -- flag )
  cast vtable swap vtable = ;
token cast search-criterion constant vtable-criterion

: ?value-definition ( caddress -> character unsigned --
  value-definition )
  [ dt value-definition vtable ] literal
  vtable-criterion search-context
  invert if -32 throw then cast value-definition ;
```

In order to check whether the input parameter of `to` may be stored in the value according to the rules of StrongForth's data type system, `?to` tries to find a version of `!` for it in the context vocabularies. Before executing `?to`, `to` temporarily restores its own input parameter on the interpreter data type heap. The second input parameter for `!` is being constructed by `?to` and also temporarily added to the interpreter data type heap. If no matching version of `!` is found, `?to` throws an exception.

```
: ?to ( definition -- )
  [ dt address dt-prefix or ] literal >dt output-params drop >dt
  " !" false match-criterion search-context
  dt-drop invert if -13 throw then drop ;
```

Compilation of `to`

During compilation, `to` can be applied to both values and locals. At the beginning, `to` tries to find the parsed name in the locals vocabulary. If it succeeds, it compiles the local. Otherwise, it tries to find a value with this name in the context vocabularies and compiles the value:

```
: to ( -- )
  parse-name over over search-local
  if nip nip
    dup ?to new to-local-definition dup compile, delete
  else drop ?value-definition
    dup ?to new to-value-definition dup compile, delete
  then ; compile-only
```

Compiling `to` for a value or a local is done in a different way than interpreting `to` for a value. After making sure that the data type of the value or local matches the item on top of the compiler data type heap, `to` creates a temporary object of class `to-value-definition` or class `to-local-definition`, respectively. Finally, `to` compiles and then immediately deletes the temporary object.

What's inside these two classes? They are derived from classes `value-definition` and `local-definition`, respectively, and they both add just a new constructor and a new version of `compile`, to their parent classes:

```

dt value-definition procreates to-value-definition
class to-value-definition
  forth definitions protected
  : to-value-definition ( value-definition to-value-definition --
    2nd )
    locals( this ) this cast value-definition copy
    0 #name ! null caddress -> character 'name !
    #params @ dup #input-params ! 'params @
    over [ 2 cells ] literal * allocate -> data-type
    dup 'params ! rot move this ;
  :noname ( compiler-workspace to-value-definition -- )
    ... ; is (compile)
endclass

dt local-definition procreates to-local-definition
class to-local-definition
  forth definitions
  : to-local-definition ( local-definition to-local-definition --
    2nd )
    swap cast value-definition swap cast to-value-definition
    to-value-definition cast to-local-definition ;
  :noname ( compiler-workspace to-local-definition -- )
    ... ; is (compile)
endclass

```

The new classes have no additional data members with respect to their parents. Their constructors copy the data members from an object of the parent class, `value-definition` or `local-definition`, and change their stack diagram by making an input parameter out of an output parameter. New dynamic memory space for the stack diagram has to be allocated, because this memory space gets freed when the object is deleted. For the same reason, address and length of the object's name are set to zero. The newly created object expects an item on the stack that has the data type of the value or the local it is derived from, and can be compiled.

An Implicit Local: `r@`

Forth 2012 specifies the words `>R`, `R>` and `R@` to transfer single-cell items between the data stack and the return stack. The stack diagrams contain everything you need to know about the semantics:

```

>R ( x -- ) ( R: -- x )
R> ( -- x ) ( R: x -- )
R@ ( -- x ) ( R: x -- x )

```

Now, let's see what StrongForth has to offer:

```

' >r . >r ( -- r-index ) compile-only ok
' r> . r> ( r-index -- ) compile-only ok
' r@ .
' r@ ? undefined word
definition

```

What's that? `>r` and `r>` seem have totally different stack diagrams with a strange data type called `r-index`. And `r@` doesn't even seem to exist! This must be a mistake.

Well, by now you should be used to the fact that StrongForth offers some surprises. What is really happening here? Let's assume StrongForth had `r@` in one of its vocabularies. What would be the stack diagram? That's difficult to say, because it depends on what data type `>r` has actually pushed onto the return stack. In the definition

```
: x1 ... 200 >r ... r@ ... r> ... ;
```

both `r@` and `r>` are expected to have the stack diagram (-- unsigned), whereas in

```
: x2 ... pad >r ... r@ ... r> ... ;
```

the stack diagrams of `r@` and `r>` should be (-- caddress -> character). Therefore, it makes no sense for StrongForth to provide a pre-defined version of `r@` in the dictionary. The easiest way to solve this problem is to make `r@` a local. Each local has a stack diagram that is specified at its definition. In this case, `>r` defines `r@` as a local, and `r>` removes the most recent local from the locals vocabulary.

That's exactly how it works. Both `>r` and `r>` are immediate words:

```
dt single procreates r-index
```

```
: >r ( -- r-index )
  " r@" (local) #locals @ cast r-index ; compile-only

: r> ( r-index -- )
  drop locals-vocabulary last name " r@" compare
  if -263 throw
  else " r@" evaluate locals-vocabulary last delete
  then ; compile-only
```

`>r` uses `(local)` with the name `r@` to create a new local and to assign it a single-cell or a double-cell item or a floating-point number. The value `>r` returns is the index of the local within the stack frame. However, its value is of no significance. Its sole purpose is to pass an item of a unique data type to `r>` in order to ensure that the two words are used pairwise and with proper nesting. Data type `r-index` is in fact only used by `>r` and `r>`. Syntax violations like

```
... r> ... >r ... ;
```

or

```
... if ... >r ... then ... r> ... ;
```

are rejected by the compiler. On the other hand, the requirement for pairwise use of `>r` and `r>` prevents usages like

```
... if ... >r ... else ... >r ... then ... r> ... ;
```

or

```
... >r ... if ... r> ... else ... r> ... then ... ;
```

which would be allowed in Forth 2012. Anyway, with some knowledge about the handling of the control-flow in StrongForth, even these programming structures can be implemented.

`r>` drops the item of data type `r-index` that was produced by `>r`. As an additional measure of type security it ensures that `r@` is indeed the last local that was added to the locals vocabulary. It fetches the local from the stack frame and then removes it from the locals vocabulary. Note that there's no need to pop something off the return stack. The stack frame is always resolved at the end of a definition.

With this technique, nesting multiple instances of `>r` and `r>` is of course possible, even with different data types and sizes. In those cases, multiple versions of `r@` would reside in the locals vocabulary. The latest version hides all previous versions. Just like in Forth 2012, you have access only to the latest one, up to the point where it is deleted from the locals vocabulary by `r>`.

The rather strict syntax applies only to `>r` and `r>`, not to `r@`. The scope of `r@` lies between `>r` and `r>`, independently of the control structure inbetween. This means, for example, that you can use `r@` within a `do` loop even if `>r` and `r>` are both outside of the loop:

```
... >r ... do ... r@ ... loop ... r> ...
```

Another interesting consequence of `r@` being a local is the fact that `to` can be applied to it the same way as with all other locals:

```
: test ( unsigned -- )
  >r r@ . r@ 2* to r@ r> . ; ok
7 test 7 14 ok
```

However, you should be careful using `r@` in such a way. Always keep in mind that `>r` and `r>` are immediate words that build a control-flow structure, just like `if ... else ... then`, `begin ... until` and `do ... loop`, and that `r@` is a local that is being defined by `>r`, and that is being removed by `r>`.

StrongForth's implementations of `>r` and `r>` allow not only transferring single-cell items to and from the returns stack. `>r` and `r>` also work for double-cell items and floating-point numbers. However, there's no replacement for the Forth 2012 words `2>R`, `2R@` and `2R>` when applied to pairs of single-cell items. If you don't need `2R@`, you can use `>r >r` instead of `2>R` and `r> r>` instead of `2R>`. Otherwise, you have to use locals.

In addition to `>R`, `R>`, `2>R` and `2R>`, Forth 2012 specifies the two words `N>R` and `NR>`, that transfer arbitrary numbers of items to and from the return stack. `N>R` and `NR>` are not available in StrongForth, because a local always consist of exactly one item. Providing these words is superfluous anyway. They become handy in connection with words like `GET-ORDER`, `SET-ORDER`, `SAVE-INPUT` and `RESTORE-INPUT` that deal with such tuples. Instead, StrongForth takes advantage of objects to store tuples and keep them away from the data stack.

16 Conditionals And Loops

Saving and Restoring Data Type Information

Let's consider the generalized machine code of a conditional clause with an `if` branch and an `else` branch, as compiled by StrongForth:

Label	Machine code	Compiler data type heap
	calculate the condition	
l1	conditional jump to l4	($x_1 \dots x_n$ flag -- $x_1 \dots x_n$)
l2	<code>if</code> branch	($x_1 \dots x_n$ -- $y_1 \dots y_m$)
l3	unconditional jump to l5	($y_1 \dots y_m$ -- $y_1 \dots y_m$)
l4	<code>else</code> branch	($x_1 \dots x_n$ -- $y_1 \dots y_m$)
l5	continue linear flow of execution	($y_1 \dots y_m$ -- ...)

The conditional jump instruction preceeding the `if` branch is generated by compiling the low-level word `(0branch)`. `(0branch)` expects a condition on the stack, which is supposed to be a flag, although it can be any single-cell item. A value of zero means `false` (`if` branch not taken), any non-zero value means `true` (`if` branch taken). At the end of the `if` branch, an unconditional jump generated by the low-level word `(branch)` jumps to the end of the `else` branch, where both branches join:

```
(0branch) ( single -- )
(branch) ( -- )
```

These two words actually compile short jump instructions, i. e., their jump offset is limited to the range -128 to +127. For long conditional clauses and loops, expecially if nested, this range might turn out to be insufficient. You could try to factor parts of the code out to make the branches shorter, but this is not always possible. For those cases, StrongForth provides additional low-level words with unlimited jump ranges:

```
(0lbranch) ( single -- )
(lbranch) ( -- )
```

To use these long jumps, special versions of `if`, `else` and so on have to be used, which can be included from StrongForth's source code library.

Now, what about the data type information? At label l1, these data types are on the compiler data type heap:

```
(  $x_1 \dots x_n$  flag )
```

During execution of the immediate word `if` in compilation state, `(0branch)` is compiled. Because `(0branch)` consumes the condition, the compiler data type heap at label l2 looks like this:

```
(  $x_1 \dots x_n$  )
```

Both the `if` branch and the `else` branch start with these data types. Since the `if` branch might change the compiler data type heap, its contents at label l2 need to be saved in some way. It will be restored at the beginning of the `else` branch, which is marked by label l4. After the `if` branch has been compiled, the compiler data type heap might look totally different:

```
(  $y_1 \dots y_m$  )
```

These are the contents of the compiler data type heap at label l3. Because of the jump instruction compiled by (branch), execution will continue at label l5. Again, the current contents of the compiler data type heap needs to be saved.

Before compiling the else branch, the contents of the compiler data type heap are restored to its contents at the beginning of the if branch. At the end of the else branch, both branches join. This means that the stack effect of the else branch has to be exactly the same as the stack effect of the if branch. This is ensured by comparing the contents of the compiler data type heap at label l5 with what has been saved at label l3. An exception is thrown if this condition is not met. At l5, execution continues with

```
( y1 ... ym )
```

on the compiler data type heap.

For saving and restoring the contents of the compiler data type heap, StrongForth provides class control-flow. This class also keeps track of the code addresses where branches and loop bodies begin, and calculates the jump offsets:

```
dt object procreates control-flow
class control-flow
  forth definitions
  virtual resolve ( control-flow -- )
  protected definitions
  null unsigned member #frozen
  null address -> data-type member 'frozen
  null address member 'location
  null origin member 'next-origin
  code ?congruent ( control-flow -- )
    \ ...
  endcode
  : save ( control-flow -- )
    locals( this ) dt-here
    if dt-depth dup #frozen !
      dt-bottom over cells 2* allocate -> data-type
      dup 'frozen ! rot move
    then ;
  forth definitions
  : relocate ( address control-flow -- )
    'location ! ;
  : control-flow ( control-flow -- 1st )
    locals( this ) erase save code-space here relocate this ;
  : restore ( control-flow -- )
    locals( this ) dt-init
    'frozen @ #frozen @ + 'frozen @ ?do i @ >dt loop ;
  : match ( control-flow -- )
    locals( this ) 'frozen @
    if dt-here
      if ?congruent
        else restore
```

```

        then
    then ;
: >flag ( single -- 1st ) ;
: add-origin ( control-flow origin -- )
  locals( this orig )
  this 'next-origin @ orig 'next-origin !
  orig this 'next-origin ! ;
:noname ( control-flow -- )
  locals( this )
  'next-origin @ if 'next-origin @ resolve then ; is resolve
:noname ( control-flow -- )
  locals( this )
  'next-origin @ if 'next-origin @ delete then
  'frozen @ if 'frozen @ free then
  [parent] delete ; is delete
endclass

```

The protected method `save` creates a copy of the compiler data type heap in dynamic memory, storing the address and the number of basic data types in the data members `'frozen` and `#frozen`. Of course, it does so only if the compiler data type heap is not locked.

`?congruent`, which is also a protected method, compares the current compiler data type heap with the one that has been saved. It throws an exception if both versions are not one-to-one identical. If necessary, you have to apply type casts to ensure data type identity in the two branches, like in this example:

```

: test ( flag -- caddress )
  if pad else null caddress then ;
  if pad else null caddress then ? data types not congruent
caddress
: test ( flag -- caddress )
  if pad cast caddress else null caddress then ; ok

```

The constructor of class `control-flow` saves the compiler data type heap and uses the method `relocate` to save the current location of the code space in the data member `'location`. This location is always assumed to be the beginning of a conditional clause or a loop, which is either the origin or the destination of a jump instruction. More precisely, it is the location at which the compiler data type heap is saved.

`restore` is the method that restores the compiler data type heap from the saved version. This happens, for example, at the beginning of an `else` branch, or, more general, whenever the compiler data type heap is locked. If it is not locked, `?congruent` is typically the preferred operation. `match` executes the proper operation depending on whether the compiler data type heap is locked or not.

`>flag` is an internal word that controls register allocation immediately before `(0branch)` gets compiled. It actually tells the compiler that the flag `(0branch)` will consume shall be transferred to the processor's `eflags` register before the compiler data type heap is saved. In the final section of this chapter you will see why `>flag` is required.

In some control-flow structures, a single copy of the compiler data type heap is associated with more than one code space location. For example, within a `case ... endcase` structure, each of `... endof` branch ends with an unconditional jump to the location at the end of the structure. Therefore, class `control-flow` builds a linked list of objects of class `origin`, which

is a child of class `control-flow`. The linked list starts at the content of member `'next-origin`. `add-origin` adds an object of class `origin` to the list.

The linked list has to be considered within the virtual methods `resolve` and `delete` as well. `resolve` is supposed to compile the jump offset once both source and destination addresses are known. The actual semantics has to be specified by the children of class `control-flow`. However, what needs to be done by the version of class `control-flow`, is to traverse the linked list and recursively resolve all control-flow origins. The same kind of recursion is used within the definition of virtual method `delete`. This word first deletes everything in the linked list before it frees the dynamic memory allocated for the saved compiler data type heap and finally its own data members.

Conditional Clauses

Now, after you've learned the basics of conditional and unconditional jumps and about keeping track of the data type information when deviating from the linear flow of execution, it's time to study the implementations of `if`, `else`, `then` and `ahead`. Actually, there's not much left:

```
: if ( -- origin )
  ['] (0branch) compile, new origin ; compile-only
: then ( origin -- )
  dup match dup resolve delete ; compile-only
: ahead ( -- origin )
  ['] (branch) compile, new origin dt-lock ; compile-only
: else ( origin -- 1st )
  dt-here
  if [compile] ahead
  else new origin null address over relocate
  then swap [compile] then ; compile-only
```

All four words are `compile-only`, which means their semantics are executed in compilation state, and the interpreter does not find them in interpretation state.

You can see that StrongForth keeps objects of class `control-flow` and of its child classes, like `origin`, on the data stack. As demonstrated in the definitions of `then` and `else`, they can be moved or copied just like all other data stack items. There's no separate control-flow stack as optionally specified by Forth 2012. The two Forth 2012 words `CS-PICK` and `CS-ROLL` have no counterparts in StrongForth.

`if` just compiles a conditional jump with a zero offset and creates a new object of class `origin`, because the conditional jump is the *origin*. However, the jump offset is not known yet, but because the location is stored in the object of class `origin`, it can be resolved at the end of the `if` branch. Class `origin` is a child of class `control-flow`. The only difference to its parent class is the virtual method `resolve`, which resolves a forward jump by storing the jump offset in the location of the operand byte of the jump instruction:

```

dt control-flow procreates origin
class origin
  forth definitions
  : origin ( origin -- 1st )
    control-flow ;
  :noname ( origin -- )
    ... ; is resolve
endclass

```

You'll see in the next section that class `control-flow` has a second subtype destination, which is created at the *destination* of a jump instruction.

If the `if` branch is terminated by `then`, i. e., if there's no `else` branch, the contents of the compiler data type heap at the beginning and at the end of the `if` branch should exactly match.

Label	Machine code	Compiler data type heap
	calculate the condition	
l1	conditional jump to l3	(x ₁ ... x _n flag -- x ₁ ... x _n)
l2	if branch	(x ₁ ... x _n -- x ₁ ... x _n)
l3	continue linear flow of execution	(x ₁ ... x _n -- ...)

Because the compiler data type heap generally is not locked when `then` is executed, `match` compares the contents of the compiler data type heap with the data types that were saved when `if` was executed. Since `then` marks the end of the `if` branch at label l3, it can now resolve the forward reference. `resolve` calculates the jump offset as the length of the `if` branch, and stores the result in the operand byte of the conditional jump instruction.

`ahead` is very similar to `if`. The differences are that `ahead` compiles an unconditional jump instruction instead of a conditional jump instruction, and that it locks the compiler data type heap. No further machine code can be compiled. The only means to continue compilation is to unlock the compiler data type heap by making this location the destination of a jump instruction. Unlocking the compiler data type heap is a side effect of `restore`.

This doesn't make sense? Well, certainly not as long as you think about compiling code that looks like this:

```
... ahead ... then ...
```

But if you look back at the code compiled by

```
... if ... else ... then ...
```

you'll see that the `else` branch is nothing else than an `ahead` branch. It is preceded by an unconditional jump instruction. And indeed, `ahead` is included in the definition of `else`. Actually, `else` has two tasks: Finish the `if` branch and start the `else` branch.

`else` begins with the second task. It executes `ahead` to save the contents of the compiler data type heap at the end of the `if` branch, and compiles an unconditional jump instruction. Then, it performs the first task by executing `then`. `then` calculates the jump offset at label l1. Since the compiler data type heap was just locked by `ahead`, `then` restores the contents the compiler data type heap to the status at label l2, instead of comparing them with the saved data types. This means, the `else` branch starts with the same compiler data type heap as the `if` branch. The item of data type `origin`, which `else` returns, contains the contents of the compiler data type heap at the end of the `if` branch.

Label	Machine code	Compiler data type heap
	calculate the condition	
l1	conditional jump to l4	(x ₁ ... x _n flag -- x ₁ ... x _n)
l2	if branch	(x ₁ ... x _n -- y ₁ ... y _m)
l3	unconditional jump to l5	(y ₁ ... y _m -- y ₁ ... y _m)
l4	else branch	(x ₁ ... x _n -- y ₁ ... y _m)
l5	continue linear flow of execution	(y ₁ ... y _m -- ...)

The final `then` at the end of the `else` branch will do the same as `then` at the end of an `if ... then` conditional clause, with the exception that it calculates the offset of an unconditional jump instruction instead of the offset of a conditional jump instruction. It compares the contents of the compiler data type heap with the data types that were saved at the end of the `if` branch.

An interesting variation are `if` and `else` branches that terminate with `exit`, like these:

```
... if ... exit then ...
... if ... exit else ... then ...
... if ... else ... exit then ...
```

In the first and the third case, `then` encounters a locked compiler data type heap. Instead of comparing the contents of the data type heap with the contents that was saved at the beginning of the branch, it just restores the data type heap, because the program flow does not continue after `exit`.

In the second case, there's no need to compile an unconditional jump instruction to the end of the `else` branch. Instead of executing ahead, `else` just creates a new object of data type `origin`. Because the compiler data type heap is locked, it is not saved by the constructor. Overwriting the code location with a null address prevents `resolve` within `then` from calculating a non-existent jump offset. Anyway, this case does not really make sense. If you exit the definition at the end of an `if` branch, you don't need an explicit `else`. Instead, you can simply write:

```
... if ... exit then ...
```

Loops

The two words `(branch)` and `(0branch)` are also used for implementing loop structures starting with `begin`. The main difference between conditionals and loops is that conditionals contain forward jumps, while loops contain backward jumps. Here's the general structure of a `begin ... until` loop:

Label	Machine code	Compiler data type heap
	linear flow of execution	
l1	loop body	(x ₁ ... x _n -- x ₁ ... x _n flag)
l2	conditional jump to l1	(x ₁ ... x _n flag -- x ₁ ... x _n)
l3	continue linear flow of execution	(x ₁ ... x _n -- ...)

And here are the definitions of `begin` and `until`:

```
: begin ( -- destination )
  ['] noop compile, new destination ; compile-only

: until ( destination -- )
  ['] >flag compile, dt-drop dup match dt-restore
  ['] (0branch) compile, dup resolve delete ; compile-only
```

`begin` does not need to compile a jump instruction. It just marks the destination of a backward jump. Compiling `noop` does not create any machine code, but it consolidates the compiler's register allocation:

```
noop ( -- )
```

To save the contents of the compiler data type heap, `begin` creates an object of class `destination`, which is another child of class `control-flow`. The class definition looks similar to the one of class `origin`. However, since resolving a forward and a backward jump is different, the virtual method `resolve` is not the same:

```
dt control-flow procreates destination
class destination
  forth definitions
  : destination ( destination -- 1st )
    control-flow ;
  :noname ( destination -- )
    ... ; is resolve
endclass
```

`until` begins by making sure that there's a single-cell item on the stack that can serve as a condition. The compiler data type heap may not be locked. Before comparing the contents of the compiler data type heap with the version saved by `begin`, this condition has to be temporarily removed. After compiling and resolving the conditional jump, the destination can be deleted.

You might ask why `until` can't simply be implemented like this:

```
: until ( destination -- )
  ['] (0branch) compile, dup match
  dup resolve delete ; compile-only \ won't work!
```

The reason why this doesn't work in all cases is that `match` and `?congruent`, might require to do some register juggling in order to equalize the register usage at the beginning and at the end of the loop. Naturally, the register movement instructions have to be compiled *before* the conditional jump instruction. The compilation of `>flag` serves the purpose of moving the condition into the processor's `eflags` register, where it doesn't interfere with any register juggling. The complicated implementation of `until` is thus a consequence of StrongForth's programming model and its optimizations.

The definition of `again` turns out to be more straightforward than the one of `until`, because `again` doesn't expect a condition on the stack:

```
: again ( destination -- )
  dup match ['] (branch) compile,
  dup resolve delete dt-lock ; compile-only
```

Instead of a conditional backward jump, `again` compiles an unconditional backward jump. At the end, `again` locks the compiler data type heap, because any code following an unconditional jump cannot be executed. The general structure of a `begin ... again` loop looks like this:

Label	Machine code	Compiler data type heap
	linear flow of execution	
11	loop body	($x_1 \dots x_n$ -- $x_1 \dots x_n$)
12	unconditional jump to 11	($x_1 \dots x_n$ -- $x_1 \dots x_n$)
	end of execution	

Loops that are constructed with `begin ... while ... repeat` are actually a combination of an infinite loop (`begin ... again`) and a conditional clause (`if ... then`):

Label	Machine code	Compiler data type heap
	linear flow of execution	
l1	loop body (part 1)	($x_1 \dots x_n$ -- ...)
l2	calculate the condition	(... -- $y_1 \dots y_m$ flag)
l3	conditional jump to l6	($y_1 \dots y_m$ flag -- $y_1 \dots y_m$)
l4	loop body (part 2)	($y_1 \dots y_m$ -- $x_1 \dots x_n$)
l5	unconditional jump to l1	($x_1 \dots x_n$ -- $x_1 \dots x_n$)
l6	continue linear flow of execution	($y_1 \dots y_m$ -- ...)

The semantics of `while` is almost identical to the semantics of `if`. In addition, `while` handles the item of data type destination, which was created by `begin`. `swap` takes care that this item stays on top of the data stack:

```
: while ( destination -- origin 1st )
  [compile] if swap ; compile-only
```

At the end of the loop, `repeat` executes first `again` and then `then`. `again` compiles an unconditional backward jump using the control-flow object of class `destination`. Now, only an object of class `origin` is left on the stack. It was created by the `if` within `while` and is consumed by the `then` within `repeat`:

```
: repeat ( origin destination -- )
  [compile] again [compile] then ; compile-only
```

The body of the `begin ... while ... repeat` loop is split into two parts. The first part starts with $x_1 \dots x_n$ on the compiler data type heap and ends with $y_1 \dots y_m$ flag. The second part starts with $y_1 \dots y_m$, because the condition is consumed by (`0branch`), and ends with $x_1 \dots x_n$. After the end of the loop, execution continues with $y_1 \dots y_m$.

Although StrongForth additionally keeps track of the data type information, the definitions of `if`, `else`, `then`, `ahead`, `begin`, `until`, `again`, `while` and `repeat` are not much more complex than the respective definitions in a Forth 2012 implementation. The objects of the two children of the `control-flow` class do most of the necessary jobs.

do Loops

`do` loops cannot be implemented by compiling (`branch`) and (`0branch`). The reason is that `do` loops need to create and discard loop control parameters on entry and on exit of the loop. These two operations are performed by (`do`) and (`loop`), the runtime code compiled by `do` and `loop`, respectively:

```
(do) ( integer 1st -- )
(do) ( address 1st -- )
(loop) ( integer -- flag )
(loop) ( address -- flag )
```

The loop control parameters may be either integers or addresses. All other data types, like logical, token or objects are not *countable* in the sense that each value has a well-defined successor and a predecessor. Just like Forth 2012, StrongForth does not allow double-cell numbers as loop indexes. This is the general structure of a `do` loop:

Label	Machine code	Compiler data type heap
	linear flow of execution	
l1	(do)	(limit start x ₁ ... x _n -- x ₁ ... x _n)
l2	loop body	(x ₁ ... x _n -- x ₁ ... x _n)
l3	fetch loop index	(x ₁ ... x _n -- x ₁ ... x _n index)
l4	(loop)	(x ₁ ... x _n index -- x ₁ ... x _n flag)
l5	conditional jump to l2	(x ₁ ... x _n flag -- x ₁ ... x _n)
l6	continue linear flow of execution	(x ₁ ... x _n -- ...)

In order to access the loop index with the data type specified on entry of the loop, *i* and *j* are defined as locals. The technique is the same as with *r@*. Defining *i* and *j* as static words within one of the vocabularies makes no sense, because their data types might change between different *do* loops. The scope of the loop index is the loop starting with *do* and ending with *loop*. *do* actually uses *create-index* to create the loop index *i* as a local:

```
: create-index ( -- local-definition )
  " i" over over search-local
  if name swap +!
  else drop
  then new local-definition dt-here over params! dup enddef ;
```

create-index considers up to two levels of nested loops. That means, if a local named *i* already exists in the locals vocabulary, the old one is renamed to *j*, before the new loop index is created as *i*. Note the programming trick hidden in *name swap +!*. This phrase takes advantage of the fact that the length of the character string returned by *name* must be 1. And adding 1 to *i* makes it *j*. Some previous Forth implementations defined *k* as a third loop index, thus allowing access to all loop indexes in threefoldly nested loops. Forth 2012 does not specify *k*, and neither does StrongForth. The term *dt-here over params!* assigns the loop index a data type that is obtained from the compiler data type heap after *(do)* has been compiled. You already know this technique from the definitions of *constant*, *(local)* and *(value)*.

Outside of a *do* loop, *i* does not exist, as this example demonstrates:

```
: invisible 10 0 do i . loop i . ; ok
invisible 10 0 do i . loop i ? undefined word
```

An interesting side effect of the loop indexes *i* and *j* being locals is the fact that their value may be changed with *to*. Here's an example:

```
: hiccup ( -- ) ok
  20 0 do i 7 = if 15 to i then i . loop ; ok
hiccup 0 1 2 3 4 5 6 15 16 17 18 19 ok
```

Another consequence of the loop index being a local is that you don't need to take care about them when you exit a loop. This is because locals are stored in the stack frame, which is automatically dissolved when a word returns to its caller. Compiling the Forth 2012 word *UNLOOP* immediately before *exit* in order to get rid of the loop control parameters is not necessary. Actually, something like *unloop* is not even defined in StrongForth. If you want to keep it for compatibility reasons, you may certainly define it as a dummy word. This example works fine:

```
: exit-loop ( -- )
  10 0 do i . i 6 > if exit then loop ; ok
exit-loop 0 1 2 3 4 5 6 7 ok
```

Now, let's have a look at the definition of `do`:

```
dt destination procreates do-destination

: begin-loop ( local-definition -- do-destination )
  new destination cast do-destination dup rot enclosing! ;

: do ( -- do-destination )
  " (do)" evaluate create-index begin-loop ; compile-only
```

You already know that `do` compiles a proper version of `(do)` and that it uses `create-index` to create the loop index as a local. `(do)` is similar to the version of `(>r)` for single-cell items. However, `(do)` transfers both the loop limit *and* the loop index to the stack frame, thus reserving space for two single-cell items instead of only one. The loop index is accessible by `i`, while the loop limit is hidden. `(loop)` and its siblings `(+loop)` and `(-loop)` are the only words that query the loop limit.

`begin-loop` saves the contents of the compiler data type heap in an object of class `destination`, which is then casted to the child class `do-destination`. `do-destination` does not have its own class definition, because it is only needed to ensure the proper syntax of loop structures. `enclosing!` stores the object of class `do-destination` in the local in order to establish a link from the definition of the loop index to the loop it belongs to.

`loop` expects an item of class `do-destination` on the stack, that was produced by `do`. Its definition begins with `?loop`, which tries to find the loop index of the innermost `do` loop. `loop` then compiles `(loop)` with the loop index as a parameter. The data type of the loop index on the compiler data type heap allows the compiler to find a matching version of `(loop)`.

```
: end-loop ( do-destination local-definition -- )
  delete " j" search-local
  if name swap -!
  else drop
  then [compile] until ;

: ?loop ( -- local-definition )
  " i" search-local
  if dup enclosing if exit then
  then -26 throw ;

: loop ( do-destination -- )
  ?loop " i (loop)" evaluate end-loop ; compile-only
```

`(loop)` increments the loop index and then compares it with the loop limit. The resulting flag tells whether the loop shall be continued or not. `end-loop` then removes the loop index from the locals vocabulary, renames the loop index of a potential outer loop back from `j` to `i`, and finally executes `until` to resolve the loop destination. `(0branch)` as compiled by `until` consumes the flag returned by `(loop)`.

At the beginning of this section about `do` loops, two overloaded versions of `(loop)`, one for integers and one for addresses, have been shown. Both of these versions increment the loop index by 1. However, since StrongForth supports address arithmetic, you'd certainly expect that a loop index of data type, say, `address -> single`, is being incremented by 1 cells, i.e., by the number of address units per single cell, on each loop run. And this is exactly what happens. To cover all cases, there are actually 8 overloaded versions of `(loop)`:

```

(loop) ( integer -- flag )
(loop) ( address -- flag )
(loop) ( address -> single -- flag )
(loop) ( address -> double -- flag )
(loop) ( address -> float -- flag )
(loop) ( caddress -- flag )
(loop) ( sfaddress -- flag )
(loop) ( dfaddress -- flag )

```

These words increment the loop index by 1, by one address unit, by the size of a cell, a cell pair or a floating-point number, and by the size of a character, a single-precision and a double-precision floating-point number, respectively. For `(do)`, on the other hand, two versions are sufficient, because this word does not incorporate any address arithmetic:

```

(do) ( integer 1st -- )
(do) ( address 1st -- )

```

?do

In addition to `do`, Forth 2012 specifies `?do` for constructing `do` loops. If `?do` is used instead of `do`, the loop index is compared to the loop limit before the loop body is executed the first time. It is thus possible to implement loops that are executed zero times, whereas loops starting with `do` are always executed at least once.

The definition of `?do` is an extended version of `do`:

```

: ?do ( -- do-destination )
  " over over <> " evaluate [compile] if [compile] do
  tuck swap add-origin ; compile-only

```

`?do` compiles an additional conditional clause that skips the loop if the loop index is equal to the loop limit before the loop body is executed for the first time. Instead of just compiling `(do)`, `?do` compiles the phrase

```
over over <> (0branch) (do)
```

`add-origin` saves the origin of the conditional jump as `'next-origin` in the object of data type `do-destination` that has been created by `begin-loop`. The conditional jump will thus be automatically resolved when the loop is resolved.

+loop and -loop

At the end of a `do` loop, `loop` may be replaced by `+loop` or `-loop`. `-loop` is not specified by Forth 2012, but you can easily guess that its semantics is the same as that of the phrase `negate +loop`. The difference in the definitions of `loop`, `+loop` and `-loop` is only in the runtime codes they compile:

```

: +loop ( do-destination -- )
  ?loop " i (+loop)" evaluate end-loop ; compile-only

: -loop ( do-destination -- )
  ?loop " i (-loop)" evaluate end-loop ; compile-only

```

The runtime codes of `+loop` and `-loop` each expect an additional parameter of data type `integer` that is added to the loop index after each execution of the loop body. This parameter can have positive and negative values. `loop`, on the other hand, always increments the loop index by *one*. But remember that the actual value of *one* depends on the data type of the loop index, because `loop` performs address arithmetic. This also applies to `+loop` and `-loop`. If the loop index is an address, the increment parameter is multiplied by the size of the item the address point to, before it

is added to or subtracted from the loop index. Naturally, the runtime codes of `+loop` and `-loop` have the same set of overloaded versions as `loop`:

```
(+loop) ( integer integer -- flag )
(+loop) ( integer address -- flag )
(+loop) ( integer address -> single -- flag )
(+loop) ( integer address -> double -- flag )
(+loop) ( integer address -> float -- flag )
(+loop) ( integer caddress -- flag )
(+loop) ( integer sfaddress -- flag )
(+loop) ( integer dfaddress -- flag )

(-loop) ( integer integer -- flag )
(-loop) ( integer address -- flag )
(-loop) ( integer address -> single -- flag )
(-loop) ( integer address -> double -- flag )
(-loop) ( integer address -> float -- flag )
(-loop) ( integer caddress -- flag )
(-loop) ( integer sfaddress -- flag )
(-loop) ( integer dfaddress -- flag )
```

The loop exit condition of `LOOP` as specified by Forth 2012 is pretty simple. The loop exits at `LOOP` if the incremented loop index equals the loop limit. In StrongForth, we have to consider the fact that the loop index is incremented according to the rules of address arithmetic. It is possible that the exit condition is never met. Therefore, StrongForth generally applies the Forth 2012 specification for `+LOOP`, which is more general: The loop exits when the loop index crosses the border between the loop limit minus one and the loop limit. Here's an example:

```
: test here -> single 5 + here 2 + -> single do i . loop ;   ok
test 8660562 8660566 8660570 8660574 8660578   ok
here -> single 5 + . 8660580   ok
```

The loop exits correctly, although the loop index never equals the loop limit.

leave

`leave` is a word that does not fit well into the structure of `do` loops, because it does not constitute a syntactical unit together with the other keywords. `leave` is typically nested inside other syntactical structures, as in the following example:

```
: test ( -- ) 10 0
do i . i 7 > if [ .s ] colon-definition do-destination origin
leave then loop ;   ok
```

`.s` has been inserted here in order to visualize the contents of the interpreter data type heap immediately before `leave` is executed. It demonstrates that the loop control flow object of data type `do-destination` is not directly available to `leave`, because `leave` is nested inside a conditional clause. Any assumptions about where to find the loop control flow object on the stack are void, because `leave` may be nested even deeper inside conditional clauses or loops. The only way by which `leave` can get access to the loop control flow object is by locating the definition of the loop index and accessing its `'enclosing-loop` member. This is exactly what `leave` does:

```
: leave ( -- )
  ?loop " (branch)" evaluate enclosing dup match
  dt-lock new origin add-origin ; compile-only
```

`leave` locates the loop index with `?loop` and compiles an unconditional jump to the end of the loop. `?loop` throws an exception if no loop index is available, meaning that `leave` was executed

outside the body of a `do` loop. `enclosing` returns the loop control-flow object, and `match` ensures that the present contents of the compiler data type heap matches the version that has been saved at the beginning of the loop body. Finally, `add-origin` adds the origin of the unconditional jump to the linked list of origins to be resolved at the end of the loop.

Between case and endcase

Conditional clauses built with `case`, `of`, `endof` and `endcase` are more complicated than simple `if ... then ... else` structures. Let's start with an example before digging into the details of the implementation:

```
: .numeral ( unsigned -- )
  case 0 of ." zero" endof
    1 of ." one" endof
    2 of ." two" endof
    drop ." many"
  endcase ; ok
1 .numeral one ok
3 .numeral many ok
```

Remember again, that other than specified by Forth 2012, the case selector has to be dropped explicitly. This means, that the default section may never be omitted. In some cases, the default section just drops the case selector, while in others it performs calculations with it. If you don't like this deviation, you can easily add `postpone drop` to the definition of `endcase`.

This is the general pattern of a `case ... endcase` structure with three different cases:

Label	Machine code	Compiler data type heap
	calculate the case selector	(... -- x ₁ ... x _n s)
11	calculate case 1	(x ₁ ... x _n s -- x ₁ ... x _n s c ₁)
12	over =	(x ₁ ... x _n s c ₁ -- x ₁ ... x _n s flag)
13	conditional jump to 17	(x ₁ ... x _n s flag -- x ₁ ... x _n s)
14	drop	(x ₁ ... x _n s -- x ₁ ... x _n)
15	body 1	(x ₁ ... x _n -- y ₁ ... y _m)
16	unconditional jump to 120	(y ₁ ... y _m -- y ₁ ... y _m)
17	calculate case 2	(x ₁ ... x _n s -- x ₁ ... x _n s c ₂)
18	over =	(x ₁ ... x _n s c ₂ -- x ₁ ... x _n s flag)
19	conditional jump to 113	(x ₁ ... x _n s flag -- x ₁ ... x _n s)
110	drop	(x ₁ ... x _n s -- x ₁ ... x _n)
111	body 2	(x ₁ ... x _n -- y ₁ ... y _m)
112	unconditional jump to 120	(y ₁ ... y _m -- y ₁ ... y _m)
113	calculate case 3	(x ₁ ... x _n s -- x ₁ ... x _n s c ₃)
114	over =	(x ₁ ... x _n s c ₃ -- x ₁ ... x _n s flag)
115	conditional jump to 119	(x ₁ ... x _n s flag -- x ₁ ... x _n s)
116	drop	(x ₁ ... x _n s -- x ₁ ... x _n)
117	body 3	(x ₁ ... x _n -- y ₁ ... y _m)
118	unconditional jump to 120	(y ₁ ... y _m -- y ₁ ... y _m)
119	default body	(x ₁ ... x _n s -- y ₁ ... y _m)
120	continue linear flow of execution	(y ₁ ... y _m -- ...)

The four keywords are all executed during compilation. Data type consistency is handled by objects of two new classes, which are both children of data type `origin`:

```
dt origin procreates of-origin
dt origin procreates endof-origin
```

Neither of these two classes has its own class definition. They just serve to enforce the proper syntax of a `case ... endcase` structure. An object of class `of-origin` represents the contents of the compiler data type heap at the beginning of the `case ... endcase` structure and at the beginning of each body, as marked by `of`. In the diagram above, this is denoted by

$x_1 \dots x_n s$

with s standing for *selector*. The contents of the compiler data type heap at the end of each body, i. e., at the location where `endof` and `endclass` are executed, is represented by an object of class `endof-origin`:

$y_1 \dots y_m$

`case` consolidates the compiler's register allocation without compiling anything:

```
: case ( -- endof-origin of-origin )
  null endof-origin
  ['] noop compile, new origin cast of-origin ; compile-only
```

The contents of the compiler data type heap is saved in an object of class `of-origin`. At the location of each `of`, the compiler data type heap shall have exactly these contents. Additionally, `case` provides an object of class `endof-origin`, but since $y_1 \dots y_m$ is unknown before the first occurrence of `endof`, its value is still null.

`of` expects the two parameters created by `case` on the data stack:

```
: of ( endof-origin of-origin -- 2nd 1st )
  " over swap =" evaluate ['] >flag compile,
  dt-drop dup match dt-restore
  ['] (0branch) compile, code-space here over relocate
  " drop" evaluate swap ; compile-only
```

However, `of` doesn't touch the object of class `endof-origin`. It compiles `over swap =` and `>flag` to compare a duplicate of the case selector with a specific value of the same data type. Evaluating `over swap =` instead of compiling these two words allows providing not only single-cell items as case selector, but also double-cell items and even floating-point numbers. Of course, `swap` is superfluous, but it helps improving the efficiency of the generated code. After executing these words at runtime, the case selector and a flag in the processor's `eflags` register are on top of the data stack. `match` ensures that the contents of the compiler data type heap without the flag are identical to the version that was saved by `case`. Now, `of` compiles a conditional jump and stores its location in the object of class `of-origin`. The object is reused for each case body, updating its location each time. By evaluating `drop`, `of` discards the case selector that is still on the data stack at runtime. Finally, `of` swaps its two parameters, so that `endof` gets them in reverse order. This technique helps keeping the proper syntax by making the interpreter accept neither `of` nor `endcase` before the order of the two objects has been restored by `endof`. `of` and `endof` have to be used alternately. Two `ofs` or two `endofs` in a row are rejected by the compiler.

The definition of `endof` contains a conditional clause that distinguishes the first from all succeeding `endofs` within a `case ... endcase` structure. At the first occurrence, the object of data type `endof-origin` is still null, because the compiler doesn't know yet the contents of its data type heap at the end of a case body:

```

: endof ( of-origin endof-origin -- 2nd 1st )
  dup
  if dup match dt-here
    if [''] (lbranch) compile, dup dt-lock new origin add-origin
    then
  else dt-here
    if drop [''] (lbranch) compile,
      new origin cast endof-origin dt-lock
    then
  then over match over resolve swap ; compile-only

```

Assuming the compiler data type heap is not locked, the first `endof` compiles an unconditional jump to the end of the `case ... endcase` structure and saves the compiler data type heap as denoted by $y_1 \dots y_m$, replacing the null object of data type `endof-origin`. Because the unconditional jump discontinues the linear flow of execution, the compiler data type heap has to be locked. All succeeding `endofs` compare the current contents of the compiler data type heap with those that were saved by the first `endof`, before compiling an unconditional jump. Furthermore, they add a new head to the linked list of the locations of all `endofs`. `endcase` will later resolve all forward references in the linked list.

After joining the two branches of the conditional clause, `endof` restores the compiler data type heap to the versions saved by `case`, denoted by $x_1 \dots x_n$ s, and then resolves the conditional jump. Finally, the order of two objects of data types `of-origin` and `endof-origin` is brought back to the original order. The compiler is now ready to find the next `of` or the default body of the structure.

Note that `endof` uses the long jump variant (`lbranch`) instead of the short jump variant (`branch`), because `case` statements often have a rather big code size.

`endcase` finalizes the `case ... endcase` structure by removing the two control flow objects that were created by `case`:

```

: endcase ( endof-origin of-origin -- )
  delete dup if [compile] then else drop then ; compile-only

```

Using `then`, `endcase` resolves the forward references of all `endofs`. Again, note that `endcase` doesn't compile a final `drop` to get rid of the case selector. In StrongForth, this has to be done manually within the default body.

It is syntactically correct to implement a `case ... endcase` structure that does not contain any `ofs` and `endofs` at all. Semantically, this makes no sense, although it might be what a code generator could produce:

```

: nonsense ( character -- )
  case . space ." default"
  endcase ; ok
char a nonsense a default ok

```

This case is handled by the `if` branch within the definition of `endcase`. If no case body is present, `endof-origin` is still null when `endcase` is encountered.

All case bodies that are located between `of` and `endof` share the same stack diagram:

```
( x1 ... xn -- y1 ... ym )
```

The stack diagram of the default body looks similar, with `s` being the case selector:

```
( x1 ... xn s -- y1 ... ym )
```

There might be cases where $y_1 \dots y_m$ is accidentally equal to $x_1 \dots x_n$ s. Although it is possible to omit the default body in those cases, it seems like a good idea to provide a default body anyway, even if it will not be executed in any scenario. If you believe it's dead code, you can simply throw an exception just in case your assumption might later turn out to be wrong.

One thing still needs to be mentioned. The data type of the case selector is not limited to single-cell items. The only place where the case selector is being processed is within the definition of `of`. Since `over swap =` as well as `drop` are overloaded for double-cell items and for floating-point numbers, `case ... endcase` structures will work with non-single-cell data types as well. However, using floating-point numbers is discouraged, because `=` for floating-point numbers might, caused by rounding differences, not always work as expected.

cast: A Sample Application Of case ... endcase

An interesting application of a `case ... endcase` structure is the implementation of `cast`. You already know that `cast` replaces the data type of the item on top of the data stack with any other data type:

```
true .s flag ok
cast signed .s . signed -1 ok
```

Basically, `cast` uses `dt-drop` to remove the original data type from the data type heap. Then, it obtains a new data type using `dt` and adds it onto the data type heap with `>dt`. Since `cast` is an immediate word, it works both in interpretation and in compilation state. `dt-drop` and `>dt` automatically apply to the data type heap that is associated with the current state:

```
: cast ( -- )
  dt-drop dt-here @ [ dt-prefix invert ] literal and dup
  size 100 * dt rot or dup size rot + 1 dt-allot
  case 0404 of
    dt-drop >dt endof
    0408 of " s>d" evaluate dt-drop >dt endof
    0410 of " s>f" evaluate dt-drop >dt endof
    0804 of " d>s" evaluate dt-drop >dt endof
    0808 of
      dt-drop >dt endof
    0810 of " d>f" evaluate dt-drop >dt endof
    1004 of " f>s" evaluate dt-drop >dt endof
    1008 of " f>d" evaluate dt-drop >dt endof
    1010 of
      dt-drop >dt endof
    \ default \ drop drop -12 throw
  endcase ; immediate
```

As you can see from the definition of `cast`, things are not as easy as describes in the summary. Two more things have to be considered. First, the attributes of the data type of the item on top of the data stack have to be preserved. I. e., except for the prefix attribute, which has to be removed, because the new data type is always a basic data type.

Another thing the implementation of `cast` needs to take into account is the sizes of the old and the new data type. If, for example, the data type of a double-cell item is replaced with the data type of a single-cell item, register assignments or the depth of the data stack would be corrupted. To avoid this problem, size conversions have to be applied whenever the old and the new data type belong to items of different sizes. For example, `cast` has to compile `d>s` when a double-cell item is being cast to a single-cell item, and it compiles `s>d` when a single-cell item is being cast to a double-cell item. `cast` temporarily adds the head of the removed (compound) data type back to the data type heap, in order to be able to compile the proper overloaded conversion word. If, on the other hand, the sizes of the old and the new data type are the same, no data type conversion is necessary. The default body of the `case ... endcase` structure just cleans up the stack and then throws an

exception. Normally, the default body is supposed to be dead code, because all possible combinations of data type sizes are being handled by regular case bodies. But who knows? It is not prohibited to create a new ancestor data type with a size of, say, 7 address units.

Conversions between single-cell integers, double-cell integers and floating-point numbers are performed by these conversion operations:

```
s>d ( integer -- integer-double )
s>d ( signed -- signed-double )
s>d ( single -- double )
s>d ( unsigned -- unsigned-double )
d>s ( double -- single )
d>s ( integer-double -- integer )
d>s ( signed-double -- signed )
d>s ( unsigned-double -- unsigned )
s>f ( signed -- float )
s>f ( single -- float )
d>f ( double -- float )
d>f ( signed-double -- float )
f>d ( float -- signed-double )
f>s ( float -- signed )
```

All of these operations leave a numerical value unchanged, if the value can be represented by the destination data type. For example, `s>d` converts 530017 correctly to 530017., but if `d>s` is applied to -6000000000., the result is obviously incorrect:

```
-6000000000. d>s . -1705032704 ok
```

Floating-point numbers are truncated:

```
pi f>s . 3 ok
3.9e0 f>s . 3 ok
-3.9e0 f>s . -3 ok
```

It has to be considered that floating-point numbers can become extremely large compared even to double-cell numbers, resulting in unpredictable results if they do not fit into the range of an integer number:

```
5.8155e18 f>d . 5815500000000000000 ok
5.8155e40 f>d . -9223372036854775808 ok
```

In the other direction, keep in mind that the resolution of a double-cell number is higher than the one of the significand of a floating-point number. As a consequence, some digits might get lost:

```
-9223372036854775808. d>f s. -9.223372036854776e+018 ok
```

Note also that some of the conversion operators are overloaded. If the source data type is a signed integer, the conversion to a destination operand has to be sign-extended. An unsigned source operand, on the other hand, has to be zero-extended. Furthermore, when converting between single-cell numbers and double-cell numbers, the data type of the destination shall be related to the one of source. That means, a signed number shall stay a signed number, an unsigned number shall stay an unsigned number, and an integer number shall stay an integer number.

17 Qualified Tokens

Saving the Data Type System

StrongForth's data type system is static, which means that the stack effect of each word is already known at compile time. When a word is executed or compiled, the interpreter or the compiler know exactly how to keep track of the respective data type heap. As a consequence, words like `?DUP`, whose stack effect depend on runtime values, cannot be implemented in StrongForth. The stack effect has to be unique.

One of those Forth 2012 words with an ambiguous stack diagram is

```
EXECUTE ( i*x xt -- j*x )
```

`EXECUTE` has an arbitrary number of unspecified input parameters, except for the last, and an arbitrary number of output parameters. From StrongForth's point of view, the situation is much worse than with `?DUP`, because there's almost *no* information about the stack effect. Should `EXECUTE` be removed as well? Certainly not! It is one of the most powerful and versatile words in Forth 2012. What can we do?

When `execute` is being *interpreted*, the value of the execution token is known. By searching the context vocabularies for a word with the given token, we could find out the definition and apply its stack effect to the interpreter data type heap before executing `(execute)`. If a word has no execution token, like many arithmetical and logical operations, we could implement a colon definition as a wrapper. You might remember that `(execute)` was introduced in connection with `interpret`. Actually, `interpret` uses a technique similar to the one explained here. For details, please see the explanation of `interpret` in chapter 14.

However, `execute` demonstrates its real value during compilation. And during compilation, the solution presented here doesn't work. Because the runtime value of an execution token is generally not known at compile time, the stack diagram can't be determined. What is required is a version of `execute` that has the runtime semantics of `(execute)` *and* takes care of the stack effects already at compile time. So, how can we tell the compiler what the stack effect of executing a yet unknown token is?

Obviously, there can be different kinds of tokens, each of which is associated with a specific stack effect. For example, if the token belongs to

```
1+ ( integer -- 1st ) or
2* ( integer -- 1st ) or
cells ( integer -- 1st ),
```

the stack effect of `execute` applied to it is `(integer token -- 1st)`. If the token belongs to `#>`, the stack effect of `execute` is `(number-double token -- address -> character unsigned)`. In order to select the correct stack effect for `execute`, all the compiler needs to know is which *type* of token is actually on the stack.

That's how StrongForth solves the problem. For each kind of token, it defines a subtype of data type token and an overloaded version of `execute` that can be applied to it. For example, in order to execute tokens of words with the stack diagram `(integer -- 1st)`, we have to define a subtype of data type token, which might be called `token(i--1)`, and an overloaded version of `execute`:

```
execute ( integer token(i--1) -- 1st )
```

A subtype of data type `token`, which is associated with a specific stack effect and for which an overloaded version of `execute` exists, is called a *qualified token*. Obviously, the number of possible qualified tokens is unlimited. Predefining all qualified tokens for all words in StrongForth's vocabulary is not practical, and predefining all qualified tokens for user defined words, including those containing user defined data types, is simply impossible. Qualified tokens need to be defined on demand, using the word `)procreates`. Let's begin with an example of how `)procreates` is to be used:

```
: cells' ( integer -- 1st ) cells ; ok
( integer -- 1st )procreates token(i--1) ok
latest prev . token(i--1) ( stack-diagram -- 1st ) ok
latest . execute ( integer token(i--1) -- 1st ) ok
18 ' cells' token cast token(i--1) .s unsigned token(i--1) ok
execute .s . unsigned 72 ok
```

`)procreates` defines two words, a new subtype of `token` and an overloaded version of `execute` that can be applied to it. The phrase

```
' cells' token cast token(i--1)
```

calculates the qualified token of the word `cells'`. But why don't we just use `cells` instead of defining `cells'`, a kind of wrapper around `cells`? The reason is that `cells` does not have a token, because it is neither a colon nor a code definition:

```
' cells token . 0 ok
```

The example code with `cells` instead of `cells'` would simply cause a crash. A special word that returns a valid token for any kind of definition will be presented in the next section.

This is the definition of `)procreates`:

```
: )procreates ( stack-diagram -- )
  dup ?stack-diagram [dt] token procreates
  latest ?data-type [ ' (execute) input-params drop @ ] literal or
  over input-param,
  " execute" ['] (execute)
  dup vtable -> definition (new) definition
  tuck params! enddef ;
```

`)procreates` terminates the specification of the stack diagram that the qualified token it creates shall be associated with. It creates a subtype of data type `token` with the name parsed by `procreates`, and an overloaded version of `execute` that is a copy of `(execute)`. The stack diagram of the new version of `execute` is composed of the stack diagram provided to `)procreates`, and the just created qualified token as an additional input parameter. The execution semantics of `execute` is identical to the one of `(execute)`. The phrase

```
[ ' (execute) input-params drop @ ] literal or
```

copies the data type attributes of the input parameter of `(execute)` to the additional input parameter of the new overloaded version of `execute`. In StrongForth these attributes contain information about the register in which the parameter shall be passed.

At compile time, the qualified token determines the specific overloaded version of `execute` that is to be compiled. Compiling this version of `execute` automatically keeps track of the stack effect of the executed qualified token.

Isn't it rather wasteful to define a specific data type and a specific overloaded version of `execute` each time an application needs to use `execute`? Well, it turns out that the number of different

stack diagrams for qualified tokens is pretty limited. StrongForth itself has only four predefined qualified tokens, that satisfy the needs of its implementation:

```
( -- )procreates (--)  
( unsigned -- )procreates (unsigned--)  
( -- address -> character unsigned )procreates (--string)  
( definition single -- flag )procreates search-criterion
```

The last of these four you already know. It's the data type of the token passed to the virtual member search of class vocabulary, which serves as the additional search criterion. For each of these four qualified tokens, an appropriate version of `execute` is implicitly defined:

```
execute ( (--) -- )  
execute ( unsigned (unsigned--) -- )  
execute ( (--string) -- address -> character unsigned )  
execute ( definition single search-criterion -- flag )
```

Creating Qualified Tokens

In the example of the previous section, we used `cast` to convert the token of `cells'` into a qualified token:

```
' cells' token cast token(i--1)
```

Search criteria, like `no-criterion`, are defined as constants using simple type casts as well:

```
:noname ( definition single -- flag )  
  drop drop true ;  
token cast search-criterion constant no-criterion
```

Both casts work perfectly well. But they might create a feeling of uneasiness. What if you make a mistake by choosing the wrong qualified token? What if you, for example, mistakenly cast the token of a word with a different stack diagram or a null token like the one of `cells` to `token(i--1)`? Executing any version of `execute` with a null token will immediately cause a crash. Executing `execute (integer token(i--1) -- 1st)` with a word whose stack diagram is, say, `(integer integer -- 1st)` will corrupt StrongForth's data type system, because the data stack and the data type heap will no longer be synchronized. Sooner or later, this will result in a crash as well. And if the error shows at some remote location in your code, it will be rather difficult to debug.

Therefore, using `token` together with a simple type cast for creating qualified tokens is not the preferred solution. To ensure that a suitable word is selected, it is strongly recommended to use the dedicated word `>token`:

```
' cells dt token(i--1) >token cast token(i--1)  ok
```

`>token` expects a definition and the data type of a qualified token on the stack. It throws an exception if the definition does not match the stack diagram represented by the qualified token according to the rules of the StrongForth data type system. Because it returns an item of data type token, the result still has to be casted. So, let's keep the qualified token in a value and try it out:

```
value test  ok  
164 test execute .s . unsigned 656  ok
```

Everything looks fine. Using `>token`, qualified tokens can be obtained in a safe way. `>token` contains quite a number of security measures in order to prevent the creation of a qualified token if the definition is not suitable:

```

' 2* dup . 2* ( integer-double -- 1st ) ok
dt token(i--1) >token cast token(i--1)
dt token(i--1) >token ? undefined word
token
' 2* prev dup . 2* ( integer -- 1st ) ok
dt token(i--1) >token cast token(i--1) ok
30 swap execute . 60 ok

```

It was stated earlier in this chapter that `cells` is not a colon definition. Although its token is null, `>token` seems to return a non-null qualified token. The reason for this to work is that `>token` automatically compiles a chunk of code to generate a valid qualified token, if necessary. The same thing happens if matching the stack diagrams of the definition and the qualified token requires some register shuffling. And it works in either compilation or interpretation state.

Now, here's the definition of `>token` along with two supporting words:

```

:noname ( definition single -- flag )
  cast data-type-attributes >data-type
  over input-params dt-stripped + @ dt-prefix and =
  swap vtable [ ' (execute) vtable ] literal = and ;
token cast search-criterion constant execute-criterion

: ?qualified-token ( data-type -- definition )
  >attributes " execute" rot
  execute-criterion search-all invert
  if -265 throw
  then ;

: >token ( definition data-type -- token )
  ?qualified-token
  true new stack-diagram tuck params-stripped, false (>token) ;

```

`>token` begins with using `?qualified-token` to find the overloaded version of `execute` that is associated with the given data type. For this purpose, `?qualified-token` has to perform a vocabulary search for a word named `execute`. The additional search criterion `execute-criterion` requires that the last input parameter of the definition is the qualified token to be searched for, and that its virtual method table is identical to the one of `(execute)`.

`>token` then creates a stack diagram that is identical to the one of the found version of `execute`. I. e., except for the last input parameter, which is the qualified token to be executed. This is the stack diagram the first input parameter of `>token` has to match. The details of this matching and of the possible creation of a code chunk are hidden in `(>token)`. `(>token)` is a rather complex word that uses a few factored words, which are deleted after its definition is completed:

```

: ?ahead ( -- origin )
  dt-here
  if [compile] ahead
  else null origin
  then ; immediate

: ?then ( origin -- )
  dup
  if [compile] then
  else drop
  then ; immediate

: compile-token ( definition -- )
  compile, [compile] exit ;

```

```

: (>token) ( definition stack-diagram flag -- token )
  locals( def sd f )
  latest #locals @ state @ ] code-space here
  [compile] ?ahead
  new code-definition cast colon-definition begin-compilation sd )
  def catch compile-token dup f or
  if dt-lock rot [compile] ?then
    rot code-space here - code-space allot null token
  else over ?alias rot [compile] ?then
    rot drop over token
  then
  rot delete rot state ! rot #locals ! rot to latest
  swap throw ;

' ?ahead delete
' ?then delete
' compile-token delete

```

Because (>token) works in interpretation state *and* in compilation state, and it might need to generate a code chunk for the qualified token, it has to save the status of a potentially ongoing compilation. `latest`, `#locals`, and `state` are saved on the data stack and restored at the end of (>token). `?ahead` compiles an unconditional jump over the code chunk and saves a copy of the current compiler data type heap, unless the compiler data type heap is locked. (>token) then starts a new colon definition for the code chunk with the stack diagram passed to (>token).

Next, (>token) tries to compile the code chunk including `exit`, which checks the output parameters. Exceptions during compilation are being caught by `catch`, which in StrongForth parses instead of expecting an execution token on the stack. An exception means that the definition does not match the given stack diagram. In this case, the code chunk is being discarded by restoring the code space pointer. Locking the compiler data type heap and then executing `?then` restores the compiler data type heap. A null token will be returned.

(>token)'s third parameter of data type `flag` is usually `false`. If it is `true`, the code chunk is discarded and a null token is returned independently of whether the definition can be compiled. In this case, (>token) serves the sole purpose to check whether the definition matches the given stack diagram.

If compiling the definition succeeds and the `flag` parameter is `false`, `?alias` may eliminate the code chunk if it consists only of a single call instruction. This is just an optimization. Again, `?then` restores the compiler data type heap, which is already locked after the compilation of `exit` was successful. However, the code chunk, if not eliminated by `?alias`, remains untouched, so the saved code space pointer can be dropped. The execution token of the code chunk will be returned.

Finally, the definition of the code chunk can be deleted. The status of the potentially ongoing compilation is restored. (>token) throws the exception that could have been caused by `compile-token` and returns the token. The three factored words `?again`, `?then` and `compile-token` are no longer required. Only their compiled machine code remains.

Selecting Overloaded Definitions

In order to apply `>token`, we need to supply it with an object of data type definition. For non-overloaded words like `cells`, this is an easy task:

```

' cells .s . definition cells ( integer -- 1st ) ok

```

For words with few overloaded words in a row, `prev` can be used to get access to the previous version:

```
' 2* . 2* ( integer-double -- 1st ) ok
' 2* prev . 2* ( integer -- 1st ) ok
```

Things get more difficult if the overloaded versions of a word are not located adjacent to each other in the vocabulary, or if they are distributed over various vocabularies. Or consider a word with more than a few overloaded versions, like `+` or, even worse, `rot` with its 27 overloaded versions. Using `prev` won't be practicable in those cases. The problem is that `'` uses no additional search criterion. It always finds the first occurrence of a word with the parsed name in the context vocabularies.

Wouldn't it be nice to have an alternative to `'` than allows specifying the stack diagram of the word as an additional search criterion? Here it is:

```
: )' ( stack-diagram -- definition )
  dup ?stack-diagram dup parse-name rot
  identity-criterion search-context rot delete invert
  if -13 throw
  then ;
```

And this is how to use it:

```
( integer integer -- 1st )' + . + ( integer integer -- 1st ) ok
( address address -- 1st )' + .
( address address -- 1st )' + ? undefined word
definition
```

`)'` finishes the stack diagram provided to it on the data stack, parses a name and then performs a context vocabulary search. As an additional search criterion, it uses `identity-criterion` with the stack diagram as its parameter. Finally, it deletes the stack diagram.

`identity-criterion` uses `dt-compare` to compare the input and output parameters of a definition with those of the stack diagram passed to it as the single-cell parameter. It returns true if and only if both stack diagrams are one-to-one identical with respect to identifier/offset and the prefix and reference attributes.

```
:noname ( definition single -- flag )
  cast stack-diagram
  over over input-params rot input-params dt-compare
  rot rot output-params rot output-params dt-compare and ;
token cast search-criterion constant identity-criterion
```

Example: A Simple Jump Table

At the end of this chapter, we'll have a look at a small example that demonstrates how a jump table can be implemented with >token:

```
( unsigned 1st -- 1st )procreates (u1--1) ok
create jump-table ( -- address -> (u1--1) ) ok
( integer integer -- 1st )' + dt (u1--1) >token , ok
( integer integer -- 1st )' - dt (u1--1) >token , ok
( integer unsigned -- 1st )' * dt (u1--1) >token , ok
( unsigned unsigned -- 1st )' / dt (u1--1) >token , ok
( unsigned unsigned -- 2nd )' mod dt (u1--1) >token , ok
( integer 1st -- 1st )' min dt (u1--1) >token , ok
( integer 1st -- 1st )' max dt (u1--1) >token , ok
( single -- )' drop dt (u1--1) >token , ok
( single single -- 2nd )' nip dt (u1--1) >token , ok
: test ( unsigned 1st -- )
  9 0 do over over jump-table i + @ execute . loop drop drop ; ok
16 7 test 23 9 112 2 2 7 16 16 7 ok
```

The jump table contains tokens of various arithmetic operations to be performed on two items of data type unsigned, and returning a result of the same data type. test is a loop that iterates over the jump table, executing all qualified tokens with the same parameters.

Most of the arithmetic operations are more general than required in order to match the qualified token. For example, * allows an item of data type integer as its first input parameter. But of course, it still matches the qualified token, because data type unsigned is a subtype of data type integer.

Even more interesting is the fact that drop (single --) matches the qualified token (u1--1). What can be seen from this example is that it is not required for both stack diagrams to be congruent in some sense. It is merely sufficient that the word has the stack effect the qualified token demands. Applying drop to two items of data type unsigned results in one item of this data type left, so the resulting stack effect is identical to that of the arithmetic operations, and it satisfies the requirements of the qualified token. Actually, if the input parameter match does not consume all of execute's input parameters, the remaining input parameters simply stay on the data type heap and become output parameters. Of course, this only works under the condition that execute's output parameter list starts with the same data types as its input parameter list. For example, the qualified token

```
( integer flag -- signed )procreates (if--s)
```

won't allow any word that doesn't consume both input parameters to match.

18 Implementing Object Orientation

You've already learned how to derive new classes from existing ones. Now it is time to have a closer look at class definitions and how StrongForth's implementation of object orientation works.

Creating the Virtual Method Table

Within a class definition, the data type attributes of the class need to be readily available to quite a number of words. That's why they are being stored in a globally accessible value:

```
null class-attributes value this-attributes
```

Furthermore, several words reference the attributes of the parent class and the virtual method table of both the class itself and the respective parent class:

```
: parent-attributes ( -- class-attributes )
  this-attributes 'parent @ cast class-attributes ;

: this-vtable ( -- vtable )
  this-attributes 'vtable @ ;

: parent-vtable ( -- vtable )
  parent-attributes 'vtable @ ;
```

Next, let's see how a virtual method table is being created. Remember that the virtual method table of a class consists of the size in address units of its objects and the execution tokens of all of its virtual methods:

virtual method table

object size
virtual method execution tokens

If the virtual method table already exists, `?create-vtable` just updates the size in address units of the objects of the class. Otherwise, it creates a new virtual method table in the data space. It initializes the object size, copies the execution tokens from the virtual method table of the parent class and finally initializes the execution tokens of added virtual methods with the token of the word `unassigned`. `unassigned` throws an exception. When defining a new virtual method, the execution token of `unassigned` in the virtual method table will be overwritten.

```
: parent-tokens, ( -- )
  parent-attributes #vtable @ 1
  ?do i parent-vtable 'virtual @ data-space ,
  loop ;

: unassigned-tokens, ( -- )
  this-attributes #vtable @ parent-attributes #vtable @
  ?do [ ' unassigned token ] literal data-space ,
  loop ;
```

```

: ?create-vtable ( object-size -- )
  this-vtable
  if this-vtable 'object-size !
  else data-space align data-space here cast vtable
    this-attributes 'vtable !
    data-space , parent-tokens, unassigned-tokens,
  then ;
' parent-tokens, delete
' unassigned-tokens, delete

```

Copying the execution tokens from the virtual method table of the parent class and initializing the execution tokens of the added virtual methods are factored out into `parent-tokens`, and `unassigned-tokens`,. Since these two words are no longer required, they can be deleted after the definition of `?create-vtable` is done.

The Class Definition

A class definition for an already existing data type, which needs to be derived from data type object, starts with `class` and ends with `endclass`:

```

: class ( -- vocabulary object-size )
  current @
  dt >class-attributes to this-attributes
  parent-vtable
  if parent-attributes #vtable @ this-attributes #vtable !
    parent-attributes 'last @ protected-vocabulary last!
    null definition private-vocabulary last!
    parent-vtable 'object-size @
  else -267 throw null object-size
  then ;

: endclass ( vocabulary object-size -- )
  ?create-vtable
  protected-vocabulary last this-attributes 'last !
  this-attributes #friends @
  if protected-vocabulary last private-vocabulary last 0<>
    if private-vocabulary first link private-vocabulary last
      then new vocabulary tuck last! this-attributes 'vocabulary !
  then current !
  [compile] protected [compile] ignore
  [compile] private [compile] ignore
  ignore-friends ;

```

Class definitions often use additional context vocabularies like `protected` and `private`, and switch the current compilation vocabulary back and forth. That's why the current compilation vocabulary is saved by `class` and restored by `endclass`. Apart from that, `class` begins with assigning a value to `this-attributes`. If the parent class is not yet defined, `class` throws an exception. Otherwise, it continues with initializing the size of the virtual method table and the last definition of its `protected` vocabulary with the values from the parent class. The `private` vocabulary is empty from the start. The object size in address units, which is placed on the stack, also begins with the value of the parent class.

`endclass` stores a pointer to the last definition of the `protected` vocabulary in the class attributes, to make the vocabulary available to derived classes. But it also has to make the

definitions in the protected vocabulary and the private vocabulary available to those classes that were declared friends. For this purpose, `endclass` creates a new vocabulary and stores a pointer to it in the attributes of the class. If the private vocabulary is empty, the new vocabulary is identical to the protected vocabulary, i. e., its last definition is the same. If the private vocabulary is not empty, `endclass` links the first definition of the private vocabulary to the last definition of the protected vocabulary, thus creating a combined vocabulary, whose last definition is the same as the last definition of the private vocabulary. Finally, `endclass` removes the private and protected vocabularies as well as the vocabularies of all friend classes from the search order.

`ignore-friends` walks through the linked context vocabulary list, removing all vocabularies that originate from friend classes.

```
:noname ( definition single -- flag )
  cast vocabulary swap data-type?
  if dup object?
    if >class-attributes 'vocabulary @ =
      else drop drop false
    then
  else drop drop false
  then ;
token cast search-criterion constant friend-criterion

: friend? ( vocabulary -- data-type flag )
  dup
  if null caddress -> character 0 rot friend-criterion search-all
    if data-type?
      else drop 0. cast data-type false
    then
  else drop 0. cast data-type false
  then ;

: ignore-friends ( -- )
  context @
  begin dup 0<>
  while dup friend? nip
    if dup next swap >context [compile] ignore
    else next
  then
  repeat drop ;
```

`friend?` finds out whether a vocabulary belongs to a friend class. With the additional search criterion `friend-criterion`, it searches all vocabularies for a class whose friend vocabulary matches the given vocabulary. If it finds one, it returns the data type of this class and a true flag. Otherwise, it returns a null data type and a false flag.

Friend Classes

A class can declare other classes to be friends. This means, these other classes are being granted access to the private and protected words of the class.

To declare a list of classes as friends, you have to use this phrase:

```
friends( <class1> <class2> ... <classn> )
```

`friends (` may only be used once within a class definition. It parses the names of the friend classes upto the closing right parenthesis, storing pointers to their class attributes in a friendship list in the data space. The address of the list and the number of list items are stored in the class attributes `'friends` and `#friends`, respectively:

```
: friends( ( object-size -- 1st )
  this-attributes #friends @ if -272 throw then
  data-space here -> class-attributes this-attributes 'friends !
  0
  begin parse-name over over " )" compare
  while ?data-type >class-attributes data-space , 1+
  repeat drop drop this-attributes #friends ! ;
```

The input parameter of data type `object-size` is returned unchanged as the output parameter. Its presence ensures that `friends (` is always used within a class definition.

The class that has in this way been declared a friend may request access to the private and protected words by using `access`, followed by the class name:

```
access <class>
```

`access` parses a class name and then searches the friendship list of the class for a match with the class attributes of its own class. This means, it checks whether its own class has really been declared a friend of the class whose private and public vocabularies it wants to access. If this check succeeds, `access` includes the requested vocabulary in the search order:

```
: access ( object-size -- 1st )
  dt >class-attributes dup 'friends @ over #friends @ 0
  ?do dup @ this-attributes =
    if drop 'vocabulary @ >context exit
    then 1+
  loop drop drop -273 throw ;
```

A demonstration of how to use `access` and `friends (` is included in chapter 13.

Data Members

Data members are defined in the similar way as variables. Each data member is an object of class `member-definition`:

```
dt definition procreates member-definition
class member-definition
  protected definitions
  null unsigned member 'offset
  forth definitions
  : member-definition ( unsigned caddress -> character unsigned
    member-definition -- 5 th )
    definition locals( this )
    address-unit-bits / 'offset ! this ;
  :noname ( compiler-workspace member-definition -- )
    ... ; is (compile)
endclass
```

The protected member `'offset` contains the offset in address units with respect to the address of the object. The constructor of class `member-definition` calculates the offset by dividing its first parameter, which is the offset in bits, by the number of bits per address unit. Really, an offset in bits? The reason is that StrongForth can also define bit fields as members. For example, you could define bits 0 to 5 as one data member, and bit 6 and bit 7 as two one-bit data members. Together, the three data members occupy the same byte within the memory space allocated for objects of the class. You'll learn more about bit fields and bit field members in chapter 30.

The stack diagram of a data member looks like this:

```
( <class> -- <addr> -> x )
```

`<class>` is the data type of the class the member belongs to. `x` is the data type of the member and `<addr>` is the data type of its address, i. e., either `address`, `caddress`, `sfaddress` or `dfaddress`. An overloaded version of `params!` constructs this stack diagram. Its first parameter is the basic data type of the address `<addr>` and its second parameter is the address of the compound data type `x` of the new member. The data type `<class>` of the presently defined class can be obtained from the global value `this-attributes`:

```
: params! ( data-type address -> data-type member-definition -- )
  swap rot dt-prefix or state @ new stack-diagram
  this-attributes >data-type
  [ ' 'parent input-params drop @ ] literal or
  over param, -- tuck param, tuck params, swap params! ;
```

Now let's begin with a simple example. Say, we want to define an array of four single-cell member variables of data type `unsigned`:

```
null unsigned 4 members
```

An item of data type `object-size` is already on the stack, because `members` is always used within a class definition. Its value is changed by `members` and returned on the stack. You might expect that the implementation of `members` is similar to the one of `variables`. However, it looks rather different:

```
: (member) ( unsigned object-size data-type -- 2nd )
  over parse-name new member-definition
  dup rot dt-here rot params!
  enddef swap + ;

: members ( object-size single unsigned -- 1st )
  nip
  [ address-unit-bits cells ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] address (member) ;
```

Because `members` are not filled with given values, the second input parameter is simply dropped. It is a dummy parameter. However, its data type is required for constructing the stack diagram of the member. A part of `members` has been factored out to `(member)`, because this part can be reused in other words that define data members.

A cell has `address-unit-bits cells` bits. At the beginning, `members` aligns the offset parameter of data type `object-size` to the first bit of a cell. At the end, it increments the offset by the number of bits reserved, which is the number of array items multiplied by the number of bits in a cell. After a new object of data type `member-definition` has been created, `(members)` uses `params!` to construct the stack diagram. `dt-here` returns a pointer to the compound data type of the dummy parameter.

members for double-cell items can build up on members for single-cell items, because the value of the dummy parameter does not matter and because the alignment is the same in both cases:

```
: members ( object-size double unsigned -- 1st )
  2* swap d>s swap members ;
```

Similar to defining variables, StrongForth provides a full set of words for defining data members. These words differ from the two previously presented versions of members in the size of the members, in their alignment, and in the data type of the address:

```
: cmembers ( object-size single unsigned -- 1st )
  nip
  [ address-unit-bits chars ] literal * swap
  [ address-unit-bits chars ] literal aligned
  [dt] caddress (member) ;

: members ( object-size float unsigned -- 1st )
  nip
  [ address-unit-bits floats ] literal * swap
  [ address-unit-bits cells 2/ ] literal aligned
  [dt] address (member) ;

: dfmembers ( object-size float unsigned -- 1st )
  nip
  [ address-unit-bits cells 2* ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] dfaddress (member) ;

: sfmembers ( object-size float unsigned -- 1st )
  nip
  [ address-unit-bits cells ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] sfaddress (member) ;
```

For defining single data members instead of arrays of data members, it is recommended to use the these definitions:

```
: member ( object-size single -- 1st )
  1 members ;

: member ( object-size double -- 1st )
  1 members ;

: cmember ( object-size single -- 1st )
  1 cmembers ;

: member ( object-size float -- 1st )
  1 members ;

: sfmember ( object-size float -- 1st )
  1 sfmembers ;

: dfmember ( object-size float -- 1st )
  1 dfmembers ;
```

Virtual Methods

Some of StrongForth's predefined classes have virtual methods. The most frequently used one is `delete`. Since this virtual method is defined in class `object`, it can be invoked with all derived classes as well. Some classes define special versions of `delete`, because they have to perform

additional actions on deletion, like freeing allocated dynamic memory. All virtual methods are objects of class `virtual-definition`:

```
dt definition procreates virtual-definition
class virtual-definition
  protected definitions
  null unsigned member 'index
  forth definitions
  : virtual-definition ( unsigned caddress -> character unsigned
    virtual-definition -- 5 th )
    definition tuck 'index ! ;
  : index ( virtual-definition -- unsigned )
    'index @ ;
  :noname ( compiler-workspace virtual-definition -- )
    ... ; is (compile)
endclass
```

The only additional member of class `virtual-definition` is the index of the execution token in the virtual method table. `'index` is being initialized by the constructor and can be obtained with the public `index` method. The code `(compile)` compiles reads the execution token with the given index from the virtual method table and executes it.

All virtual methods that are added to a class with respect to its parent have to be defined with `virtual` before the virtual method table is created. The virtual method table is created with `?create-table` when the first virtual method is assigned an execution token. If the class definition contains no such assignment at all, the virtual method table is being created by `endclass`.

`virtual` throws an exception if the virtual method table already exists. Otherwise, it increments the size `#vtable` of the virtual method table and creates a new virtual method with the new size as the `index` parameter:

```
: virtual ( object-size -- 1st )
  this-vtable
  if -276 throw
  else this-attributes #vtable dup @ dup 1+ rot !
    parse-name new virtual-definition enddef
  then ;
```

You already know how to apply `is` within a class definition. To ensure that this word is used only with a class definition, it expects an object of data type `object-size` as the first input parameter, and returns it unchanged as its output parameter. At the beginning, `is` creates the virtual method table, if it does not yet exist:

```
: is ( object-size definition -- 1st )
  over ?create-vtable
  parse-name [ dt virtual-definition vtable ] literal
  vtable-criterion search-context
  if cast virtual-definition tuck >token
    swap index this-vtable 'virtual !
  else drop drop -270 throw
  then ;
```

is searches the context vocabularies for a virtual definition with the parsed name. If it finds one, it uses a special version of `>token` to check whether the stack diagram of the word provided as its second input parameter matches the virtual definition. It then stores the word's token at the correct position in the virtual method table. If it does not find a virtual definition with the parsed name, it throws an exception.

But why do we need a special overloaded version of `>token`? The first reason is that the stack diagram of the definition whose execution token shall be returned is not matched against a qualified token, but against a virtual definition. To understand the second reason, let's have a look at the assignment of `refill` within the class definition of class `terminal-input-stream`:

```
:noname ( terminal-input-stream -- flag )
  locals( this )
  0 >in ! 'buffer @ /buffer @ accept #buffer ! true ; is refill
```

And this is the definition of the virtual method `refill` within the class definition of class `input-stream`:

```
virtual refill ( input-stream -- flag )
```

The new version does not match the virtual method, because it only applies to objects of class `terminal-input-stream`, not to objects of its parent class `input-stream`. This means, the stack diagram provided to `(>token)`, i. e., the stack diagram of the virtual method, has to be slightly changed for the match to succeed. `params-virtual`, creates a stack diagram that is the same as the one of a given virtual method, but it exchanges the last input parameter with the data type of the class presently being defined. In the example, this is `terminal-input-stream`.

```
: params-virtual, ( definition stack-diagram -- )
  locals( sd ) dup input-params dt-stripped sd params-alias,
  this-attributes >data-type over input-params dt-stripped + @
  [ dt-prefix dt-reference or invert ] literal and or
  sd param, sd -- swap output-params rot params-alias, ;

: >token ( definition virtual-definition -- token )
  true new stack-diagram tuck params-virtual, false (>token) ;
```

With this version of `>token`, assigning an execution token to a virtual method is possible. In many cases, the new word is defined with `:noname`. However, it is also possible to assign an existing, named word to a virtual method, using `'` or `)'` or some other means to get hold of it.

Early Binding

When executing a virtual method, the actual execution token is obtained from the virtual method table of the object that is on top of the stack. The class this object belongs to is generally not known at compile time. For example, if you compile `refill` for an object of data type `input-source`, it is possible that the actual object at runtime is of data type `terminal-input-source`, which means that the version of `refill` for `terminal-input-source` is executed instead of the version for `input-source`. This behaviour is called *late binding*, because the decision of which method to execute is made at runtime, and runtime is *later* than compile time.

In some situations, it is desirable to determine the method to be executed already at compile time. This is called *early binding*. One of these situations is `delete` in the class definition of class `stack-diagram`:

```
:noname ( stack-diagram -- )
  dup 'saved-state @ state ! [parent] delete ; is delete
```


After restoring the value of `state`, the object to be deleted has to execute the version of `delete` that belongs to the parent class, which is class object. Writing `delete` instead of `[parent]` `delete` would end up in an endless recursion. `[parent]` `delete` means that it's not the virtual method `delete` that is to be executed at runtime, but the instance of `delete` that belongs to the parent of the presently defined class.

Similarly,

```
[bind] <class> <method >
```

with parsed `<class>` and `<method>` does not compile a virtual method, but the code of the version of `<method>` that belongs to an arbitrary class `<class>`.

The definitions of `[parent]` and `[bind]` both depend on `(bind)`, which expects the class attributes of the parent class or of `<class>` as an input parameter, respectively:

```
:noname ( definition single -- flag )
  over vtable [ dt virtual-definition vtable ] literal =
  if match-criterion execute
  else drop drop false
  then ;
token cast search-criterion constant virtual-match-criterion

: (bind) ( class-attributes -- )
  latest swap
  parse-name over over
  false virtual-match-criterion search-context
  if nip nip true
  else drop " this" evaluate
    false virtual-match-criterion search-context
  then
  if cast virtual-definition tuck dup index rot
    over over #vtable @ <
    if 'vtable @ 'virtual @ new code-definition
      state @ new stack-diagram
      rot over params-alias, over params! dup compile, delete
    else drop drop drop drop -12 throw
    then
  else drop drop -270 throw
  then to latest ;
```

`(bind)` parses the name of a word. It then tries to find a virtual method with that name in the context vocabularies whose stack diagram matches the data type heap. If it doesn't succeed, it tries again after evaluating `this`. `(bind)` throws an exception if both of these tries fail. Otherwise, the next check is whether the index of the virtual method is less than the size of the virtual method table of the class. `(bind)` continues defining a temporary code definition based on a dedicated constructor:

```
: code-definition ( virtual-definition token code-definition
  -- 3rd )
  locals( this ) erase
  this 'token ! 'attributes @ 'attributes ! this ;
```

This constructor assigns the temporary code definition the execution token from the virtual method table and the attributes from the found virtual method. The code definition gets the same stack diagram as the virtual method. After the temporary code definition has been compiled, it can be deleted.

Note that `(bind)` has to save and restore `latest`, because this value is changed by defining the temporary code definition. `(bind)` is always executed during compilation.

With `(bind)`, it is easy to define `[bind]` as well as `[parent]`:

```
: [bind] ( -- )
  dt >class-attributes (bind) ; compile-only

: [parent] ( -- )
  parent-attributes (bind) ; compile-only
```

19 Deferred Definitions

As a general rule, every Forth word has to be defined before it can be either compiled or executed. However, there are cases in which it is desirable to compile a word before it is actually defined. Compiling a word that has not yet been defined is called a *forward reference*. Forth 2012 specifies the words `DEFER`, `DEFER@`, `DEFER!` and `ACTION-OF` to allow defining words that can be compiled before they are being defined, and whose semantics can be changed, affecting all uses of these words, even if they are already compiled. This feature is usually implemented with created definitions that store an execution token in their data fields and whose runtime code fetches the token and executes it.

In StrongForth, the stack diagrams of deferred words have to be considered. And of course, deferred words are objects of a specific class:

```
dt definition procreates deferred-definition
class deferred-definition
  protected definitions
  null unsigned cmember #pop-params
  null address -> token member 'deferred
  forth definitions
  : defer@ ( deferred-definition -- token )
    'deferred @ @ ;
  : defer! ( token deferred-definition -- )
    'deferred @ ! ;
  : deferred-definition ( address -> token caddress -> character
    unsigned deferred-definition -- 6 th )
    definition tuck 'deferred ! ;
  :noname ( compiler-workspace deferred-definition -- )
    ... ; is (compile)
endclass
```

Of the two protected members, you may ignore the first one. `#pop-params` determines the number of cells to be popped off the stack after compiling an *MSVCRT* library function. All these functions are deferred words. The second data member `'deferred` contains the address of the execution token of the deferred word. `(compile)` compiles an indirect call instruction with the content of `'deferred` being the parameter. The constructor of class `deferred-definition` extends the semantics of the constructor of class `definition` by initializing this data member.

The methods `defer@` and `defer!` perform read and write access to the execution token, respectively. But why do we need an additional level of indirection? Wouldn't it be easier to directly keep the execution token in objects of class `deferred-definition` instead of storing the execution token in a cell at a different location? Yes, that would work as well. On the other hand, the execution token might get changed at runtime. StrongForth's philosophy is not to store anything needed at runtime in objects of class `definition` or any class derived from it, because definitions may be deleted when they are no longer needed. Deferred definition shall still work properly even after their created definitions do not exist any longer.

`defer` creates a new object of class `deferred-definition`. The actual execution token is stored in the data space. It is being initialized with the token of `unassigned`, which does nothing else but throwing an exception:

```
: defer ( -- )
  data-space here -> token parse-name new deferred-definition
  [ ' unassigned token ] literal data-space , enddef ;
```

StrongForth comes with four predefined deferred definitions:

```
defer prompt ( -- )
defer error ( signed -- )
defer accept ( caddress -> character integer -- 3rd )
defer (quit) ( -- )
```

Note that all three have explicit stack diagrams. `prompt` is executed by `quit` to display the system prompt. `error` handles exceptions if no exception handler is present. These two words will be explained in detail the next chapter. `accept` is a deferred definition, because the predefined semantics does not allow any command line editing apart from backspace:

```
msvcrt

:noname ( caddress -> character integer -- 3rd )
  cast unsigned locals( buf limit ) 0
  begin key dup 13 cast character \ cr \ <>
  while dup 8 cast character \ bs \ =
    if drop dup
      if 1- 8 cast character \ bs \ _putch bl _putch drop _putch
      else 7 cast character \ bel \ _putch
      then drop
    else over limit < over bl >= and
      if dup emit over buf swap + ! 1+
      else drop 7 cast character \ bel \ _putch drop
      then
    then
  repeat drop space cast integer ; is accept

ignore
```

You can chose to define a more sophisticated semantics and assign its execution token to the deferred word `accept`.

Finally, `(quit)` is executed within `quit`. This deferred definition may be used to perform system initialization tasks qhose specification needs to be deferred.

`action-of` is a parsing word that searches the context vocabulary for a deferred word with the given name. If it does not find a definition with this name or if it is not a deferred definition, it throws an exception. Since `action-of` is specified to work both in interpretation state and in compilation state, StrongForth defines two different overloaded version of it. `parse-deferred-definition` has been factored out, because the parsing and the vocabulary search is used by both versions. Note that the dictionary search is done with an additional search criterion that selects definitions with the virtual method table of a deferred definition:

```
: parse-deferred-definition ( -- deferred-definition )
  parse-name [ dt deferred-definition vtable ] literal
  vtable-criterion search-context invert
  if -269 throw
  then cast deferred-definition ;
```

```

: action-of ( -- token )
  parse-deferred-definition defer@ ; execute-only

: action-of ( -- )
  parse-deferred-definition [literal]
  ['] defer@ compile, ; compile-only

```

Other than specified by Forth 2012, `is` does not expect an execution token on the stack. Because it is most frequently used together with `'` or with `:noname`, it rather expects an item of data type definition. It reuses `parse-deferred-definition` to select a deferred definition, and an overloaded version of `>token` tailored for deferred definitions. Again, since `is` is supposed to work in interpretation state as well as in compilation state, StrongForth provides two different versions of it:

```

: >token ( definition deferred-definition -- token )
  true new stack-diagram tuck params-alias, false (>token) ;

: is ( definition -- )
  parse-deferred-definition tuck >token swap defer! ;

: is ( -- )
  parse-deferred-definition [literal]
  " tuck >token swap defer! " evaluate ; compile-only

```

Typical applications of forward references are recursions that extend over more than one word. If word `a` uses word `b` and word `b` in turn uses word `a`, none of these two words can be defined first:

```

: a ( ... -- ... ) ... b ... ;
: b ( ... -- ... ) ... a ... ;

```

Using execution tokens is a possible solution for this problem:

```

( ... -- ... )procreates t
null t value (b)
: a ( ... -- ... ) ... (b) execute ... ;
: b ( ... -- ... ) ... a ... ;
' b dt t >token cast t to (b)

```

Deferred words provide a more elegant solution:

```

defer b ( ... -- ... )
: a ( ... -- ... ) ... b ... ;
:noname ( ... -- ... ) ... a ... ; is b

```

The definition `b` is *deferred* until after `a` has been defined. `defer` creates a deferred definition that executes a yet to be defined word. `is` calculates the token of this word and assigns it to the deferred definition `b`. Of course, `is` checks that `b` is a deferred definition and that `b` and the word have exactly the same stack diagrams.

Here's a simple example that uses `defer` and `is` to resolve a recursion that extends over two words:

```

defer flip ( unsigned -- 1st ) ok
: flop ( unsigned -- 1st )
  ." flop " dup 1 > if 1- flip then ; ok
:noname ( unsigned -- 1st )
  ." flip " flop ; is flip ok
3 flip flip flop flop flop flop flop ok
3 flop flop flip flop flop flop ok

```

20 Exception Handling

catch in StrongForth

Exception handling in Forth 2012 is based on the two words `CATCH` and `THROW`. `CATCH` creates an exception frame and executes a token. If no exception was thrown during execution of the token, `CATCH` removes the exception frame and continues execution after pushing zero as a no-error code onto the stack. If, on the other hand, an exception is thrown before the execution of the token is finished, the exception frame is removed by `THROW`. `THROW` ensures that the flow of execution is continued at the same point where `CATCH` would have returned if the execution of the token had terminated normally, with the error code on the stack.

The Forth 2012 stack diagram of `CATCH` looks quite similar to the one of `execute`, because both words execute a token:

```
EXECUTE ( i*x xt -- j*x )
CATCH   ( i*x xt -- j*x 0 | i*x n )
```

In order to keep the consistency of StrongForth's data type system, `catch` needs to consider the stack effect of executing the token. However, the actual value of the token at runtime is not known at compile time. This is the same problem as with `execute`, and consequently, it could be resolved in the same way. If the token provided to `catch` were a qualified token, the compiler would know its stack effect. We could make `)procreates` not only define a dedicated version of `execute` for each qualified token, but also a dedicated version of `catch`.

However, a different solution was chosen for StrongForth. Because there's another problem. `CATCH` as specified by Forth 2012 does not have unique output parameters, even if the stack diagram of the word belonging to the token were known. If no exception is thrown during execution of the token, `CATCH` has the same stack effect as if the token had been executed by `EXECUTE`, plus an error code as an additional output parameter. If an exception is thrown, the depth of the data stack is supposed to remain unchanged, except for the error code. For example, if the stack effect of the token were

```
( addr u -- flag )
```

the stack effect of `CATCH` would be

```
( addr u token -- flag 0 | addr u n )
```

with output parameters `addr u` having undefined values. Since StrongForth cannot handle ambiguous stack diagrams, `catch` needs to have the same stack effect in both cases, i. e.,

```
( addr u token -- flag n )
```

Because this is a major deviation from the Forth 2012 standard, another deviation doesn't add much pain. Since dealing with qualified tokens is pretty cumbersome, StrongForth uses a different approach to implement the `catch ... throw` mechanism. Instead of expecting a qualified token on the stack, it parses the input stream for the name of a word, finds a matching overloaded version, determines its stack diagram and compiles runtime code whose semantics resembles the one specified for `CATCH` by Forth 2012. The parsing syntax is easier to use than a syntax that expects an execution token on the stack.

You already saw an application of `catch` in the definition of `(>token)`. Here's a simpler example of how to use it. `?data-type` throws an exception if its parameter of data type definition is not associated with a data type:

```

: test ( definition -- )
  catch ?data-type if drop ." none" else . then ; ok
' signed test signed ok
' spaces test none ok

```

In both cases, with a valid and an invalid parameter, `catch ?data-type` produces the same stack effect, leaving an item of data type definition and a signed error-code on the stack. The value of `?data-type`'s output parameter is invalid in the second case, so it is discarded.

Class exception-handler

`catch` stores all data that is required to restore the state after an exception has been thrown in a so-called exception frame. An exception frame is an object of class `exception-frame`. It is created before the word whose exceptions shall be caught is executed, and deleted after the word has finished execution. If no exception has been thrown, the exception frame is deleted by the code compiled by `catch`. Otherwise, `throw` deletes the exception frame.

```

dt object procreates exception-frame
null exception-frame variable current-exception-frame
class exception-frame
  private definitions
  null exception-frame member 'previous-frame
  null signed member 'error-code
  null input-stream member 'saved-default-input-stream
  null input-stream member 'saved-input
  null unsigned member 'saved-fp
  null address member 'saved-esp
  null address member 'saved-eip
  forth definitions private
  : exception-frame ( unsigned address address exception-frame
    -- 4 th )
    locals( this )
    current-exception-frame @ 'previous-frame !
    this current-exception-frame !
    +0 'error-code !
    default-input-stream @ dup 'saved-default-input-stream !
    save-input 'saved-input !
    'saved-eip ! 'saved-esp ! 'saved-fp ! this ;
  : prev ( exception-frame -- exception-frame )
    'previous-frame @ ;
  :noname ( exception-frame -- )
    dup prev current-exception-frame !
    dup 'saved-input @ dup
    if delete
    else drop
    then [parent] delete ; is delete

```

```

private definitions
  \ ip@ ( -- address )
  \ ip! ( address -- )
  \ stack-tare ( -- address )
  \ fp-tare ( definition -- signed )
  \ update-offsets ( address single -- )

+0 constant (0)

: (begin-catch) ( signed -- )
  unpack sp@ swap + ip@ rot + rot fp@ swap +
  rot rot new exception-frame drop drop ;

: (end-catch) ( -- signed )
  current-exception-frame @ dup 'error-code @ swap delete ;

forth definitions

: catch ( -- )
  ['] (0) compile, ['] (begin-catch) compile,
  code-space here 0 over stack-tare
  parse-name true match-criterion search-context
  if dup compile,
    fp-tare rot rot code-space here rot - stack-tare rot -
    pack update-offsets ['] (end-catch) compile,
  else drop drop drop drop drop -13 throw
  then ; compile-only

: throw ( signed exception-frame -- )
  locals( this )
  'error-code !
  'saved-default-input-stream @ default
  'saved-input @ restore-input drop
  null input-stream 'saved-input !
  'saved-fp @ fp!
  'saved-esp @ sp!
  'saved-eip @ ip! ;

endclass

```

Exception frames may be nested, e. g., if the word whose exceptions shall be caught itself catches exceptions caused by the words it uses. Therefore, exception frames build a linked list, with its head stored in the variable `current-exception-frame`. Whenever a new exception frame is created, it is placed at the top of the list and links to the previous top. During deletion of the exception frame at the top of the list, the previous top is being restored. If the list is empty, `current-exception-frame` contains a null object.

Besides the pointer to the previous exception frame in the linked list, class `exception-frame` has six more private members. All of them are being initialized by the constructor. The error code is initialized with zero and may be overwritten by `throw`. The remaining five members save the program status, so that it can be restored when an exception is thrown. The status information consists of the default input stream, the input source, the return stack pointer, the hardware floating-point stack pointer and the instruction pointer. The three pointers are being initialized with the input parameters of the constructor. You'll see in a moment, how these parameters are being calculated.

The destructor unlinks the exception frame from the linked list and restores the previous exception frame as `current-exception-frame`. It further deletes the copy of the saved input stream, if it still exists, before deleting the exception frame itself.

`catch` is a compile-only word that compiles the following code sequence into the current definition:

- A literal containing parameters for `(begin-catch)`
- `(begin-catch)`
- The word that might throw exceptions
- `(end-catch)`

`(begin-catch)` creates an exception frame. The three parameters of the constructor are calculated from the literal parameter of `(begin-catch)`. This literal packs four character-size numbers of data type `single` into one cell:

- The net effect in address units of the code compiled between `(begin-catch)` and `(end-catch)` on the stack pointer. Adding this offset to the known value of the stack pointer after `(begin-catch)` yields the value of the stack pointer after the code has been executed without throwing an exception. This is the value `throw` has to restore.
- The net effect in address units of the code compiled between `(begin-catch)` and `(end-catch)` on the instruction pointer. This is actually the size of the compiled code. Adding this offset to the return address of `(begin-catch)` yields the address of the call to `(end-catch)`. This is the address `throw` has to restore in order to continue execution.
- The net effect of the code compiled on the hardware floating-point stack pointer. Adding this offset to the floating-point stack pointer during execution of `(begin-catch)` yields the value of the floating-point stack pointer after executing the compiled code. `throw` sets the floating-point stack pointer to this value.
- The fourth signed number is unused and has to be dropped.

Since the three actually used numbers are unknown before the code has been compiled, a dummy value is compiled as the packed literal. Its value is changed after the numbers become known.

Before we continue, let's have a closer look at `pack` and `unpack`:

```
pack ( single single single single -- single )
unpack ( single -- single single single single )
unpack ( signed -- signed signed signed signed )
```

`pack` packs the least significant bytes of four single-cell items in one item of data type `single`. `unpack` restores the four single-cell items. Two overloaded versions of `unpack` are required, because restoring signed numbers requires sign extension, whereas restoring unsigned numbers and all other items requires zero extension.

`(end-catch)` restores the previous exception frame. Before deleting the exception frame that was created by `(begin-catch)`, it retrieves the error code deposited by `throw`.

Now here's what `catch` is actually doing. It begins compiling a dummy literal for `(begin-catch)` and then `(begin-catch)` itself. But before compiling the code that might throw an exception, it obtains the current code space pointer, and with `stack-tare` the stack usage during compilation so far. If no match for the parsed name can be found, `catch` throws an exception. Otherwise, the target word is being compiled. At this point, the compiler is able to calculate the offset parameters for the creation of an exception frame with `(begin-catch)`. 0 as the fourth, unused value for packing is already on the stack. `fp-tare` uses the stack diagram of the target word to calculate its net effect on the hardware floating-point stack pointer by subtracting the number of floating-point output parameters from the number of floating-point input parameters. The length of the generated code is the difference between the code space pointer after and before compiling the target word. The net effect on the stack pointer by the code of the target word is calculated as the difference in the values returned by `stack-tare`. These four offsets are then

packed and stored as the value of the literal parameter for `(begin-catch)`. Finally, `catch` compiles `(end-catch)`.

The definition of `catch` as it is shown here is a simplified version, which may serve as a model. The real implementation has to consider quite a number of additional details regarding code generation and optimization.

Since `catch` is a compile-only word, it cannot be used in interpretation state. It seems obvious that a word that compiles a sequence of other words can not be interpreted. Nevertheless, it is possible to define an execute-only version of `catch`. This version can be included from the source file `catch.sf`.

`catch` has been made a member of class `exception-frame`, because its definition uses some of its private members. Anyway, it does not have an object of this class as an input parameter. This is different with `throw`, another public member of class `exception-frame`. Of course, this is not the version you'd typically use in your code. `throw` stores the error code in the `'error-code` member of class `exception-frame`. Then it restores the input source and sets the instruction pointer, the stack pointer and the hardware floating-point stack pointer to the values they would have if no exception had been thrown and the target word had executed to the point just before the call to `(end-catch)`. Execution continues with `(end-catch)`, which restores the previous exception frame, deletes the current one and returns the error code.

Note that `ip@` (in `(begin-catch)`) and `ip!` do not really work as shown in this programming model. They are actually implemented by accessing the return addresses on the return stack.

Exceptions are actually thrown by executing `throw` with a single parameter of data type `signed`. StrongForth provides an overloaded version, that implements the word `THROW` specified in Forth 2012:

```
: throw ( signed -- )
  dup
  if current-exception-frame @ dup
    if throw
    else drop error
    then
  else drop
  then ;
```

`throw` does nothing if the error code is zero. Otherwise, the existence of an exception frame determines, whether its method `throw` is executed, or whether the exception is handled by `error`, which simply displays an error message. The semantics of `error` will be explained in the next section.

The Exception Extension Word Set

StrongForth supports the Forth 2012 *Exception Extension* word set, which consists of the words `abort` and `abort"`. `abort` is defined exactly as suggested in the standard:

```
: abort ( -- )
  -1 throw ;
```

`abort"` is specified to execute `-2 throw`. If no exception frame is present, `throw` with the parameter `-2` shall display the given character string. In order to pass the character string to `throw`, the runtime code `(abort")` of `abort"` copies it into the transient area starting at `line`. The transient area has a length of `/hold` or 130 characters, which should be sufficient for most error messages. Trailing blanks fill up the remaining space of the transient buffer. The fact that the

transient area is used for other purposes as well, e. g., for pictured numeric output and for escaped string conversion, is supposed not to cause collisions.

```
: (abort") ( single caddress -> character unsigned -- )
  rot
  if line /hold blank line swap /hold min move -2 throw
  else drop drop
  then ;

: abort" ( -- )
  [compile] " ['] (abort") compile, ; compile-only
```

`abort"` is a compile-only word that simply parses and compiles a string literal and then compiles the runtime code `(abort")`.

The exceptions thrown by `abort` and `abort"` can be caught by an exception handler. If no exception handler is present, they get a special treatment within `error`:

```
defer error ( signed -- )

:noname ( signed -- )
  case -0 of exit endof
    -1 of endof
      -2 of user-output-device default
        line /hold -trailing type endof
      \ default \ user-output-device default
      cr .source ." ? " dup catch .message
      if .error else drop then cr catch .s drop
    endcase cr quit ; is error
```

`error` is a deferred word. You can provide your own version that will be executed whenever an exception is not caught by an exception handler. The original reason for making `error` a deferred word was that its definition contains high-level words, although it is itself used by `throw`, which needs to be available even to low-level words. High-level words like `case`, `-trailing` and `."` are compiled from source code when StrongForth is starting up.

`error` distinguishes different error codes with a `case ... endcase` conditional clause. An error code of zero requires no handling at all. Any non-zero value finally leads to the execution of `quit`, which initializes the system and restarts the interpreter loop with the console as input source. Error code `-1`, as produced by `abort`, does a silent restart, i. e., without displaying any message. Error code `-2` displays the string literal that `(abort")` has copied into the transient area, before performing a restart. All other error codes display

- the currently interpreted line upto the word that caused the exception,
- an error message that depends on the error code and
- a dump of the data type heap.

The phrase `user-output-device default` ensures that error messages are displayed on the console.

`.source` displays the already parsed words of the interpreted line, delivering the context of the exception:

```
: .source ( -- )
  source drop >in @ type ;
```

`.message` displays an error message for a given error code.

```
: .message ( signed -- )
  dup 1- -400 -0 within
  if abs " StrongForth.msg" r/o open
    swap c/l * cast unsigned-double over reposition
    dup line tuck c/l rot read -trailing type close
  else dup 1- -512 -400 within
    if abs 400 - msvcrt strerror string ignore type
    else throw
    then
  then ;
```

If the error code is between -400 and -1, a specific error message is loaded from the text file *StrongForth.msg*. In this file, every error message occupies `c/l` (64) characters, padded by spaces. Error codes between -511 and -400 display error messages that are produced by the operating system. All other error codes lead to an exception being thrown. However, this exception, and all other exception that occur during execution of `.message`, like a file read error, are caught by an exception handler within `error`. If `.message` fails, `.error` simply displays the error code as a decimal number:

```
: .error ( signed -- )
  ." Error " base @ decimal swap . base ! ;
```

The third line `error` displays is the dump of the data type heap. It is necessary to catch possible exceptions of `.s`, because `error` might get lost in an infinite recursion if something goes wrong during the execution of `.s`.

Now let's have a look at the definition of `quit`. Actually, `quit` performs some initialization work before entering the interpreter loop, as specified in the Forth 2012 standard:

```
: quit ( -- )
  end-compilation
  [ stack-space here stack-space unused + ] literal sp!
  dt-init
  [compile] private [compile] ignore
  [compile] protected [compile] ignore
  [compile] assembler [compile] ignore
  (quit)
  begin current-exception-frame @
  while current-exception-frame @ delete
  repeat
  user-input-device default
  user-output-device default
  msvcrt
  _iob to stdin
  _iob cast address 8 cells + cast file to stdout
  _iob cast address 16 cells + cast file to stderr
  ignore
  begin refill
  while interpret prompt
  repeat bye ;
```

Naturally, the initialization of StrongForth involves more work than that of a standard Forth 2012 system. `end-compilation` turns to interpretation state, if the system was in compilation state. Because StrongForth has no physical data stack, initializing the return stack with `sp!` initializes the virtual data stack as well. This is an important deviation from the semantics specified by Forth

2012. In addition, `dt-init` initializes the interpreter data type heap. Vocabularies `private` and `protected` are removed from the context vocabulary list, because their typical scope is limited to a class definition. Likewise, vocabulary `assembler`, whose scope begins with `code` and ends with `endcode`, is removed from the context. All exception frames are being deleted. The default input and output streams are assigned to the console.

`(quit)` is a deferred word whose semantics is, by default, the one of `ignore-friends`. This ensures that the search order is being cleaned up if `quit` happens to be executed during a class definition, e. g., because an unhandled exception has been thrown. You can assign additional initialization tasks to `(quit)` that shall be executed by `quit`:

```
defer (quit) ( -- )
' ignore-friends is (quit)
```

With initialization being done, `quit` enters the interpreter loop, which repeatedly executes these steps:

- Fill the input buffer.
- Interpret or compile the contents of the input buffer.
- If in interpretation state, display a prompt to request more input.

`prompt` displays `ok` if in interpretation state:

```
defer prompt ( -- )
:noname ( -- )
  state @ invert if ." ok" then cr ; is prompt
```

`prompt` is, like `(quit)`, a deferred definition. You may change StrongForth's prompt to whatever you prefer. For example, you might consider to display the contents of the data type heap:

```
:noname ( -- )
  state @ if ." Compiler" else ." Interpreter" then
  ." data type heap: "
  dt-depth if postpone .s else ." <empty>" then cr ; ok
  is prompt Interpreter data type heap: <empty>
8 base Interpreter data type heap: unsigned caddress -> unsigned
drop drop Interpreter data type heap: <empty>
```

Once `quit` cannot refill the input buffer, it exits StrongForth by executing `bye`. The semantics of `bye` is identical to that of the corresponding Forth 2012 word. StrongForth provides an overloaded version of `bye` which returns a numerical program exit status to the operating system:

```
: bye ( integer -- )
  msvcrt exit ignore ;

: bye ( -- )
  0 bye ; 1 retreat
```

Since input provided by the user input device typically never ceases, `quit` is actually an endless loop. But it is not only used upon system startup. `quit` may also be used as a simple means for suppressing the prompt, and for error recovery.

Error Codes

In the previous section, you've learned that error codes 0 to -511 are reserved by StrongForth:

Error code	Meaning
0	no error
-1	abort
-2	abort "
-3 to -58	same meaning as specified by Forth 2012
-59 to -255	reserved
-256 to -303	specific StrongForth error codes
-304 to -399	reserved
-400 to -511	C runtime library error codes

To display a list of all error messages, you can define this word and execute it with appropriate parameters, e. g., -511 and -0, to see all messages:

```
: .messages ( signed 1st -- )
  do cr i 3 .r space i .message -1 +loop ;
```

Operating System Exceptions

StrongForth catches exceptions raised by the operating system by translating them into StrongForth exceptions that can be handled with StrongForth exception handlers:

Exception	Error code	Caused by
Access violation	-9	Read or Write to invalid addresses
Datatype misalignment	-23	Not applicable
Array bounds exceeded	-289	bound, machine instruction
Integer divide by zero	-10	div, and idiv, machine instructions
Integer overflow	-288	into machine instruction
Illegal instruction	-294	ud2, machine instruction and other invalid opcodes
Privileged instruction	-290	hlt, machine instruction and other privileged instructions
Page fault	-277	Not applicable

A typical example is a division by zero:

```
10 0 /
10 0 / ? division by zero
unsigned
```

If / had been enclosed by an exception frame, catch would return -10 as the error code.

Note that entering control-C does not raise an exception. You can enter control-C just like any other key:

```
key cast integer . <ctrl-C> 3 ok
```

In contrast to those operating system exceptions, exceptions caused by the hardware floating-point processor are masked. Since floating-point exceptions are not automatically cleared, you can retrieve the error code associated with them with this word:

```
fpe ( -- signed )
```

fpe stands for *floating-point exception*, and the output parameter is an error code. Additionally, fpe will clear all exception flags within the hardware floating-point processor. This is an example of how fpe can be used:

```
pi 0e0 / . infinity ok  
fpe throw  
fpe throw ? floating-point divide by zero
```

21 C Runtime Library

The `msvcrt` Vocabulary

StrongForth is embedded into the *Windows* operating system. Being a console application, StrongForth takes advantage from some services provided by the *MSVCRT C runtime library*. In the previous chapter, we have already used two different C runtime functions. `.message` uses `strerror` to deliver the system error message assigned to a given error code, and `bye` uses `exit` to return control to the operating system.

Many of the *MSVCRT* functions are made available to StrongForth in the `msvcrt` vocabulary. The Forth 2012 *File-Access* and *Memory-Allocation* word sets, including extensions, are implemented in StrongForth based on the `msvcrt` vocabulary. You can get a list of all words in the `msvcrt` vocabulary with

```
msvcrt words
```

All these words have the same names and semantics as in the *MSVCRT* library. However, in most cases, you will define a wrapper that is adapted to StrongForth's style. One of the reasons is that `msvcrt` words expect parameters on the stack instead of in registers. Even more important, `msvcrt` words expect parameters in reverse order, which might be confusing. For example,

```
_write ( unsigned caddress -> character file-descriptor
        -- unsigned )
```

is the StrongForth version of the *MSVCRT* function `_write`:

```
int _write(
    int fd,
    const void *buffer,
    unsigned int count
)
```

A wrapper word could look like this:

```
forth definitions msvcrt

: _write ( caddress -> character unsigned file-descriptor -- 3rd )
  rot rot swap rot _write ;

ignore
```

Data types like `file-descriptor` have been created for several kinds of items used by *MSVCRT* library functions. Some of these, like `file-status` and `file-control`, are bit fields or enumerations. The predefined bits and values for items of these data types can be included as constants from the source file `msvcrt.sf`. Here are a few examples:

```
0 bit cast file-control constant O_WRONLY \ open for writing only
3 cast locking-mode constant LK_RLCK \ lock for writing
-1 cast character constant EOF \ end of file
```


Error Handling

Forth 2012 specifies that words in the *File-access* and *Memory-allocation* word sets return an *I/O result code* on top of the stack. In StrongForth, the I/O result codes have been abandoned. Instead, if an error occurs during execution of these words, an exception is being thrown. You can retrieve the I/O result code by executing a word with an exception frame. The I/O result code is then simply the value returned by `catch`.

Most *MSVCRT* library functions signal an error during execution by returning a specific value. Sometimes zero, sometimes a non-zero value, and sometimes -1, indicate that something went wrong. In those cases, the I/O result code can afterwards be retrieved with one of these two words:

```
doserrno ( -- signed )
errno ( -- signed )
```

The I/O result codes delivered by *MSVCRT* library functions are positive numbers. They are transformed by adding 400 and negating the result in order to become StrongForth error codes. One of these three words is typically executed after an *MSVCRT* word that might fail:

```
: ?ferror ( single -- )
  if errno @ 400 + negate throw then ;

: 0?ferror ( single -- 1st )
  dup 0= ?ferror ;

: -1?ferror ( single -- 1st )
  dup cast signed -1 = ?ferror ;
```

Note that `?ferror` consumes 1st input parameter, while `0?ferror` and `-1?ferror` return it unchanged for further processing.

Memory Allocation

StrongForth's dynamic memory allocation is based on these *MSVCRT* library functions:

```
malloc ( unsigned -- address )
free ( address -- )
realloc ( unsigned address -- 2nd )
_msize ( address -- unsigned )
```

`allocate` expects the number of address units and returns the address of an allocated block of dynamic memory with the given size:

```
: allocate ( unsigned -- address )
  malloc dup 0= if -268 throw then ;
```

`allocate` does not return an I/O result code. if `malloc` returns a null address to indicate that it did not allocate anything, `allocate` throws an exception.

Since `allocate` returns an unspecified address, you usually have to cast this address to a compound data type, like in this example:

```
20 cells allocate -> signed constant signed-array
```

If you need to allocate dynamic memory for a character string, you might think about doing it this way:

```
80 chars allocate cast address -> character constant line-of-text
```

But there's an easier way to accomplish this task:

80 chars callocate -> character constant line-of-text

ccallocate has exactly the same semantics as allocate. However, it returns an item of data type caddress instead of an item of data type address. Similarly, variants of allocate are available for single-precision and double-precision floating-point numbers:

```
: callocate ( unsigned -- caddress )
  allocate cast caddress ;

: sfallocate ( unsigned -- sfaddress )
  allocate cast sfaddress ;

: dfallocate ( unsigned -- dfaddress )
  allocate cast dfaddress ;
```

Note that all these variants still expect the number of address units on the stack, so you can't avoid using chars, sfloats or dfloats.

free has actually the same semantics like the equally named *MSVCRT* library function. It has to be defined anyway to make it available in the forth vocabulary:

```
: free ( address -- )
  free ;
```

Because the order of the two input parameters of realloc are reversed with respect to resize, they have to be swapped. Otherwise, the definition of resize resembles the one of allocate. realloc, just like malloc, returns a null pointer if the requested dynamic memory could not be allocated.

```
: resize ( address unsigned -- 1st )
  swap realloc dup 0= if -268 throw then ;
```

Finally, the *MSVCRT* library provides a function that obtains the size in address units of a memory block allocated with allocate or resize. This function can be used to define an overloaded version of size that is not included in the Forth 2012 specification:

```
: size ( address -- unsigned )
  _msize ;
```

Note that all overloaded versions of size return the size in *address units* of the item they are applied to.

Zero-terminated Strings

Many C runtime functions deal with character strings that have a terminating null character:

S	t	r	o	n	g	F	o	r	t	h	□
---	---	---	---	---	---	---	---	---	---	---	---

StrongForth, on the other hand, represents character strings in the format `caddress -> character unsigned`, i. e., as the address of the first character and the character count. Creating a so-called zero-terminated string from a string in the usual *address-and-count* format cannot be accomplished by just appending a null character after the last character of the string, because the terminating character might overwrite something else. It is necessary to copy the string to a buffer that is big enough for one additional character, and then append the null character to the copy of the string.

First, we create a new data type for zero-terminated strings. This data type, as well as all other words presented in this section, are included in the `msvcrt` vocabulary:

```
dt single procreates zero-terminated-string
```

Next, we need a word that converts a string from the address-and-count format to a zero-terminated string, allocating dynamic memory:

```
: zero-terminated-string ( caddress -> character unsigned
  -- zero-terminated-string )
  dup 1+ chars callocate -> character swap locals( c-addr u )
  c-addr u move null character c-addr u + !
  c-addr cast zero-terminated-string ;
```

Once the zero-terminated string is no longer needed, the dynamic memory allocated for it should be freed:

```
: free ( zero-terminated-string -- )
  cast caddress free ;
```

Of course, we also need a word that converts a zero-terminated string into a character string in the address-and-count format. It uses the *MSVCRT* function `strlen` to count the characters up to the null character:

```
: string ( zero-terminated-string
  -- caddress -> character unsigned )
  dup cast caddress -> character swap strlen ;
```

There's also an overloaded version of `.`, which directly displays a zero-terminated string:

```
: . ( zero-terminated-string -- )
  string type ;
```

You'll see several examples of how to deal with zero-terminated strings in the following sections of this chapter.

File Access

In StrongForth, the Forth 2012 *File-Access* word set including extensions is implemented using *MSVCRT* library functions. Data type `file` is actually a pointer to one of the *MSVCRT* I/O blocks:

```
dt single procreates file
```

Each I/O block is 8 cells long. The first three I/O blocks are reserved for the standard I/O devices of the C runtime library. Being values, the standard I/O devices can be changed in order to redirect standard input and/or output:

```
_iob value stdin
_iob cast address 8 cells + cast file value stdout
_iob cast address 16 cells + cast file value stderr
```

When creating and opening files, a file access method has to be specified. Forth 2012 specifies the three constants `R/O`, `W/O` and `R/W` as file access methods, plus a modifier `BIN`, which should be applied if the file is to be interpreted as binary code instead of as text. StrongForth defines a dedicated data type `fam` for the file access method. The modifier `bin` is a dummy word, because the *MSVCRT* library does not distinguish between binary code and text.

```
dt single procreates fam

1 cast fam constant r/o
2 cast fam constant w/o
3 cast fam constant r/w

: bin ( fam -- 1st ) ;
```

With the two new data types, `create` and `open` can be defined based on the *MSVCRT* library function `fopen`. In Forth 2012, these two words are called `CREATE-FILE` and `OPEN-FILE`. Generally, StrongForth abandons the `-FILE` postfix or the `FILE-` prefix in the names of words, because it can distinguish overloaded words by the data types of their input parameters. For example, `create` for files cannot be mixed up with `create` for definitions, which has no input parameters at all. Just make sure that you don't try to create a definition when accidentally a character string and a file access method are on top of the stack.

```
: create ( caddress -> character unsigned fam -- file )
  rot rot zero-terminated-string swap
  case w/o of [ here " wb" ", 0 c, cast zero-terminated-string ]
    literal over fopen 0?error endof
  r/w of [ here " w+b" ", 0 c, cast zero-terminated-string ]
    literal over fopen 0?error endof
  \ default \ drop -303 throw null file
endcase swap free ;

: open ( caddress -> character unsigned fam -- file )
  rot rot zero-terminated-string swap
  case r/o of [ here " rb" ", 0 c, cast zero-terminated-string ]
    literal over fopen 0?error endof
  w/o of [ here " wb" ", 0 c, cast zero-terminated-string ]
    literal over fopen 0?error endof
  r/w of [ here " r+b" ", 0 c, cast zero-terminated-string ]
    literal over fopen 0?error endof
  \ default \ drop -303 throw null file
endcase swap free ;
```

Both words start with creating a zero-terminated string as the file name. They then enter a `case ... endcase` conditional clause with the file access method as the case selector. Each case executes the *MSVCRT* library function `fopen` with the file name and a zero-terminated string literal specifying the file access as parameters. `0?error` throws an exception if `fopen` returns a null file instead of a valid file handle. An invalid file access method also leads to an exception. Finally, the dynamic memory allocated for the zero-terminated string containing the file name is freed.

Note that the `r/o` file access method cannot be applied to `create`.

The *MSVCRT* library function `_stat` copies information about the status of a file with a given name into a memory block of 9 cells. `status`, the StrongForth equivalent of the Forth 2012 word `FILE-STATUS`, allocates 9 cells of dynamic memory for the file status and returns its address after executing `_stat`. A non-zero result of this function causes an exception to be thrown. Information about the contents of the 9 status cells can be obtained from the *MSVCRT* documentation. To avoid memory leakage, the dynamic memory block should be freed after processing the status information.

```
: status ( caddress -> character unsigned -- address )
  zero-terminated-string 9 cells allocate
  swap over over _stat ?error free ;
```

`delete` and `rename` can easily be implemented with the *MSVCRT* library functions `remove` and `rename`, respectively. Character strings in the address-and-count format have to be converted to zero-terminated strings. Errors have to be handled. Finally, the dynamic memory allocated for the zero-terminated strings has to be freed:

```
: delete ( caddress -> character unsigned -- )
  zero-terminated-string dup remove ?error free ;
```

```
: rename ( caddress -> character unsigned
  caddress -> character unsigned -- )
  zero-terminated-string rot rot zero-terminated-string
  over over rename ?ferror free free ;
```

The position of a file can be obtained with the *MSVCRT* library function `ftell`. This function returns an unsigned number, or `max-unsigned` if an error occurred during execution:

```
: position ( file -- unsigned-double )
  ftell -1?ferror cast unsigned-double ;
```

Other StrongForth words use the *MSVCRT* library function `fseek` instead of `ftell`. `fseek` sets the file position relative to the beginning of the file, to the end of the file, or to the current file pointer, depending on the value of a switch of data type `seek-origin` and an offset parameter of data type `signed`:

```
fseek ( seek-origin signed file -- integer )
```

Switch	Value	New position
SEEK_SET	0	= offset
SEEK_CUR	1	= current position ± offset
SEEK_END	2	= size of file ± offset

Bear in mind that the offset parameter is a single-cell value. Files might be larger than the values that can be represented in a single-cell value.

With `fseek`, the implementation of reposition becomes straightforward. `fseek` returns a non-zero value if it fails for whatever reason:

```
: reposition ( unsigned-double file -- )
  null seek-origin \ SEEK_SET \ rot cast signed rot fseek
  ?ferror ;
```

Querying the size of a file is a little bit tricky. We have to position the file pointer to the end of the file and then obtain its value, which is actually the file size. In order to leave the file pointer unchanged, we have to save its value at the beginning and restore it at the end of the definition of `size`:

```
: size ( file -- unsigned-double )
  dup position cast signed
  over 2 cast seek-origin \ SEEK_END \ +0 rot fseek ?ferror
  over position
  rot rot swap null seek-origin \ SEEK_SET \ rot rot fseek
  ?ferror ;
```

Resize can be implemented using the *MSVCRT* library function `_chsize`. It also returns a non-zero value if the operation doesn't succeed. Because `_chsize` expects a file descriptor as an input parameter, `_fileno` has to be used to convert the I/O block of data type `file` into a file descriptor:

```
: resize ( unsigned-double file -- )
  swap cast unsigned swap _fileno _chsize ?ferror ;
```

Now, let's have a look at file read and write operations. `fread` and `fwrite`, the two *MSVCRT* library function used for reading and writing, return the number of characters read or written, respectively. `ferror` returns the error code associated with the most recent I/O operation on a specific file:

```
ferror ( file -- integer )
```

The error code is available until it is explicitly cleared with `clearerr`:

```
clearerr ( file -- )
```

Based on these *MSVCRT* library functions, the Forth 2012 word `READ-FILE` can be implemented in StrongForth:

```
: read ( caddress -> character unsigned file -- 3rd )
  locals( addr u f )
  f u 1 addr fread dup u <
  if f ferror cast signed dup
    if f clearerr 400 + negate
      then throw
  then ;
```

The literal 1 is a parameter for `fread` that specifies the number of address units per character. If the number of characters read is lower than the number of characters requested, `read` checks whether an error or an end-of-file condition has occurred. It clears the error and throws an exception if necessary.

`write` looks very similar to `read`. The differences are that it calls `fwrite` instead of `fread` and that it doesn't return the number of characters processed:

```
: write ( caddress -> character unsigned file -- )
  locals( addr u f )
  f u 1 addr fwrite u <
  if f ferror cast signed dup
    if f clearerr 400 + negate
      then throw
  then ;
```

`write-eol`, which is not specified by Forth 2012, writes a carriage-return/line-feed sequence to the destination file. The creation of a string consisting of the two control characters in the data space is embedded in the definition of `write-eol`:

```
: write-eol ( file -- )
  [ chere -> character 13 c, 10 c, align ] literal 2 rot write ;
```

Based on `write` and `write-eol`, the Forth 2012 word `WRITE-LINE` can be implemented:

```
: write-line ( caddress -> character unsigned file -- )
  dup >r write r> write-eol ;
```

The implementation of `read-line` is more complicated:

```
: read-line ( caddress -> character unsigned file -- 3rd flag )
  locals( a n f ) a n 2 + f read dup
  if 0
    begin dup n <
      while a over + @ 10 cast character \ lf \ <>
        while 1+
          repeat
            then dup dup
            if a over 1- + @ 13 cast character \ cr \ = if 1- swap then
              then 1+ over n >=
              if drop drop n dup
                then cast signed rot - 1 cast seek-origin \ current \ swap
                f fseek ?ferror true
            else false
          then ;
```

`read-line` starts trying to read as many characters as fit into the given buffer, plus two characters for the line terminator sequence. Next, it searches the buffer for the first occurrence of a line feed character, which is the Unix line terminator or the second character of the Windows line termination sequence. If the line feed character was not the first character of the line, `read-line` looks for an additional carriage return character. If the line is shorter than the maximum size specified, the file is repositioned back to the position after of the line feed character, which is where the next line starts. Otherwise, the line read so far is not yet complete and the file position is set to the character after the last character returned in the buffer. `read-line` returns `true` if and only if at least one character has been successfully read from the file.

`flush` uses the *MSVCRT* library function `fflush` and throws an exception if `fflush` returns a non-zero value:

```
: flush ( file -- )
  fflush ?ferror ;
```

The interpreter and the compiler can distinguish this overloaded version of `flush` from the word with the same name in the *Block* word set, because it has an input parameter of data type `file`.

Finally, `close` closes a file using the *MSVCRT* library function `fclose`. An exception is thrown if the result code is non-zero:

```
: close ( file -- )
  fclose ?ferror ;
```

Loading Source Files

In order to interpret the contents of a source file, Forth 2012 specifies three words:

```
INCLUDE-FILE ( i * x fileid - - j * x )
INCLUDED ( i * x c-addr u - - j * x )
INCLUDE ( i * x "name" - - j * x )
```

In StrongForth, all these words share the same name `include`, because the interpreter can distinguish the three overloaded versions by their stack diagrams. Let's begin with the first one, which expects an item of data type `file` on the stack:

```
: include ( file -- )
  default-input-stream @ locals( file stream )
  file 258 new file-input-stream default
  begin refill
  while interpret
  repeat
  file close default-input-stream @ delete stream default ;
```

This version of `include` can be applied to a source file that has already been opened. It saves the present input stream, creates a new file input stream with a buffer size of 258 characters, and switches to the source file as the new input stream. Then it enters an interpreter loop, repeatedly refilling and interpreting the input buffer. At the end of the file, `include` closes the file, deletes the file input stream and restores the previous input stream.

For the other two versions, we need a means to remember that the source file has been included. The easiest way to implement this feature is to define a new word that has the same name as the source file in the current compilation vocabulary:

```
: new-included-file ( caddress -> character unsigned -- )
  (create) does> ( address -- ) drop -300 throw ;
```

If the name of a source file contains spaces, `new-included-file` defines a StrongForth word with spaces as well. This is not a problem, because the created words are not supposed to be executed or compiled. If that happens anyway, the runtime semantics just throws an exception.

With this defining word, StrongForth's version of the Forth 2012 word `INCLUDED` looks like this:

```
: include ( caddress -> character unsigned -- )
  over over new-included-file r/o open include ;
```

We could, for example, include a source file called `bounds.sf`, which contains a single colon definition named `bounds`:

```
" bounds.sf" include ok
latest . bounds ( -- ) immediate ok
latest prev . bounds.sf ( -- ) ok
bounds.sf
bounds.sf ? already included
```

`include` has used `new-included-file` to create `bounds.sf`, before it actually loads the source code stored in the file. This definition can be used by `require` to find out whether the source file has already been included.

Before we turn to `require`, here's the parsing version of `include`:

```
: include ( -- )
  parse-name include ; 2 retreat
```

Since this version has no parameters, it has to be re-linked within the vocabulary in such a way that the other two versions are found first. Otherwise, the parsing `include` would hide all other overloaded versions.

Finally, let's have a look at `require`. This word executes `include` only if the respective source file hasn't been included yet. `require` searches all vocabularies for a definition created by `new-included-file` that has the same name as the source file. The phrase `latest prev prev prev` is used to obtain the runtime code of `new-included-file`, which is located five definitions before the definition of `require`. Unfortunately, there is no easier way, because the runtime code is defined with `:noname`.

```
: require ( caddress -> character unsigned -- )
  over over [ latest prev prev prev prev ] literal
  runtime-criterion search-all nip
  if drop drop else include then ;
```

Of course, StrongForth also provides a parsing version of `require`, which has to be re-linked as well:

```
: require ( -- )
  parse-name require ; 1 retreat
```

Arguments and Environment

The *MSVCRT* library grants access to the command line arguments and the environment strings of the terminal window that called StrongForth. These function are available within StrongForth:

```
argc ( -- unsigned )
argv ( -- address -> zero-terminated-string )
environ ( -- address -> zero-terminated-string )
```


`argc` returns the number of command line arguments as an unsigned number. Since the path to the StrongForth executive counts as parameter, the minimum value is one. `argv` returns the address of an array of zero-terminated strings. Each string contains one argument. The first one is the path to the StrongForth executive. The list is terminated with a null zero-terminated string.

`environ` also returns the address of an array of zero-terminated strings, each of which represents one environment variable and its value. The array is terminated by a null zero-terminated string. With the following word, you can display a list of all environment variables:

```
: environment ( -- )
  msvcrt environ
  begin dup 1+ swap @ dup
  while cr .
  repeat drop drop ignore ;
```

The System Clock

The *MSVCRT* library function `clock` returns the number of milliseconds elapsed since StrongForth was started, or -1 if an error occurred. With `clock`, the word `ticks` in the Forth vocabulary can be implemented:

```
: ticks ( -- unsigned )
  clock -1?error ;
```

Based on ticks, it is possible to implement the Forth 2012 word `MS`:

```
: ms ( unsigned -- )
  ticks swap +
  begin ticks over >
  until drop ;
```

Okay, a permanently querying loop is not the most elegant solution. But it works as expected.

Another Forth 2012 word related to the system time is `TIME&DATE`. Again, StrongForth uses *MSVCRT* library functions for the implementation:

```
time&date ( -- unsigned unsigned unsigned unsigned unsigned
  unsigned )
```

`time&date` is the predefined word with the largest number of output parameters.

Keyboard Events

A keyboard event happens each time you press a key or a combination of keys on your keyboard. The *MSVCRT* library function `_kbhit` returns a non-zero value if a keyboard event is pending. `ekey?` uses this function:

```
: ekey? ( -- flag )
  _kbhit 0<> ;
```

`_getch` waits for a keyboard event and then returns a character-size item representing the key or a combination of keys that has been pressed. Some of the special keys require two calls of `_getch`. If the first call returns a prefix code, either 0 or 224, a second call is required to construct a complete keyboard event. In StrongForth, keyboard events have a dedicated data type:

```
dt single procreates keyboard-event
```

```
: ekey ( -- keyboard-event )
  _getch dup 0=
  if drop _getch 256 +
  else dup 224 cast character =
    if drop _getch 512 +
    then
  then cast keyboard-event ;
```

`ekey` returns a keyboard event. If the event has a zero prefix, 256 is added to the value returned by the second call to `_getch`. If the event has prefix 244, 512 is added to the value returned by the second call to `_getch`. In total, `ekey` can produce 766 different keyboard events, 254 non-prefixed and two times 256 prefixed ones.

`ekey>char` converts non-prefixed keyboard events to characters and returns an additional `true` flag. If a keyboard event is prefixed, it returns a `false` flag together with the original keyboard event as an invalid character.

```
: ekey>char ( keyboard-event -- character flag )
  dup cast character swap cast unsigned max-character <= ;
```

Prefixed keyboard events are being caused by pressing function keys, the reverse tabulator key, the six-key block to the left of the enter key, and the cursor keys. For these keys except for the reverse tabulator key, Forth 2012 specifies a number of unsigned key constants:

```
1 constant k-F1
2 constant k-F2
3 constant k-F3
4 constant k-F4
5 constant k-F5
6 constant k-F6
7 constant k-F7
8 constant k-F8
9 constant k-F9
10 constant k-F10
11 constant k-F11
12 constant k-F12
13 constant k-reverse-tab
16 constant k-insert
17 constant k-delete
18 constant k-home
19 constant k-end
20 constant k-prior
21 constant k-next
22 constant k-left
23 constant k-right
24 constant k-up
25 constant k-down
```

Using a logical `or`, these constants can be combined with bit masks if one or more of the respective modifier keys are being pressed simultaneously:

```
5 bit constant k-shift-mask
6 bit constant k-ctrl-mask
7 bit constant k-alt-mask
```

The task of `ekey>fkey` is to convert a prefixed keyboard event into an unsigned value that can be compared with key constants. It returns `true` if the conversion succeeds and `false` if the keyboard event is non-prefixed. `ekey>fkey` uses an inlined conversion table in the data space:

```
: ekey>fkey ( keyboard-event -- unsigned flag )
  ekey>char
  if cast unsigned false
  else [ data-space chere -> unsigned hex
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 00 prefix 00
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 08
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 10
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 18
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 20
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 28
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 30
    00 c, 00 c, 00 c, 01 c, 02 c, 03 c, 04 c, 05 c, \ 38
    06 c, 07 c, 08 c, 09 c, 0A c, 00 c, 00 c, 00 c, \ 40
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 48
    00 c, 00 c, 00 c, 00 c, 21 c, 22 c, 23 c, 24 c, \ 50
    25 c, 26 c, 27 c, 28 c, 29 c, 2A c, 41 c, 42 c, \ 58
    43 c, 44 c, 45 c, 46 c, 47 c, 48 c, 49 c, 4A c, \ 60
    81 c, 82 c, 83 c, 84 c, 85 c, 86 c, 87 c, 88 c, \ 68
    89 c, 8A c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 70
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 78
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 80
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 88
    00 c, 00 c, 00 c, 96 c, 4D c, 97 c, 00 c, 92 c, \ 90
    98 c, 94 c, 00 c, 00 c, 00 c, 00 c, 00 c, 93 c, \ 98
    99 c, 95 c, 90 c, 91 c, 00 c, 00 c, 00 c, 00 c, \ A0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ A8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ B0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ B8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ C0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ C8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ D0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ D8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ E0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ E8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ F0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ F8
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 00 prefix E0
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 08
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 10
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 18
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 20
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 28
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 30
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 38
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 12 c, \ 40
    18 c, 14 c, 00 c, 16 c, 00 c, 17 c, 00 c, 13 c, \ 48
    19 c, 15 c, 10 c, 11 c, 00 c, 00 c, 00 c, 00 c, \ 50
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 58
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 60
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 68
    00 c, 00 c, 00 c, 56 c, 57 c, 53 c, 55 c, 52 c, \ 70
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 78
    00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 0B c, 0C c, 2B c, \ 80
```

```

2C c, 4B c, 4C c, 8B c, 8C c, 58 c, 00 c, 00 c, \ 88
00 c, 59 c, 50 c, 51 c, 00 c, 00 c, 00 c, 00 c, \ 90
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ 98
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ A0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ A8
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ B0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ B8
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ C0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ C8
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ D0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ D8
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ E0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ E8
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ F0
00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, 00 c, \ F8
decimal ] literal swap 256 - + @ true
then ;

```

The Forth 2012 word `KEY` processes only non-prefixed characters. In StrongForth, `key` executes `ekey` within a loop until a non-prefixed key has been pressed:

```

: key ( -- character )
  begin ekey ekey>char invert
  while drop
  repeat ;

```

There is no function in the *MSVCRT* library that indicates whether a non-prefixed key has been pressed. If any key has been pressed at all, `key?` has to read it in order to find out whether it is prefixed or not. If it is, the key is discarded and `_kbhit` is executed again. Otherwise, the *MSVCRT* library function `_ungetch` pushes the key back to the input device, before `key?` returns a `true` flag. A `false` flag is returned if no more keys are available:

```

: key? ( -- flag )
  begin _kbhit
  while ekey ekey>char
    if _ungetch drop true exit
    else drop
    then
  repeat false ;

```

Unsupported Features

Forth 2012 specifies `EMIT?` as the counterpart of `key?`. `EMIT?` returns a flag that indicates whether the output device is ready to accept the next character. Since the *MSVCRT* library does not provide a suitable function, StrongForth's version of `emit?` always returns a `true` flag:

```

: emit? ( -- flag )
  true ;

```

22 Programming-Tools Word Set

Some words of StrongForth's implementation of the Forth 2012 *Programming-Tools* word set have already been presented in previous chapters. In this chapter, we'll have a look at the remaining words.

?

? is a very simple word. In many Forth 2012 systems, it is implemented like this:

```
: ? ( a-addr -- ) \ Forth 2012
  @ . ;
```

In StrongForth, this code doesn't work:

```
: ? ( address -- ) @ . ;
: ? ( address -- ) @ ? undefined word
address
```

All versions of @ that are provided by StrongForth expect the address of a specific data type on the stack. Our first approach might look like this:

```
: ? ( address -> single -- ) @ . ; ok
1234 variable test ok
test ? 1234 ok
```

This works fine, but it has a couple of drawbacks. First, we need a separate overloaded version of ? for each kind of address: For address -> single, address -> double, address -> float, caddress -> single, sfaddress -> float and dfaddress -> float. Second, all items will be displayed as unsigned single or double-precision numbers, or as floating-point numbers without exponent. Even items of data types signed, flag, character, data-type and definition will be displayed as unsigned numbers. ? wouldn't care about the different overloaded versions of . that display data nicely according to their data type. That is, unless we decide to overload ? by defining separate versions for all overloaded versions of ..

There's a much simpler solution. Only one version of ? that covers all combinations, plus all future kinds of addresses and all future overloaded versions of .. Here it is:

```
: ? ( -- )
  " @ ." evaluate ; immediate
```

The only drawback is that this version of ? compiles two words instead of one. As a compensation, it executes a little bit faster at runtime, because there's no additional subroutine level required.

dump

StrongForth provides three overloaded versions of dump in order to cover the data sizes single-cell, double-cell and character:

```
: dump ( address unsigned -- )
  swap -> single swap 0
  ?do i 4 mod 0= if cr dup .addr then dup @ space .cell 1+
  loop drop ;

: dump ( address -> double unsigned -- )
  2* dump ;

: dump ( caddress unsigned -- )
  swap -> single swap 0
  ?do i 16 mod 0= if cr dup .addr then dup @ space .byte 1+
  loop drop ;
```

.byte and .cell display a byte or a cell as 2- or 8-digit hexadecimal numbers, respectively.

.addr displays an 8-digit hexadecimal address with a trailing colon:

```
: .byte ( single -- )
  base @ hex swap [ 255 cast logical ] literal and 2 0.r base ! ;

: .cell ( single -- )
  base @ hex swap 8 0.r base ! ;

: .addr ( address -- )
  .cell [char] : . ;
```

The first overloaded version of dump is a catchall for any kind of addresses. It displays the contents of up to four cells as 8-digit hexadecimal numbers on each line, preceeded by the address of the first cell. The version for addresses of double-cell items does the same, but it displays the contents of twice as many cells. The version of dump for character addresses displays up to 16 2-digit hexadecimal numbers per line. The second parameter of dump always determines the number of items to be displayed, not the number of address units as in Forth 2012.

Here are some examples:

sp@ 15 dump

```
00870050: 76191E31 0803588B 00882128 008423A0
00870060: 0000000B F0E0D0C0 9C7F3048 00000023
00870070: 00870074 00400000 FFFFFFFF8 20202020
00870080: 9C4FD028 80000020 20202020 ok
```

dt-bottom 10 dump

```
0041AC50: 0040A238 00040008 0040B148 00000000
0041AC60: 0040A180 00000002 0040A1BC 00000000
0041AC70: 0040A1FC 00000000 0040A1FC 00000000
0041AC80: 0040A1FC 00000001 0041395C 00000000
0041AC90: 0040A238 00040001 0041395C 00000000 ok
```

code-space chere 1000 - 72 dump

```
00428384: 23 E0 FD FF 57 E8 AC 7B FF FF 5F C3 E8 96 DD FD
00428394: FF 57 83 C7 18 8B 07 5F E8 0A E0 FD FF 8B 4B 04
004283A4: 8B 1B 81 E1 00 00 04 00 57 50 51 53 BB 38 A2 40
004283B4: 00 B9 00 00 00 00 81 C9 00 00 04 00 58 5A E8 50
004283C4: D5 FD FF 85 C9 89 D1 89 ok
```

see

StrongForth provides two overloaded versions of `see`. The first one is a virtual method of class `definition`:

```
virtual see ( definition -- )
:noname ( definition -- )
  drop ; is see
```

By default, this virtual method does nothing at all. You have to include the source file `see.sf` to assign the virtual methods `see` of several children and grandchildren of class `definition` semantics that display meaningful information. When `see` is applied to objects of classes `code-definition` or `colon-definition`, it will produce a disassembly:

```
' spaces see
code spaces ( ecx: integer -- ecx: changed )
0041FC6C: ecx +00 cmp,
0041FC6F: 0041FC79 jle,
0041FC71: 00406102 call, space
0041FC76: ecx dec,
0041FC77: 0041FC6C jmp,
0041FC79: ret,
endcode ok
```

Objects of other classes display other information. Feel free to try it out:

```
' c/l see
64 cast unsigned constant c/l ok
```

Forth 2012 specifies `SEE` without a parameter. It is supposed to parse the name of the definition. And this is exactly what StrongForth's second overloaded versions of `see` does:

```
: see ( -- )
  ' see ; 1 retreat
```

With this word, which has to step back behind the virtual method version, you can apply `see` in the same way as `SEE` in Forth 2012:

```
see space
code space ( -- )
00406102: eax push,
00406103: al 20 mov,
00406105: 004060D2 call, .
0040610A: eax pop,
0040610B: ret,
endcode ok

see latest
9205448 cast definition value latest ok
```

[if], [else] and [then]

The implementations of [if], [else] and [then] are identical to those suggested in section A.15 of the ANS Forth specification, the predecessor of Forth 2012. Some minor deviations are caused by words that are not available in StrongForth, like WORD, 2DUP, 2DROP and S":

```
: [else] ( -- )
  1
  begin
    begin parse-name dup
    while over over " [if]" compare 0=
      if drop drop 1+
      else over over " [else]" compare 0=
        if drop drop 1- dup
          if 1+
          then
            else " [then]" compare 0=
              if 1-
              then
            then
          then dup 0=
          if drop exit
          then
        repeat drop drop refill invert
    until drop ; immediate
: [if] ( single -- )
  0= if [compile] [else] then ; immediate
: [then] ; immediate
```

[defined] and [undefined]

[defined] and [undefined] both return a flag indicating whether a word with a given name has been defined or not, respectively. The flag is typically consumed by [if]. The implementation consists of a context vocabulary search with no additional search criterion:

```
: [defined] ( -- flag )
  parse-name null single no-criterion search-context
  nip ; immediate
: [undefined] ( -- flag )
  [compile] [defined] invert ; immediate
```

Note that [defined] only checks for the existence of a word with the parsed name. It can not look for a word with a specific stack diagram, as might make sense if overloaded versions exist.

SYNONYM vs. alias

In the *Programming-Tools Extension* word set, Forth 2012 specifies a word called `SYNONYM`, that creates a copy of an existing word with a new name. `SYNONYM` parses first the new name and then the old name.

This technique only works if the words in the vocabulary have unique names. A StrongForth version of `SYNONYM` wouldn't be able to select a specific overloaded version of a word. Let's assume we wanted to define a synonym for `bye` with the name `goodbye`, that returns zero as an error code to the calling process.

```
synonym goodbye bye
```

wouldn't do the job, because StrongForth has two overloaded versions of `bye`:

```
words bye  
bye ( integer -- )  
bye ( -- ) ok
```

The hypothetical implementation of `synonym` would find the first version of `bye`, which expects an error code on the stack, instead of the second version with no input parameter.

Before Forth 2012, a word called `ALIAS` was quite popular in Forth:

```
ALIAS ( xt "name" -- )
```

`ALIAS` defines a word with the parsed name that has the semantics specified by the execution token `xt`. With an equivalent word `alias` in StrongForth, the above example could be solved like this:

```
( -- )' bye alias goodbye ok  
latest . goodbye ( -- ) ok
```

The definition of `alias` in StrongForth is a bit tricky. Because the data type of the created alias must be the same as the one of the prototype, alias has to use `(new)` instead of `new`. The parameter of `(new)` is the virtual method table of the class of the original definition:

```
: alias ( definition -- )  
  dup parse-name rot dup vtable -> definition (new) definition  
  state @ new stack-diagram rot over params-alias, over params!  
  enddef ;
```

The constructor of class `definition` has to be executed explicitly. It is a dedicated version of the constructor that copies the members of the original definition. However, a special treatment is required for the stack diagram. The members `#params`, `#input-params` and `'params` may not simply be copied by the constructor. Otherwise, deleting either the original or the alias would free the dynamic memory allocated for the stack diagram shared by both definitions. Instead, `alias` creates a new stack diagram for the alias definition that is a one-to-one copy of the stack diagram of the original definition.

You can even specify a new stack diagram for the alias definition, like in this example:

```
' here . here ( memory-space -- address ) ok  
' here alias chere ( memory-space -- caddress ) ok  
latest . chere ( memory-space -- caddress ) ok
```

This works fine in this case. However, defining an alias with a new stack diagram is potentially dangerous, because the stack diagram is not type checked. Consider the following example:

```
: 2+ ( integer -- 1st ) 2 + ; ok
7 2+ . 9 ok
latest alias 2+ ( address -- 1st ) ok
here dup . 2+ . 8922704 8922704 ok
```

The address remains unchanged! So, what went wrong? The problem becomes obvious if we see the two versions of 2+:

```
latest prev see
code 2+ ( ecx: integer -- ecx: 1st ecx: changed )
00428764: ecx +02 add,
00428767: ret,
endcode ok
latest see
code 2+ ( ebx: address -- ebx: 1st ecx: changed )
00428764: ecx +02 add,
00428767: ret,
endcode ok
```

The default register of data type `integer` is `ecx`, while the default register of data type `address` is `ebx`. These registers are assigned to the input and output parameters within the stack diagrams. And this means, we can't simply apply the code of 2+ for items of data type `integer` to the version for items of data type `address`. The above example of `chere` works only, because the default register for items of data type `address` is also `ebx`.

This problem is the reason why `alias` is not predefined in Strongforth. `alias` is not required anyway, because it is always possible to define a simple colon definition. For example,

```
: bin ( fam -- 1st ) ;
: chere ( memory-space -- caddress ) here cast caddress ;
```

and

```
' noop alias bin ( fam -- 1st )
' here alias chere ( memory-space -- caddress )
```

are equivalent. There are not even performance penalties, because `?alias` within the definition of `; eliminates` the additional level of procedure calls.

If you want to use the definition of `alias` presented above anyway, you have to be careful when assigning explicit stack diagrams to alias definitions. Sometimes it works, sometimes not. The good news is: With aliases of `noop` or other words that have no input and output parameters, you won't get into trouble.

23 The Search Order

Displaying Vocabularies

You already know that each vocabulary is included in one of two linked lists, the context vocabulary list and the hidden vocabulary list. The context vocabulary list determines the search order.

`order` displays the names of the current vocabulary and a list of the names of all context vocabularies. The implementation is straightforward. However, we first need an overloaded version of `.` for displaying the name of a vocabulary:

```
: . ( vocabulary -- )
  dup friend? if . drop else drop >definition .name then ;

: order ( -- )
  cr ." current: " current @ .
  cr ." context: " context @
  begin dup 0<>
  while dup . next
  repeat drop ;
```

By default, `>definition` obtains the created definition the vocabulary in its data field is associated with. `.` then displays the name of this definition as the name of the vocabulary. But there's a second case to consider. The combined private and protected vocabularies of classes have no associated created definition. Those vocabularies can be recognized by the fact that they have friend classes. So, if a vocabulary has friends, the name of the class it belongs to is displayed instead. `friend?` returns the class as an object of data type `data-type` together with a `true` flag.

Using `order`, you can usually see that integer and floating-point numbers are implemented as special vocabularies:

```
order
current: forth
context: forth integer-lit float-lit  ok
```

The Search-Order Word Set

The words presented in this section are usually not required in StrongForth applications. They are provided to support compatibility with Forth 2012. They can be included from the source file `order.sf`.

Forth 2012 specifies the word `GET-ORDER`, which returns the search order in the following format:

```
GET-ORDER ( -- widn ... wid1 n )
```

The stack diagram is ambiguous, because the number of word lists cannot be determined at compile time. Since ambiguous stack diagrams are not allowed in StrongForth, an alternative solution has to be found. StrongForth's version of `get-order` allocates a chunk of dynamic memory where the context vocabulary list and the hidden vocabulary list are being stored. To ensure that `get-order`

`und` `set-order` are used strictly pairwise, `get-order` returns the pointer to this memory chunk as a new data type:

`dt single procreates search-order`

First, we need a word that determines the length of a vocabulary list:

```
: (#order) ( unsigned vocabulary -- 1st )
  begin dup 0<>
  while swap 1+ swap next
  repeat drop ;
```

Next, two internal words save and restore linked lists of vocabularies, starting at a given vocabulary. The linked list is stored as an array of vocabularies. A null vocabulary indicates the end of the array:

```
: (get-order) ( vocabulary address -> vocabulary -- 2nd )
  begin over over ! 1+ over 0<>
  while swap next swap
  repeat nip ;

: (set-order) ( address -> vocabulary -- 1st )
  begin dup @ dup 0<>
  while swap 1+ tuck @ swap next!
  repeat drop 1+ ;
```

`get-order` and `set-order` can be defined based on these three internal definitions:

```
: get-order ( -- search-order )
  2 context @ (#order) hidden @ (#order)
  cells allocate -> vocabulary
  dup context @ swap (get-order) hidden @ swap (get-order) drop
  cast search-order ;

: set-order ( search-order -- )
  cast address -> vocabulary dup
  dup @ context ! (set-order)
  dup @ hidden ! (set-order) drop free ;
```

`get-order` starts with determining the size of the array. The array shall have enough space for all vocabularies in the context and the hidden vocabulary lists, plus two null vocabularies indicating the ends of these lists. After allocating the array, `get-order` uses two times `(get-order)` to save the two vocabulary lists in the array.

`set-order` restores the two vocabulary lists with `(set-order)` and then frees the dynamic memory that has been allocated by `get-order`.

The three words `(#order)`, `(get-order)` and `(set-order)` are pure internal definitions. They are deleted within `order.sf` after they have been used in `get-order` and `set-order`:

```
' (#order) delete
' (get-order) delete
' (set-order) delete
```

Forth 2012 specifies that `SET-ORDER` shall empty the search order if its size parameter is zero. `SET-ORDER` shall further restore the default search order if it is executed with `-1` as the size parameter. Both special operations are not possible with the StrongForth version of `set-order`, because there is no size parameter. To empty the search order, i. e., the context vocabulary list, `ignore-all` can be used. `only` restores the default search order:

```

: ignore-all ( -- )
  begin context @ 0<>
  while [compile] ignore
  repeat ;
: only ( -- )
  ignore-all [compile] forth ;

```

The Forth 2012 word `also` is not required in StrongForth. `>context` adds a vocabulary to the context vocabulary list without the necessity to allocate space for it. For example, instead of

```
also msvcrt
```

you can simply write

```
msvcrt
```

For compatibility reasons, `also` is provided as a dummy word:

```
: also ( -- ) ;
```

At the end of this section, three words from the Forth 2012 *Search-Order* word sets are still missing. Here they are:

```

forth context @ constant forth-vocabulary
: get-current ( -- vocabulary )
  current @ ;
: set-current ( vocabulary -- )
  current ! ;

```

marker

The semantics of the Forth 2012 word `MARKER` is somewhat related to `GET-ORDER` and `SET-ORDER`. In StrongForth, `marker` is a defining word, whose execution semantics is to restore certain system states that were saved in an object of a dedicated class at the time `marker` was executed:

```

dt object procreates marker-class
class marker-class
  private definitions
  null definition member 'latest
  null memory-space member 'default-memory-space
  null address 3 members 'memory-space-pointers
  null address -> vocabulary member 'next
  null address -> definition member 'last

  : vocabularies, ( vocabulary -- )
    begin dup , dup 0<>
    while next
    repeat drop ;

  : last-definition, ( vocabulary -- )
    begin dup 0<>
    while dup last , next
    repeat drop ;

```

```

: restore-vocabularies ( address -> vocabulary marker-class
  -- 1st )
  local this
  begin dup @ dup 0<>
  while 'last @ @ over last! 1 'last +!
    swap 1+ tuck @ swap next!
  repeat drop ;

forth definitions

: marker-class ( marker-class -- 1st )
  local this
  latest 'latest !
  default-memory-space @ 'default-memory-space !
  data-space here 'memory-space-pointers !
  stack-space here 'memory-space-pointers 1+ !
  code-space here 'memory-space-pointers 2 + !
  data-space default
  here -> vocabulary 'next !
  current @ ,
  context @ vocabularies,
  hidden @ vocabularies,
  here -> definition 'last !
  context @ last-definition,
  hidden @ last-definition,
  this ;

:noname ( marker-class -- )
  local this
  'next @
  dup @ dup current ! last swap 1+
  dup @ context ! restore-vocabularies 1+
  dup @ hidden ! restore-vocabularies drop
  current @ last swap (forget)
  'memory-space-pointers 2 + @
  code-space here - code-space allot
  'memory-space-pointers 1+ @
  stack-space here - stack-space allot
  'memory-space-pointers @
  data-space here - data-space allot
  'default-memory-space @ default
  'latest @ to latest
  this [parent] delete ; is delete

endclass

: marker ( -- )
  new marker-class create ,
  does> ( address -> marker-class -- ) @ delete ;

```

Class `marker-class` has a number of private data members in which the constructor saves `latest`, `default-memory-space` and the memory space pointers of the code space, the data space and the stack space. Additional memory spaces created by the application are not considered. Furthermore, the constructor of class `marker-class` creates two arrays in the data space that save the vocabulary structure. The vocabulary structure consists of the contents of `current`, `context` and `hidden` as well as the two private members `'last-definition` and `'next-vocabulary` of all vocabularies.

Three private methods create the two arrays in the data space and restore the vocabulary status from the saved data. `vocabularies`, starts with the contents of either `context` or `hidden`, traversing the linked list and saving all vocabularies on the way, including the terminating null vocabulary. `last-definition`, also traverses a vocabulary list, saving the respective last definition of each vocabulary. `restore-vocabularies` works through a part of the two arrays in the data space to restore the `context` or `hidden` vocabulary structure in a single pass each. Private member `'last` is updated in each turn of the loop.

The runtime code of the word created by `marker` just deletes the object of class `marker-class` that is stored in its data field. `marker-class`'s version of `delete` restores the system state based on the data saved in the data space and in the members. Finally, the object deletes itself.

24 Floating-Point Numbers

We've already stumbled over floating-point numbers in quite a number of overloaded words, like those for stack shuffling, memory access, arithmetic and literals. You also know the specific versions of words for single-precision and double-precision floating-point numbers, whose name begin with `sf` or `df`. Here are some examples:

```
nip ( float double -- 2nd )
@ ( address -> float -- 2nd )
, ( float -- )
fill ( address -> float unsigned 2nd -- )
* ( float float -- 1st )
abs ( float -- 1st )
0< ( float -- flag )
1+ ( address -> float -- 1st )
literal ( float -- )
constant ( float -- )
sfhere ( -- sfaddress )
```

Other words that deal with floating-point numbers are the data conversion operations:

```
s>f ( signed -- float )
s>f ( single -- float )
d>f ( double -- float )
d>f ( signed-double -- float )
f>d ( float -- signed-double )
f>s ( float -- signed )
```

Not all applications use floating-point numbers. That's the reason why some of the words that have already been presented in earlier chapters, and quite a number of additional words from the Forth 2012 *Floating-Point* word set are defined in a separate source file named `float.sf`. Before you begin using floating-point numbers, you have to include `float.sf`:

```
include float.sf
```

The Floating-Point Stack

The StrongForth programming model is based on a unified stack model for floating-point numbers. This means, floating point numbers are placed in stack diagrams inbetween single-cell and double-cell data types, as if all items were located on a common data stack. You do not need to keep track of two stacks in parallel. Instead, you can simply assume floating-point numbers were on the data stack, like in this example, which demonstrates the use of `rot (single double float - 2nd 3rd 1st)`:

```
true -123456789123456789. pi .s flag signed-double float ok
rot .s signed-double float flag ok
. . . true 3.141592653589793 -123456789123456789 ok
```

Floating-point numbers are kept on the hardware floating-point stack of the processor's floating-point unit. This technique has a considerable performance advantage. The attributes of the data types on the data type heaps tell if a floating-point number is located on the floating-point heap.

The hardware floating-point stack has a depth of 8. It wraps around from 0 to 7 and from 7 to 0, so it's a bad idea to cause the floating-point stack to overflow. The stack pointer can be fetched and stored:

```
fp@ ( -- unsigned )
fp! ( unsigned -- )
```

fp@ actually returns the number of unused positions on the floating-point stack:

```
quit    ok
fp@ . 8  ok
pi fp@ . 7  ok
drop fp@ . 8  ok
```

Using fp@ and especially fp! in applications is dangerous. If you need to know the current depth of the hardware floating-point stack, fdepth is the preferred word:

```
: fdepth ( -- unsigned )
  8 fp@ - ;
```

Floating-Point Arithmetic

As already stated in connection with + and -, StrongForth omits the leading F in the names of floating-point words as specified by Forth 2012. Operator overloading allows StrongForth to use the same names as those of the words for integer arithmetic. Consequently, the leading F can be omitted for dedicated floating-point arithmetic words as well. For example, the Forth 2012 word FSIN is just named sin in StrongForth. You can use floating-point numbers in the same way and with the same operators as you use single-precision or double-precision integer numbers:

```
6.279e+6 +100000.e0 +200000.e0 .s float float float  ok
max dup . 200000.  Ok
+ . 6479000.  ok
```

Here are those specific operations that can be directly calculated by the hardware floating-point unit. They are implemented as pure machine code words in the kernel of StrongForth:

```
pi ( -- float )
round ( float -- 1st )
floor ( float -- 1st )
trunc ( float -- 1st )
ln ( float -- 1st )
lnp1 ( float -- 1st )
log ( float -- 1st )
sin ( float -- 1st )
cos ( float -- 1st )
sincos ( float -- 1st 1st )
atan2 ( float float -- 1st )
sqrt ( float -- 1st )
** ( float float -- 1st )
alog ( float -- 1st )
expm1 ( float -- 1st )
exp ( float -- 1st )
```

pi returns the value of π . This word is not specified by Forth 2012, but it is provided by StrongForth, because it is a predefined constant of the hardware floating-point unit.

And these are the derived floating-point words, whose definitions are contained in `float.sf`:

```
: tan ( float -- 1st )
  sincos / ;

: tanh ( float -- 1st )
  expm1 dup dup 1e0 + / over over + rot rot - 2e0 + / ;

: sinh ( float -- 1st )
  expm1 dup dup 1e0 + / + 2e0 / ;

: cosh ( float -- 1st )
  expm1 dup dup 1e0 + / - 2e0 / 1e0 + ;

: atan ( float -- 1st )
  1e0 atan2 ;

: asin ( float -- 1st )
  dup dup * negate 1e0 + sqrt atan2 ;

: acos ( float -- 1st )
  dup dup * negate 1e0 + sqrt swap atan2 ;

: atanh ( float -- 1st )
  dup lnpl swap negate lnpl - 0.5e0 * ;

: asinh ( float -- 1st )
  dup dup * 1e0 + sqrt / atanh ;

: acosh ( float -- 1st )
  dup dup * 1e0 - sqrt swap / atanh ;

: ~ ( float 1st float -- flag )
  dup 0= if drop = exit then dup 0>
  if rot rot - abs swap <
  else rot rot over over - abs rot abs rot abs + rot abs * <
  then ;
```

Floating-Point Numbers In Memory

When storing floating-point numbers in memory locations, e. g. in a variable or an array, the kind of address decides in which format it is stored. A floating-point number stored in an address of data type `sfaddress` always has single-precision (32 bit) format. A floating-point number stored in an address of data type `dfaddress` always has double-precision (64 bit) format. And finally, a floating-point number stored in address `-> float` uses all 80 bits of precision. Words for calculating the size in address units of floating-point numbers can thus be defined like this:

```
: sfloats ( integer -- 1st )
  cells ;

: dfloats ( integer -- 1st )
  2* cells ;

: floats ( integer -- 1st )
  10 * ;
```

In StrongForth, these three words are only needed for low-level address arithmetic with unspecified addresses, because address arithmetic with explicit floating-point addresses automatically considers the size of floating-point numbers in memory:

```

here .s . address 8988220 ok
here 2 floats + . 8988240 ok
here -> float 2 + . 8988240 ok

```

Because single-precision and double-precision floating-point numbers fill exactly one cell and two cells of memory, respectively, their alignment is the same as for cells:

```

: sfloat ( memory-space -- )
  align ;

: sfloat ( -- )
  align ; 1 retreat

: sfaligned ( address -- 1st )
  aligned ;

: dfloat ( memory-space -- )
  align ;

: dfloat ( -- )
  align ; 1 retreat

: dfaligned ( address -- 1st )
  aligned ;

```

With full-precision floating-point numbers, things look differently. Their size of 80 bits corresponds to two and a half cells. If an array of 80-bit floating-point numbers is packed tightly, not all individual numbers can be cell aligned. An alignment of two address units is more appropriate.

80-bit floating-point number										80-bit floating-point number									
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
cell				cell				cell				cell				cell			

```

: ffloat ( memory-space -- )
  local this this here 1 cast logical and if 1 this allot then ;

: ffloat ( -- )
  default-memory-space @ ffloat ; 1 retreat

: faligned ( address -- 1st )
  cast unsigned 2 aligned cast address ;

```

Floating-Point Literals

For converting a character string representation of a floating-point number into the internal binary representation, Forth 2012 specifies the word `>FLOAT`. The conversion complies with a quite universal rule, which even allows character strings like `.4d8` or `6-3` to be converted into floating-point numbers:

```

convertible string := <significand>[<exponent>]
<significand>      := [<sign>]{<digits>[.<digits0>] | .<digits>}
<exponent>        := <marker><digits0>
<marker>          := {<e-form> | <sign>}
<e-form>          := <e-char>[<sign>]
<sign>            := { + | - }
<e-char>          := { D | d | E | e }
<digits>          := <digit><digits0>
<digits0>         := <digit>*
<digit>           := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

```

The interpreter's conversion rule is somewhat more restricted:

```

convertible string := <significand><exponent>
<significand>      := [<sign>]<digits>[.<digits0>]
<exponent>        := e[<sign>]<digits0>
<sign>            := { + | - }
<digits>          := <digit><digits0>
<digits0>         := <digit>*
<digit>           := { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }

```

In the same way as for integer numbers, these rules are handled by a vocabulary. At the beginning, we have a small set of words that support the conversion:

```

: /10^n ( float signed -- 1st )
  +10 s>f swap
  begin dup 0<>
  while +2 /mod swap
    if rot rot tuck / swap rot
    then swap dup * swap
  repeat drop drop ;

: *10^n ( float signed -- 1st )
  dup 0< if negate /10^n exit then
  +10 s>f swap
  begin dup 0<>
  while +2 /mod swap
    if rot rot tuck * swap rot
    then swap dup * swap
  repeat drop drop ;

: not-decimal? ( -- flag )
  base @ 10 <> ;

: ?decimal ( -- )
  not-decimal? if -40 throw then ;

```

/10^n divides a floating-point number by 10 raised to the power of the second parameters. The second parameter of data type signed must be non-negative. *10^n multiplies a floating-point number by 10 raised to the power of the second parameter. This word allows for the second parameter to be positive, zero or negative.

According to the Forth 2012 specification, floating-point numbers shall always be represented by decimal digits. The number-conversion radix base has to be 10 whenever a floating-point number is being converted or interpreted. not-decimal? returns a true flag if this condition is violated, otherwise it returns a false flag. ?decimal throws an exception is the number-conversion radix base is different from 10.

Now, here is the vocabulary class that implements the conversion rules for floating-point numbers:

```
dt vocabulary procreates float-vocabulary
class float-vocabulary
  forth definitions
  : float-vocabulary ( float-vocabulary -- 1st )
    vocabulary ;
  private definitions
  null float constant float-number
  : >char ( caddress -> character unsigned 2nd -- 1st 3rd flag )
    locals( c ) dup
    if over @ c = if /string true else false then
    else false
    then ;
  : >period ( caddress -> character unsigned -- 1st 3rd flag )
    [char] . >char ;
  : >sign ( caddress -> character unsigned -- 1st 3rd signed )
    dup
    if over @ >sign dup if rot rot /string rot then
    else +0
    then ;
  : >e-char ( caddress -> character unsigned -- 1st 3rd flag )
    dup
    if over @ [ 0 bit 5 bit or invert ] literal and [char] D =
      dup if rot rot /string rot then
    else false
    then ;
  : >marker ( caddress -> character unsigned -- 1st 3rd signed )
    >e-char >r >sign r> over
    if drop else if 1+ then then ;
  : >marker' ( caddress -> character unsigned -- 1st 3rd signed )
    [char] e >char
    if >sign dup 0= if 1+ then else +0 then ;
  : >exp ( caddress -> character unsigned signed
    -- 1st 3rd 4 th flag )
    dup
    if >r 0. rot rot >number rot cast signed r> * true
    else false
    then ;
  : >digits ( float caddress -> character unsigned
    -- 1st 2nd 4 th signed )
    dup >r
    begin dup
    while over @ digit?
      while >r rot +10 s>f * r> s>f + rot rot /string
      repeat drop
    then r> over - cast signed ;
```

```

: >significand' ( caddress -> character unsigned
  -- float 1st 3rd signed flag )
  [ null float ] literal rot rot >digits
  if >period if >digits else +0 then true
  else +0 false
  then ;

: >significand ( caddress -> character unsigned
  -- float 1st 3rd signed flag )
  >period
  if [ null float ] literal rot rot >digits dup 0<>
  else >significand'
  then ;

: >float' ( caddress -> character unsigned -- float flag )
  >sign +0 locals( sign part ) dup 0= not-decimal? or
  if drop drop [ null float ] literal false
  else >significand'
    if to part >marker' >exp
      if rot drop swap
        if drop drop [ null float ] literal false
        else part - *10^n sign ?negate true
        then
      else drop drop drop drop [ null float ] literal false
      then
    else drop drop drop drop [ null float ] literal false
    then
  then ;

```

forth definitions

```

: >float ( caddress -> character unsigned -- float flag )
  +0 +0 locals( sign part )
  over over -trailing nip 0=
  if drop drop [ null float ] literal true exit
  then
  >sign to sign dup 0= not-decimal? or
  if drop drop [ null float ] literal false
  else >significand
    if to part >marker >exp rot
      if drop drop drop drop [ null float ] literal false
      else rot drop
        if part - *10^n
        else drop part /10^n
        then sign ?negate true
      then
    else drop drop drop drop [ null float ] literal false
    then
  then ;

```

```

private definitions
:noname ( caddress -> character unsigned single search-criterion
float-vocabulary -- definition flag )
drop locals( scp sc ) >float'
if ['] float-number tuck cast float-definition assign
dup scp sc execute
else drop null definition false
then ; is search
endclass

```

The constructor of class `float-vocabulary` is exactly the same as the one of its parent class. Data members are not required, and since there exists only one object of the class, there's no need to have this object as the last input parameters of the methods. All methods except for `>float` are private, because `>float` is the only word that needs to be visible outside the class definition. The interpreter's floating-point number conversion is implemented as a dedicated version of the `search` virtual method.

The two conversion rules give guidance for a reasonable factoring of the StrongForth word `>float` and the private variant `>float'`, that is used by `search`.

`float-number` is the definition that will be returned by `search` after assigning a floating-point number to it, if the conversion was successful.

`>char` checks whether the first character of a string is a given character. If it is, the character is removed from the string and a `true` flag is returned. An empty string or a string that does not start with this character are left unchanged, and a `false` flag is returned. An obvious application of `>char` is `>period`.

`>sign`, `>e-char` and `>marker` are implementations of the sub-rules `<sign>`, `<e-char>` and `<marker>` in the above conversion rules, respectively. The overloaded version `>sign (character -- signed)` in the definition of `>sign` you already know from the class definition of class `integer-vocabulary` in chapter 11. Both `>sign` and `>marker` return the converted sign as `-1` or `+1`, or `0` if the conversion did not succeed. But there's another version of `>marker`, called `>marker'`. This one is for the interpreter's conversion rule. It actually implements the sub-rule `e[<sign>]` within `<exponent>`.

`>exp` and `>digits` both implement `<digits0>`, for the exponent and for the significant, respectively. `>exp` expects the sign converted by `>marker` and returns the converted exponent as a signed single-precision number. `>digits` accumulates decimal digits into a floating-point number and returns the number of digits that were converted:

The conversion rule for the significand allows two different formats:

```

[<sign>]<digits>[.<digits0>]
[<sign>].<digits>}

```

`>significand'` converts a character string according to the first rule, starting after the sign. The conversion fails if the string does not begin with a decimal digit. `>significand` converts according to either of both rules. If the character string starts with a decimal point, it converts the string according to the second rule. Otherwise, it tries the first rule by executing `>significand'`. If the conversion succeeds in either of the two words, they return the converted floating-point number, the remaining string, zero or the number of digits after the decimal point, and a `true` flag. Otherwise, the floating-point number is invalid, the number of digits after the decimal point is zero and the flag is `false`.

With all these private methods of class float-vocabulary, it is possible to implement >float. >float has several exit points that are taken if the format does not comply with the conversion rule. As a special case, >float converts an empty character string into a valid floating-point number with the value zero.

>float' is a variant of >float that applies to the interpreter's conversion rule. An empty string is not considered as a valid floating-point number. Instead of >significand and >marker, >float' uses >significand' and >marker'. The virtual method search uses >float' to convert a character string into a floating-point number. If the conversion succeeds, the floating-point number is assigned to the private constant float-number, which is returned by search.

Finally, a single object of class float-vocabulary is being created. This object becomes the body of a vocabulary named float-lit, whose execution semantics is to make float-vocabulary the head of the context vocabulary list. After executing float-lit and forth, float-lit is at the second position of the context vocabulary list. Floating-point numbers can now be found by the interpreter:

```
create float-lit new float-vocabulary
cast address latest ?created-definition body!
does-vocabulary immediate
float-lit forth
```

Floating-Point Representation

Pictured numeric output of floating-point numbers is based on the Forth 2012 word REPRESENT, which is called represent in StrongForth. represent creates a representation of the absolute value of the significand of a floating-point number and stores it into a buffer. Furthermore, it returns the sign of the significand, the exponent, and a flag telling whether the floating-point number is a valid number:

hex

```
: represent ( float caddress -> character unsigned
  -- signed flag flag )
  ?decimal over over [char] 0 fill rot fcam
  0200 over set? +0 locals( s e ) 4500 swap and
  case 0400 of abs swap max-precision min tuck cast signed
    over log f>s dup to e - *10^n
    over [ 1 s>f ] literal swap cast signed *10^n
    over round > invert
    if +1 /10^n (represent) e 1+ to e
    else (represent)
    then true endof
  4000 of drop drop drop true endof
  0000 of drop " unsupported" fill false endof
  0100 of drop " nan" fill false endof
  0500 of drop " infinity" fill false endof
  4400 of drop " denormalized" fill false endof
  \ default \ drop drop " free" fill false
  endcase e s rot ;
```

decimal

`fxam` is a word that returns the contents of the hardware floating-point processor's status word register after examining the number on top of the floating-point stack:

```
fxam ( -- logical )
```

Bits 14, 10, 9 and 8 are set depending on the status of this number:

Status	bit 14	bit 10	bit 9	bit 8
unsupported	0	0	s	0
NaN (Not a Number)	0	0	s	1
valid	0	1	s	0
infinity	0	1	s	1
zero	1	0	s	0
free	1	x	x	1
denormalized	1	1	s	0

`x` is undefined. `s` is 0 if the number is positive, and 1 if it is negative. `represent` begins with filling the character buffer with '0' digits and initializing two locals with the sign bit `s` of the floating-point number and zero as the signed exponent `e`. Then, a `case ... endcase` structure handles the different cases as indicated by bits 14, 10 and 8 of the floating-point processor's status word.

If the floating-point number is zero, we're already done. The character buffer is already filled with '0' digits and the exponent is zero. All that remains is cleaning up the stack and returning a `true` flag.

If the floating-point number is invalid (unsupported, NaN, infinity, free or denormalized), an overloaded version of `fill` is used to fill the buffer with a character string indicating its status with a text string. If the buffer is longer than the character string, trailing blanks are added. If the buffer is too short to take the whole character string, the character string is cut off:

```
: fill ( caddress -> character unsigned 1st 3rd -- )
  locals( to-caddr to-len from-caddr from-len )
  from-caddr to-caddr to-len from-len min move
  from-len to-len <
  if to-caddr to-len from-len /string blank
  then ;
```

Now, what happens with a valid floating-point number? `represent` calculates the decimal exponent of the floating-point number and then scales the number in such a way that its integer part contains as many digits as fit into the buffer. If the buffer size is larger than the maximum possible precision of the significand, the number of digits is reduced accordingly. (`represent`) uses the hardware floating-point unit's BCD arithmetic to convert the integer part of the floating-point number right-aligned to a character string:

```
(represent) ( caddress -> character unsigned float -- )
```

With the hardware floating point processor, the maximum precision of a floating number is 18 digits. The number of significant digits for pictured numeric output of floating-point numbers is controlled by the value `precision`. By default, StrongForth displays up to 16 digits. The maximum is 18, but with 18 significant digits rounding errors become visible even after simple arithmetic operations. `set-precision` implements a Forth 2012 word that changes the precision:

```
18 constant max-precision
16 value precision

: set-precision ( unsigned -- )
  max-precision min to precision ;
```

The overloaded version of `.` for floating-point numbers uses `represent` and a few simple new words. The semantics of `.` is the same as the one of `F.` in Forth 2012.

```
: period ( -- )
  [char] . . ;

: .sign ( flag -- )
  if [char] - . then ;

: . ( float -- )
  line precision represent rot local exp
  if .sign line precision exp 0>
    if over over exp cast unsigned min type
      exp precision - zeros
      period precision exp cast unsigned min /string
    else zero period exp negate zeros
    then [char] 0 -trailing
  else drop line precision -trailing
  then type space ;
```

`period` simply displays one period character. An optional sign character is displayed by `.sign`. `.` starts with converting a floating-point number into the number of desired digits. The transient area that is used for numeric output of integers is the character buffer `line`. Four cases have to be considered, the first two of which are handled by the same conditional branch within the definition of `.`:

- The integer part of the absolute value of the number has more digits than specified by `precision`: `.` displays an optional sign, all significant digits, the required number of zeros, and a decimal point. Example: `12345678900000.`
- The integer part of the absolute value of the number has less digits than specified by `precision`, but it's greater than or equal to 1: `.` displays an optional sign, the digits that belong to the integer part, a decimal point, and then the digits that belong to the decimal fraction. Example: `-12345.6789`
- The absolute value of the number is less than 1: `.` displays an optional sign, a zero digit, a decimal point, the required number of zeros, and finally the significant digits. Example: `0.00123456789`
- The number is invalid: `.` displays the character string provided by `represent` after removing trailing spaces. Example: `infinity`

The format without exponent is not very handy whenever floating-point numbers get really small or really large. In such cases, scientific or engineering representation with an explicit exponent makes more sense. The exponent is always displayed with a sign character, even if it is positive or zero, and three digits:

```
: .sign+ ( flag -- )
  if [char] - else [char] + then . ;

: .exponent ( signed -- )
  [char] e . dup 0< .sign+
  abs +999 min s>d <# # # # #> type ;
```

`s.` and `e.` both use `(se.)` to display a floating-point number with an explicit exponent. `(se.)` has an additional modulus parameter of data type `signed` that is used to adjust the exponent. The exponent is always a multiple of the modulus, and the number of digits to the left of the decimal point is between 1 and the modulus. For scientific representation, the modulus is +1, for engineering representation, it is +3:

```

: (se.) ( float signed -- )
  local exp line precision represent
  if .sign line @ [char] 0 = if 1+ then
    dup 1- exp mod dup 0< if exp + then 1+ to exp
    line precision over exp cast unsigned type
    period exp /string type exp - .exponent
  else drop drop line precision -trailing type
  then space ;

: s. ( float -- )
  +1 (se.) ;

: e. ( float -- )
  +3 (se.) ;

```

Here are a few examples:

```

123456789.e5 s. 1.234567890000000e+013 ok
-123456789.e5 e. -12.34567890000000e+012 ok
+123456789.e-15 e. 123.4567890000000e-009 ok

```

Finally, remember that StrongForth provides a word named `split` that returns the lower and upper half of a double-cell item as two single-cell items:

```
split ( double -- single single )
```

An overloaded version of `split` is available for floating-point numbers:

```
split ( float -- signed 1st )
```

This word returns the exponent and the significand of a floating-point number as separate values, like in these examples:

```

pi split . . 1.570796326794897 1 ok
1024.885e0 split . . 1.0008642578125 10 ok

```

Note that the numeric base of exponents is binary, not decimal. A floating-point number is thus represented like this:

```
<value> = <significand> * (2 ** <exponent>)
```

25 Blocks

The Block File

Although block support is a somewhat anachronistic feature of Forth 2012, StrongForth provides the complete *Block* and *Block Extension* word sets as specified by Forth 2012. However, since blocks are not required in many applications, the StrongForth block word set is included in the `block.sf` source file. If you want to use blocks in StrongForth, you have to include this source file:

```
include block.sf
```

In its present version, StrongForth supports only one block buffer, which is initialized with spaces:

```
1024 constant c/b
bl c/b cvariables block-buffer
```

All blocks are contained in a single file with the name *StrongForth.blk*, which is located in the same directory as the executable *StrongForth.exe*. All blocks in the block file have been filled with spaces. Although the block file already exists, you can imagine it being created with the following code. If you prefer a different size than 4096 blocks, feel free to delete the block file and re-create it.

```
: init ( file unsigned -- 1st )
  0 do dup block-buffer c/b rot write loop ;
" StrongForth.blk" w/o create
4096 init close
latest delete
```

By including `block.sf`, the block file is opened and its handle is assigned to the constant `block-file`. `#blocks` is the number of blocks as calculated from the size of the block file:

```
" StrongForth.blk" r/w open constant block-file
block-file size c/b m/ constant #blocks
```

Block numbers may be between 1 and `#blocks`. A check for valid block numbers is implemented in `?block`. `?block` throws an exception if its parameter is either zero or greater than `#blocks`:

```
: ?block ( unsigned -- 1st )
  dup #blocks 1+ 1 within if -35 throw then ;
```

`position-block` sets the file pointer for subsequent block read or write operations within the block file:

```
: position-block ( unsigned -- )
  ?block 1- c/b m* block-file reposition ;
```

`blk#` is a variable that contains the number of the block the block buffer is assigned to, or zero if the block buffer is not assigned to any block:

```
0 variable blk#
```

Another variable contains a flag that tells whether the block in the block buffer has been changed with respect to the contents of the block file. To set the update flag, `update` can be used to mark the block buffer as updated:

```
false variable updated
```

```
: update ( -- )  
  true updated ! ;
```

`save-buffers` writes the contents of the block buffer to the corresponding position within the block file, if the block buffer has been updated. After this has been done, the update flag can again be cleared.

```
: save-buffers ( -- )  
  updated @  
  if blk# @ position-block  
    block-buffer c/b block-file write  
    false updated !  
  then ;
```

Based on these words, here are the implementations of the Forth 2012 words `EMPTY-BUFFERS` and `FLUSH`:

```
: empty-buffers ( -- )  
  0 blk# ! false updated ! ;  
  
: flush ( -- )  
  save-buffers empty-buffers ; 1 retreat
```

Note the phrase `1 retreat` at the end of the definition of `flush`. This is necessary, because `flush (--)` overloads an already existing version with a parameter:

```
flush ( file -- )
```

This is actually the StrongForth version of the Forth 2012 word `FLUSH-FILE`. The name has been changed, because the two versions can be distinguished by the parameter of data type `file`. If the versions for blocks remained the more recent one in the vocabulary, it would hide the version for files, which would thus become inaccessible.

Now we can define the basic block create and read words. The assigned and updated block buffer is automatically written to the block file whenever the block buffer is flushed or re-assigned:

```
: buffer ( unsigned -- caddress -> character )  
  ?block dup blk# @ =  
  if drop  
  else save-buffers blk# !  
  then block-buffer ;  
  
: block ( unsigned -- caddress -> character )  
  ?block dup blk# @ =  
  if drop  
  else save-buffers dup position-block  
    block-buffer c/b block-file read drop blk# !  
  then block-buffer ;
```

`buffer` expects a block number and returns the address of the block buffer. It does not read an existing block from the block file. If the block buffer is already assigned to a different block *and* has been updated, `buffer` saves the block buffer and then reassigns it to the new block. Since StrongForth has only one block buffer, the output parameter of `buffer` is always the same.

The second method to assign the block buffer to a block is to access an already existing block. The difference to the definition of `buffer` is that `block` performs a physical read operation after saving the previous contents of the block buffer. Of course, this is only necessary if the new block is not identical to the old one.

Both words throw exceptions if an attempt is made to use an invalid block:

```
#blocks . 4096 ok
1 block drop ok
4096 block drop ok
0 block
0 block ? invalid block number
caddress -> character
4097 block
4097 block ? invalid block number
caddress -> character
```

Block Structure

According to the Forth 2012 specification, a block is just *1024 characters of data on mass storage, designated by a block number*. However, it is usually interpreted as being divided into 16 lines of 64 characters each. StrongForth defines the number of characters per line:

```
64 constant c/l
```

The contents of a block can be displayed with `list`. For later reference, `list` stores the block number in the variable `scr`:

```
0 variable scr
: list ( unsigned -- )
  scr ! base @ decimal scr @ block cr ." scr #" scr @ . cr
  [ c/b c/l / ] literal 0
  do i 2 .r space dup i c/l * + c/l -trailing type cr
  loop drop base ! ;
```

Because the block number and the line numbers are to be displayed as decimal numbers, `list` has to temporarily switch to decimal numeric base. This is an example of how `list` displays an empty block, consisting of 1024 space characters:

```
1 list
scr #1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
ok
scr @ . 1 ok
```

Blocks As Input Source

A block can be made the input source, just like the user input device, a source file or a character string. All that needs to be done is deriving a dedicated child class from class `input-source`:

```
dt input-stream procreates block-input-stream
class block-input-stream
  forth definitions protected
  null unsigned member blk
  : block-input-stream ( unsigned block-input-stream -- 2nd )
    locals( this ) erase
    ?block dup blk ! block this 'buffer ! this ;
  :noname ( block-input-stream -- flag )
    locals( this ) 0 this >in !
    this 'buffer @ [ block-buffer c/b + c/l - ] literal < dup
    if #buffer @
      if c/l this 'buffer +!
      else c/l #buffer !
      then
    then ; is refill
  :noname ( input-stream block-input-stream -- flag )
    locals( this ) cast block-input-stream
    dup 'buffer @ block-buffer dup c/b + within
    if dup blk @ this blk @ =
      if dup 'buffer @ this 'buffer !
      dup #buffer @ this #buffer !
      dup >in @ this >in ! false
      else true
      then
    else true
    then swap [parent] delete ; is restore-input
endclass
```

Class `block-input-stream` has an additional public member `blk` with respect to its parent class. In contrast to most data members, this one is public, because it corresponds to the Forth 2012 variable `BLK`. There might be some instances where access to this variable from outside the class definition is required, although `blk` does not, as in Forth 2012, specify the input source.

The constructor expects a block number on the stack. This is the block that will subsequently serve as the input source, with `'buffer` being initialized with the address of the first line. Note that this constructor leaves the buffer size `#buffer` being zero. Its value will be changed to `c/l` once the input buffer is made available with `refill`. The content of the member `/buffer` also remains zero, indicating that the buffer of this input stream was not allocated from dynamic memory and may not be freed when the object of class `block-input-stream` is being deleted.

The contents of a block are parsed line by line. `'buffer` contains a pointer to first character of the currently parsed line. As long as the last line has not been parsed, `refill` makes the next line available by resetting `>in` to zero and adding `c/l` to the buffer pointer. An exception is the first time `refill` is executed after creating the block input stream, which is indicated by `#buffer` still being zero. In this case, instead of advancing the buffer pointer, the buffer size is set to the line length. This value remains until the block input stream is being deleted.

`save-input` reuses the mechanism of the parent class `input-source` to create a copy of an existing block input source. Restoring a block input source requires that the buffer address is within the limits of the block buffer, and that the block number is the same. If these conditions are met, all that needs to be done is restoring the data members `'buffer`, `#buffer` and `>in` by copying their values from the saved block input source to the current one.

With an object of class `block-input-source`, it's pretty easy to implement the Forth 2012 word `LOAD` in the similar way as `evaluate`:

```
: load ( unsigned -- )
  ?block
  default-input-stream @ local stream
  new block-input-stream default
  begin refill
  while interpret
  repeat
  default-input-stream @ delete stream default ;
```

`load` temporarily switches the default input source to a newly created object of class `block-input-stream` and then enters an interpreter loop. The loop exits after the last line of the block buffer has been parsed. The default input source has to be saved in a local, because the data stack might contain items that are used by the interpreted block.

`thru` allows interpreting a sequence of blocks. Its definition contains nothing more than a check for valid input parameters and a loop around `load`:

```
: thru ( unsigned 1st -- )
  over over >
  if drop drop -35 throw
  else 1+ swap do i load loop
  then ;
```

A Line Editor

StrongForth comes with a very simple block line editor, whose source code is contained in the source file `editor.sf`. It actually is a port of the old *FIG* line editor. To make the line editor commands available, its source code has to be compiled by entering

```
include editor.sf
```

at the StrongForth command prompt. The line editor interprets each block as a screen with 16 lines of 64 characters. Since a screen usually does not contain control characters like carriage return, line feed and tab, lines have to be padded with trailing spaces in order to contain exactly 64 characters.

The line editor frequently uses `pad` to store character strings or complete lines. `pad` can be used to copy one line to another line within the same screen or between two different screens. The length of the string in `pad` is stored in the variable `count`. Another variable, `r#`, contains the position of the editing cursor as an offset to the start of the current screen:

```
0 variable count
0 variable r#
```

The following is a short description of the line editor commands. `list` and `flush` are already included in the `forth` vocabulary. All other words belonging to the line editor are contained in the `new editor` vocabulary:

```
wordlist editor immediate
editor definitions
```


Line editing commands

Command	Description
<i>n d</i>	Delete line <i>n</i> by moving all succeeding lines one line up. Hold the original contents of line <i>n</i> in <i>pad</i> .
<i>n e</i>	Erase line <i>n</i> with spaces.
<i>n h</i>	Hold line <i>n</i> in <i>pad</i> .
<i>n i</i>	Insert the contents of <i>pad</i> in line <i>n</i> and move the original line <i>n</i> and all following lines one line down. Line 15 is lost.
<i>n p text</i>	Put <i>text</i> into line <i>n</i> , overwriting the original contents.
<i>n r</i>	Replace line <i>n</i> with the contents of <i>pad</i> .
<i>n s</i>	Spread line <i>n</i> by moving it and all following lines one line down. Line <i>n</i> is erased with spaces. Line 15 is lost.

Cursor movement commands

Command	Description
<i>b</i>	Used after <i>f</i> to move the cursor back to the start of the found text. Then display the current line, marking the cursor position with an underscore character.
<i>n m</i>	Move the editing cursor by the signed amount <i>n</i> . Then display the current line, marking the cursor position with an underscore character.
<i>n t</i>	Type line <i>n</i> and copy it's contents into <i>pad</i> . Move the editing cursor to the start of line <i>n</i> .
<i>top</i>	Move the cursor to the start of line 0 of the screen.

String editing commands

Command	Description
<i>c text</i>	Copy <i>text</i> to the current line at the cursor position and move the rest of the line to the right. Then display the current line, marking the cursor position with an underscore character.
<i>f text</i>	Find <i>text</i> starting at the cursor position. If <i>text</i> is found, move the editing cursor to the end of the found text. Then display the current line, marking the cursor position with an underscore character. Otherwise move the cursor to the start of line 0 of the screen.
<i>n</i>	Used after <i>f</i> to find the next occurrence of the found text.
<i>till text</i>	Find <i>text</i> starting at the cursor position until the end of the current line. If <i>text</i> is found, delete all characters from the cursor position up to and including <i>text</i> by moving the remaining characters to the left and filling the line up with spaces. Then display the current line, marking the cursor position with an underscore character.
<i>x text</i>	Find <i>text</i> starting at the cursor position. If <i>text</i> is found, delete it by moving the remaining characters to the left and filling the line up with spaces. Move the cursor to the position where <i>text</i> was found. Then display the current line, marking the cursor position with an underscore character. Otherwise move the cursor to the start of line 0 of the screen.

Screen commands

Command	Description
<i>n</i> list	List screen <i>n</i> and select it for editing.
<i>n</i> clear	Clear screen <i>n</i> with blanks and select it for editing.
<i>n1 n2</i> copy	Copy the contents of screen <i>n1</i> to screen <i>n2</i> .
flush	Write all modified screens to the block file.
<i>n1 n2</i> index	Display the first line of each screen from <i>n1</i> to <i>n2</i> .
l	List the current screen. Then display the current line, marking the cursor position with an underscore character.

26 String Extensions

With the *String Extension* word set, Forth 2012 specifies three words that provide means to substitute macros enclosed in delimiters with given substitution character strings. `replaces` defines the substitution strings, `substitute` performs multiple substitutions for multiple macros, and `unescape` duplicates delimiters within a string.

StrongForth extends the functionality of the specification by

- allowing arbitrary characters as delimiters,
- adding an overloaded version of `replaces` that uses string-generating words instead of constant substitution strings,
- adding overloaded versions of `substitute` that redirect the target string to output streams and to files, and
- extending the semantics of `unescape` to prevent buffer overflows.

To make these words available to StrongForth, the source file `strext.sf` has to be included:

```
include strext.sf
```

The delimiter is implemented as a value instead of a constant. It can be changed with `to:`

```
char % value delimiter
```

The substitution strings are kept in dynamic memory as counted strings. `allocate-counted-string` expects a character string as an address and a count, and returns the address of a counted string in dynamic memory. The semantics of `count` is equivalent to `COUNT` in Forth 2012:

```
: allocate-counted-string ( caddress -> character unsigned
  -- 1st )
  local len len 1+ callocate len over -> unsigned !
  -> character tuck 1+ len move ;
```

```
: count ( caddress -> character -- 1st unsigned )
  dup cast caddress -> unsigned @ swap 1+ swap ;
```

`(replaces)` is a defining word that allocates a counted string in dynamic memory. Created definitions assume the name of a macro and return the assigned substitution string:

```
: (replaces) ( caddress -> character unsigned
  caddress -> character unsigned -- )
  (create) allocate-counted-string ,
  does> ( address -> caddress -> character
    -- caddress -> character unsigned )
  @ count ;
```

The semantics of `replaces` is more complex, because it allows changing the substitution strings of already existing macros without redefining them. Existing macros are looked for in the current compilation vocabulary:

```
: replaces ( caddress -> character unsigned 1st 3rd -- )
  over over [ latest prev ] literal \ runtime code of (replaces)
  runtime-criterion current @ search
  if nip nip >body -> caddress -> character
    dup @ free rot rot allocate-counted-string swap !
  else drop (replaces)
  then ;
```

And here's the overloaded versions for string-generating words:

```
: (replaces) ( (--string) caddress -> character unsigned -- )
  (create) ,
  does> ( address -> (--string)
    -- caddress -> character unsigned )
  @ execute ;

: replaces ( (--string) caddress -> character unsigned -- )
  over over [ latest prev ] literal \ runtime code of (replaces)
  runtime-criterion current @ search
  if nip nip >body -> (--string) !
  else drop (replaces)
  then ;
```

These two definitions look quite similar to the previous two. Instead of allocating a counted string, (replaces) compiles the execution token of a word that returns a character string. The runtime code executes this token. replaces differs from the previous version only in one point: There's no need to free a previously allocated counted string if the token of the string generating word is being updated. Here's an example of how to use it:

```
: "context ( -- caddress -> character unsigned )
  context @ >definition name ; ok
"context type forth ok
wordlist macros ok
macros definitions ok
' "context dt (--string) >token cast (--string) ok
" context" replaces ok
latest . context ( -- caddress -> character unsigned ) ok
context type macros ok
" The context is %context%." pad c/l substitute . 1 ok
type The context is macros. ok
ignore ok
```

First, we define a string generating word "context that returns the name of the head of the context vocabulary list. replaces assigns this string generating word to the macro context. In order to avoid name conflicts, it is recommended to define all macros in a dedicated vocabulary. The application of substitute finally demonstrates how the macro substitutes its name within a given string.

substitute is also based on an internal word. (substitute) is executed whenever substitute finds a delimiter. Its parameters are the number of substitutions so far and the remaining character string. If (substitute) finds a matching delimiter, it tries to find the enclosed macro in the first context vocabulary. If a macro with that name exists, it types the substitution string on the default output stream and increments the substitution count. Otherwise, it types two delimiters with the original text inbetween. If (substitute) does not find a matching delimiter, it just types the delimiter and the remaining text:

```

: (substitute) ( unsigned caddress -> character unsigned
-- 1st 2nd 4 th )
locals( addr len ) addr len
chere -> character delimiter c, 1 search
[ -1 chars ] literal allot
if len over -
  addr over false no-criterion context @ search
  if nip [dt] (--string) >token cast (--string) execute type
    rot 1+ rot rot
  else drop delimiter emit addr swap type delimiter emit
  then /string
else delimiter emit addr swap type len + 0
then ;

```

But why does (substitute) *emit* and *type* instead of copy delimiters and substrings to a destination character string? Well, there's an overloaded version of substitute that directs delimiters and substrings to the default output stream:

```

: substitute ( caddress -> character unsigned -- unsigned )
0 rot rot
begin dup
while over @ delimiter <>
  if over @ emit /string
  else /string dup
    if over @ delimiter =
      if delimiter emit /string
      else (substitute)
      then
    else delimiter emit
    then
  then
repeat
drop drop ;

```

substitute traverses the source string character by character, typing each character up to the first delimiter. If the delimiter is the last character of the string, or if it is immediately followed by another delimiter, one delimiter is typed. Otherwise, an attempt is made to detect a macro by executing (substitute).

With this version of substitute, it is easy to implement SUBSTITUTE as specified by Forth 2012:

```

: substitute ( caddress -> character unsigned
caddress -> character unsigned -- 4 th 6 th unsigned )
new string-output-stream
default-output-stream @ locals( stream saved )
stream default substitute saved default
stream string stream delete rot ;

```

Using an object of class string-output-stream, the destination string becomes the temporary default output stream. After the substitution is done, the default output stream is restored and the object is deleted.

The same technique works for files as well. By temporarily making a file the default output stream, its contents can be processed by substitute line by line, counting the total number of substitutions:

```

: substitute ( file file -- unsigned )
  new file-output-stream default-output-stream @ over default
  258 callocate -> character 0
  locals( src stream saved buf count )
  begin buf dup 256 src read-line
  while substitute cr count swap + to count
  repeat
  drop drop saved default stream delete count ;

```

A version of `substitute` for files can turn out to be pretty useful, for example, if you need to instantiate source file templates. Templates will be explained in chapter 32.

For `unescape`, StrongForth provides again three overloaded versions for the default output stream, for character strings and for files. The implementation of the version for the default output stream is straightforward, while the definitions of the other two overloaded versions look similar to the respective ones of `substitute`:

```

: unescape ( caddress -> character unsigned -- )
  over swap + swap
  ?do i @ delimiter =
    if delimiter emit
    then
    i @ emit
  loop ;

: unescape ( caddress -> character unsigned
  caddress -> character unsigned -- 4 th 6 th )
  new string-output-stream
  default-output-stream @ locals( stream saved )
  stream default unescape saved default
  stream string stream delete ;

: unescape ( file file -- )
  new file-output-stream default-output-stream @ over default
  258 callocate -> character locals( src stream saved buf )
  begin buf dup 256 src read-line
  while unescape cr
  repeat
  drop drop saved default stream delete ;

```

However, there's a major difference to the Forth 2012 specification of `UNESCAPE`. StrongForth does not allow a buffer overflow. The version of `unescape` for character strings requires an additional size parameter for the destination string. If the buffer is not large enough, an exception will be thrown.

27 Structures

In the *Facility* word set, Forth 2012 specifies a collection of words that support data structures. A structure is an ordered set of data items. The main difference to an array is the fact that a structure may contain different data items, such as cells, double-cells, characters and floating-point numbers. Each item can be referred to with a specific name.

That sounds pretty similar to the description of an object. In fact, a structure can be viewed as a kind of class with no methods, like the predefined classes `data-attributes` and `class-attributes`, which have constructors as their only methods. Structures don't even have constructors.

Although StrongForth supports structures, it is recommended to use classes instead. Classes offer nearly the same features, but they are much more powerful, because they additionally offer methods, encapsulation and many other things. If you prefer to stick with structures anyway, you have to include the source code of the words that will be presented in this chapter from

```
struct.sf:
```

```
include struct.sf
```

Let's begin with an example of a simple structure:

```
begin-structure example ok
null signed-double member a ok
null logical member b ok
null character 4 cmembers c ok
end-structure ok
' example . example ( stack-diagram -- 1st ) ok
' a . a ( example -- address -> signed-double ) ok
' b . b ( example -- address -> logical ) ok
' c . c ( example -- address -> character ) ok
dt example structure-size allocate cast example constant x ok
+4400000000000000. x a ! ok
5 bit 20 bit or x b ! ok
char S x c ! ok
char F x c 1+ ! ok
x a @ . 4400000000000000 ok
x b @ hex . decimal 100020 ok
x c 2 type SF ok
```

The definition of a structure is enclosed between `begin-structure` and `end-structure`. But instead of `+FIELD`, `FIELD:`, `CFIELD:`, `FFIELD:`, `SFFIELD:` and `DFFIELD:`, the fields of a StrongForth structure are defined with `member` or `members` and their variants. The words that are used to define the data members of classes can be reused for structures. They create words with stack diagrams in the format

```
( <struct> -- <addr> -> x )
```

`<struct>` is the data type of the structure, `<addr>` can be one of these: `address`, `caddress`, `sfaddress` or `dfaddress`. `x` is the data type of the field or member. I. e., a member expects an instance of a structure on the stack and returns the address of the data item within the instance of the structure.

The definition of `begin-structure` resembles the one of `procreates`, because it also creates a new data type. Just like classes, each structure has its own data type. The data type of a structure has to be directly or indirectly derived from the predefined data type `structure`:

```
dt single procreates structure

: begin-structure ( -- structure-attributes object-size )
  create [dt] structure dup size new structure-attributes
  dup cast address latest ?created-definition body!
  dup cast class-attributes to this-attributes
  null object-size does-data-type ;
```

Instead of creating an object of class `data-type-attributes`, `begin-structure` creates an object of class `structure-attributes`. `structure-attributes` is a child of `data-type-attributes`. It has an additional data member `'length`, which stores the size in bits of the structure. It is initialized with zero:

```
dt data-type-attributes procreates structure-attributes

class structure-attributes
  forth definitions
  null object-size member 'length

  : structure-attributes ( data-type unsigned
    structure-attributes -- 3rd )
    data-type-attributes ;

endclass
```

A pointer to the new object of class `structure-attributes` is stored in the `data` field of the created definition. `begin-structure` returns the object of data type `structure-attributes` to be used by `end-structure`, plus the structure size as an item of data type `object-size`, initialized with zero.

`end-structure` stores the size of the structure in bits, as accumulated by member definitions, in the structure attributes:

```
: end-structure ( structure-attributes object-size -- )
  swap 'length ! ;
```

In analogy to the words `object?` and `>class-attributes`, StrongForth provides the words `structure?` and `>structure-attributes` for structures. The definitions are very similar to the respective definitions for classes and objects:

```
: structure? ( data-type -- flag )
  [dt] structure ancestor? ;

: >structure-attributes ( data-type -- structure-attributes )
  dup structure?
  if >attributes cast structure-attributes
  else drop -271 throw null structure-attributes
  then ;
```


Determining the size of a structure is different from determining the size of an object. The size of an object is stored in the first cell of its virtual method table. But a structure has no virtual method table. Its size is stored in the 'length member of its attributes, which can be obtained from the structure's data type using >structure-attributes. So, here's a word that can be used to determine the size in address units of a structure:

```
: structure-size ( data-type -- unsigned )
  >structure-attributes 'length @ cast unsigned
  address-unit-bits aligned address-unit-bits / ;
```

Note that it is not possible to overload size, because an overloaded version of size for items of data type data-type already exists. It returns the size of an item of a given data type on the data stack. For structures, it always returns 1 cells.

There is one feature of Forth 2012 structures that is still not directly supported by StrongForth structures. Because Forth 2012 specifies that the runtime semantics of the word created by begin-structure returns the size of the structure, it can embed a structure within another structure. The following example is copied from the Forth 2012 specification:

```
BEGIN-STRUCTURE point \ -- a-addr 0 ; -- lenp
FIELD: p.x            \ -- a-addr cell
FIELD: p.y            \ -- a-addr cell*2
END-STRUCTURE

BEGIN-STRUCTURE rect  \ -- a-addr 0 ; -- lenr
point +FIELD r.tlhc   \ -- a-addr cell*2
point +FIELD r.brhc   \ -- a-addr cell*4
END-STRUCTURE
```

Something similar to +FIELD is not available in StrongForth. It wouldn't work anyway, because a StrongForth structure is a data type. Executing it does not return the size of the structure. To embed structures within another structure, StrongForth provides smembers and smember. These two words expect the data type of a structure on the stack, and smembers additionally the number of structures. The above Forth 2012 example could be implemented in StrongForth like this:

```
begin-structure point
null signed member p.x
null signed member p.y
end-structure

begin-structure rect
dt point smember r.tlhc
dt point smember r.brhc
end-structure
```

r.tlhc expects a structure of data type rect and returns a structure of data type point, to which then p.x or p.y can be applied:

```
' r.brhc . r.brhc ( rect -- point ) ok
here cast rect dup . 8726100 ok
r.brhc .s dup . point 8726108 ok
p.y .s . address -> signed 8726112 ok
```

Here are the implementations of `smembers` and `smember`. A new overloaded version of `params!` creates the stack diagram with the presently defined structure as the input parameter and the embedded structure as the output parameter:

```
: params! ( data-type member-definition -- )
  swap state @ new stack-diagram
  this-attributes >data-type
  [ ' 'parent input-params drop @ ] literal or
  over param, -- tuck param, swap params! ;

: smembers ( object-size data-type unsigned -- 1st )
  >r over parse-name new member-definition
  over over params!
  enddef structure-size r> * address-unit-bits * + ;

: smember ( object-size data-type -- 1st )
  1 smembers ;
```

Note that `smembers`, other than `members` and its variants, doesn't do any kind of alignment. Neither does the Forth 2012 word `+FIELD`, in contrast to `FIELD:` and its variants.

28 The Assembler

First steps

StrongForth comes with an assembler and a disassembler, whose source code is contained in the file `asm.sf`. Before using the assembler or the disassembler, you have to load it with

```
include asm.sf.
```

A few words related to the assembler have already been presented in chapter 10:

```
code ( -- code-definition )
;code ( colon-definition -- code-definition ) compile-only
label ( -- )
assembler ( -- ) immediate
```

`code` initiates a code definition, `;code` begins the assembler runtime code of a creating word.

`label` defines an assembler code label as a constant assigned to the current code location.

`assembler` is the vocabulary that contains all assembler instructions and addressing modes.

Let's begin with a very simple code definition:

```
code 2+ ( eax: integer -- eax: 1st eax: changed )   ok
eax inc,      ok
eax inc,      ok
ret,          ok
endcode       ok
4 2+ . 6       ok
```

As usual, defining a new StrongForth word requires supplying a stack diagram. In addition to the stack diagram of a colon definition, we now have to provide information about the processor registers in which the input parameters are expected and in which the output parameters are returned. Generally, StrongForth passes all single-cell parameters in the general-purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi` and `edi`. Double-cell parameters are passed in the register pairs `eax/edx`, `ebx/ecx` and `esi/edi`. floating-point numbers reside on the hardware floating-point stack. In special cases, it is possible to pass items of data type `flag` in the processor's `eflags` register. Single- and double-cell items may also be passed on the stack, although this is not as efficient as passing parameters in registers. You can even pass input parameters on the stack, but these parameters require special treatments by code definitions implemented in assembly language, or by *MSVCRT* library functions. Colon definitions cannot handle parameters passed on the stack.

A number of words add the information about the location of a parameter within a stack diagram:

```
eax: ( stack-diagram -- 1st )
ecx: ( stack-diagram -- 1st )
edx: ( stack-diagram -- 1st )
ebx: ( stack-diagram -- 1st )
esi: ( stack-diagram -- 1st )
edi: ( stack-diagram -- 1st )
eax/edx: ( stack-diagram -- 1st )
ebx/ecx: ( stack-diagram -- 1st )
esi/edi: ( stack-diagram -- 1st )
eflags: ( stack-diagram -- 1st )
float: ( stack-diagram -- 1st )
stack: ( stack-diagram -- 1st )
```

```
lit: ( stack-diagram -- 1st )
any: ( stack-diagram -- 1st )
```

The phrase `eax: changed` is a hint that register `eax` is changed by executing this word. This hint is necessary to keep StrongForth's register allocation from becoming corrupted. Colon definitions automatically keep track of changed registers. In assembly language, everything is possible. You as the programmer should know which registers are changed by your code and which remain unchanged.

Other than normal assemblers, Forth assemblers prefer postfix notation, which means the operands precede the instruction words, as in `eax inc, .` In another aspect, Strongforth adheres to conventional x86 assemblers: in dual-operand assembly instructions, the destination operand always is the first one, the source operand is the second one. For example

```
eax ecx add,
```

means that the contents of register `ecx` (the source) is added to registers `eax` (the destination).

Generally, a comma is appended to the names of all assembler instruction words in order to mark the end of the assembly instruction, and to indicate that something is appended to the code space. Typically, the last assembler instruction in a code definition is `ret, ,` a return from subroutine. Finally, `endcode` links the new code definition to the current compilation word list. `end-code` also removes the assembler vocabulary from the context vocabulary list:

```
: endcode ( code-definition -- )
  enddef [compile] assembler [compile] ignore ;
```

Note that a code definition is compiled while staying in interpretation state. Since `code`, other than `:`, does not switch to compilation state, all assembly instructions are executed, although they are not immediate words.

Now, let's try the disassembler:

```
see 2+
code 2+ ( eax: integer -- eax: 1st eax: changed )
00428794: eax inc,
00428795: eax inc,
00428796: ret,
endcode ok
```

`see` is a virtual method of class definition. The versions of `see` for `code` and `colon` definitions call the disassembler to show the assembly code. Here's an example of a colon definition:

```
see char
code char ( -- eax: character eax: ecx: ebx: changed )
0041EA78: 00406060 call, parse-name
0041EA7D: ecx ecx test,
0041EA7F: 0041EA86 jz,
0041EA81: eax byte[ebx] movzx,
0041EA84: 0041EA8B jmp,
0041EA86: eax 00000020 mov,
0041EA8B: ret,
endcode ok
```

As a second example, we use `;code` to start a sequence of assembly instructions:

```
: :n+ ( integer -- )
  create , ;code ( eax: integer ebx: address -> integer ok
  -- eax: 1st eax: changed ) ok
```

```

        eax [ebx] add,    ok
        ret,    ok
    endcode    ok
3 :n+ 3+    ok
4 3+ . 7    ok

```

:n+ is a defining word that creates words that add a constant value to an integer. It stores the constant value in the data field of the new definition. ;code is succeeded by the stack diagram of the definitions created by :n+, with an additional input parameter for the address of the data field. The address of the data field is always passed in register ebx.

The code consists of a single add, assembler instruction, and a ret, instruction. According to the usual 8086 assembler syntax, the destination operand eax comes first. The source operand is the value ebx points to.

There's one more thing that is worth to be noted about ;code. It's the way how the stack diagrams of the defined words are being constructed. There are actually three possible ways:

- You may specify the stack diagram at the beginning of the runtime code, including the address of the data field in register ebx as the last input parameter. This technique is used in the example of :n+. Its disadvantage is that the stack diagrams of all defined words are identical. Nevertheless, it is the preferred technique.
- You may compile the stack diagram explicitly between create and ;code, using a suitable version of params! to construct the stack diagram. This technique is useful if compiling defining words like constant or variable that create words with varying data types in their stack diagrams.
- You may specify the stack diagram individually for each defined word, i. e., after executing the defining word. This technique allows the greatest flexibility. Anyway, it is more dangerous, because it allows specifying stack diagrams that do not fit to the runtime code of the defined words, because there is no type checking.

Whatever you chose is best for a specific defining word, make sure to apply only one of these techniques. Any mixture will most likely lead to unpredictable results.

Addressing modes

So far, you've already seen two addressing modes: *register direct* and *register indirect*. For example, eax is the name of an x86 32-bit general-purpose register, but in the context of the StrongForth assembler it denotes an addressing mode. eax is a so-called *register direct* addressing mode, which simply determines that the operand is in the eax register. Another example is [ebx]. This addressing mode determines that the operand is in the memory cell pointed to by the ebx register. All addressing modes return an item of data type mode, which is directly derived from data type double:

```
dt double procreates mode
```

Items of this data type consist of two cells, one of which specifies the addressing mode, while the other one contains an optional displacement parameter. Here is a list of all x86 addressing modes explicitly defined by the StrongForth assembler:

Cell-size Operands

eax (-- mode)	Register direct
ecx (-- mode)	
edx (-- mode)	

<pre> ebx (-- mode) esp (-- mode) ebp (-- mode) esi (-- mode) edi (-- mode) </pre>	
<pre> [eax] (-- mode) [ecx] (-- mode) [edx] (-- mode) [ebx] (-- mode) [esp] (-- mode) [ebp] (-- mode) [esi] (-- mode) [edi] (-- mode) </pre>	Register indirect
<pre> [eax]+ (single -- mode) [ecx]+ (single -- mode) [edx]+ (single -- mode) [ebx]+ (single -- mode) [esp]+ (single -- mode) [ebp]+ (single -- mode) [esi]+ (single -- mode) [edi]+ (single -- mode) </pre>	Register indirect with displacement
<pre> [] (single -- mode) </pre>	Memory indirect

Byte-size Operands

<pre> al (-- mode) cl (-- mode) dl (-- mode) bl (-- mode) ah (-- mode) ch (-- mode) dh (-- mode) bh (-- mode) </pre>	Register direct
<pre> byte[eax] (-- mode) byte[ecx] (-- mode) byte[edx] (-- mode) byte[ebx] (-- mode) byte[esp] (-- mode) byte[ebp] (-- mode) byte[esi] (-- mode) byte[edi] (-- mode) </pre>	Register indirect
<pre> byte[eax]+ (single -- mode) byte[ecx]+ (single -- mode) byte[edx]+ (single -- mode) byte[ebx]+ (single -- mode) byte[esp]+ (single -- mode) byte[ebp]+ (single -- mode) byte[esi]+ (single -- mode) </pre>	Register indirect with displacement

<code>byte[edi]+ (single -- mode)</code>	
<code>byte[] (single -- mode)</code>	Memory indirect

Word-size Operands

<code>ax (-- mode)</code> <code>cx (-- mode)</code> <code>dx (-- mode)</code> <code>bx (-- mode)</code> <code>sp (-- mode)</code> <code>bp (-- mode)</code> <code>si (-- mode)</code> <code>di (-- mode)</code>	Register direct
<code>16-bit-operand: [eax] (single -- mode)</code> <code>16-bit-operand: [ecx] (single -- mode)</code> <code>16-bit-operand: [edx] (single -- mode)</code> <code>16-bit-operand: [ebx] (single -- mode)</code> <code>16-bit-operand: [esp] (single -- mode)</code> <code>16-bit-operand: [ebp] (single -- mode)</code> <code>16-bit-operand: [esi] (single -- mode)</code> <code>16-bit-operand: [edi] (single -- mode)</code>	Register indirect
<code>16-bit-operand: [eax]+ (single -- mode)</code> <code>16-bit-operand: [ecx]+ (single -- mode)</code> <code>16-bit-operand: [edx]+ (single -- mode)</code> <code>16-bit-operand: [ebx]+ (single -- mode)</code> <code>16-bit-operand: [esp]+ (single -- mode)</code> <code>16-bit-operand: [ebp]+ (single -- mode)</code> <code>16-bit-operand: [esi]+ (single -- mode)</code> <code>16-bit-operand: [edi]+ (single -- mode)</code>	Register indirect with displacement
<code>16-bit-operand: [] (single -- mode)</code>	Memory indirect

Floating-Point Operands

<code>st0 (-- mode)</code> <code>st1 (-- mode)</code> <code>st2 (-- mode)</code> <code>st3 (-- mode)</code> <code>st4 (-- mode)</code> <code>st5 (-- mode)</code> <code>st6 (-- mode)</code> <code>st7 (-- mode)</code> <code>st (-- mode)</code>	Floating-point register direct
---	--------------------------------

As can be seen, addressing modes for byte, word and cell operands are clearly distinguished. All addressing modes not listed here are implicit to specific assembly instructions, like `ret`, `.`. This applies also to immediate addressing, because immediate operands do not require a special word to indicate the addressing mode. For example,

```
ah 23 add,
```

adds the immediate value 23 to the content of the `al` register. `add`, and several other assembler instruction words are actually overloaded. One version expects an addressing mode and an

immediate operand as input parameters, whereas the second version requires two addressing modes:

```
assembler words add,  
add, ( mode mode -- )  
add, ( mode single -- ) ok
```

The processor provides two variants of the addressing mode *register indirect with displacement*, one with an 8-bit signed displacement, which suffices in many cases, and one with a 32-bit displacement. StrongForth's corresponding addressing modes automatically select the required processor addressing mode depending on the value of the displacement n . If n is between -128 and +127, the addressing mode with an 8-bit displacement is compiled, otherwise the addressing mode with a 32-bit displacement.

Please note the name conflict between `bl (-- character)` in the `forth` vocabulary and `bl (-- mode)` in the `assembler` vocabulary. If the `assembler` vocabulary is the head of the context vocabulary list, `bl (-- character)` is invisible to the interpreter. However, choosing a different name for either of these two overloaded words would obviously not have been a good idea.

By default, the x86 processor assumes 32-bit operands. It temporarily switches to 16-bit operands whenever an instruction is preceded by a special prefix byte. This prefix is compiled by the word

```
16-bit-operand: ( -- )
```

`16-bit-operand:` is implicitly compiled if a 16-bit register like `ax` is specified as register direct addressing mode. Another prefix can be specified to force 16-bit addresses instead of 32-bit addresses:

```
16-bit-address: ( -- )
```

Anyway, 16-bit addresses are not used by StrongForth's compiler.

We're not yet done with addressing modes. First, there are the scaled index addressing modes, which can extend all *register indirect* and *register indirect with displacement* addressing modes with an additional scaled index register:

Scaled Index

```
index[eax] ( mode -- mode )  
index[ecx] ( mode -- mode )  
index[edx] ( mode -- mode )  
index[ebx] ( mode -- mode )  
index[ebp] ( mode -- mode )  
index[esi] ( mode -- mode )  
index[edi] ( mode -- mode )  
index[eax*2] ( mode -- mode )  
index[ecx*2] ( mode -- mode )  
index[edx*2] ( mode -- mode )  
index[ebx*2] ( mode -- mode )  
index[ebp*2] ( mode -- mode )  
index[esi*2] ( mode -- mode )  
index[edi*2] ( mode -- mode )  
index[eax*4] ( mode -- mode )  
index[ecx*4] ( mode -- mode )  
index[edx*4] ( mode -- mode )  
index[ebx*4] ( mode -- mode )  
index[ebp*4] ( mode -- mode )  
index[esi*4] ( mode -- mode )
```



```

index[edi*4] ( mode -- mode )
index[eax*8] ( mode -- mode )
index[ecx*8] ( mode -- mode )
index[edx*8] ( mode -- mode )
index[ebx*8] ( mode -- mode )
index[ebp*8] ( mode -- mode )
index[esi*8] ( mode -- mode )
index[edi*8] ( mode -- mode )

```

For example, the assembler instruction

```
1000 [ebx] index[edi*4] inc,
```

increments a cell operand whose address is calculated by adding 1000 to the content of register ebx, and then adding 4 times the content of register edi.

Some overloaded versions of the assembler instructions `mov`, `push`, and `pop`, access the processors segment registers, control registers and debug registers. For these registers, constants of data types `sreg`, `creg` and `dreg` are defined, respectively:

```

dt unsigned procreates sreg \ segment register
dt unsigned procreates creg \ control register
dt unsigned procreates dreg \ debug register

```

Inherent Register Operands

<pre> es (-- sreg) cs (-- sreg) ss (-- sreg) ds (-- sreg) fs (-- sreg) gs (-- sreg) </pre>	Segment register addressing mode
<pre> cr0 (-- creg) cr1 (-- creg) cr2 (-- creg) cr3 (-- creg) cr4 (-- creg) </pre>	Control register addressing mode
<pre> dr0 (-- dreg) dr1 (-- dreg) dr2 (-- dreg) dr3 (-- dreg) dr4 (-- dreg) dr5 (-- dreg) dr6 (-- dreg) dr7 (-- dreg) </pre>	Debug register addressing mode

Here are a few examples of how to use these registers or addressing modes:

```

ds push,
eax cr3 mov,
dr0 edx mov,

```

Floating-point assembler instructions that access memory locations require an additional addressing mode modifier to determine the size of the operand:

Modifier	Size in memory
m16 (mode -- mode)	16 bits
m32 (mode -- mode)	32 bits (single-precision)
m64 (mode -- mode)	64 bits (double-precision)
m80 (mode -- mode)	80 bits

Here are two examples of how to use these modifiers:

```
[edi] m16 fiadd,
4 floats [ebx]+ m80 fld,
```

Instruction words

The StrongForth assembler does not provide the complete set of all x86 assembler instructions. Especially some of those related to various coprocessor units are missing. By studying the sources of the assembler, you can easily find out how to define missing assembler instructions.

Most assembly instructions expect one or two operands on the stack, which can be immediate values, jump destination addresses, registers or memory locations, or anything else. Registers and memory addresses are always specified by addressing modes as explained in the previous section.

A new data type `segment` has been created for those words that expect a segment on the stack:

```
dt address procreates segment
```

Note that in all instructions requiring two operands, like `add`, and `mov`, , the destination operand comes before the source operand. A list of all StrongForth assembly instructions is given below.

Assembler instructions

<code>aaa, (--)</code>	ASCII adjust after addition
<code>aad, (--)</code>	ASCII adjust <code>ax</code> before division
<code>aad, (unsigned --)</code>	ASCII adjust <code>ax</code> before division w/base
<code>aam, (--)</code>	ASCII adjust <code>ax</code> after multiply
<code>aam, (unsigned --)</code>	ASCII adjust <code>ax</code> after multiply w/base
<code>aas, (--)</code>	ASCII adjust <code>al</code> after subtraction
<code>adc, (mode mode --)</code>	Add with carry
<code>adc, (mode single --)</code>	Add with carry
<code>add, (mode mode --)</code>	Add
<code>add, (mode single --)</code>	Add
<code>and, (mode mode --)</code>	Logical <i>and</i>
<code>and, (mode single --)</code>	Logical <i>and</i>
<code>arpl, (mode mode --)</code>	Adjust RPL field of segment selector
<code>bound, (mode mode --)</code>	Check array index <i>idx</i> against bounds <i>bnd</i>
<code>bsf, (mode mode --)</code>	Bit scan forward
<code>bsr, (mode mode --)</code>	Bit scan reverse
<code>bswap, (mode --)</code>	Byte swap
<code>bt, (mode mode --)</code>	Bit test
<code>bt, (mode unsigned --)</code>	Bit test
<code>btc, (mode mode --)</code>	Bit test and complement
<code>btc, (mode unsigned --)</code>	Bit test and complement
<code>btr, (mode mode --)</code>	Bit test and reset
<code>btr, (mode unsigned --)</code>	Bit test and reset
<code>bts, (mode mode --)</code>	Bit test and set
<code>bts, (mode unsigned --)</code>	Bit test and set
<code>call, (address --)</code>	Call procedure near

<code>call, (mode --)</code>	Call procedure near
<code>callf, (segment address --)</code>	Call procedure far
<code>callf, (mode --)</code>	Call procedure far
<code>cbw, (--)</code>	Convert byte to word
<code>cdq, (--)</code>	Convert doubleword to quadword
<code>clc, (--)</code>	Clear carry flag
<code>cld, (--)</code>	Clear direction flag
<code>cli, (--)</code>	Clear interrupt flag
<code>clts, (--)</code>	Clear task-switched flag in CR0
<code>cmc, (--)</code>	Complement carry flag
<code>cmova, (mode mode --)</code>	Conditional move if above
<code>cmovae, (mode mode --)</code>	Conditional move if above or equal
<code>cmovb, (mode mode --)</code>	Conditional move if below
<code>cmovbe, (mode mode --)</code>	Conditional move if below or equal
<code>cmovc, (mode mode --)</code>	Conditional move if carry
<code>cmove, (mode mode --)</code>	Conditional move if equal
<code>cmovg, (mode mode --)</code>	Conditional move if greater
<code>cmovge, (mode mode --)</code>	Conditional move if greater or equal
<code>cmovl, (mode mode --)</code>	Conditional move if less
<code>cmovle, (mode mode --)</code>	Conditional move if less or equal
<code>cmovna, (mode mode --)</code>	Conditional move if not above
<code>cmovnae, (mode mode --)</code>	Conditional move if not above or equal
<code>cmovnb, (mode mode --)</code>	Conditional move if not below
<code>cmovnbe, (mode mode --)</code>	Conditional move if not below or equal
<code>cmovnc, (mode mode --)</code>	Conditional move if not carry
<code>cmovne, (mode mode --)</code>	Conditional move if not equal
<code>cmovng, (mode mode --)</code>	Conditional move if not greater
<code>cmovnge, (mode mode --)</code>	Conditional move if not greater or equal
<code>cmovnl, (mode mode --)</code>	Conditional move if not less
<code>cmovnle, (mode mode --)</code>	Conditional move if not less or equal
<code>cmovno, (mode mode --)</code>	Conditional move if not overflow
<code>cmovnp, (mode mode --)</code>	Conditional move if not parity
<code>cmovns, (mode mode --)</code>	Conditional move if not sign
<code>cmovnz, (mode mode --)</code>	Conditional move if not zero
<code>cmovo, (mode mode --)</code>	Conditional move if overflow
<code>cmovp, (mode mode --)</code>	Conditional move if parity
<code>cmovpe, (mode mode --)</code>	Conditional move if parity even
<code>cmovpo, (mode mode --)</code>	Conditional move if parity odd
<code>cmovs, (mode mode --)</code>	Conditional move if sign
<code>cmovz, (mode mode --)</code>	Conditional move if zero
<code>cmp, (mode mode --)</code>	Compare two operands
<code>cmp, (mode single --)</code>	Compare two operands
<code>cmpsb, (--)</code>	Compare byte string operands
<code>cmpsd, (--)</code>	Compare doubleword string operands
<code>cmpsw, (--)</code>	Compare word string operands
<code>cmpxchg, (mode mode --)</code>	Compare and exchange
<code>cmpxchg8b, (mode --)</code>	Compare and exchange bytes
<code>cpuid, (--)</code>	CPU identification
<code>cs: (--)</code>	Code segment prefix
<code>cwd, (--)</code>	Convert word to doubleword <code>dx:ax</code>
<code>cwde, (--)</code>	Convert word to doubleword <code>eax</code>
<code>daa, (--)</code>	Decimal adjust <code>al</code> after addition
<code>das, (--)</code>	Decimal adjust <code>al</code> after subtraction
<code>dec, (mode --)</code>	Decrement by 1
<code>div, (mode --)</code>	Unsigned divide
<code>ds: (--)</code>	Data segment prefix
<code>enter, (unsigned unsigned --)</code>	Make stack frame for procedure parameters

es: (--)	Extra segment prefix
f2xml, (--)	Compute 2^x-1
fabs, (--)	Floating-point absolute value
fadd, (mode --)	Floating-point add
fadd, (mode mode --)	Floating-point add
faddp, (--)	Floating-point add and pop
faddp, (mode mode --)	Floating-point add and pop
fbld, (mode --)	Load floating-point from BCD
fbstp, (mode --)	Store floating-point to BCD integer and pop
fchs, (--)	Change floating-point sign
fclex, (--)	Clear floating-point exceptions
fcmovb, (mode mode --)	FP conditional move if below
fcmovbe, (mode mode --)	FP conditional move if below or equal
fcmove, (mode mode --)	FP conditional move if equal
fcmovnb, (mode mode --)	FP conditional move if not below
fcmovnbe, (mode mode --)	FP conditional move if not below or equal
fcmovne, (mode mode --)	FP conditional move if not equal
fcmovnu, (mode mode --)	FP conditional move if not unordered
fcmovu, (mode mode --)	FP conditional move if unordered
fcom, (--)	Compare floating-point values
fcom, (mode --)	Compare floating-point values
fcom, (mode mode --)	Compare floating-point values
fcomi, (mode mode --)	Compare FP values and set eflags
fcomip, (mode mode --)	Compare FP values, set eflags and pop
fcomp, (--)	Compare floating-point values and pop
fcomp, (mode --)	Compare floating-point values and pop
fcomp, (mode mode --)	Compare floating-point values and pop
fcompp, (--)	Compare floating-point values and pop both
fcos, (--)	Floating-point cosine
fdecstp, (--)	Decrement Floating-point stack-top pointer
fdiv, (mode --)	Floating-point divide
fdiv, (mode mode --)	Floating-point divide
fdivp, (--)	Floating-point divide and pop
fdivp, (mode mode --)	Floating-point divide and pop
fdivr, (mode --)	Floating-point reverse divide
fdivr, (mode mode --)	Floating-point reverse divide
fdivrp, (--)	Floating-point reverse divide and pop
fdivrp, (mode mode --)	Floating-point reverse divide and pop
ffree, (mode --)	Free floating-point register
fiadd, (mode --)	Floating-point add integer
ficom, (mode --)	Compare floating-point with integer
ficomp, (mode --)	Compare floating-point with integer and pop
fidiv, (mode --)	Floating-point divide integer
fidivr, (mode --)	Floating-point reverse divide integer
fild, (mode --)	Floating-point load integer
fimul, (mode --)	Floating-point multiply integer
fincstp, (--)	Increment Floating-point stack-top pointer
finit, (--)	Initialize floating-point unit
fist, (mode --)	Store floating-point as integer
fistp, (mode --)	Store floating-point as integer and pop
fisttp, (mode --)	Store FP as integer with truncation and pop
fisub, (mode --)	Floating-point subtract integer
fisubr, (mode --)	Floating-point reverse subtract integer
fld, (mode --)	Load floating-point value
fldl, (--)	Load floating-point constant +1.0
fldcw, (mode --)	Load x87 FPU control word
fldenv, (mode --)	Load x87 FPU environment 28 bytes

<code>fldenvw, (--)</code>	Load x87 FPU environment 14 bytes
<code>fldl2e, (--)</code>	Load floating-point constant $\log_2 e$
<code>fldl2t, (--)</code>	Load floating-point constant $\log_2 10$
<code>fldlg2, (--)</code>	Load floating-point constant $\log_{10} 2$
<code>fldln2, (--)</code>	Load floating-point constant $\log_e 2$
<code>fldpi, (--)</code>	Load floating-point constant π
<code>fldz, (--)</code>	Load floating-point constant +0.0
<code>fmul, (mode --)</code>	Floating-point multiply
<code>fmul, (mode mode --)</code>	Floating-point multiply
<code>fmulp, (--)</code>	Floating-point multiply and pop
<code>fmulp, (mode mode --)</code>	Floating-point multiply and pop
<code>fnclex, (--)</code>	Clear exceptions
<code>fninit, (--)</code>	Initialize floating-point unit
<code>fnop, (--)</code>	Floating-point no Operation
<code>fnsave, (mode --)</code>	Save x87 FPU state 108 bytes
<code>fnsavew, (mode --)</code>	Save x87 FPU state 94 bytes
<code>fnstcw, (mode --)</code>	Store x87 FPU control word
<code>fnstenv, (mode --)</code>	Store x87 FPU environment 28 bytes
<code>fnstenvw, (mode --)</code>	Store x87 FPU environment 14 bytes
<code>fnstsw, (mode --)</code>	Store x87 FPU status word
<code>fpatan, (--)</code>	Floating-point partial arctangent
<code>fprem, (--)</code>	Floating-point partial remainder
<code>fpreml, (--)</code>	Floating-point IEEE partial remainder
<code>fptan, (--)</code>	Floating-point partial tangent
<code>frndint, (--)</code>	Floating-point round to integer
<code>frstor, (mode --)</code>	Restore x87 FPU state 108 bytes
<code>frstow, (mode --)</code>	Restore x87 FPU state 94 bytes
<code>fs: (--)</code>	F segment prefix
<code>fsave, (mode --)</code>	Save x87 FPU state 108 bytes
<code>fscale, (--)</code>	Floating-point scale
<code>fsin, (--)</code>	Floating-point sine
<code>fsincos, (--)</code>	Floating-point sine and cosine
<code>fsqrt, (--)</code>	Floating-point square root
<code>fst, (mode --)</code>	Store floating-point value
<code>fstcw, (mode --)</code>	Store x87 FPU control word
<code>fstenv, (mode --)</code>	Store x87 FPU environment 28 bytes
<code>fstenvw, (mode --)</code>	Store x87 FPU environment 14 bytes
<code>fstp, (mode --)</code>	Store floating-point value and pop
<code>fstsw, (mode --)</code>	Store x87 FPU status word
<code>fsub, (mode --)</code>	Floating-point subtract
<code>fsub, (mode mode --)</code>	Floating-point subtract
<code>fsubp, (--)</code>	Floating-point subtract and pop
<code>fsubp, (mode mode --)</code>	Floating-point subtract and pop
<code>fsubr, (mode --)</code>	Floating-point reverse subtract
<code>fsubr, (mode mode --)</code>	Floating-point reverse subtract
<code>fsubrp, (--)</code>	Floating-point reverse subtract and pop
<code>fsubrp, (mode mode --)</code>	Floating-point reverse subtract and pop
<code>ftst, (--)</code>	Floating-point test
<code>fucom, (--)</code>	Unordered compare floating point values
<code>fucom, (mode --)</code>	Unordered compare floating point values
<code>fucomi, (mode mode --)</code>	Compare FP values and set eflags
<code>fucomip, (mode mode --)</code>	Compare FP values, set eflags and pop
<code>fucomp, (--)</code>	Unordered compare FP values and pop
<code>fucomp, (mode --)</code>	Unordered compare FP values and pop
<code>fucompp, (--)</code>	Unordered compare FP values and pop both
<code>fwait, (--)</code>	Wait
<code>fxam, (--)</code>	Examine floating point

<code>fxch, (--)</code>	Floating-point exchange register contents
<code>fxch, (mode --)</code>	Floating-point exchange register contents
<code>fxch, (mode mode --)</code>	Floating-point exchange register contents
<code>fxtract, (--)</code>	FP extract exponent and significand
<code>fyl2x, (--)</code>	Floating-point compute $y * \log_2 x$
<code>fyl2xpl, (--)</code>	Floating-point compute $y * \log_2(x + 1)$
<code>getsec, (--)</code>	GETSEC leaf functions
<code>gs: (--)</code>	G segment prefix
<code>hlt, (--)</code>	Halt
<code>idiv, (mode --)</code>	Signed divide
<code>imul, (mode --)</code>	Signed multiply
<code>imul, (mode mode --)</code>	Signed multiply
<code>imul, (mode mode single --)</code>	Signed multiply
<code>in, (mode mode --)</code>	Input from port
<code>in, (mode unsigned --)</code>	Input from port
<code>inc, (mode --)</code>	Increment by 1
<code>insb, (--)</code>	Input from port to byte string
<code>insd, (--)</code>	Input from port to doubleword string
<code>insw, (--)</code>	Input from port to word string
<code>int, (unsigned --)</code>	Call to interrupt procedure
<code>into, (--)</code>	Call to interrupt procedure if overflow
<code>invd, (--)</code>	Invalidate internal caches
<code>invlpg, (mode --)</code>	Invalidate TLB entries
<code>iret, (--)</code>	Interrupt return 16 bit
<code>iretd, (--)</code>	Interrupt return 32 bit
<code>ja, (address --)</code>	Jump if above
<code>jae, (address --)</code>	Jump if above or equal
<code>jb, (address --)</code>	Jump if below
<code>jbe, (address --)</code>	Jump if below or equal
<code>jc, (address --)</code>	Jump if carry
<code>jcxz, (address --)</code>	Jump if <code>cx</code> register zero
<code>je, (address --)</code>	Jump if equal
<code>jecxz, (address --)</code>	Jump if <code>ecx</code> register zero
<code>jg, (address --)</code>	Jump if greater
<code>jge, (address --)</code>	Jump if greater or equal
<code>jl, (address --)</code>	Jump if less
<code>jle, (address --)</code>	Jump if less or equal
<code>jmp, (address --)</code>	Unconditional relative jump near
<code>jmp, (mode --)</code>	Unconditional absolute jump near
<code>jmpf, (segment address --)</code>	Unconditional absolute jump far
<code>jmpf, (mode --)</code>	Unconditional absolute jump far
<code>jna, (address --)</code>	Jump if not above
<code>jnae, (address --)</code>	Jump if not above or equal
<code>jnb, (address --)</code>	Jump if not below
<code>jnb, (address --)</code>	Jump if not below or equal
<code>jnc, (address --)</code>	Jump if not carry
<code>jne, (address --)</code>	Jump if not equal
<code>jng, (address --)</code>	Jump if not greater
<code>jnge, (address --)</code>	Jump if not greater or equal
<code>jnl, (address --)</code>	Jump if not less
<code>jnl, (address --)</code>	Jump if not less or equal
<code>jno, (address --)</code>	Jump if not overflow
<code>jnp, (address --)</code>	Jump if not parity
<code>jns, (address --)</code>	Jump if not sign
<code>jnz, (address --)</code>	Jump if not zero
<code>jo, (address --)</code>	Jump if overflow
<code>jp, (address --)</code>	Jump if parity

jpe, (address --)	Jump if parity even
jpo, (address --)	Jump if parity odd
js, (address --)	Jump if sign
jz, (address --)	Jump if zero
lahf, (--)	Load status flags into ah register
lar, (mode mode --)	Load access rights byte
lds, (mode mode --)	Load far pointer
lea, (mode mode --)	Load effective address
leave, (--)	High level procedure exit
les, (mode mode --)	Load far pointer
lfs, (mode mode --)	Load far pointer
lgdt, (mode --)	Load global descriptor table register
lgs, (mode mode --)	Load far pointer
lidt, (mode --)	Load interrupt descriptor table register
lldt, (mode --)	Load local descriptor table register
lmsw, (mode --)	Load machine status word
lock (--)	Assert LOCK# signal prefix
lodsb, (--)	Load from byte string
lodsd, (--)	Load from doubleword string
lodsw, (--)	Load from word string
loop, (address --)	Loop according to ecx counter
loope, (address --)	Loop according to ecx counter if equal
loopne, (address --)	Loop according to ecx counter if not equal
loopnz, (address --)	Loop according to ecx counter if not zero
loopz, (address --)	Loop according to ecx counter if zero
lsl, (mode mode --)	Load segment limit
lss, (mode mode --)	Load far pointer
ltr, (mode --)	Load task register
lzcnt, (mode mode --)	Count number of leading zero bits
mov, (mode mode --)	Move
mov, (mode single --)	Move
mov, (mode sreg --)	Move from segment register
mov, (sreg mode --)	Move to segment register
mov, (mode creg --)	Move from control register
mov, (creg mode --)	Move to control register
mov, (mode dreg --)	Move from debug register
mov, (dreg mode --)	Move to debug register
movsb, (--)	Move byte string
movsd, (--)	Move doubleword string
movsw, (--)	Move word string
movsx, (mode mode --)	Move with sign extension
movzx, (mode mode --)	Move with zero extension
mul, (mode --)	Unsigned multiply
neg, (mode --)	Two's complement negation
nop, (--)	No operation
nop, (mode --)	Multi-byte no operation
not, (mode --)	One's complement negation
or, (mode mode --)	Logical <i>or</i>
or, (mode single --)	Logical <i>or</i>
out, (mode mode --)	Output to port
out, (unsigned mode --)	Output to port
outsb, (--)	Output from byte string to port
outsd, (--)	Output from doubleword string to port
outsw, (--)	Output from word string to port
pop, (mode --)	Pop value from stack
pop, (sreg --)	Pop segment register from stack
popa, (--)	Pop all 16-bit general-purpose registers

popad, (--)	Pop all 32-bit general-purpose registers
popf, (--)	Pop stack into lower 16 bits eflags register
popfd, (--)	Pop stack into eflags register
push, (mode --)	Push value onto stack
push, (sreg --)	Push segment register onto stack
push, (single --)	Push literal onto stack
pusha, (--)	Push all 16-bit general-purpose registers
pushad, (--)	Push all 32-bit general-purpose registers
pushf, (--)	Push lower 16 bits of eflags register
pushfd, (--)	Push eflags register
rcl, (mode mode --)	Rotate left through carry
rcl, (mode unsigned --)	Rotate left through carry
rcr, (mode mode --)	Rotate right through carry
rcr, (mode unsigned --)	Rotate right through carry
rdmsr, (--)	Read from model specific register
rdpmc, (--)	Read performance-monitoring counters
rdtsc, (--)	Read time-stamp counter
rep (--)	Repeat string operation prefix
repe (--)	Repeat while equal string operation prefix
repne (--)	Repeat while not equal string operation prefix
repnz (--)	Repeat while zero string operation prefix
repz (--)	Repeat while not zero string operation prefix
ret, (--)	Near return from procedure
ret, (unsigned --)	Near return from procedure and pop
retf, (--)	Far return from procedure
retf, (unsigned --)	Far return from procedure and pop
rol, (mode mode --)	Rotate left
rol, (mode unsigned --)	Rotate left
ror, (mode mode --)	Rotate right
ror, (mode unsigned --)	Rotate right
rsm, (--)	Resume from system management mode
sahf, (--)	Store ah to flags
sal, (mode mode --)	Shift arithmetic left
sal, (mode unsigned --)	Shift arithmetic left
sar, (mode mode --)	Shift arithmetic right
sar, (mode unsigned --)	Shift arithmetic right
sbb, (mode mode --)	Subtract with borrow
sbb, (mode single --)	Subtract with borrow
scasb, (--)	Scan byte string
scasd, (--)	Scan doubleword string
scasw, (--)	Scan word string
seta, (mode --)	Set byte if above
setae, (mode --)	Set byte if above or equal
setb, (mode --)	Set byte if below
setbe, (mode --)	Set byte if below or equal
setc, (mode --)	Set byte if carry
sete, (mode --)	Set byte if equal
setg, (mode --)	Set byte if greater
setge, (mode --)	Set byte if greater or equal
setl, (mode --)	Set byte if less
setle, (mode --)	Set byte if less or equal
setna, (mode --)	Set byte if not above
setnae, (mode --)	Set byte if not above or equal
setnb, (mode --)	Set byte if not below
setnbe, (mode --)	Set byte if not below or equal
setnc, (mode --)	Set byte if not carry
setne, (mode --)	Set byte if not equal

setng, (mode --)	Set byte if not greater
setnge, (mode --)	Set byte if not greater or equal
setnl, (mode --)	Set byte if not less
setnle, (mode --)	Set byte if not less or equal
setno, (mode --)	Set byte if not overflow
setnp, (mode --)	Set byte if not parity
setns, (mode --)	Set byte if not sign
setnz, (mode --)	Set byte if not zero
seto, (mode --)	Set byte if overflow
setp, (mode --)	Set byte if parity
setpe, (mode --)	Set byte if parity even
setpo, (mode --)	Set byte if parity odd
sets, (mode --)	Set byte if sign
setz, (mode --)	Set byte if zero
sgdt, (mode --)	Store global descriptor table register
shl, (mode mode --)	Shift left
shl, (mode unsigned --)	Shift left
shld, (mode mode mode --)	Double precision shift left
shld, (mode mode unsigned --)	Double precision shift left
shr, (mode mode --)	Shift right
shr, (mode unsigned --)	Shift right
shrd, (mode mode mode --)	Double precision shift right
shrd, (mode mode unsigned --)	Double precision shift right
sidt, (mode --)	Store interrupt descriptor table register
sldt, (mode --)	Store local descriptor table register
smsw, (mode --)	Store machine status word
ss: (--)	Stack segment prefix
stc, (--)	Set carry flag
std, (--)	Set direction flag
sti, (--)	Set interrupt flag
stosb, (--)	Store to byte string
stosd, (--)	Store to doubleword string
stosw, (--)	Store to word string
str, (mode --)	Store Task Register
sub, (mode mode --)	Subtract
sub, (mode single --)	Subtract
syscall, (--)	Fast System Call
sysenter, (--)	Fast System Call
sysexit, (--)	Fast return from fast system call
sysret, (--)	Return from fast system call
test, (mode mode --)	Logical compare
test, (mode single --)	Logical compare
tzcnt, (mode mode --)	Count number of trailing zero bits
ud2, (--)	Undefined instruction
verr, (mode --)	Verify a Segment for Reading
verw, (mode --)	Verify a Segment for Writing
wait, (--)	Wait
wbinvd, (--)	Write back and invalidate cache
wrmsr, (--)	Write to model specific register
xadd, (mode mode --)	Exchange and add
xchg, (mode mode --)	Exchange register/memory with register
xlat, (--)	Table look-up translation
xlatb, (--)	Table look-up translation
xor, (mode mode --)	Logical <i>xor</i>
xor, (mode single --)	Logical <i>xor</i>

Instructions whose name does not end with a comma are prefixes that have to be used in combination with another instruction. The syntax for prefixes in relation to the assembly instruction they are applied to is rather obvious: Prefixes have to be executed immediately before the assembly instruction, but they may be mixed with addressing modes and immediate values if convenient. For example, these two phrases generate the same code:

```
es: eax [edi] mov,  
eax es: [edi] mov,
```

In addition to the assembler instructions StrongForth provides four instructions for compiling bytes, words, cells and double cells into the code space. These words are mostly used by the assembler itself:

```
: db, ( single -- )  
  code-space c, ;  
  
: dw, ( single -- )  
  cast unsigned 256 /mod swap db, db, ;  
  
: dd, ( single -- )  
  code-space , ;  
  
: dq, ( double -- )  
  code-space , ;
```

Furthermore, StrongForth provides two macro instructions that can be used in the same way as assembler instructions. Macro instructions are usually compiled into a series of assembler instructions.

```
push, ( data-type -- )  
pop, ( data-type -- )
```

These two macro instructions push or pop all general-purpose registers that are included in the attributes of the given data type. Depending on the attributes, zero to six registers are being pushed or popped. General purpose registers are `eax`, `ebx`, `ecx`, `edx`, `esi` and `edi`.

Register Usage

At the beginning of this chapter, you learned that the registers whose contents are changed by your assembly code, need to be explicitly specified with `changed` in the output parameter list of the stack diagram. This rule applies only to code definitions. Colon definitions automatically determine the registers that are changed by their code, because the changed registers are stored in the attributes of each definition.

You can specify multiple registers as `changed`, as in this example:

```
code dummy ( eax: single -- eax: 1st eax: ebx: ecx: changed )  
...  
ret,  
endcode
```

On the other hand, it is always an advantage to have as few registers changed as possible. The reason is simple. If a colon definition uses a word that changes a register that is in use, it needs to save this register before compiling the code for this word. It can either push the register onto the stack, or it can move its value to a yet unused register that is left unchanged. In the above example, it might be a good idea to prevent changing registers `ebx` and `ecx` by pushing them onto the stack at the beginning of the definition and popping them before the `ret,` instruction:

```

code dummy ( eax: single -- eax: 1st eax: changed )
ebx push,
ecx push,
...
ecx pop,
ebx pop,
ret,
endcode

```

Of course, register `eax` should not be marked as unchanged if changing its value is part of the semantics of this word.

Within colon definitions, you can't manually insert `push`, and `pop`, instructions into the code. Nevertheless, the code of many definitions destroys registers that are not used as output parameters. Let's have a look at an example:

```

: new-erase ( address -> single unsigned -- )
  null single fill ; ok
  see new-erase
code new-erase ( ebx: address -> single ecx: unsigned -- eax: ecx:
edi: changed )
0042B47A: eax 00000000 mov,
0042B47F: edi ebx mov,
0042B481: 00405FB2 jmp, fill
endcode ok

```

Register `eax` is changed, because it is loaded with the fill value. The register move from `ebx` to `edi` is required, because `fill` expects the address of the array in `edi`. And finally, `ecx` changes, because `fill` doesn't bother to save it. This means, if `new-erase` is used within a colon definition, and one or more of these three registers contain values that are needed later on, the compiler has to save them before and restore them after calling `new-erase`. You can prevent the destruction of registers by explicitly marking them as unchanged:

```

: new-erase ( address -> single unsigned --   ok
  eax: ecx: edi: unchanged )
  null single fill ; ok
  see new-erase
code new-erase ( ebx: address -> single ecx: unsigned -- )
0042B486: edi push,
0042B487: ecx push,
0042B488: eax push,
0042B489: eax 00000000 mov,
0042B48E: edi ebx mov,
0042B490: 00405FB2 call, fill
0042B495: eax pop,
0042B496: ecx pop,
0042B497: edi pop,
0042B498: ret,
endcode ok

```

changed and unchanged are always used within a stack diagram:

```

changed ( stack-diagram -- 1st )
unchanged ( stack-diagram -- 1st )

```

The compiler doesn't even forget to `pop` the unchanged registers if `exit` is used within the colon definition. Instead of simply compiling `ret`,, `exit` (and `;`) executes the `exit`, macro instruction:

```
exit, ( unsigned -- )
```

The parameter of data type `unsigned` is the size of the stack frame in cells, or zero if no stack frame exists. `exit,` does three things:

- It compiles `pop,` instructions for all registers that have been marked unchanged.
- If a stack frame exists, it compiles a `leave,` instruction.
- It compiles a `ret,` instruction.

You can even use `unchanged` combined with `exit,` within code definitions:

```
code dummy ( eax: single -- eax: 1st   ok  
  eax: changed ebx: ecx: unchanged )   ok  
...  
0 exit,   ok  
endcode   ok  
see dummy  
code dummy ( eax: single -- eax: 1st eax: changed )  
004287BE: ecx push,  
004287BF: ebx push,  
...  
004287C0: ebx pop,  
004287C1: ecx pop,  
004287C2: nop,  
004287C3: ret,  
endcode ok
```

However, taking advantage of this technique is discouraged. Explicitly compiling `push,` and `pop,` instructions in combination with `ret,` instead of `exit,` produces better readable code and is less bound to errors.

Conditionals

The StrongForth assembler supports structured programming by using the following instructions instead of conditional and unconditional jump instructions and explicit labels:

```
ifcc,      ( -- origin-address )  
else,      ( origin-address -- 1st )  
then,      ( origin-address -- )  
ahead,     ( -- origin-address )  
begin,     ( -- destination-address )  
untilcc,   ( destination-address -- )  
again,     ( destination-address -- )  
whilecc,   ( destination-address -- origin-address 1st )  
repeat,    ( origin-address destination-address -- )
```

`ifcc`, `untilcc`, `whilecc`, each stand for a whole group of instructions, with `cc` being a condition derived from the bits of the x86 `eflags` register:

<code>cc</code>	Condition
<code>a</code>	above
<code>ae</code>	above or equal
<code>b</code>	below
<code>be</code>	below or equal
<code>c</code>	carry
<code>e</code>	equal
<code>g</code>	greater
<code>ge</code>	greater or equal
<code>l</code>	less
<code>le</code>	less or equal
<code>na</code>	not above
<code>nae</code>	not above or equal
<code>nb</code>	not below
<code>nbe</code>	not below or equal
<code>nc</code>	not carry
<code>ncxz</code>	not <code>cx</code> zero
<code>ne</code>	not equal
<code>necxz</code>	not <code>ecx</code> zero
<code>ng</code>	not greater
<code>nge</code>	not greater or equal
<code>nl</code>	not less
<code>nle</code>	not less or equal
<code>no</code>	not overflow
<code>np</code>	not parity
<code>ns</code>	not sign
<code>nz</code>	not zero
<code>o</code>	overflow
<code>p</code>	parity
<code>pe</code>	parity even
<code>po</code>	parity odd
<code>s</code>	sign
<code>z</code>	zero

The above instructions are the assembler's equivalent to the words `if`, `else`, `then`, `ahead`, `begin`, `until`, `again`, `while` and `repeat`. The data types `origin-address` and `destination-address` are used in a similar way as `origin` and `destination`:

```
dt address procreates origin-address
dt address procreates destination-address
```

This means that structures like

```
... ifcc, ... then, ...
... ifcc, ... else, ... then, ...
... ahead, ... then, ...
... begin, ... untilcc, ...
... begin, ... again, ...
... begin, ... whilecc, ... repeat, ...
```

can be inserted into the assembly code. These structures can also be nested if desired. They will be translated into appropriate conditional and unconditional jump instructions. Structured assembly greatly reduces the need for labels and jump instructions. If you want to use labels anyway, you can use the word `label`, that has been presented in chapter 10. Now, let's try an example:

```

code > ( ecx: unsigned ebx: 1st -- ecx: flag ecx: changed ) ok
ecx ebx cmp, ok
ifa, ecx true mov, ok
else, ecx ecx xor, ok
then, ok
ret, ok
endcode ok
see >
code > ( ecx: unsigned ebx: 1st -- ecx: flag ecx: changed )
004287D2: ecx ebx cmp,
004287D4: 004287DD jbe,
004287D6: ecx FFFFFFFF mov,
004287DB: 004287DF jmp,
004287DD: ecx ecx xor,
004287DF: ret,
endcode ok

```

As expected, `ifa`, generates a conditional jump for the inverse condition. *Above* turns to *below or equal*, because the `if` branch should be skipped whenever the condition is *not* true. `else`, resolves the conditional jump generated by `ifa`, and compiles an unconditional jump that skips the `else` branch. Finally, `then`, resolves the unconditional jump compiled by `else`,.

A second example demonstrates a typical loop structure. `bits` expects an item of data type `single` and returns the number of bits required to represent its value. `bits` actually returns the bit number of the highest 1-bit in `single`, plus 1.

```

code bits ( eax: single -- ecx: unsigned ecx: changed ) ok
  eax push, ok
  ecx ecx xor, ok
  eax eax test, ok
  begin, ok
  whilenz, ok
    ecx inc, ok
    eax 1 shr, ok
  repeat, ok
  eax pop, ok
  ret, ok
endcode ok
see bits
code bits ( eax: single -- ecx: unsigned ecx: changed )
004287E0: eax push,
004287E1: ecx ecx xor,
004287E3: eax eax test,
004287E5: 004287EC jz,
004287E7: ecx inc,
004287E8: eax 1 shr,
004287EA: 004287E5 jmp,
004287EC: eax pop,
004287ED: ret,
endcode ok

```

`whilenz`, compiles a conditional jump for the inverse condition, and `repeat`, compiles an unconditional jump and resolves the jump addresses of the complete loop structure. It is not necessary to define any labels. Note that register `ecx` has been marked as changed, while register `eax` is not, because its content is saved on the stack.

The Disassembler

The usage of the disassembler has already been demonstrated several times in the previous sections. The `assembler` vocabulary contains special versions of

```
. ( data-type -- ),  
.params ( address -> data-type unsigned -- ) and  
. ( definition -- )
```

which additionally display the register attributes of each data type.

```
disassemble ( address -- )
```

displays a disassembly of the machine code starting at the address provided as a parameter.

Disassembly automatically stops at the first `ret`, assembly instruction that is not being jumped over by a conditional or unconditional forward jump instruction.

If you include the source file `see.sf`, the virtual member `see` of class `code-definition` is being updated by a definition that uses `disassemble` to display the assembly code of an object of this class:

```
:noname ( code-definition -- )  
  cr ." code " dup .  
  dup token cast address disassemble  
  cr ." endcode"  
  inline? if ." inline" then ; \ is see  
token 5 dt code-definition vtable 'virtual !
```

29 ASCII Characters

Lower Case And Upper Case

StrongForth provides words to convert characters representing letters as well as character strings into lower case or upper case. Non-letter characters remain unchanged. The definitions of the four overloaded words are included in the source file `ascii.sf`:

```
: upcase ( character -- 1st )
  dup [char] a [ char z 1+ ] literal within
  if [ char a char A - ] literal -
  else
    case [char] ä of [char] Ä endof
        [char] ö of [char] Ö endof
        [char] ü of [char] Ü endof
    endcase
  then ;

: locase ( character -- 1st )
  dup [char] A [ char Z 1+ ] literal within
  if [ char a char A - ] literal +
  else
    case [char] Ä of [char] ä endof
        [char] Ö of [char] ö endof
        [char] Ü of [char] ü endof
    endcase
  then ;

: upcase ( caddress -> character unsigned -- )
  over swap + swap ?do i @ upcase i ! loop ;

: locase ( caddress -> character unsigned -- )
  over swap + swap ?do i @ locase i ! loop ;
```

The conversion considers the standard ASCII characters a to z and A to Z as well as all German umlauts. The usages of these words are straightforward:

```
include ascii.sf
char @ locase . @ ok
char A locase . a ok
parse-name StrongForth over over upcase type STRONGFORTH ok
```

The phrase `over swap + swap` used in the definitions of both `upcase` and `locase` converts the specification of an array from an address-plus-size representation to the limit-plus-index representation required by `do` loops. Since this calculation is used pretty often, it makes sense to factor it out. Defining a universal macro instead of a set of overloaded words for different data types is the preferred solution:

```
: bounds ( -- )
  " over swap + swap" evaluate ; immediate
```

However, this macro is not by default included in StrongForth. Note that `over` and `swap` typically compile no machine instructions at all. As a result, using `bounds` won't cause a penalty neither in code size nor in execution speed.

Non-Graphic Characters

The ANS Forth definitions `char` and `[char]` are very handy when it comes to producing single character literals. However, these words can only produce graphic ASCII characters. Whenever ASCII control characters are required, it might be useful to have a set of corresponding constants available.

`ctrl` and `[ctrl]` are variations of `char` and `[char]`, respectively. They accept only letters in lower or upper case and return the corresponding non-graphic character.

```
: ctrl ( -- character )
  parse-name
  if @ upcase dup [ char Z 1+ ] literal [char] A within
    if -273 throw
    else [char] @ -
    then
  else drop null character
  then ;

: [ctrl] ( -- )
  ctrl [literal] ; compile-only
```

The source file `ascii.sf` contains these two definitions as well as all the non-graphic character constants shown below. Here's an example of how to use the two words:

```
ctrl g . beep! Ok
: bell [ctrl] g . ; ok
bell beep! ok
```

Constants for 26 ASCII control characters can be produced using `ctrl`:

```
ctrl a constant <soh>
ctrl b constant <stx>
ctrl c constant <etx>
ctrl d constant <eot>
ctrl e constant <enq>
ctrl f constant <ack>
ctrl g constant <bel>
ctrl h constant <bs>
ctrl i constant <ht>
ctrl j constant <lf>
ctrl k constant <vt>
ctrl l constant <ff>
ctrl m constant <cr>
ctrl n constant <so>
ctrl o constant <si>
ctrl p constant <dle>
ctrl q constant <dc1>
ctrl r constant <dc2>
ctrl s constant <dc3>
ctrl t constant <dc4>
ctrl u constant <nak>
ctrl v constant <syn>
ctrl w constant <etb>
ctrl x constant <can>
ctrl y constant <em>
ctrl z constant <sub>
```

The remaining 7 ASCII control characters are defined using explicit type casts:

```
    null character constant <nul>
27 cast character constant <esc>
28 cast character constant <fs>
29 cast character constant <gs>
30 cast character constant <rs>
31 cast character constant <us>
127 cast character constant <del>
```

30 Bit Fields

In Forth 2012, the minimum size of a variable is one cell. In StrongForth, you can also define character-size variables using `cvariable` and `cvariables`. Since a single cell and even more a character is pretty small compared to the totally available memory of a system, not much memory is wasted if the data to be represented by a variable requires fewer bits than the number of bits in a cell or in a character. However, this statement can become wrong when arrays are being considered. A large array of numbers that are all less than, say, 4, actually is a waste of memory, if each such number occupies one cell or one character. Forth 2012 specifies access to character size items in memory with `C@` and `C!`. You can define arrays of characters with

```
CREATE <name> <n> CHARS ALLOT
```

or

<n> CHARS BUFFER: <name>

StrongForth supports character size items with the dedicated address data type `caddress`. Overloaded versions of `@` and `!` deal with these addresses in order to fetch and store character size items, respectively. Proper address arithmetic is also provided. And within the definition of structures and objects, you can even define character size members with `cmember` and `cmembers`. But that's still not enough. It is also possible to define bit fields that occupy a given number of bits within a class:

```
null unsigned 5 bmember channel
null flag 1 6 bmembers switches
```

define a data member of data type `unsigned` that is 5 bits long, plus an array of 6 flags that each occupy only one bit.

channel					switches																										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

When `channel` and `switches` are executed, they return compound data types `baddress -> unsigned` and `baddress -> flag`, respectively. `baddress` is a new address data type that is a child of data type `double`:

```
dt double procreates baddress
3 bit 1 bit or \ ebx/ecx \ dt baddress >attributes 'register !
```

The second line changes the default registers for items of data type `baddress` from those they inherited from data type `double`, `eax/edx`, to `ebx/ecx`. Items of data type `baddress` contain information about

- the address of the cell in which the least significant bit of the bit field is located, (usually in register `ebx`)
- the position of the least significant bit of the bit field within the cell (usually in register `cl`), and
- the size of the bit field (usually in register `ch`).

The size of a bit field may not exceed the size of one cell. However, a bit field may cross the border between two succeeding cells, e. g., on a 32-bit system, a 24 bits long bit field that starts at bit 19 is still handled correctly. This bit field extends over the 13 most significant bits of one cell and the 11 least significant bits of the next cell.

Here are some more examples:

```
hex data-space default ok
12345678 , 9ABCDEF0 , ok
test 3 dump
008D27FC: 45678E2B 12345678 9ABCDEF0 ok
test 1+ cast address -> signed 19 9 bit-field constant field2 ok
-1 field2 ! ok
test 3 dump
008D27FC: 45678E2B FE345678 9ABCDEF3 ok
+0 field2 ! ok
test 3 dump
008D27FC: 45678E2B 00345678 9ABCDEF0 ok
+13B field2 ! ok
test 3 dump
008D27FC: 45678E2B 76345678 9ABCDEF2 ok
field2 @ . -C5 ok
```

For dealing with arrays of bit fields, overloaded versions of `fill` and `erase` as well as the usual address arithmetic words are available:

```
fill ( baddress -> single unsigned 2nd -- )
erase ( baddress -> single unsigned -- )
+ ( baddress integer -- 1st )
- ( baddress integer -- 1st )
1+ ( baddress -- 1st )
1- ( baddress -- 1st )
+! ( integer baddress -> integer -- )
+! ( integer address -> baddress -- )
-! ( integer baddress -> integer -- )
-! ( integer address -> baddress -- )
```

Again, let's view some examples of their application:

```
decimal ok
test 12 4 bit-field constant field3 ok
field3 10 erase ok
test 3 dump
008D27FC: 00000E2B 76300000 9ABCDEF2 ok
field3 10 9 fill ok
test 3 dump
008D27FC: 99999E2B 76399999 9ABCDEF2 ok
7 field3 3 + ! ok
test 3 dump
008D27FC: 97999E2B 76399999 9ABCDEF2 ok
8 field3 1+ +! ok
test 3 dump
008D27FC: 97919E2B 76399999 9ABCDEF2 ok
```

Bit field members defined in classes are always packed as tightly as possible. In the example at the beginning of this section, bit field `channel` starts at bit position 0, and the bit field array `switches` starts at bit position 5. The next bit field would then start at bit position 11, unless the next member (if any) is defined with `member`, `cmember`, `sfmember` or `dfmember` instead of `bmember`. Members defined with these other variants of `member` always have to be aligned in some way.

To be able to use bit fields and bit members in your programs, you first have to include the sources from `baddress.sf`:

```
include baddress.sf
```

Since many of the words presented in this chapter are defined with `code`, the StrongForth assembler needs to be included as well.

Here are the definitions of `bmembers` and `bmember`:

```
: params! ( data-type address -> data-type bmember-definition -- )
  swap rot dt-prefix or state @ new stack-diagram
  this-attributes >data-type over param, --
  tuck param, tuck params, swap params! ;

: bmembers ( object-size single unsigned unsigned -- 1st )
  rot drop over 1- ?bit over * rot rot
  over parse-name new bmember-definition
  [dt] baddress over dt-here swap params! enddef swap + ;

: bmember ( object-size single unsigned -- 1st )
  1 bmembers ;
```

The definition of `bmembers` has some similarities with the one of `members`. However, there are differences. `bmembers` has to check the size of the desired bit field and does not perform any alignment, because bit fields can start at any bit position. It creates an object of data type `bmember-definition` instead of an object of data type `member-definition`. And the output parameter of a bit field member is of data type `baddress` instead of `address`.

So finally, here's the class definition of class `bmember-definition`:

```
dt member-definition procreates bmember-definition
class bmember-definition
  protected definitions
  null unsigned cmember 'position
  null unsigned cmember 'length

  forth definitions
  : bmember-definition ( unsigned unsigned
    caddress -> character unsigned bmember-definition -- 6 th )
    definition locals( this )
    [ address-unit-bits cells ] literal /mod cells 'offset !
    'position ! 'length ! this ;

  :noname ( compiler-workspace bmember-definition -- )
    ... ; is (compile)
endclass
```

Not surprisingly, class `bmember-definition` has two additional private members with respect to its parent class. `'position` and `'length` keep the position and the length of the bit field, respectively. Both data members are initialized by the constructor.

31 Environmental Queries

Forth 2012 specifies a word that can be used to query certain environmental parameters:

```
ENVIRONMENT? ( c-addr u - - false | i * x true )
```

You can see that StrongForth has a problem implementing this word, because its stack diagram isn't unique. However, there is a pretty nice way in which it could be implemented:

```
environment? ( caddress -> character unsigned -- address flag )
```

Instead of a varying number of output parameters, this version returns an address and a flag. The address points to a location from where the value or the values can be fetched. If the flag is false, a null address will be returned. A typical usage would look like this:

```
" /pad" environment? . -> unsigned @ . true 84 ok
```

Anyway, `environment?` does not exist in StrongForth. Forth 2012 specifies this word mainly for the purpose to query parameters without knowing whether they exist. The mayor application is to find out which word sets are available.

Since StrongForth supports all word sets specified by Forth 2012 except for the *Extended-Character* word set, there is no necessity for something like `ENVIRONMENT?`. Those Forth 2012 environment queries that are not related to the presence of word sets have in StrongForth been implemented as constants:

```
16382 constant return-stack-cells
6 constant stack-cells
8 constant address-unit-bits
84 constant /pad
255 constant /counted-string
address-unit-bits cells 4 * 2 + constant /hold
false constant floored
255 constant max-character
+2147483647 constant max-signed
4294967295 constant max-unsigned
+9223372036854775807. constant max-signed-double
18446744073709551615. constant max-unsigned-double
8 constant floating-stack
1.7976931348623149e+308 constant max-float
```

`#locals` and `wordlists` have not been provided, because StrongForth has no explicit limitation on these parameters. You can define as many locals as fit in a 128-byte stack frame, depending on the size of the locals. The size of the search order is unlimited, because it is implemented as a linked list.

An interesting fact is that `stack-cells` has only a value of 6. This seems to be a rather low value for a Forth data stack. 6 is actually the number of general-purpose registers StrongForth uses to emulate the data stack. A physical data stack does not exist in StrongForth. This means, 6 cells of data can be dealt with simultaneously. Additional stack cells have to be pushed to the return stack and are temporarily unavailable.

32 Utilities

StrongForth comes with the source files of a number of utilities. These utilities may be included on demand. Some of them demonstrate specific features of StrongForth, while others point out certain aspects of data typing and may help overcoming difficult situations.

Permutation

`permutation` calculates a permutation of an array of single-cell items with a given length. If n is the number of single cells, $n!$ different permutations exist. The last input parameter of `permutation` specifies the index of the generated permutation, which ranges from 0 to $n!-1$. These are the definitions in the source file *permutat.sf*:

```
: mswap ( address -> single 1st -- )
  dup @ rot dup @ rot rot ! swap ! ;

: fak ( unsigned -- 1st )
  dup
  if 1 swap 1+ 1 do i * loop
  else drop 1
  then ;

: permutation ( address -> single unsigned unsigned -- )
  locals( addr count index )
  count 1- fak index count 1- 0
  do over /mod
    addr swap + i + addr i + mswap
    swap count 1- i - / swap
  loop drop drop ;
```

`mswap` swaps the contents of two memory cells. `fak` calculates the faculty of an unsigned number. Overloaded definitions of `mswap` and `permutation` for arrays of double-cell items, of character-size items or of three kinds of floating-point numbers can be produced without having to change anything but the stack diagrams. The consistent use of overloaded definitions for words being applied to different operands makes this possible.

Help from the Glossary

Whenever a Forth programmer is in doubt about the stack diagram and/or the semantics of a word, the glossary is the first place to look for help. In StrongForth, an extended version of `words` displays the stack diagrams of all words with a given name in the vocabulary at the head of the context vocabulary list:

```
words /string
/string ( caddress -> character unsigned -- 1st 3rd )
/string ( caddress -> character unsigned integer -- 1st 3rd ) ok
```

In addition to this feature, the contents of the glossary can be displayed with `help`. `help` shows the glossary entries of all words with a given name in all word sets, accessing a pure ASCII text version of the glossary:

help /string

/string (caddress -> character unsigned -- 1st 3rd)
Adjust the character string at caddress -> character with length unsigned by one character. The resulting character string, specified by 1st 3rd, begins at caddress -> character plus one character and is unsigned minus one characters long.

/string (caddress -> character unsigned integer -- 1st 3rd)
Adjust the character string at caddress -> character with length unsigned by integer characters. The resulting character string, specified by 1st 3rd, begins at caddress -> character plus integer characters and is unsigned minus integer characters long.

Note: integer may be a negative value.

ok

If you are interested in the implementation, have a look at the source file `help.sf`.

Qualified Token Literals

Creating qualified token literals that can be passed to `execute` is pretty inconvenient in StrongForth:

```
( unsigned 1st -- 1st )procreates (u1--1)
( unsigned unsigned -- 1st )' / dt (u1--1) >token cast (u1--1)
```

Defining a data type for the qualified token can't be omitted, because using qualified tokens is necessary for retaining the consistency of the data type system. Next, we have to

- select a suitable overloaded version of the desired definition,
- provide the data type for the qualified token,
- execute `>token` and
- cast the unqualified token to a qualified token.

The two words `'token` and `['token]`, which can be included from the source file `token.sf`, provide a more convenient way to create qualified tokens in interpretation and compilation state, respectively. With `'token`, the previous example can be simplified:

```
'token / (u1-1)
```

`'token` parses the name of a word and the data type of the qualified token. It finds an overloaded version of the word whose execution token matches the desired qualified token and returns it directly as an item with the data type of the qualified token. Since it internally uses `>token`, it is type-safe. `['token]` is a variant of `'token` to be used in compilation state.

Replacements for ?dup

StrongForth requires all words to have well-defined stack diagrams. It is not possible to define Forth 2012 words like `?DUP`, `FIND`, `PICK` and `ROLL`, whose stack effects depend on conditions that are generally not known at compile time. Some of these words, like `FIND`, can be modified to have unambiguous stack diagrams. Most other words with ambiguous stack diagrams are dispensable. The necessity for `PICK` and `ROLL`, for example, arises only in badly factored code or can be avoided by using locals.

Nevertheless, many Forth programmers will miss ?DUP, because this word helps in some situations to keep the code short. The typical situation in which ?DUP is used is immediately preceding a conditional branch:

```
?dup if
?dup while
?dup until
```

But not even StrongForth's data type system requires that both branches start with identical compiler data type heaps, and this means that combinations of ?DUP and conditional branches can be implemented without corrupting the data type system. The source file `qdup.sf` contains definitions for the three words. ?if, ?while and ?until, which can be used instead of the above phrases. Their stack diagrams are identical to those of if, while and until, respectively. Let's begin with ?if, and compare it with the definition of if:

```
: ?if ( -- origin )
  " dup (0branch)" evaluate dt-drop new origin dt-restore ;
  compile-only

: if ( -- origin )
  [''] (0branch) compile, new origin ; compile-only
```

The single-cell item that is used as the condition is duplicated before the conditional branch, just like ?dup would have done. The trick is that in StrongForth, dup does not compile any code, if the item to be duplicated already occupies a register. No code is required to *un-duplicate* the item, and no else branch. The only things that need to be done is dropping the condition from the compiler data type heap before creating the object of data type origin, and to restore it afterwards.

Here's how ?if works in a possible implementation of spaces:

```
: spaces ( signed -- )
  +0 max ?if 0 do space loop then ;
```

Without ?if, spaces would have to be implemented like this:

```
: spaces ( signed -- )
  +0 max dup if 0 do space loop else drop then ;
```

Based on ?if, ?while can easily be implemented:

```
: ?while ( destination -- origin 1st )
  [compile] ?if swap ; compile-only
```

Using ?while, the definition of words, for example, could be simplified:

```
: words ( -- ) \ using while
  context @ last parse-name locals( addr count )
  begin dup 0<>
  while count
    if dup name addr count compare 0= else true then
    if dup cr . then prev
  repeat drop ;

: words ( -- ) \ using ?while
  context @ last parse-name locals( addr count )
  begin
  ?while count
    if dup name addr count compare 0= else true then
    if dup cr . then prev
  repeat ;
```

The implementation of `?until` in StrongForth is more complicated. The reason is that there might not be enough registers available for the emulated data stack. It is possible that the compiler data type heap saved by `begin` uses all six available registers, so there is none left for the additional condition expected by `until` or `?until`. `until` solves this problem by transferring the condition into the processor's `eflags` register, which remains untouched by possibly necessary data movement instructions. For `?until`, this technique does not work, because the condition is a single-cell item that needs to be retained. One solution is to temporarily store the condition as a local in the stack frame:

```
: ?until ( destination -- )
  " dup local r@" evaluate [compile] until " r@" evaluate
  locals-vocabulary last delete ; compile-only
```

For the local, any other name could have been chosen as well. Remember that StrongForth allows defining locals at multiple locations within a definition. As a demonstration, a simple and rather meaningless example shall suffice:

```
: test ( unsigned -- )
  begin 2* dup 7 bit and ?until . . ; ok
3 test 128 192 ok
5 test 128 160 ok
```

Related to `?if`, `?while` and `?until` is a macro that can be included from the source file `qq.sf`. The parsing macro

```
?? <name>
```

is a replacement for

```
if <name> then
```

where *name* is the name of a word with a symmetrical stack diagram, i. e., a stack diagram that doesn't change the data type heap. Typical words are `exit`, `leave`, `negate` or `cr`. For example, instead of

```
: ?negate ( integer signed -- 1st ) 0< if negate then ;
```

you can alternatively write

```
: ?negate ( integer signed -- 1st ) 0< ?? negate ;
```

in order to produce identical code. Here's the definition of `??`:

```
: ?? ( -- )
  [compile] if parse-name evaluate [compile] then ; immediate
```

Long Branches

For efficiency reasons, the unconditional and conditional jump instructions compiled by `(branch)` and `(0branch)` have 8-bit displacements. Sometimes, conditional clauses and loops are longer than the range of -128 to +127 code bytes that can be covered with an 8-bit displacement. It is not always possible to factor the source code in such a way that conditional clauses and loops are short enough. In those cases, an exception will be thrown at compile time:

```
out of branching range
```

Fortunately, the x86 processor provides long jump instructions with 32-bit displacement, which can jump to any code location within the address range. To compile long jumps, we have to use the two words `(lbranch)` and `(0lbranch)`, which were already presented in chapter 16. The source file `long.sf` contains long-branch versions of all those words that compile jump

instructions either directly or indirectly. The only difference to the definitions of the respective original instructions is that (branch) is replaced by (lbranch) and (0branch) is replaced by (0lbranch):

```
long-ahead ( -- origin )
long-if ( -- origin )
long-else ( origin -- 1st )
long-again ( destination -- )
long-until ( destination -- )
long-while ( destination -- origin 1st )
long-repeat ( origin destination -- )
long-of ( endof-origin of-origin -- 2nd 1st )
?long-do ( -- do-destination )
long-end-loop ( do-destination local-definition -- )
long-loop ( do-destination -- )
+long-loop ( do-destination -- )
-long-loop ( do-destination -- )
long-leave ( -- )
```

Whenever you encounter the message out of branching range during compilation of a word with long conditional clauses or long loops that can't be factored, consider using one of these words.

Command Line Input

By default, StrongForth provides a very simple version of accept. The only means of command line editing is the backspace key, which deletes the most recent character. This version does certainly not fulfil the expectations of a professional programmer.

Fortunately, accept is a deferred definition. You can write your own, more comfortable version of accept and assign it with is accept. The source file accept.sf contains such a new version. Its features are similar to those of the old MS-DOS command line, although this is far from being state of the art. If you're interested or want to extend those features, have a look at the source file. Instead of discussing the sources in detail, we will just present a list of the control keys you can use during line input.

Key	Description
Backspace	Delete the character left of the cursor.
Escape	Delete all characters left of the cursor.
Tab	Complete a word with the first match in the context vocabulary.
Reverse Tab	Delete the characters belonging to the word left of the cursor.
F1	Copy one character from the previous command line.
F2+c	Copy characters from the previous command line upto and not including c.
F3	Copy the remainder of the previous command line.
F4+c	Skip characters from the previous command line upto and not including c.
F5	Display @. Discard the command line and add it to the command line history.
F6	Add ctrl z to the command line
Cursor up	Recall the previous command line from the command line history.
Cursor down	Recall the next command line from the command line history.
Insert	Switch between overwrite and insert mode. Overwrite mode is the default.
Delete	Skip one character from the previous command line.

The implementation is mainly based on a class called line-editor. A 1024 characters long buffer stores the history of the most recently entered command lines. Typing control-V invokes a

debug function that displays the data members of the `line-editor` class. This function will help you with implementing your own extensions.

With the parsing word `shortcut`, you can assign macros to unused function keys like F7 to F12, Home, End, Page up, Page down, Cursor left and Cursor right:

```
shortcut StrongForth Type shortcut key: <F7>7 ok
```

From now on, whenever you press F7 while typing a command line, the character string `StrongForth` will be added to the line.

Two overloaded versions of `shortcut` allow specifying the code returned by `ekey>fkey` directly instead of manually typing the desired shortcut key, or specifying ASCII control characters as shortcut keys:

```
words shortcut  
shortcut ( character -- )  
shortcut ( unsigned -- )  
shortcut ( -- ) ok  
k-F10 shortcut over over ok  
ctrl c shortcut cast ok
```

Using these overloaded versions, you can create a file with your preferred shortcuts, to be included at the beginning of all future `StrongForth` sessions.

Quicksort

The source file `qsort.sf` contains an implementation of the *quicksort* algorithm:

```
( single 1st -- flag )procreates (s1--f)  
: mean ( address -> single 1st -- 1st )  
  over - 2/ + ;  
: mswap ( address -> single 1st -- )  
  dup @ rot dup @ rot rot ! swap ! ;  
: partition ( address -> single 1st (s1--f) -- 1st 1st 1st 1st )  
  >r over over mean @ r> locals( pivot comp ) over over  
  begin  
    swap begin dup @ pivot comp execute while 1+ repeat  
    swap begin pivot over @ comp execute while 1- repeat  
    over over <= if over over mswap swap 1+ swap 1- then  
    over over >  
  until swap rot ;  
: quicksort ( address -> single 1st (s1--f) -- )  
  locals( comp ) comp partition  
  over over < if comp recurse else drop drop then  
  over over < if comp recurse else drop drop then ;
```

This implementation of `quicksort` demonstrates both recursion and the usage of qualified tokens. `quicksort` sorts a sequence of single cell items that are stored in memory starting at the address given by its first parameter. The second parameter is the address of the last cell to be included in the sort. The third parameter of data type `(s1--f)` is the qualified token of a word that compares a couple of single cell items, returning an item of data type `flag`. By choosing an appropriate comparison word, items of any single-cell data type can be sorted in any possible way.

The items to be sorted may even be objects, provided they can be compared and arranged in some meaningful way.

The definition of `quicksort` uses the word `partition` that reorders the sequence of single-cell items in two parts, where each item belonging to the first part is lower with respect to the comparison word than each item belonging to the second part. `partition` in turn uses `mswap` and `mean`. `mswap`, which you already know from the section about permutation at the beginning of this chapter, swaps the contents of two single-cell memory locations. `mean` calculates the mean between two addresses.

Here's a simple example that sorts 10 unsigned numbers in either direction, using `<` and `>` as comparison words:

```
data-space default ok
here -> unsigned constant start ok
1 , 6 , 0 , 2 , 5 , 9 , 3 , 8 , 7 , 4 , ok
here -> unsigned constant end ok
: print end start do i @ . loop ; ok
print 1 6 0 2 5 9 3 8 7 4 ok
:noname ( single 1st -- flag )
  cast integer swap cast integer swap < ; ok
start end 1- rot dt (sl--f) >token cast (sl--f) quicksort ok
print 0 1 2 3 4 5 6 7 8 9 ok
:noname ( single 1st -- flag )
  cast integer swap cast integer swap > ; ok
start end 1- rot dt (sl--f) >token cast (sl--f) quicksort ok
print 9 8 7 6 5 4 3 2 1 0 ok
```

The fact that the comparison words generally expect two items of data type `single` on the stack means that `>token` cannot be directly applied to words like `<` and `>`. A word that expects items of data type `single` matches items of data types `unsigned` on the stack, but not vice versa. We have to define variants of `<` and `>` for two items of data type `single`. Of course, these two variants do not need to have names.

Only one restriction remains: The items to be sorted have to be cell sized. If you want to sort items of double-cell or character size, or floating-point numbers, appropriate overloaded versions of `mswap`, `partition` and `quicksort` have to be defined.

Templates

The technique of overloading words often results in sets of definitions that look very similar. A good example are the three versions of `constant`, whose definitions differ only in the ancestor data type they are applied to:

```
: constant ( single -- )
  parse-name new single-definition
  dt-here over params! tuck assign enddef ;

: constant ( double -- )
  parse-name new double-definition
  dt-here over params! tuck assign enddef ;

: constant ( float -- )
  parse-name new float-definition
  dt-here over params! tuck assign enddef ;
```

We could actually create a source file called `constant.sfx` which contains a definition of `constant` with a placeholder instead of the data type:

```
: constant ( %DT% -- )
  parse-name new %DT%-definition
  dt-here over params! tuck assign enddef ;
```

This is called a template. Of course, trying to directly include this file causes an exception to be thrown, because the word `%DT%` will not be found in any vocabulary. We have to perform a string substitution on the file to convert it to something that can be compiled:

```
" constant.sfx" r/o open constant src ok
" constant.sf" r/w create constant dst ok
" single" " DT" replaces ok
src dst substitute . 2 ok
0. src reposition ok
" double" " DT" replaces ok
src dst substitute . 2 ok
0. src reposition ok
" float" " DT" replaces ok
src dst substitute . 2 ok
src close ok
dst close ok
```

The newly created file `constant.sf` contains exactly the three definitions of `constant` shown above, to be included with

```
include constant.sf
```

Of course, using a template in such a simple case doesn't make much sense. Things look differently if the templates get more complex, or if we don't know in advance, which combinations of actual strings will be inserted instead of the placeholders. The definitions that implement the quicksort algorithm, for example, need two placeholders. A suitable template file called `qsort.sfx` looks like this:

```
: mean ( %ADDR->ITEM% 1st -- 1st )
  over - 2/ + ;

: mswap ( %ADDR->ITEM% 1st -- )
  dup @ rot dup @ rot rot ! swap ! ;

: partition ( %ADDR->ITEM% 1st %QTOKEN% -- 1st 1st 1st 1st )
  >r over over mean @ r> locals( pivot comp ) over over
  begin
    swap begin dup @ pivot comp execute while 1+ repeat
    swap begin pivot over @ comp execute while 1- repeat
    over over <= if over over mswap swap 1+ swap 1- then
    over over >
  until swap rot ;

: quicksort ( %ADDR->ITEM% 1st %QTOKEN% -- )
  locals( comp ) comp partition
  over over < if comp recurse else drop drop then
  over over < if comp recurse else drop drop then ;
```

The contents of `qsort.sfx` differs from the contents of `qsort.sf`, as presented in the previous section, only in replacing two data types by the placeholders `%ADDR->ITEM%` and `%QTOKEN%`. With a simple string substitution, you can generate `qsort.sf` from `qsort.sfx`, or you can generate versions of the quicksort algorithm for character-size items or for double-cell items:

```

" qsort.sfx" r/o open constant src ok
" qsort2.sf" r/w create constant dst ok
( single 1st -- flag )procreates (s1--f) ok
" caddress -> single" " ADDR->ITEM" replaces ok
" (s1--f)" " QTOKEN" replaces ok
src dst substitute . 6 ok
0. src reposition ok
( double 1st -- flag )procreates (d1--f) ok
" address -> double" " ADDR->ITEM" replaces ok
" (d1--f)" " QTOKEN" replaces ok
src dst substitute . 6 ok
0. dst reposition ok
src close ok
dst include ok

```

Including the destination file compiles a total of 8 words:

```

quicksort ( address -> double 1st (d1--f) -- )
partition ( address -> double 1st (d1--f) -- 1st 1st 1st 1st )
mswap ( address -> double 1st -- )
mean ( address -> double 1st -- 1st )
quicksort ( caddress -> single 1st (s1--f) -- )
partition ( caddress -> single 1st (s1--f) -- 1st 1st 1st 1st )
mswap ( caddress -> single 1st -- )
mean ( caddress -> single 1st -- 1st )

```

The words are shown in reverse order. Finally, here's an example using quicksort for character strings:

```

" The quick brown fox jumps over the lazy dog." tuck pad swap move
ok
dup pad dup rot + 1- ok
:noname ( single 1st -- flag )
  swap cast integer swap cast integer < ; ok
dt (s1--f) >token cast (s1--f) quicksort ok
pad swap type .Tabcdeeeefghhijklmnooopqrrstuvwxyz ok

```

Of course, you can also include conditional compilation with [if] ... [else] ... [then] in a source template. The conditions can be calculated based on the actual values of placeholders.

The StrongForth Test Suite

Almost all StrongForth words have been tested by a comprehensive test suite. However, the sources of the test suite will not be published. You can write your own tests with the words used by the StrongForth test suite, whose sources can be found in the file `test.sf`:

```

: ?test ( single 1st -- )
  <> if -298 throw then ;

: ?test ( double 1st -- )
  <> if -298 throw then ;

: ?test ( float 1st -- )
  <> if -298 throw then ;

```



```
: ?empty ( -- )
  dt-depth fp@ 8 < or if -299 throw then ;
```

The three overloaded versions of `?test` throw exceptions if the two input parameters of data types `single`, `double` or `float` do not have the same value. `?empty` throws an exception if the data type heap and/or the hardware floating-point stack are not empty after execution of a word. In order to allow rounding differences for floating-point numbers, an additional word is handy when testing the results of floating-point numbers:

```
: ~test ( float 1st float -- )
  ~ invert if -298 throw then ;
```

Furthermore, a means is provided to ensure that the execution of a word does not cause side effects by allotting memory space in the data, code and stack space. The parsing word `test:` prints the parsed name, typically the name of the word to be tested, and stores the three memory space pointers in a variable:

```
null address 3 variables spacetop
```

```
: test: ( -- )
  spacetop @ if ;test then
  data-space here spacetop !
  stack-space here [ spacetop 1+ ] literal !
  code-space here [ spacetop 2 + ] literal !
  null character parse cr type space ;
```

`;test`, which is also executed by `test:`, throws an exception if one of the memory space pointers has been changed since the beginning of the most recent test:

```
: ;test ( -- )
  data-space here spacetop @ <> abort" data space leak"
  stack-space here [ spacetop 1+ ] literal @ <>
  abort" stack space leak"
  code-space here [ spacetop 2 + ] literal @ <>
  abort" code space leak"
  null address spacetop ! ;
```

This word should be placed at the end of each test file. As an example, here's a typical test file that tests the words defined in the source file `bcd.sf`, which supports double-cell binary coded decimal numbers:

```
require test.sf

test: bcd

( bcd dup input-params 1 ?test
@ dt bcd ?test delete ?empty
dt bcd size 2 cells ?test ?empty
dt bcd parent dt double ?test ?empty
dt bcd vtable null vtable ?test ?empty

test: bcd>

hex 3088719546092291. decimal cast bcd bcd>
3088719546092291. ?test ?empty

test: >bcd

4195220691443962. >bcd
hex 4195220691443962. decimal cast bcd ?test ?empty

;test
```

execute-parsing

The definitions of the two words presented in this section are contained in the source file `xparsing.sf`. `execute-parsing` allows a parsing word to be executed with `execute` that parses a given character string instead of the default input stream:

```
: execute-parsing ( caddress -> character unsigned (-- ) -- )
  default-input-stream @ locals( stream )
  rot rot new string-input-stream default
  execute
  default-input-stream @ delete stream default ;
```

The semantics are pretty simple. First, `execute-parsing` saves the default input stream. It then creates a new string input stream that is initialized with the character string to be parsed, and makes it the new default input stream. After executing the word, the default input stream is restored.

A variant of `execute-parsing` is `execute-parsing-file`, which makes the word to be executed parse a file instead of a character string:

```
: execute-parsing-file ( file (-- ) -- )
  default-input-stream @ locals( stream )
  swap /pad new file-input-stream default
  refill drop execute
  default-input-stream @ delete stream default ;
```

Because both words use a version of `execute` for a specific qualified token, overloaded versions have to be defined for each qualified token `execute-parsing` or `execute-parsing-file` is to be applied to. You might consider using a template for this purpose.

Executing catch

It was already stated in chapter 20 that although `catch` is a compile-only word in StrongForth, it is possible to provide a version for interpretation state as well. The file `catch.sf` contains the source code for an execute-only version of `catch`:

```
require xparsing.sf

dt colon-definition procreates temp-colon-definition
class temp-colon-definition
  private definitions
  null definition member 'save-latest
  null address member 'save-code
  forth definitions private

  : temp-colon-definition ( caddress -> character unsigned
    temp-colon-definition -- 4 th )
    latest locals( this lat ) code-definition
    lat 'save-latest !
    code-space here over 'save-code ! ;

  : params! ( definition temp-colon-definition -- )
    state @ new stack-diagram rot over params,
    [dt] signed over param, swap params! ;
```

```

:noname ( temp-colon-definition -- )
  dup 'save-code @ code-space here - code-space allot
  dup 'save-latest @ to latest
  [parent] delete ; is delete
endclass

: catch ( -- )
  parse-name over over false match-criterion search-context
  if rot rot new temp-colon-definition begin-compilation
    tuck params! dup params>dt
    dup name [ ' catch dt (--) >token cast (--) ] literal
    execute-parsing [compile] ;
    latest name evaluate latest delete
  else drop drop drop -13 throw
  then ; execute-only

```

It is still not possible to directly catch an exception during interpretation. What the `execute-only` version of `catch` does is compile a temporary definition using the `compile-only` version of `catch`, execute this definition and then delete it.

First, `catch` tries to find a word with the given name and a matching stack diagram:

```
<name> ( <input-parameters> -- <output-parameters> )
```

The temporary colon definition compiled by `catch` looks like this:

```
: <name> ( <input-parameters> -- <output-parameters> signed )
  catch <name> ;
```

Note that it has the same name as the word whose exception shall be caught. The `catch` used in this definition needs to be compiled by `execute-parsing`, because it is a parsing word that should parse `<name>`. After the temporary colon definition is finished, it is executed with `evaluate`. Finally, the temporary definition is being deleted.

The temporary colon definition is an object of class `temp-colon-definition`, which is a child of class `colon-definition`. Its constructor is the same as the one of classes `code-definition` and `colon-definition`, except that it additionally saves the value of `latest` and the current code space pointer. The public method `params!` copies the stack diagram of its first parameter and then adds data type `signed` as the last output parameter. `delete` removes all traces of the temporary colon definition by not only deleting the object itself, but also by restoring the value of `latest` and by de-allotting its machine code.

Here's a simple example:

```
-129 catch ?byte . .s . -289 signed 4346708 ok
```

Bear in mind that `catch` cannot guarantee that the a word that caused an exception returns meaningful results at all.

33 Inside StrongForth

StrongForth's data stack is purely virtual. Single-cell and double-cell items are stored in the processor's six general-purpose registers `eax`, `ebx`, `ecx`, `edx`, `esi` and `edi`. Conditions may additionally be stored as a combination of flags in the `eflags` register. A data stack with a size of only six cells is definitely not large enough. So, if registers are used up, currently unused cells are temporarily pushed onto the return stack. The data type system takes care of not mixing up data items, return addresses and stack frames. The information about where stack items are actually stored, in which register or on the return stack, is kept in the attributes of the data types on the data type heaps. As a result, no word can have more items as input or output parameters than fit into six cells. This has turned out not to be a serious restriction. Forth 2012 specifies only two words using up this limitation:

```
2ROT ( x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2 )
TIME&DATE ( -- +n1 +n2 +n3 +n4 +n5 +n6 )
```

Anyway, such a lot of parameters are difficult to handle, and readability suffers. Therefore, it can be stated that this is not really a limitation.

The lack of a physical data stack whose depth could be calculated also means that Strongforth has no match for the Forth 2012 word `DEPTH`. In order to find out what's on the virtual data stack, you have to investigate the respective data type heap.

Generally, a definition's stack diagram determines in which registers it expects its input parameters and in which registers it returns its output parameters:

assembler

```
' #> . #> ( eax/edx: number-double -- ebx: caddress -> character
ecx: unsigned eax: ecx: ebx: changed ) ok
```

We have to use the assembler's version of `.` for items of data type definition to show the registers. The first input parameter is expected in register pair `eax/edx`, and the two output parameters are returned in register `ebx` and `ecx`. The contents of registers `eax`, `ebx` and `ecx` are potentially changed by this word. If the parameters are in different registers, appropriate register move or exchange instructions have to be compiled before the call instruction. If the parameters are located on the return stack, they have to be popped into the appropriate registers.

You might expect that register juggling is required rather often to provide the input parameters to the next word. But this is not true. Each data type has a default register that is inherited by its children. The register code is stored in the member `'register` of class `data-type-attributes`. Here are some examples:

Data type	Default register
single	<code>eax</code>
double	<code>eax/edx</code>
address	<code>ebx</code>
unsigned	<code>ecx</code>
logical	<code>ecx</code>
file	<code>edx</code>
object	<code>esi</code>
definition	<code>edi</code>
vocabulary	<code>edi</code>
data-type	<code>ebx/ecx</code>

In most cases, output parameters of a certain data type are returned in their default register, and input parameters are expected in their default register as well. No register movement instructions need to be compiled between the call instructions of two succeeding words. Here's an example of a word that actually uses #>:

```
' picture prev see
code picture ( eax/edx: double -- ebx: caddress -> character
ecx: unsigned eax: ecx: edx: ebx: changed )
00422C69: 00422B2D call, <#
00422C6E: 00422C5D call, #s
00422C73: 00422B35 jmp, #>
endcode ok
```

Because all parameters occupy their default registers, including the words compiled in the definition, register movement instructions are not required.

Code and colon definitions are objects of class `code-definition`, whose version of the virtual method (`compile`) compiles a call to the definition's runtime code in the code space. If the definition is marked as `inline`, the runtime code is compiled byte by byte, without the expense of a subroutine call and return. Inline definitions usually consist of only few machine code instructions. They may not contain subroutine calls.

However, many basic forth words like `dup`, `drop`, `swap`, `over` and `rot`, `+` and `-`, `@` and `!`, `*` and `/`, `max` and `min`, `1+` and `1-`, `2*` and `2/`, `lshift` and `rshift`, `<` and `>`, `0<` and `0>`, and `,or`, `xor` and `invert`, and several others, are not colon definitions. Their code are compiled by dedicated versions of (`compile`), that create machine code instructions depending on the registers occupied by the input parameters, and assigning suitable registers to the output parameters. Let's view an example:

```
( integer -- 1st ) ' 1+ . 1+ ( any: integer -- 1st ) ok
```

`any:` means that the input parameter may reside in *any* register. Compiling this version of `1+` produces machine code that fits to the actual register. Again: No register juggling is required, like in this example:

```
: 3+ ( integer -- 1st ) 1+ 1+ 1+ ; ok
see 3+
code 3+ ( ecx: integer -- ecx: 1st ecx: changed )
00428594: ecx inc,
00428595: ecx inc,
00428596: ecx inc,
00428597: ret,
endcode ok
```

Next, we'll have a look at another special *register*:

```
12 constant dozen ok
latest . dozen ( -- lit: unsigned ) ok
```

What does `lit:` mean? Yes, `lit:` stands for *literal*. The constant `dozen` compiles a literal, that has not yet been assigned a register. The assignment is postponed by the compiler upto a point where it knows what to do with the literal. What this means is best demonstrated by another example:

```
: dozen+@ ( address -> single -- 2nd ) dozen + @ ; ok
latest see
code dozen+@ ( ebx: address -> single -- eax: 2nd
eax: ebx: changed )
00428598: ebx +30 add,
```

```

0042859B: eax [ebx] mov,
0042859D: ret,
endcode ok

```

Three words in the source code are compiled into two machine code instructions. +30 is a hexadecimal value equal to 48 decimal, which actually is *a dozen cells*. The secret behind this optimization lies in the definition of +, which can process a literal as the second input parameter:

```
+ ( any: address -> single lit: any: integer -- 1st )
```

Both input parameters can be placed in any register. The second input parameter may also be a literal. If it is, the literal is included in the add instruction as an immediate operand. You can also see that this example leaves room for another optimization by combining the complete semantics of this word into one single machine code instruction.

The same example will also work if we replace dozen with an explicit numeric literal:

```
: dozen+@ ( address -> single -- 2nd ) 12 + @ ; ok
```

The last special register we shall have a look at is `eflags`:. `eflags`: applies to conditions of data type `flag` that are combinations of bits in the processor's `eflags` register. Sounds complicated? Here's another example:

```

( signed 1st -- flag )' < cr .
< ( any: signed lit: any: 1st -- eflags: flag ) ok
: ?low ( signed -- ) +10 < if ." too low" then ; ok
latest see
code ?low ( eax: signed -- ecx: ebx: changed )
0042B584: eax +0A cmp,
0042B587: 0042B598 jge,
0042B589: ecx 00000007 mov,
0042B58E: ebx 008F2710 mov,
0042B593: 00406AAC call, type
0042B598: ret,
endcode ok

```

< accepts its operands in arbitrary registers. The second operand may also be a literal. The result is returned as a condition in the `eflags` register. This condition is actually compiled by (`0branch`) within `if` as a conditional jump instruction.

Other words that return conditions in the `eflags` register are comparison words, zero comparison words, the runtime codes of `loop`, `+loop` and `-loop`, and words that directly query bits in the `eflags` register like `carry?`.

Finally, we'll have a look at the implementations of `dup`, `drop`, `swap`, `over`, `rot`, `nip` and `tuck` for single-cell and double-cell items. The interesting aspect is that these words do not change any data. They just manipulate the data stack. Now, what does that mean if we have no physical data stack and all items are stored in registers?

Let's begin with `swap`:

```

( single single -- 2nd 1st )' swap cr .
swap ( lit: any: single lit: any: single -- 2nd 1st ) ok

```

The two operands may be in any registers and even be provided as literals. The compiler can simply swap the operands by renaming the registers and/or literals. If, for example, the top of the data stack is in register `eax` and the next of stack is in register `ebx`, it is sufficient to swap the register assignments on the data type heap. Now, the top of stack is in register `ebx` and the next of stack is in register `eax`. No code needs to be compiled. `rot` is similar.

What about dup? Again, the operand is in a register, or it is a literal:

```
( single -- 1st 1st )' dup cr .  
dup ( lit: any: single -- 1st 1st ) ok
```

No code needs to be compiled, and only the data type heap needs to be changed. The two output parameters are now both in the same register. A register move becomes necessary only if this register is being changed by a word compiled later. `over` and `tuck` work the same way.

`drop` and `nip` are even simpler. Since the item to be dropped is in a register, there's no need for stack manipulations. It is sufficient to drop the data type off the data type heap. The register is now available to be used by other items.

To illustrate the fact that pure stack manipulations actually come for free, here's how a word with multiple stack manipulation words is being compiled:

```
: /string ( caddress -> character unsigned integer -- 1st 3rd )  
  tuck - rot rot + swap ; ok  
latest see  
code /string ( ebx: caddress -> character ecx: unsigned  
eax: integer -- ebx: 1st ecx: 3rd ecx: ebx: changed )  
0042B599: ecx eax sub,  
0042B59B: ebx eax add,  
0042B59D: ret,  
endcode ok
```

You can see: StrongForth's optimizing compiler can generate pretty efficient code compared to a Forth system with a physical data stack. If you like, you can run the *BYTE Magazine* sieve benchmark:

```
8190 CONSTANT SIZE  
-1 VARIABLE FLAGS SIZE ALLOT  
  
: PRIMES  
  FLAGS SIZE -1 FILL  
  0 SIZE 0  
  DO I FLAGS + C@  
    IF I 2 * 3 + DUP I +  
      BEGIN DUP SIZE <  
        WHILE DUP FLAGS + 0 SWAP C! OVER +  
        REPEAT DROP DROP 1+  
      THEN  
    LOOP ;
```

Some minor changes have to be made to adapt this benchmark to StrongForth:

```
8190 constant size-sieve  
true size-sieve cvariables flags  
  
: primes ( -- unsigned )  
  flags size-sieve true fill  
  0 size-sieve 0  
  do flags i + @  
    if i 2 * 3 + dup i +  
      begin dup size-sieve <  
        while flags over + false swap ! over +  
        repeat drop drop 1+  
      then  
    loop ;
```

Note that `2 *` is written instead of `2*`, because this is what the original version does as well. The name of the constant has to be changed, because `size` already exists in several overloaded versions. A new version with no input parameters at all would hide them all.

Appendix A: Starting StrongForth

StrongForth does not need to be installed. It is sufficient to copy a number of files into a new directory:

- `StrongForth.exe` is the executable.
- `StrongForth.blk` is the default block file with 4096 empty blocks.
- `StrongForth.msg` is another block file with 32 blocks. Each line contains one of 512 messages padded with space characters.
- `glossary.txt` is a pure text file that contains a copy of the StrongForth forth glossary. `help` uses this file to display glossary entries for all overloaded words with a given name.
- `*.sf`, with `*` being the usual wildcard, are StrongForth source files.
- `*.sfx`, with `*` being the usual wildcard, are StrongForth template files.

`StrongForth.exe` starts with including the source file `StrongForth.sf`. Many of StrongForth's predefined words are not part of the executable. Instead, they have to be compiled from the sources in `StrongForth.sf`. At the end of `StrongForth.sf`, you can add commands to include additional sources, for example `float.sf` if you need floating-point support, or `block.sf`, if you want to work with blocks. You can also include your own source files, so your application compiles and starts automatically. Here's an alphabetical list of the source files and template files that are available out of the box. You might call it a library:

Filename	Description
<code>2dup.sf</code>	Words dealing with pairs of single-cell items: <code>2dup</code> and <code>2drop</code> (see chapter 1), <code>2@</code> and <code>2!</code> (see chapter 2) and <code>2constant</code> (see chapter 10).
<code>accept.sf</code>	<code>accept</code> with advanced line-editing functions (see chapter 32).
<code>alias.sf</code>	<code>alias</code> is a replacement for <code>SYNONYM</code> (see chapter 22).
<code>ancestor.sf</code>	Obtain the ancestor of a given data type (see chapter 7).
<code>ascii.sf</code>	<code>upcase</code> , <code>locase</code> and ASCII control characters (see chapter 29).
<code>asm.sf</code>	Assembler and disassembler (see chapter 28).
<code>baddress.sf</code>	Words dealing with bit addresses and bit fields (see chapter 30).
<code>base.sf</code>	Temporarily changing and restoring the number conversion radix.
<code>bcd.sf</code>	Conversion words for double-cell binary coded decimal numbers.
<code>bench.sf</code>	Sieve benchmark from <i>Byte</i> magazine (see chapter 33).
<code>bits.sf</code>	Assembler demo: Calculate the number of the highest 1 bit in a single cell (see chapter 28).
<code>block.sf</code>	Implementation of the <i>Block</i> word set (see chapter 25).
<code>bounds.sf</code>	Convert <i>address-and-size</i> to <i>limit-and-index</i> representation (see chapter 29).
<code>catch.sf</code>	<code>execute-only</code> version of <code>catch</code> (see chapter 32).
<code>cflag.sf</code>	Convert a Forth flag (false or true) to a C flag (0 or 1).
<code>complex.sf</code>	Support for complex floating-point numbers (see appendix B).
<code>constant.sfx</code>	Template for creating constant definitions (see chapter 32).
<code>countbit.sf</code>	Count the number of 1 bits in a single-cell or double-cell value.
<code>dot.sf</code>	Formatted display of time, date, weekday and roman numbers.
<code>dpl.sf</code>	Vocabulary for numbers entered in fixed-point notation.
<code>editor.sf</code>	Port of the <i>FIG</i> Forth block line editor (see chapter 25).
<code>escape.sf</code>	Source code of <code>\</code> for escaped string literals (see chapter 6).
<code>float.sf</code>	Support for floating-point numbers (see chapter 24).
<code>fxam.sf</code>	Query the state of the top-most floating-point number on the stack.

help.sf	Display glossary entries (see chapter 32).
long.sf	Conditionals and loops with long branches (see chapter 32).
macro.sf	Defining word for macros that evaluate given character strings.
model.sf	Equivalent (sometimes simplified) definitions for StrongForth words that are not compiled from source code. This file is for reference only and should not be included.
mrg-splt.sf	Versions of <code>merge</code> and <code>split</code> that parse the destination data types.
msvcrt.sf	Constants for <i>MSVCRT</i> data types (see chapter 21).
on_off.sf	Assign <code>true</code> or <code>false</code> flags to single-cell and character-size variables.
order.sf	<i>Seach-Order</i> word set (see chapter 23).
permutat.sf	Calculate permutations of an array (see chapter 32).
propagat.sf	Propagate a compound data type from the compiler data type heap to the interpreter data type heap.
qdup.sf	Replacements for <code>?DUP</code> : <code>?if</code> , <code>?while</code> and <code>?until</code> (see chapter 32).
qq.sf	<code>?? <name></code> is a shortcut for <code>if <name> then</code> (see chapter 32).
qsort.sf	Quicksort algorithm for single-cell items (see chapter 32).
qsort.sfx	Template of quicksort algorithm (see chapter 32).
sdump.sf	Reverse and destroying stack dump in data type specific formats.
see.sf	Assign de-compilation and disassembly words to the virtual method <code>see</code> of class definition and its child classes (see chapter 22).
self.sf	<code>self</code> is used in the stack diagram of class methods to specify a data type reference to the last input parameter.
sgn.sf	Signum function for single-cell and double-cell signed numbers.
smartptr.sf	The smart pointer <code>@@</code> compiles <code>@</code> repeatedly while a suitable overloaded version is available.
sqrt.sf	Calculate the square root of single-cell and double-cell unsigned numbers.
stack.sf	Container classes for <i>stack</i> and <i>queue</i> (see chapter 13).
strext.sf	<i>String Extension</i> word set (see chapter 26).
StrongForth.sf	Definitions of StrongForth words that are to be compiled from source code. This file is automatically included on startup.
struct.sf	Words supporting structures (see chapter 27).
test.sf	Words used by the StrongForth test suite (see chapter 32).
token.sf	Interpreting and compiling qualified token literals (see chapter 32).
xparsing.sf	<code>execute-parsing</code> and <code>execute-parsing-file</code> (see chapter 32).

Appendix B: Complex Numbers

Scientific applications sometimes have to calculate with so-called *complex numbers* instead of with normal (scalar) floating-point numbers. A complex number is composed of a real part and an imaginary part:

$$z = x + iy$$

i is the imaginary unit, the square root of minus one. Alternatively, complex numbers can be expressed by their absolute value, i. e., the geometric distance from the zero point, and their angle with respect to the real axis of the complex plane:

$$z = r \cdot \exp(i\varphi)$$

These are the relationships between the parameters:

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \varphi &= \arctan(y/x) \\ x &= r \cdot \sin(\varphi) \\ y &= r \cdot \cos(\varphi) \end{aligned}$$

Using complex numbers is especially useful for describing the amplitude and the phase of alternating voltage and current in electrical engineering. Furthermore, the mathematics of quantum computers is based on complex numbers.

An extended version of StrongForth 3.1 supports complex numbers. In order to gain full access to complex numbers, you have to include the source file `complex.sf`:

```
include complex.sf
```

Each complex number consists of two floating-point numbers, which are aggregated in a new data type:

```
null data-type 20 procreates complex
```

A complex number occupies 20 bytes in memory, because one floating-point number needs 10 bytes. It would have been possible to implement complex number support as a source library, but integrating them into the system offers more opportunities for code optimizations. This means that several words have to be overloaded with versions for complex numbers or addresses of complex numbers, including single- and double-precision.

For `dup` and `drop`, one additional version each suffices:

```
dup ( complex -- 1st 1st )
drop ( complex -- )
```

For `swap`, `over`, `nip` and `tuck`, we now need 16 versions each to be able to deal with all combinations of single, double, float and complex parameters. To show the pattern, these are the overloaded versions required for `swap`:

```
swap ( single single -- 2nd 1st )
swap ( single double -- 2nd 1st )
swap ( single float -- 2nd 1st )
swap ( single complex -- 2nd 1st ) \ new
swap ( double single -- 2nd 1st )
swap ( double double -- 2nd 1st )
swap ( double float -- 2nd 1st )
swap ( double complex -- 2nd 1st ) \ new
```

```

swap ( float single -- 2nd 1st )
swap ( float double -- 2nd 1st )
swap ( float float -- 2nd 1st )
swap ( float complex -- 2nd 1st ) \ new
swap ( complex single -- 2nd 1st ) \ new
swap ( complex double -- 2nd 1st ) \ new
swap ( complex float -- 2nd 1st ) \ new
swap ( complex complex -- 2nd 1st ) \ new

```

For rot, even 64 versions are required. Still, this is not supposed to be a problem, because they all share the same name and the semantics are somehow identical.

To access complex numbers in memory and in the stack frame, overloaded versions of some more words have to be provided:

```

! ( complex address -> 1st -- )
! ( complex sfaddress -> 1st -- )
! ( complex dfaddress -> 1st -- )
@ ( address -> complex -- 2nd )
@ ( sfaddress -> complex -- 2nd )
@ ( dfaddress -> complex -- 2nd )
+! ( complex address -> complex -- )
+! ( complex sfaddress -> complex -- )
+! ( complex dfaddress -> complex -- )
-! ( complex address -> complex -- )
-! ( complex sfaddress -> complex -- )
-! ( complex dfaddress -> complex -- )
, ( complex memory-space -- )
sf, ( complex memory-space -- )
df, ( complex memory-space -- )
, ( complex -- )
sf, ( complex -- )
df, ( complex -- )
fill ( address -> complex unsigned 2nd -- )
fill ( sfaddress -> complex unsigned 2nd -- )
fill ( dfaddress -> complex unsigned 2nd -- )
erase ( address -> complex 2nd -- )
erase ( sfaddress -> complex 2nd -- )
erase ( dfaddress -> complex 2nd -- )
move ( address -> complex 1st unsigned -- )
move ( sfaddress -> complex 1st unsigned -- )
move ( dfaddress -> complex 1st unsigned -- )
(>r) ( complex -- )

```

In order to fully support address arithmetic for items of data type complex, all words with built-in address arithmetic need to be overloaded:

```

+ ( address -> complex integer -- 1st )
+ ( sfaddress -> complex integer -- 1st )
+ ( dfaddress -> complex integer -- 1st )
- ( address -> complex integer -- 1st )
- ( sfaddress -> complex integer -- 1st )
- ( dfaddress -> complex integer -- 1st )
- ( address -> complex 1st -- signed )
- ( sfaddress -> complex 1st -- signed )
- ( dfaddress -> complex 1st -- signed )

```

```

1- ( address -> complex -- 1st )
1- ( sfaddress -> complex -- 1st )
1- ( dfaddress -> complex -- 1st )
1+ ( address -> complex -- 1st )
1+ ( sfaddress -> complex -- 1st )
1+ ( dfaddress -> complex -- 1st )
-! ( integer address -> address -> complex -- )
-! ( integer address -> sfaddress -> complex -- )
-! ( integer address -> dfaddress -> complex -- )
+! ( integer address -> address -> complex -- )
+! ( integer address -> sfaddress -> complex -- )
+! ( integer address -> dfaddress -> complex -- )
(loop) ( address -> complex -- flag )
(loop) ( sfaddress -> complex -- flag )
(loop) ( dfaddress -> complex -- flag )
(+loop) ( integer address -> complex -- flag )
(+loop) ( integer sfaddress -> complex -- flag )
(+loop) ( integer dfaddress -> complex -- flag )
(-loop) ( integer address -> complex -- flag )
(-loop) ( integer sfaddress -> complex -- flag )
(-loop) ( integer dfaddress -> complex -- flag )

```

Overloaded versions of `split` and `merge` perform conversions between complex numbers and floating-point numbers in both directions. `split` returns the real part and the imaginary part (in this order) of a complex number, while `merge` joins the real part and the imaginary part into one complex number. Note that both words, just like the versions of `split` and `merge` for double-cell items, do not have any runtime semantics.:

```

split ( complex -- float float )
merge ( float float -- complex )

```

`merge` can be used to easily enter a complex number as a literal. However, it seems better to use an alias name of `merge` for this purpose:

```
' merge alias i*+
```

`i*+` makes obvious, that the imaginary part is being multiplied by the imaginary unit `i` and then added to the real part. For example,

```
+3.4e-3 -2.7e-3 i*+
```

creates a complex number with `+3.4e-3` being the real part and `-2.7e-3` being the imaginary part. For obtaining the real part and the imaginary part of a complex number, it is also possible to use these words:

```

( float -- )' drop alias re ( complex -- float )
( float float -- 2nd )' nip alias im ( complex -- float )

```

To be able to compile complex literals, overloaded versions of `(literal)` and `literal,` are required:

```

(literal) ( complex data-type -- )
literal, ( complex address -> data-type -- )

```

Using `literal,`, an overloaded version of `literal` for complex numbers can be defined:

```

: literal ( complex -- )
  [compile] [ dt-here ] literal, ; compile-only

```

Some predefined complex numbers are available:

```
0e0 1e0 i*+ constant i
0e0 -1e0 i*+ constant -i
0e0 0e0 i*+ constant 0i0
```

Note the name conflict with the innermost loop index `i`. However, this conflict arises only within the scope of `do` loops. You can get around it by renaming the constant `i` to `+i`, or by using an explicit literal instead:

```
... [ 0e0 1e0 i*+ ] literal ...
```

Here's the definition of the overloaded version of `constant` for complex numbers. It looks almost the same as the other versions:

```
: constant ( complex -- )
  parse-name new complex-definition
  dt-here over params! tuck assign enddef ;
```

Class `complex-definition` is a child of class `definition` that compiles a complex number literal. It has two constructors to create definitions either with a name or without a name. The class definition looks similar to the one of `float-definition`:

```
dt definition procreates complex-definition
class complex-definition
  protected definitions
  null complex member 'value
  forth definitions
  : complex-definition ( complex-definition -- 1st )
    definition ;
  : complex-definition ( caddress -> character unsigned
    complex-definition -- 4 th )
    definition ;
  : assign ( complex complex-definition -- )
    'value ! ;
  :noname ( compiler-workspace complex-definition -- )
    locals( this ) drop
    'value @ output-params drop @ (literal) ; is (compile)
endclass
```

Furthermore, we need overloaded words for creating variables and values of complex numbers. Again, the definitions of these words look very much alike to those for floating-point numbers:

```
: variables ( complex unsigned -- )
  data-space align [dt] address (variable) rot rot
  data-space here -> complex over 2* floats
  data-space allot swap rot fill
enddef ;

: sfvariables ( complex unsigned -- )
  data-space align [dt] sfaddress (variable) rot rot
  data-space sfhere -> complex over 2* sfloats
  data-space allot swap rot fill
enddef ;
```

```

: dfvariables ( complex unsigned -- )
  data-space align [dt] dfaddress (variable) rot rot
  data-space dfhere -> complex over 2* dfloats
  data-space allot swap rot fill
  enddef ;

: variable ( complex -- )
  1 variables ;

: sfvariable ( complex -- )
  1 sfvariables ;

: dfvariable ( complex -- )
  1 dfvariables ;

: value ( complex -- )
  (value) swap data-space , enddef ;

: to ( complex -- )
  parse-name ?value-definition
  dt-restore dup ?to dt-drop store ; execute-only

```

Within the definition of `to` for complex numbers, an overloaded version of `store` is required, which is defined as a method of class `value-definition`:

```
store ( complex value-definition -- )
```

And here are the defining words for class members:

```

: members ( object-size complex unsigned -- 1st )
  nip
  [ address-unit-bits floats 2* ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] address (member) ;

: dfmembers ( object-size complex unsigned -- 1st )
  nip
  [ address-unit-bits dfloats 2* ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] dfaddress (member) ;

: sfmembers ( object-size complex unsigned -- 1st )
  nip
  [ address-unit-bits sfloats 2* ] literal * swap
  [ address-unit-bits cells ] literal aligned
  [dt] sfaddress (member) ;

: member ( object-size complex -- 1st )
  1 members ;

: dfmember ( object-size complex -- 1st )
  1 dfmembers ;

: sfmember ( object-size complex -- 1st )
  1 sfmembers ;

```

Each complex number occupies 20 bytes in memory, or 5 cells in a 32-bit system like StrongForth. It therefore makes sense to align complex numbers to cell size. Note that this is different from the alignment of floating-point numbers, which occupy 10 bytes or 2.5 cells each:

```
: xalign ( memory-space -- )
  align ;

: xalign ( -- )
  align ; 1 retreat

: xaligned ( address -- 1st )
  aligned ;
```

Since complex numbers are distributed over a plane, they cannot be ordered in any meaningful sense. Comparison words for complex numbers are thus limited to 0=, 0<>, = and <>. The definitions of these words just separately compare the real part and the imaginary part:

```
: 0= ( complex -- flag )
  split 0= swap 0= and ;

: = ( complex 1st -- flag )
  split rot split rot = rot rot = and ;

: 0<> ( complex -- flag )
  split 0<> swap 0<> or ;

: <> ( complex 1st -- flag )
  split rot split rot <> rot rot <> or ;
```

To display complex numbers, their real part and their imaginary part are simply processed separately:

```
: . ( complex -- )
  split swap . ." + " . ." i " ;

: s. ( complex -- )
  split swap s. ." + " s. ." i " ;

: e. ( complex -- )
  split swap e. ." + " e. ." i " ;
```

Here's an example:

```
+3.4e-3 -2.7e-3 i + . 0.0034 + -0.0027 i ok
```

Now it's time to show some the operations that can be applied to complex numbers. Calculating the conjugate complex number is pretty simple, because it only means negating the imaginary part. The negative value of a complex number is calculated by negating both the real and the imaginary part:

```
: conj ( complex -- 1st )
  split negate merge ;

: negate ( complex -- 1st )
  split swap negate swap negate merge ;
```

The absolute value of a complex number is the distance from the zero point of the complex plane. The result is a scalar floating-point number:

```
: abs**2 ( complex -- float )
  split dup * swap dup * + ;

: abs ( complex -- float )
  abs**2 sqrt ;
```


The angle with respect to the real axis can be calculated using the *arcus tangent* of the quotient of the imaginary and the real part. However, a number of cases have to be considered:

```
: arg ( complex -- float )
  split over 0=
  if nip dup 0=
    if drop 0e0
    else 0>
      if [ pi 0.5e0 * ] literal
      else [ pi 1.5e0 * ] literal
      then
    then
  else over 0>
    if dup 0<
      if swap atan2 [ pi 2e0 * ] literal +
      else swap atan2
      then
    else swap / atan pi +
    then
  then ;
```

Addition and subtraction are defined straightforward. StrongForth additionally provides words that add or subtract a scalar value from the real part of a complex number:

```
: + ( complex float -- 1st )
  swap split >r + r> merge ;

: + ( complex complex -- 1st )
  split rot split \ re2 im2 re1 im1
  >r rot + r> rot + merge ;

: - ( complex float -- 1st )
  swap split >r swap - r> merge ;

: - ( complex complex -- 1st )
  split rot split \ re2 im2 re1 im1
  >r rot - r> rot - merge ;
```

Multiplication and division are somewhat more complex. Again, versions with a scalar number as the second operand are provided:

```
: * ( complex float -- 1st )
  >r split swap r@ * swap r> * merge ;

: * ( complex complex -- 1st )
  split locals( re2 im2 ) split locals( re1 im1 )
  re1 re2 * im1 im2 * -
  re1 im2 * im1 re2 * + merge ;

: / ( complex float -- 1st )
  >r split swap r@ / swap r> / merge ;

: / ( complex complex -- 1st )
  >r r@ conj * r> abs**2 / ;
```

Note that the definition of the complex multiplication uses complex locals. It is possible to define the multiplication without locals. However, locals are preferred because otherwise, the number of floating-point numbers on the hardware floating-point stack would have to be greater than 4 at certain times during the calculation. Since the hardware floating-point stack of an x86 system is only 8 numbers deep, requiring too many intermediate values could easily lead to a stack overflow. For the same reason, complex and floating-point numbers are temporarily stored on the return stack

in the above definitions, using `>r`. This is possible, because `(>r)` has been overloaded for complex numbers.

Other frequently used operations are multiplying complex numbers with the imaginary unit and with its negative value:

```
: i* ( complex -- 1st )
  split negate swap merge ;
: -i* ( complex -- 1st )
  split swap negate merge ;
```

`0i+` converts a scalar number into a complex number with the given real part and a zero imaginary part:

```
: 0i+ ( float -- complex )
  0e0 i*+ ;
```

And here are the definitions of complex logarithmic, exponential and trigonometric functions. Using overloaded operations, calculations with complex numbers are as easy and straightforward as calculating with single-cell integer numbers:

```
: ln ( complex -- 1st )
  dup abs ln swap split swap atan2 merge ;
: log ( complex -- 1st )
  ln [ 10e0 ln ] literal / ;
: exp ( complex -- 1st )
  split sincos rot exp tuck * rot rot * merge ;
: sqrt ( complex -- 1st )
  ln 0.5e0 * exp ;
: ** ( complex complex -- 1st )
  swap ln * exp ;
: alog ( complex -- 1st )
  [ 10e0 ln ] literal * exp ;
: sin ( complex -- 1st )
  i* dup exp swap negate exp - [ -i 0.5e0 * ] literal * ;
: cos ( complex -- 1st )
  i* dup exp swap negate exp + 0.5e0 * ;
: tan ( complex -- 1st )
  i* dup exp swap negate exp over over - rot rot + / i / ;
: sinh ( complex -- 1st )
  dup exp swap negate exp - 0.5e0 * ;
: cosh ( complex -- 1st )
  dup exp swap negate exp + 0.5e0 * ;
: tanh ( complex -- 1st )
  dup exp swap negate exp over over - rot rot + / ;
: asin ( complex -- 1st )
  dup i* swap dup * negate 1e0 + sqrt + ln -i* ;
: acos ( complex -- 1st )
  dup dup * negate 1e0 + sqrt i* + ln -i* ;
```

```

: atan ( complex -- 1st )
  i* dup 1e0 + swap negate 1e0 + / ln [ -i 0.5e0 * ] literal * ;

: asinh ( complex -- 1st )
  dup dup * 1e0 + sqrt + ln ;

: acosh ( complex -- 1st )
  dup dup * 1e0 - sqrt + ln ;

: atanh ( complex -- 1st )
  dup 1e0 + swap negate 1e0 + / ln 0.5e0 * ;

```

Finally, this is the definition of an overloaded version of `~` for complex numbers. It actually compares the absolute value of the difference of two complex numbers, which is a scalar number, with a given delta value:

```

: ~ ( complex 1st float -- flag )
  locals( x1 x2 delta ) x1 x2 delta 0=
  if =
  else - abs delta 0>
    if delta <
    else x1 abs x2 abs + delta abs * <
    then
  then ;

```