Object-Oriented Programming in ANSI-C

Dr. Fritz Solms, Solms Training & Consulting
`<Fritz@SolmsTraining.co.za>`

Object-Oriented Programming in ANSI-C

by Dr. Fritz Solms

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

History of *C*

C was developed in 1972 by Dennis Ritchie at Bell Labs as a language which was specifically designed to develop the *Unix* operating system. The language was very successful, being useful for both, low-level development and relatively high-level development - at least from the perspectives of the 1970s and 80s.

C was named *C* because it was based on a language *B* which was also developed by Bell Labs.

Soon various flavours of mutually incompatible *C* were used. The need to standardize was addressed by the *ANSI-C* standard was developed and published by the American National Standards Institute.

Why Use C?

C is a relatively old language, yet it has survived reasonably for more than 3 decades. It has also been the foundation for a range of other languages including *Pascal* and its derivatives, *C++* and hence *Java*. Today *C* is largely used for low-level development and for developing for the Unix-based platforms (e.g. Linux). There are several reasons why *C* as a procedural language has survived so well:

- **C is a powerful and flexible language:** *C* is used from writing code for very low-level and resource-constrained devices to games, graphics applications, spreadsheets and even compilers and run-time environments for other languages (e.g. Java). A range of god optimizing *C* compilers exist and one write very efficient *C* code which compiles to very efficient binary code. In this course we shall show that one can even program *C* in an object-oriented way.

- **C is reasonably portable:** The ANSI-*C* standard ensures that *C* ports relatively well across hardware platforms like microchips, limited devices like mobile phones and PDAs, personal computers and servers and main frames and across operating systems like Unix and its derivatives like Linux, Palm OS, Symbian OS and Windows. Unlike *Java* code, *C* code is however not fully portable, partially because the range and resolution of the basic data types are not part of the ANSI standard.

- **C is a compact language:** *C* has only few keywords and language constructs and learning the language is simple. Learning good *C* development practices and the ability of mapping complex object-oriented designs onto *C* requires, however, a lot of technical skills.

- **C is modular:** C is modular in that individual functions or groups of functions can be packaged individual source and compiled units and can be reused across applications.

Which C Compiler?

This course sticks to the ANSI-*C* standard and as such you can use any C compiler which supports this standard which includes most compilers (e.g. Borland, IBM, Microsoft, Zortec, ...).

The GNU *C* Compiler

We shall use the GNU C compiler which is an open-source C compiler distributed under GNU public licensing. This compiler is the de-facto standard for Linux and most other Unix-based systems, but is also widely used on other platforms like Microsoft Windows.

Part I. Procedural Programming in ANSI C

Abstract

C is a procedural programming language and in this part of the book we cover standard procedural programming in *C* sticking carefully to the ANSI-*C* standard.

Table of Contents

Chapter 2. C Fundamentals

Steps for developing a C/C++ application

When developing a *C/C++* application one typically performs the following steps:

- **Write the *C/C++* source code.** You can use any text editor of your choice but it should typically at least support line numbers. You may use something as low-level as *Vi* or *Vim Vi-(improved)* or something more resource intensive like emacs, Glimmer, Kate or Jedit up to a full blown IDE like kDevelop, Anjuta, IBM VisualAge, Borland C-Builder.
- **Compile and link the source code.** You can use any ANSI C/C++ compatible compiler. We shall use the GNU C/C++ compiler, **gcc**.

Your first C application

In this section we write a simple HelloWorld application in C, compile it and run it.

Example 2.1. helloWorld.c

```
           //
/* A simple Hello-World application in C.
    It only contains a single function, main. */

#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hi there, mate!!!\n");
  return 0;
}

//
```

We shall discuss some of the elements in more detail later but for now our application has the following elements:

- **A comment block.** Comment blocks start with a **/*** and ends with a ***/**. Note that some compilers support the **//** remainder-of-line comment introduced in *C++* and supported in languages like *Java*, but this is not part of the ANSI-*C* standard and should hence be avoided.
- **Importing the standard I/O library.** This is done via an **include** statement which includes the header files providing the compiler with the information of the functions exported by that library. We shall discuss header files in more detail later in the book.
- **The function main.** The first line in the **main** function is is the starting point of the application. The function which has to take an integer reporting the number of command-line parameters and a pointer to a character array for the actual command line parameters as arguments.

Compiling the application

To compile the application under *Linux* you simply type

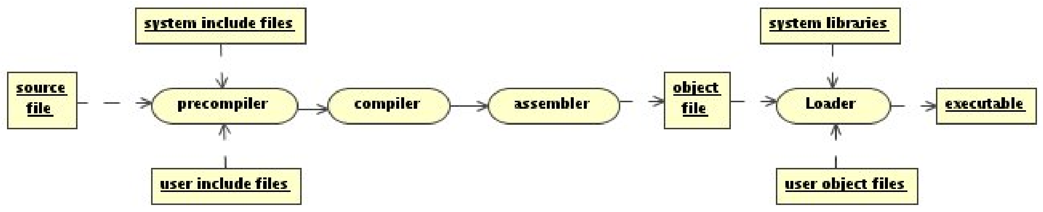
```
cc -ansi -Wall -o helloWorld helloWorld.c
```

which invokes the GNU-*C* compiler with the language option, **-x**, set to **c** specifying *C*source code, the option, **-ansi** requesting validation against the ANSI-C spec, the option **-Wall** which requests that all warning should be reported and the **-o helloWorld** option which requests that the compiled and linked application should be stored in the executable **helloWorld**. The final command-line argument is the source file itself.

A shorthand alternative is

```
cc -ansi -Wall -o helloWorld helloWorld.c
```

Figure 2.1. The steps of the compilation process.



By default the compiler will perform the following steps

- a *pre-processing step* which expands any macros, including the **#include** pre-compiler directives.
- a *compilation step* which typically includes at least two parses in order to resolve forward references and generates the mapping of the *C* code onto assembly language,
- an *assembly step* which translates the assembler code into machine code generating and object file and finally
- a *linkage step* which links the binary code fragments icluding system and user libraries and user object files into a single executable.

Running the application

To run the compiled application on *Linux* or *Unix*-based systems you simply type **./helloWorld** and to run the executable on *Windows* type in **helloWorld** followed by the **enter** key.

Variables

A variable is a data field of a certain type whose value can vary over time. A variable is thus a named storage location in memory which represents a certain size (number of bytes) and whose value is interpreted in a way determined by the data type of the variable.

Valid variable names

Variable names in *C* must start with a letter or underscore character and may contain any further number of letters, underscore characters or digits. Variable names are case-sensitive in *C*.

Primitive data types

C defines a range of primitive data types including characters, signed and unsigned integral data types and signed floating point data types. The primitive data types together with their range and resolutions are listed in table Table 2.1, “Primitive data types in C”.

Table 2.1. Primitive data types in *C*

Keyword	Data type	Size	Range
char	Single-byte character	1 byte	-128 to 127
int	Short integer	2 bytes	-32768 to 32767
short	Short integer	2 bytes	-32768 to 32767
long	Long integer	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned char	Unsigned character	1 byte	0 to 255
unsigned int	Unsigned short integer	2 bytes	0 to 65535
unsigned short	Unsigned short integer	2 bytes	0 to 65535
unsigned long	Unsigned long integer	4 bytes	0 to 4,294,967,295
float	IEEE single precision-floating point number	4 bytes	approx 3e+-38 with approx 7 digits resolution
double	IEEE double precision floating point number	8 bytes	approx 2e+-308 with approx 18 digits resolution
long double	10 bytes	3.4e+-4932 with approx 21 digits resolution	

Declaring and initializing variables

Variables are declared with a statement of the form **type variableName;**. Once can declare multiple variables of the same type within a single statement by comma-delimiting the variables. Example 2.2, “variableDeclarations.c” declares three variables, **i**, **k** and **l** in a single statement.

Example 2.2. variableDeclarations.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i, k, l=3;         /* Declares variables k & l and initializes l */
  {                      /* Start of nested block */
    //printf("i = %d" + i);
    k = l+1;             /* Initializes k */
    l = 2*i+k;           /* Error: using uninitialized variable, i
                            leads to incorrect results.*/
    int m = 2*l+k;       /* Declares & initializes variable.
                       local to block */
    printf("m = %d", m); /* Prints text & value of integer variable m. */
  }                      /* End of nested block. */
//  k = m;                  /* Illegal because m no longer exists. */

  return 0;
}
```

Variable initialization

Variables can be initialized within the declaration statement (like the variable **l** in our example) or later on in the code. Variables must be initialized before their value can be used.

Warning

Compilers may check your code for the possibility of using un-initialized variables but you may also simply get incorrect results.

The scope of a variable

Statements are grouped together in blocks of code by encapsulating them within curly brackets. Example 2.2, “variableDeclarations.c” contains one block for the group of statements for the function **main** and a second, nested block containing, among other elements, the declaration of the variable, **m**.

Variables can be declared where they are required (anywhere in the code). The life-span of a variable ranges from the point of declaration to the end of the block in which it is declared. Hence, the use of the variable **m** after the nested block in Example 2.2, “variableDeclarations.c” is illegal since the variable no longer exists.

Constants

A constant is a named value - it is like a variable whose value may not change. Constants can be declared in macros (we shall discuss macros in section the section called “Macros”. To declare a constant you simply insert the **const** keyword infront of a variable declaration:

const double pi = 3.1415926, radius=2.5;

const double area = pi*radius*radius;

Note that the **const** keyword specifies that all variables on that line are to be declared constants.

Specifying the data type of literal constants

When specifying a number like **2.232323** we do not specify a resolution and different resolutions (e.g. **float** and **double**. The different data types may provide different binary approximations to a specified decimal number.

C uses **int** as default integral data type and **double** as the default floating point data type. But *C* also enables you to specify the data type which should be used for the literal constants:

13L	long integer 234
O13	the octal integer 13 (decimal 11)
0x13	the hexadecimal (base 16) number 13 which corresponds to decimal 19
2.13e-3	The double-precision floating-point number 0.00213
2.13e-3F	The single-precision floating-point number 0.00213
2.13e-3L	The long double floating-point number 0.00213

Enumeration types

Enumeration types are really collections of named integer constants. Their main benefit is that they can, at times, make code more readable. Below we define an enumeration type months:

```
enum months {January=1, February, March, April, May, June, July,
         August, September, October, November, December};
```

Here each of January, ..., December is an integer constant. The first we set to one. If we do not specify an integer value for a particular named integer constant in an enumeration list, then that constant has as value one more than the previous member in the list. Hence, February, ..., December have values `2`, ... `12`. If we do not specify a value for the first named constant in the enumeration list, its value is assumed to be zero.

We can now define variables of the enumeration type as follows:

```
enum months month1, month2;
```

with the intention that the values of these two variables will be between `January`, ..., `December` or more accurately between `1`, ... `12`. This is, however not enforced. Both, `month1` and `month2` are really of type int and setting their value equal to, say, June is completely equivalent to setting the value equal to `6`.

Warning

Making use of enumeration types like `month1` and `month2` is no safer than using integers. Nothing prevents anybody from setting their value to `13` or even `-1`. Enumeration types can thus give a false sense of security and should be used with caution.

We can also declare an enumeration type and instance variables in a single statement as is done below:

```
enum months {January=1, February, March, April, May, June, July,
         August, September, October, November, December} month1, month2;
```

The core benefit of using enumerations is that the resultant code may be a little more readable. We can, for example, say

```
enum months month;
for (month=January; month<=December; ++month)
...
```

or

```
if (month == July)
...
```

The values of the variables in the enumeration list do not have to be consecutive integers. For example, we can define an enumeration of control characters (characters are bytes and can hence be represented by integers) as is done below

```
enum controlChars {bell='\a', backspace='\b', newline='\n', return='\r', tab='\
```

Operators

C defines a set of binary and unary operators similar to that found in most languages including arithmetic, assignment, logical, relational and bit operators.

Arithmetic operators

Arithmetic operators include **+**, **-**, *****, **/** and **%** for addition, subtration, multiplication, division and remainder (modulus) respectively.

Increment/Decrement operators

There are two increment and two decrement operators. The pre- and post-increment operators both increment the variable they operate on, but while the former, **++k** increments the variable and returns the result, the latter, **k++** increments the variable, but returns the original value of the variable.

Example 2.3. prePostIncrement.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int k = 3;

  printf("initial value of k: %d\n", k);

  int l = ++k;

  printf("l=++k;  ->  k=%d, l=%d\n", k, l);

  k = 3;
  l = k++;
  printf("l=k++;  ->  k=%d, l=%d\n", k, l);

  return 0;
}
```

The code in Example 2.3, “prePostIncrement.c” demonstrates the difference between pre- and stpoincrement. While the value of k is incremented in both cases, the value of l is equal to the final value of k in the case of using the increment operator in prefix mode, while it is equal to the original value of k when used in postfix mode. The output of the application is

```
initial value of k: 3
l=++k;  ->  k=4, l=4
l=k++;  ->  k=4, l=3
```

Assignment operators

The assignment operator is simply **=**. There are however a number of abbreviated operators (contractions) like **+=**, **-=**, ***=**, **/=** and **%=** which resemble the following abbreviations:

x += y	x = x + y
x -= y	x = x - y

x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Type-cast operator

C enables one to cast one type to another via the type-cast operator. Consider the code shown in Example 2.4, “testCast.c”. The first statement performs **2 / 3** which is the division of one integer by another. The operator used is the division operator for the integer data type which results in an answer of zero.

Example 2.4. testCast.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  double x = 2/3;
  printf("x = %f\n", x);

  double y = (double)2/3;
  printf("y = %f\n", y);

  int k=2,l=3;
  float result = k/(float)l;  /* explicit up-cast*/
  printf("%d/%d = %f\n", k, l, result);

  float kFloat = k;           /* implicit up-cast */
  result = kFloat/l;
  printf("%d/%d = %f\n", k, l, result);

  result = (int)result;       /* explicit down-cast */
  printf("After down-casting to an integer: %f\n", result);

  result = (float)2/3;
  int resultInt = result;
  printf("After down-casting to an integer: %f\n", resultInt);

  return 0;
}
```

Up-casting

If we want to perform floating-point division to the resolution of one of the floating point data types (e.g. float, double or long double), then we have to cast one of the operands to that data type. *C* will perform the calculation to the resolution of the longest data type.

So, in Example 2.4, “testCast.c” we show that casting either the first or the second operand to a floating point data type results in floating point division being used.

Down-casting

We can also down-cast a higher-resolution variable via the cast operator as is done in line 19 in Example 2.4, “testCast.c”. This will typically result in loss of information.

Warning

C will allow you to implicitly down-cast as is done in the last code block, but we strongly recommend that you refrain from doing this and most compilers will provide the settings which will issue compiler warnings in such cases.

Relational operators

The relational operators compare expression like **x <= y** and the result evaluates to either *true* (**1**) or *false* (**0**).

They include **==** for the logical *is equal to*, the standard >, <, >= and <= as well as the logical *is not equal to* represented in *C* by **!=**.

Logical operators

Logical operators evaluate logical statements and return either *true* (**1**) or *false* (**0**). They include **&&** for logical AND, || for logcal OR and **!** for logical NOT.

Logical operators are often used to assemble a logical expression from a combination of relational expressions like **(x & y) && (x != 4)** returns `1` (*true*) if `x` is less than `y` and the value of `x` is not equal to 4. Otherwise this expression returns `0`.

Important

The logical operators in *C* stretch the concept of truth a little by regarding any expression which results in a value 0 as *false* and any expression which results in a non-zero result as *true*. Thus, if `x` has a value `--2.5` then the expression **if (x) ...** will evaluate to *true* and **!x** will evaluate to *false*.

Binary operators

Binary operators perform operations which conceptually apply to the individual bits. In order to understand their functioning one needs to understand the binary representation of numbers in bits. Any integral number can be expanded in a series in powers of 2. For example, 10 is 1x2^3 + 0x2^2 + 1x2^1 + 0*2^0 and hence is represented by the binary number 1010.

Bitwise AND

The bitwise AND operator, **&**, sets a bit if and only if the corresponding bits of both operands is set. Table 2.2, “Bitwise AND” shows the truth table for the bitwise AND operator.

Table 2.2. Bitwise AND

&	**0**	**1**
0	0	0
1	0	1

Example 2.5. Bitwise AND

3 & 6 = 0000 0011 & 0000 0110 = 0000 0010 = 2

Bitwise OR

The bitwise OR operator, |, sets a bit if either one the corresponding bits of both operands is set. Table 2.3, “Bitwise OR” shows the truth table for the bitwise OR operator.

Table 2.3. Bitwise OR

|	**0**	**1**
0	0	1
1	1	1

Example 2.6. Bitwise OR

3 | 6 = 0000 0011 | 0000 0110 = 0000 0111 = 7

Bitwise XOR

The bitwise exclusive-or operator, ^, sets a bit if and only if the corresponding bit is set in only one of the operands. Table 2.4, “Bitwise XOR” shows the truth table for the bitwise XOR operator.

Table 2.4. Bitwise XOR

^	**0**	**1**
0	0	1
1	1	0

Example 2.7. Bitwise XOR

3 & 6 = 0000 0011 ^ 0000 0110 = 0000 0101 = 5

Bitwise NOT

The NOT operator, ~, is a unary operator which inverts the bits of its operands. Table 2.5, “Bitwise NOT” shows the truth table for the bitwise NOT operator.

Table 2.5. Bitwise NOT

~	
0	1
1	0

Example 2.8. Bitwise NOT

~6 = ~(0000 0110) = 1111 1001 = 249

Shit-left operator

The shift-left operator, << shifts each bit **n** binary digits to the left, padding with zeros from the rear.

Example 2.9. Shift-Left

6 << 2 = (0000 0110) << 2 = 0001 1000 = 24

Shit-right operator

The shift-right operator, >> shifts each bit **n** binary digits to the right, padding with zeros from the front.

Example 2.10. Shift-right

6 >> 2 = (0000 0110) >> 2 = 0000 0001 = 1

Precedence levels of operators

C preserves the presedence levels of the algebraic operators. For example,

```
1 - 8/2 + 2*2
```

yields `1`, i.e. multiplication and division take presedence over addition and subtraction. The presedence can be modified via brackets. For example,

```
1 - 8/(2 + 2)*2
```

yields `-3`. Table 2.6, "C Operators in decreasing presedence level. The precedence levels are separated by horizontal lines." shows the *C* operators in order of their presendence levels.

Table 2.6. C Operators in decreasing presedence level. The precedence levels are separated by horizontal lines.

Operator	Operator name	Example
()	function call	*functionName(expressionList)*
[]	array element access	*arrayName[elementNo]*
->	element access	*member-pointerToStructure->Name*
.	element access	*structureName.memberName*
!	logical NOT (true if non-zero)	*!expression*
~	bitwise NOT	*~expression*
++, --	pre- and post-increment/decrement	++*expression* or *expression*++
+, -	unary plus and minus	-*k*
*	dereference	**pointerVariable*
&	address-of	*&variableName*
sizeof	size-of memory area	*sizeof expression*
()	type-cast operator	*(type)expression*
*, /, %	multiply, divide, remainder (modulus)	*expression* * *expression*

Operator	**Operator name**	**Example**
<<, >>	shift-left and shift-right bit operators	*expression << expression*
<, <=, >, >=	relational operators	*expression <= expression*
==	relational equal	*expression == expression*
!=	relational not-equal	*expression != expression*
&, |	bitwise AND and OR	*expression | expression*
^	bitwise exclusive OR	*expression ^ expression*
&&	logical AND	*expression && expression*
||	logical OR	*expression || expression*
? :	conditional	*expression ? expression : expression*
=	assignment	*lvalue = expression*
+=, -=, *=, /=, %=	arithmetic-operator-and-assign	*lvalue += expression*
&=, |=, ^=, <<=, >>=	bitwise-operator-and-assign	*lvalue &= expression*
,	sequencing	*expression, expression*

Operator directionality

The directionality of most operators in *C* is left-to-right, i.e. that the left operand is evaluated prior to the right operand being evaluated. Exceptions are:

- The logical NOT operator **!**.
- The bitwise NOT operator **~**.
- The unary increment, **++**, decrement, **--**, and negative (unary minus), **-**, operators.
- The dereferencing, *****, and address-of, **&** operators.
- The type-cast operator, **()**.
- The assignment operators, **=**, **+=**, **-=**, *****=**, **/=**, **%=**, **&=**, **|=**, **<<=** and **>>=**.

Basic keyboard and screen input/output

In order to be able to develop some sensible applications we need to be able to write to the screen and read from the keyboard. This is typically done with the **printf** and **scanf** commands respectively.

Writing to the screen via printf

The arguments to **printf** are

- **A formatting string.** The formatting string contains
  - literal text,
  - conversion specifiers for the variables whose values have to be converted to text and inserted into the literal string at the position of the conversion specifier, and
  - control characters like the end-of-line character, **\n**.

- **An argument list of variables.** The values of the variables in the arguments list will be converted to text using the corresponding conversion specifiers and the resultant text is inserted into the formatting string at the positions of the conversion specifiers.

Conversion specifiers

Table 2.7, “Text conversion specifiers.”shows the conversion specifiers available in *C*.

Table 2.7. Text conversion specifiers.

Specifier	Description	Data types converted
%c	single character	char
%i, %d	signed decimal integer	int
%u	unsigned decimal integer	unsigned int
%o	octal number	unsigned int
%x, %X	hexadecimal number	unsigned int
%f, %m.nf [a]	decimal floating point number	float,double
%lf, %m.nlf [b]	double-precision floating point number	double
%e, %E	scientific notation	float, double
g, G	scientific/floating point [c]	float, double
%%	percent	simply print a percentage

[a] The format **%m.nf** is used to specify the field width, m and the number of decimal digits behind the decimal point, n. If a minus signed is inserted as in **%-7.3f**, then the converted text is left-justified.
[b] Same as **%f** except that this is for double.
[c] Selects the more compact between %f and %e.

Example 2.11, “conversionSpecifiers.c” illustrates the use of conversion specifiers for both, output to stdout (typically the screen) and input from stdin (typically the keyboard).

Example 2.11. conversionSpecifiers.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int score1, score2;

  printf("Enter score for team 1:");
  scanf("%d", &score1);

  printf("Enter score for team 2:");
  scanf("%d", &score2);

  float temp;
  printf("Enter current temperature:");
  scanf("%f", &temp);
  printf("temp:%f\n", temp);

  printf("The score is %d:%d and the temperature is %10.2f degrees celsius.",
         score1, score2, temp);
  return 0;
}
```

Control characters

We have used the *end-of-line* character, **/n** already. The ASCII data set specifies a range of control characters. Some of the most frequently used control characters are shown in Table 2.8, “Some commonly used control characters.”.

Table 2.8. Some commonly used control characters.

Character	Description
\b	backspace
\n	new-line
\t	horizontal tab
\\	backslash
\?	question mark
\'	single quote

Reading from the keyboard via scanf

The **scanf** function extracts variables from the standard input stream and puts the bytes resembling the conversion into an address for a variable which should typically be of the data type specified in the conversion.

In Example 2.11, “conversionSpecifiers.c” we read the two scores into the addresses ofr the corresponding variables, score1 and score2. We use the interger conversion specifier, **%d** to convert the text representation to the binary representation and the *address-of* operator, **&** to specifiy the target addresses of the memory used by the variables score1 and score2.

Conditional program flow

C provides 2 constructs for conditional program flow, **if** statements and **switch** statements.

The if statement

The **if** statement takes an expression as argument. If that expression evaluates to `0` (`false`) then the corresponding block of code is not executed and potentially an alternative block of code specified in an **else** clause is executed.

Any non-zero value is, however, interpreted as `true`. Hence, if the expression in the **if** clause evaluates to anything else but `0`, the associated block of code is run.

Example 2.12. ifStatement.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int k;

  printf("Enter k: ");
  scanf("%d", &k);

  if (k == 3)
```

```
    {
      printf("k is equal to 3.\n");
      ++k;
    }
  else
    printf("k is not equal to 3.\n");

  printf("Value of k after if: %d\n", k);

  double x = -0.2;

  if (x)
    printf("if clause evaluated to true, clause value = %f\n", x);

  x = 12.3;
  printf("%8.4f\n", x);

  return 0;
}
```

Consider Example 2.12, “ifStatement.c”. Here we first use a relational operator which will evaluate to a boolean (0 or 1) for our first **if** clause. That **if** clause has a **else** clause containing a statement which is executed if the **if** clause evaluates to `0` (false).

Note that we only need to use curly brackets if a group of statements which have to be executed conditionally. For a single conditional statement we can ommit the curly brackets.

The conditional operator

C also supports a ternary conditional operator, **? :**. Its usage is illustrated in Example 2.13, condi-“tionalOperator.c”. If the expression in the left-most operand evaluates to `true` (non-zero), then the values of the second expression (the one before the colon). Otherwise the third expression is returned. In our example we assign max to the return value of the conditional operator.

Example 2.13. conditionalOperator.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int a, b, max;
  printf("Enter a b: ");
  scanf("%d %d", &a, &b);

  max = (a>b) ? a : b;

  printf("max = %d\n", max);

  return 0;
}
```

The switch statement

The **switch** statement is like the *case* statement found in some other languages. It takes an expression of argument and has different labels for different constant values. Consider Example 2.14, “switchStatement.c”. Here we jump to statements with different labels depending on the value of the expression in the **switch** statement.

Warning

You need to put a **break** statement after each conditional group statements in the **switch** statement if you don't want the con secutive statements for the other clauses to be executed as well, i.e. a **switch** statement is really a glorified **goto** statement.

Example 2.14. switchStatement.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int day, month, year;

  printf("Enter date: (dd mm yyyy) ");
  scanf("%d %d %d", &day, &month, &year);

  printf("%d ", day);
  switch (month)
  {
    case 1: printf("January"); break;
    case 2: printf("February"); break;
    case 3: printf("March"); break;
    case 4: printf("April"); break;
    case 5: printf("May"); break;
    case 6: printf("June"); break;
    case 7: printf("July"); break;
    case 8: printf("August"); break;
    case 9: printf("September"); break;
    case 10: printf("October"); break;
    case 11: printf("November"); break;
    case 12: printf("December"); break;
    default: printf("Invalid month.\n");
  }
  printf(" %d\n", year);
  return 0;
}
```

Iterating over blocks of code

C supports 3 types of explicit looping constructs, **for** statements, **while** loops and **do while** loops.

The for loop

The **for** loop has the following syntax

```
        for (initializationExpression(s); testExpression; lastStatement(s))
          statement or blockOfStatements;
```

The elements are

- **An initialization expression.** The initialization expression is evaluated once prior to the first iteration through the loop. One can initialize multiple comma separated variables in the initialization expression. In Example 2.15, “forLoop.c” we initializes the variables i, j and k in the initialization expression.

- **A test expression.** The test expression is evaluated once at the beginning of each iteration. If it evaluates to true (non-zero) the bock of code in the for loop is executed once again. If it evaluates to false, execution procedes with the first statement after the for loop.
- **Statement(s) to be inserted at the end of the loop.** This can be one or more comma-separated statements which will be the last statements in the iteration.

Example 2.15. forLoop.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i,j,k;

  for (i=0,j=0,k=3; ((i<=k) && (k!=j)); ++i, k-=j, printf("%d\n", j))
  {
    j += i%2;
    printf("%d,%d,%d\n", i, j, k);
  }
  return 0;
}
```

The output of this application is

```
0,0,3
0
1,1,3
1
2,1,2
1
```

The while loop

A **while** loop is similar to a **for** loop in that the test is done before each iteration (a **for** loop can be seen as a special form of a **while** loop. The syntax is simply

```
while (testExpression;)
statement or blockOfStatements;
```

If you want to perform initializations, the initialization statement(s) have to be inserted before the loop. The while-loop equivalent of Example 2.15, “forLoop.c” is shown in Example 2.16, “whileLoop.c”.

Example 2.16. whileLoop.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i,j,k;

  i=0;j=0;k=3;
  while (((i<=k) && (k!=j)))
  {
    j += i%2;
    printf("%d,%d,%d\n", i, j, k);
    ++i;
    k-=j;
    printf("%d\n", j);
  }
  return 0;
}
```

The output of this application is the same as for Example 2.15, “forLoop.c”.

The do-while loop

A **do-while** loop is the same as a **while** except that the test is done at the end of each iteration and not at the start of the iteration. The consequence of this is that the block of statements in the **do-while** loop is executed at least once, even if the test criterion is never met. Example 2.17, “doWhileLoop.c” illustrates the difference between the **while** and **do-while** statements.

Example 2.17. doWhileLoop.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int n0, n;
  printf("Enter initial loop index: ");
  scanf("%d", &n0);
  printf("Enter final loop index: ");
  scanf("%d", &n);

  printf("while loop:\n");
  int i=n0;
  while (i<=n)
  {
    printf("%d", i);
    ++i;
  }
  printf("\n");

  printf("do-while loop:\n");
  i=n0;
  do
  {
    printf("%d", i);
    ++i;
  } while (i<=n);
  printf("\n");

  return 0;
}
```

The application has a **while** and **do-while** loop over the same loop variables. Running the application with `2` and `5` as initial and final loop indeces yields the same results for both constructs:

```
Enter initial loop index: 2
Enter final loop index: 5
while loop:
2345
do-while loop:
2345
```

On the other hand, switching the loop variables around such that the test is never satisfied results in the following output which illustrates that the **do-while** loop is executed at least once:

```
Enter initial loop index: 5
Enter final loop index: 2
while loop:

do-while loop:
5
```

break and continue

C provides

- a statement for breaking out of the current loop, the **break** statement, and
- a statement for abandoning the remainder of the loop and jumping back to the start of the first statement in the loop, the **continue** statement.

The goto statement

C enables one to label statements and to request that the execution thread should jump to the the line with the specified label. This can be done using a **goto** statement. Example 2.18, “goto.c” demonstrates how the while loop of Example 2.16, “whileLoop.c” can be constructed using a **goto** statement:

Example 2.18. goto.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i,j,k;

  i=0;j=0;k=3;
  a:if (((i<=k) && (k!=j)))
  {
    j += i%2;
    printf("%d,%d,%d\n", i, j, k);
    ++i;
    k-=j;
    printf("%d\n", j);
    goto a;
  }
  return 0;
}
```

Warning

There is no need to ever use a **goto** statement and using it often results in so-called spaghetti code.

Functions and Procedures

Conceptually we have *functions* which perform operations which result in a return value (e.g. the caluclation of a result) and *procedures* which perform a sequence of operations but do not return a result. In *C* both are represented by functions, the former by a function with a return value and the latter by a function without a return value (returning **void**).

Functions

A function

- has a name,
- receives a number of arguments and
- has a return value of a specified type.

```
returnType functionName(arg1Type, arg1, ..., argNType, argN)
{
  functionBodyStatements;
  returnType x = ...;
  return x;
}
```

The variable (or constant) returned must be of the return type of the function.

Example 2.19. power.c

```
#include <stdio.h>

float power(float a, int k)
{
  float result = 1;  int i;

  if (k>0)
    for (i=0; i<k; ++i)
      result *= a;
  else
    for (i=k; i<0; ++i)
      result /= a;

  return result;
}

int main(int argc, char *argv[])
{
  float x;
  printf("x = "); scanf("%f", &x);

  int n;
  printf("n = "); scanf("%d", &n);

  float xPn = power(x, n);

  printf("x^n = %f\n", xPn);

  return 0;
}
```

Procedures

A procedure has no return value. This is specified in *C* by making the return type **void** and omitting a return statement.

Example 2.20. procedure.c

```
#include <stdio.h>

void printAverage(float a, float b)
{
  printf("The average is %f\n", (a+b)/2);
}

int main(int argc, char *argv[])
{
  float a, b;
  printf("Enter a b: ");
  scanf("%f %f", &a, &b);

  printAverage(a, b);

  return 0;
}
```

How are function arguments passed?

Function arguments are always copied from the calling code to the function (i.e. passed by value). Modifying the arguments in the function body will thus not affect the source arguments as is demonstrated in Example 2.21, “passByValue.c”.

Example 2.21. passByValue.c

```
#include <stdio.h>

void f(int a, int b)
{
  printf("a, b at start of f = %d,%d\n", a, b);
  a = 2;
  b = 3;
  printf("a, b at end of f = %d,%d\n", a, b);
}

int main(int argc, char *argv[])
{
  int a=1, b=1;
  printf("a, b in main before calling f = %d,%d\n", a, b);
  f(a,b);
  printf("a, b in main after calling f = %d,%d\n", a, b);
  return 0;
}
```

The output of the program is

```
a, b in main before calling f = 1,1
a, b at start of f = 1,1
a, b at end of f = 2,3
a, b in main after calling f = 1,1
```

After having discussed pointers we shall see how variables can be effectively passed by reference.

Static variables

Local variables of a function (including function arguments) exist for the duration. Their values are not preserved accross function calls. If you need a variable whose value is maintained accross function calls, you declare it **static**. Example 2.22, “staticVariables.c” demonstrates the difference between statis and non-static variables.

Example 2.22. staticVariables.c

The function **testStatic** has a static and a non-static integer variable. Both are initialized to zero and incremented within the function.

```
#include <stdio.h>

void testStatic()
{
  int i=0;
  static int j=0;
  printf("i, j = %d, %d\n", i, j);
  ++i;
  ++j;
}

int main(int argc, char *argv[])
{
  int k;
  for (k=0; k<10; ++k)
    testStatic();

  return 0;
}
```

The static variable is initialized only once, the first time the function is called, and its value is preserved accross service requests. The output of the proram is listed below:

```
i, j = 0, 0
i, j = 0, 1
i, j = 0, 2
i, j = 0, 3
i, j = 0, 4
```

```
i, j = 0, 5
i, j = 0, 6
i, j = 0, 7
i, j = 0, 8
i, j = 0, 9
```

Recursion

If a function calls itself directly or indirectly, it is called recursion. Recursive algorithms can be very compact and conceptually simple and clean. Typical examples include certain mathematical functions like the factorial function discussed in Example 2.23, "factorial.c" and tree processing algorithms (e.g. directory trees or XML DOM trees).

Example 2.23. factorial.c

The factorial function is defined by

```
n! = n x (n-1) x (n-2) x ... x 2 x 1
```

with `0!` defined as 1.

```
#include <stdio.h>

long factorial(long n)
{
  if (n < 0) return -1;  /* error code */

  if (n == 0) return 1;

  return n * factorial(n-1);  /* recursion (function calls itself) */
}

long factorial2(long n)
{
  if (n < 0) return -1;  /* error code */

  long product = 1, i;

  for (i=n; i>0; --i)
    product *= i;

  return product;
}

int main(int argc, char *argv[])
{
  long n;
  printf("Enter n: "); scanf("%ld", &n);

  printf("n! = %ld\n", factorial(n));

  return 0;
}
```

Warning

One should, however, be aware that recursive algorithms may put significant strain on the stack (which is a limited resource) and may incur significant processing overheads in the construction and unraveling of the stack frames. For each level of recursion a new stack frame must be constructed keeping track of the local variable and the function pointer for that level.

Recursive algorithms can always be mapped onto non-recursive algorithms and in cases where there are potentially many levels of recursion, they should typically be avoided.

Function prototyping

For the compiler to resolve the function calls, the function must be known when it is used. That is why we declared the **factorial** function, for example, before **main** where we call **factorial**.

This may not always be possible. For example, if two functions call each other we need a mechanism to inform the compiler of a function which is defined lower in the listing.

Prototyping enables one to publish the function interface (header) separately from the implementation. This mechanism will also be useful when we seperate the interface and implementation into different files (header and c-implementation files) which will enable us to distribute compiled libraries and to speed up the compilation of large applications.

Example 2.24. prototyping.c

Below we define a prototype for the **factorial** function separately from its implementation which is lower down in the code:

```
#include <stdio.h>

long factorial(long n);

int main(int argc, char *argv[])
{
  long n;
  printf("Enter n: "); scanf("%ld", &n);

  printf("n! = %ld\n", factorial(n));

  return 0;
}

long factorial(long n)
{
  if (n < 0) return -1;  /* error code */

  if (n == 0) return 1;

  return n * factorial(n-1);  /* recursion (function calls itself) */
}
```

Exercises

1. Write a program which reads in data values and reports the running average accross the data values read in so far.
2. Write a function, **even** which takes an integer as argument and returns 1 (true) if the integer is

even and 0 (false) otherwise. Write a **main** routine which tests this function.

3. Consider integers **k=3**, **l=13** and **m=128** and calculate (by hand) the following binary expressions:

```
(k << 2) & l
((k >> 1) ^ l) | m
```

4. Modify exercise 2 such that you have a a function, **float addToRunningAverage(int newDataPoint)** which you call repetitatively and which keeps track and reports the running average.

5. Write a function which takes as arguments a **char** representing a byte and an integer representing the bit number. The function should return 1 if the specified bit number is set in the supplied byte and false otherwise.Write a **main** routine which tests this function.

6. Write a recursive implementation of the power function discussed in Example 2.19, “power.c”.

Chapter 3. Compound Data Types

Introduction

Though *C* does not explicitly support user-defined data types (classes), it does provide the facility to assemble ones own data types from primitive data types. The two mechanisms for this are arrays (including character arrays used for strings) and structures.

Arrays

An array is a contiguous memory space for a collection of data fields of a specified type. The array occupies a block of **arrayLength x sizeof(dataType)** bytes. The n'th array element of an array, **vec**, is retrieved by extracting **sizeof(dataType)** bytes from memory at memory position **&vec + (n-1) * sizeOf(dataType)**.

Single-dimensional arrays

The syntax for declaring a single dimensional array is

```
dataType arrayName[numberOfElements];
```

The first array element is accessed as **arrayName[0]**. In general, the i'th array element is accessed as **arrayName[i-1]**.

Note

The element access operator, **[]**, is used for both,

- array element retrieval and
- array element assignment,

i.e., for both read and write access to array elements. For example,

```
array[2] = array[3];
```

assigns the value of the third array element to that of the fourth.

Warning

Array element indeces start at 0. The last array element is **arrayName[length-1]**

We can declare and initialize arrays in a single statement via

```
double vec[4] = {1.1, 2.2, 3.3, 4.4};
```

Example 3.1. vectorNorm.c

In this example we read in a vector from the keyboard and then call a function which returns the norm of the vector. We also appended two statements to **main** which demonstrate that *C* does not do run-time bounds checking.

```
#include <math.h>
#include <stdio.h>

float norm(float vec[], int length);

int main(int argc, char *argv[])
{
  int length;
  printf("vector length = "); scanf("%d", &length);

  float vec[length];

  int i;
  for (i=0; i<length; ++i)
  {
    printf("vec[%d] = ", i);
    scanf("%E", &vec[i]);
  }

  float vecNorm = norm(vec, length);

  printf("The norm of the vector is %f\n", vecNorm);

  printf("out-of-bound array elements: %f, %f\n", vec[length], vec[length+1]);

  vec[length+1] =12.345;

  return 0;
}

float norm(float vec[], int length)
{
  float sum = 0;
  int i;
  for (i=0; i<length; ++i)
    sum += vec[i]*vec[i];
  return sqrt(sum);
}
```

To run the application, we have to include the **-lm** command line argument, i.e. we run it via

```
gcc -x c -lm -Wall -o vectorNorm vectorNorm.c
```

The command-line argument **-lm** specifies that the math library should be included during the linking (loading) phase.

C is not a "safe" language like Java

Note that in the last **printf** we access array elements which do not exist, i.e. we read from and in the following statement we even write to memory which is not used by the vector - who knows what we are overwriting? Note that the target addresses are simply calculated via the formula given in the section called “Single-dimensional arrays”.

Multi-dimensional arrays

Multidimensional arrays are still stored under the hood as one long contiguous memory block and the elements of the n'th row are simply concatenated after the elements of the row (n-1). We can go to as many dimensions as we like. For example

```
double tensor[5][10][5];
```

declares a 3-dimensional array. The array elements are retrieved via **tensor[i][j][k]** with **tensor[0][0][0]** being the first and **tensor[4][9][4]** being the last element.

Example 3.2, “matrixProduct.c” demonstrates two-dimensional arrays via a matrix multiplication program.

Example 3.2. matrixProduct.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* Read in dimensions of first matrix. */
  int nRows1, nCols1;
  printf("mat1: no of rows, columns = ");
  scanf("%d %d", &nRows1, &nCols1);
  double mat1[nRows1][nCols1];

  /* Read in first matrix */
  int nr, nc;
  for (nr=0; nr<nRows1; ++nr)
    for (nc=0; nc<nCols1; ++nc)
      {
        printf("mat[%d][%d] = ", nr, nc);
        scanf("%lf", &mat1[nr][nc]);
      }

  /* Read in dimensions of second matrix. */
  int nCols2, nRows2;
  printf("mat2: no of rows, columns = ");
  scanf("%d %d", &nRows2, &nCols2);
  double mat2[nRows2][nCols2];

  /* Read in second matrix. */
  for (nr=0; nr<nRows2; ++nr)
    for (nc=0; nc<nCols2; ++nc)
      {
        printf("mat2[%d][%d] = ", nr, nc);
        scanf("%lf", &mat2[nr][nc]);
      }

  /* Check the dimensions. */
  if (nCols1 != nRows2)
  {
    printf("*** ERROR ***: illegal matrix dimensions in matrixProduct.");
    return -1;
  }

  /* declare result matrix of correct size. */
  double result[nRows1][nCols2];

  /* Calculate matrix product. */
  for (nr=0; nr<nRows1; ++nr)
  {
    for (nc=0; nc<nCols2; ++nc)
    {
      double sum = 0;
      int n;
      for (n=0; n<nCols1; ++n)
        sum += mat1[nr][n] * mat2[n][nc];
      result[nr][nc] = sum;
    }
  }

  printf("\nmatrix product:\n");
  for (nr=0; nr<nRows1; ++nr)
  {
    for (nc=0; nc<nCols2; ++nc)
      printf("%10.4lf ", result[nr][nc]);
    printf("\n");
  }
  return 0;
}
```

An example output of the application is shown below:

```
mat1: no of rows, columns = 2 3
mat[0][0] = 1
mat[0][1] = 2
mat[0][2] = 3
mat[1][0] = 2
mat[1][1] = 3
mat[1][2] = 4
mat2: no of rows, columns = 3 3
mat2[0][0] = 1
mat2[0][1] = 2
mat2[0][2] = 2
mat2[1][0] = 3
mat2[1][1] = 2
mat2[1][2] = 3
mat2[2][0] = 4
mat2[2][1] = 3
mat2[2][2] = 5

matrix product:
19.0000    15.0000    23.0000
27.0000    22.0000    33.0000
```

Note

We have quite some code duplication and we have also not localized the responsibilities in separate function. Furthermore, all arrays are created on the stack. The stack is a on many operating systems a very limited resource and to create arrays, particularly large arrays, on the stack is not a good idea. Once we have covered pointers we shall show how arrays are created on the heap and how we can write an application with re-usable functions.

Declaring and initializing multi-dimensional arrays

As with single-dimensional arrays, we can also declare and initialize multi-dimensional arrays in a single statement. Since an array is simply a contiguous memory bock (irrespective of whether it is single or multi-dimensional) we could simply initialize that memory block in the same way we did earlier. For example

```
double mat[2][2] = {1, 2, 3, 4};
```

initializes the first row with elements 1 and 2 and the second row with elements 3 and 4.

C provides, however, a much more intuitive notation for initializing multidimensional arrays by encapsulating the individual rows within curly brackets:

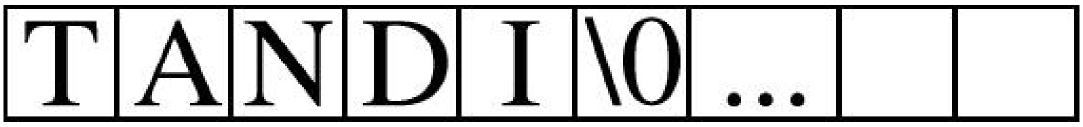
```
double mat[2][2] = {{1, 2}, {3, 4}};
```

Strings

C does not support strings directly. Instead a string is modelled as a null-terminated character array. *C* does provide a large number of string handling routines.

A character array holding a string must be at least one character longer than the string for it to be able to accommodate the null-terminating character.

Figure 3.1. String are represented as null-terminated character arrays.



Initializing a string

A string (character array) can be initialized just like any other array. For example, below we initialize a 20 element character array with a 5 character string which requires 6 characters storage space:

```
char name[20] = {'T', 'A', 'N', 'D', 'I', '\0'};
```

However, *C* also supports a more convenient form of string initialization:

```
char name[20] = "TANDI";
```

Warning

Be careful to use single quotes for characters and double-quotes for strings. Also, if you use the first form, do not forget to append the terminating null character and if you use the latter, keep in mind that you must reserve space for the terminating null character which will be appended automatically, i.e. the minimum size is

```
char name[6] = "TANDI";
```

Some commonly used string manipulation routines

In the appendix we list the contents of the ANSI-C standard library. The library, `stdlib.h`, contains a wide range of string manipulations routines. Some of the most commonly used string functions include

- `strlen` returns the length of a string.
- `strcpy` copies one string into another or alternatively a number of characters from one string into another.
- `strcmp` compares 2 strings and determines which comes first according to alphabetical order.

- `strncat` appends one string onto another or alternatively a number of characters from one string to another.
- `atoi`, `atol`, `atof` convert a string to an integer, a long and a double-precision floating point number respectively.

C provides many more string manipulation functions not shown here.

Command-line arguments

Recall that `main` took an integer and a array of strings as arguments. The first is the number of command-line arguments (including the command which launched the application) and the latter is the arguments itself. Example 3.3, “toPolar.c” lists an application which takes Cartesian coordinates as command-line arguments and calculates the polar coordinates:

Example 3.3. toPolar.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

const double pi = 3.1415926;

int main(int argc, char *argv[])
{
  if (argc != 3)
  {
    printf("Usage: toPolar xCoordinate yCoordinate\n");
    return -1;
  }

  double x = atof(argv[1]);  /* note: argv[0] is the command which */
  double y = atof(argv[2]);  /* launched the application.          */

  double radius = sqrt(x*x+y*y);

  double theta = asin(x/radius);

  double thetaInDegrees = theta * 180 / pi;

  printf("polar coordinates: radius=%f, theta=%f (%f degrees)\n",
         radius, theta, thetaInDegrees);

  return 0;
}
```

Structures

Structures are *C*'s version of records. They assemble a range of data fields into a single data structure which can be parsed along as a unity. In many ways a structure can be seen as a class without methods.

Declaring structures

The syntax for declaring a structure is

```
struct StructureName
```

```
{
field1Type field1;
...
fieldNType fieldN;
};
```

For example, we can declare a point as

```
struct Point
{
double x, y;
};
```

Using structures

We can declare a variable of the structure type via

```
struct Point p1;
```

and we can use the member access operator, **.**, to get both, read and write access to the structure fields, e.g.

```
p1.x = 2*p1.y;
```

Example 3.4. parabolaStruct.c

Below we define two structures, a structure encapsulating the data fields for a two-dimensional point and a structure encapsulating the parabola coefficients. The `calcTurningPoint` gets a `Parabola` structure as argument, calculates the turning point and returns it as a `Point`.

```
#include <stdio.h>
#include <stdlib.h>

struct Parabola
{
  double a, b, c;
};

struct Point
{
  double x, y;
};

struct Point calcTurningPoint(struct Parabola p)
{
  struct Point turningPoint;
  turningPoint.x = -p.b/(2*p.a);
  turningPoint.y = p.a*turningPoint.x*turningPoint.x + p.b*turningPoint.x + p.c
  return turningPoint;
}

int main(int argc, char *argv[])
{
  struct Parabola parabola = {1,0,0};
  if (argc == 4)
    {
      parabola.a = atof(argv[1]);
      parabola.b = atof(argv[2]);
      parabola.c = atof(argv[3]);
    }
  else
    {
      printf("Enter parabola coefficients, a b c: ");
      scanf("%lf %lf %lf", &parabola.a, &parabola.b, &parabola.c);
    }

  struct Point turningPoint = calcTurningPoint(parabola);

  printf("turning point of %fx^2 + %fx + %f = (%f,%f)\n",
         parabola.a, parabola.b, parabola.c,
         turningPoint.x, turningPoint.y);

  return 0;
}
```

Using typedefs with structures

The listing in Example 3.4, “parabolaStruct.c” is in many ways a little cumbersome because we had to drag the keyword **struct** along with the structure name.

C does, however, support a **typedef** statement which enables one to declare the structure a new data type and use the data type directly without dragging the **struct** keyword along. The syntax is

```
struct structureName
{
field1Type field1;
...
fieldNType fieldN;
} typeName;
```

We use this notation in Example 3.5, “parabolaStruct.c” making the code significantly more readable than Example 3.4, “parabolaStruct.c”.

Example 3.5. parabolaStruct.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct ParabolaStruct
{
  double a, b, c;
} Parabola;

typedef struct PointStruct
{
  double x, y;
} Point;

Point calcTurningPoint(Parabola p)
{
  Point turningPoint;
  turningPoint.x = -p.b/(2*p.a);
  turningPoint.y = p.a*turningPoint.x*turningPoint.x + p.b*turningPoint.x + p.c
  return turningPoint;
}
```

```
int main(int argc, char *argv[])
{
  Parabola parabola = {0,0,0};
  if (argc == 4)
    {
      parabola.a = atof(argv[1]);
      parabola.b = atof(argv[2]);
      parabola.c = atof(argv[3]);
    }
  else
    {
      printf("Enter parabola coefficients, a b c: ");
      scanf("%lf %lf %lf", &parabola.a, &parabola.b, &parabola.c);
    }

  Point turningPoint = calcTurningPoint(parabola);

  printf("turning point of %fx^2 + %fx + %f = (%f,%f)\n",
         parabola.a, parabola.b, parabola.c,
         turningPoint.x, turningPoint.y);

  return 0;
}
```

Initializing structures

Structures can be initialized when they are declared. The syntax is similar to that of array initialization:

```
Point p1 = {0,0};
```

We use this notation in Example 3.5, “parabolaStruct.c” to initialize `parabola` in the first line of `main`.

Structures containing structures

Since structures assemble a new data type from any existing data types, one can assemble structures from structures. For example, we could define an `Address` structure as

```
typedef struct AddressStruct
{
char[5] streetNumber;
char[30] streetName, suburbName;
unsigned int postalCode;
} Address;
```

and then a `Contact` structure containing elements of `Address` as fields:

```
typedef struct ContactStruct
{
char[30] surname;
char[50] firstNames;
char[12] workTelNo, homeTelNo;
Address workAddress, homeAddress;
};
```

To extract the postal code of the work address we simply state

```
Contact someContact;
...
someContact.workAddress.postalCode;
```

Arrays and structures

It is common to have arrays of structures. For example,

```
typedef struct PointStruct
{
double x, y;
} Point;

typedef Point experimentResults[100];
```

defines `experimentResults` as an array of 100 points.

Similarly, we can define structures which contain arrays:

```
typedef struct PointStruct
{
double x, y;
} Point;

typedef struct DateTimeStruct
{
int year, month, day, hour, minute, second;
} DateTime;

typedef ExperimentStruct
{
DateTime dateTime;
Point readings[100];
Contact supervisor;
} Experiment;

Experiment anExperiment;
```

and we can access elements like

```
anExperiment.dateTime.year,

anExperiment.readings[23],

anExperiment.supervisor.address.postalCode.
```

Unions

A union is like a polymorphic structure which may hold different elements of different types at different times. In Example 3.6, “scalar.c”we specify a polymorphic type, Scalar which may be a short, an int, a long, a float, a double, or even a Complex or a Rational number.

Example 3.6. scalar.c

The listing below shows an example of a polymorphic Scalar data type which may hold either a short, an int, a long, a float, a double, a Complex structure or a Rational structure:

```
#include <stdio.h>

typedef struct RationalStruct
{
  long numerator, denominator;
} Rational;

typedef struct ComplexStruct
{
  double real, imaginary;
} Complex;

typedef union ScalarStruct
{
  short shortNumber;
  int intNumber;
  long longNumber;
  float floatNumber;
  double doubleNumber;
  Rational rationalNumber;
  Complex complexNumber;
} Scalar;

int main(int argc, char *argv[])
{
  Scalar scalar;
  printf("sizeof(scalar) = %d bytes\n", sizeof(scalar));

  printf("initial values:\n");
  printf("scalar.intNumber = %d\n", scalar.intNumber);
  printf("scalar.rationalNumber = %ld/%ld\n",
         scalar.rationalNumber.numerator,
         scalar.rationalNumber.denominator);
  printf("scalar.complexNumber = %f + i%f\n",
         scalar.complexNumber.real,
         scalar.complexNumber.imaginary);

  scalar.rationalNumber.numerator = 2;
  scalar.rationalNumber.denominator = 3;

  printf("After initializing with rational number (fraction) 2/3:\n");
  printf("scalar.intNumber = %d\n", scalar.intNumber);
  printf("scalar.rationalNumber = %ld/%ld\n",
         scalar.rationalNumber.numerator,
         scalar.rationalNumber.denominator);
  printf("scalar.complexNumber = %f + i%f\n",
         scalar.complexNumber.real,
         scalar.complexNumber.imaginary);

  scalar.complexNumber.real = 1.5;
  scalar.complexNumber.imaginary = 2.3;

  printf("After initializing with complex number (fraction) 1.5 - i2.3:\n");
  printf("scalar.intNumber = %d\n", scalar.intNumber);
  printf("scalar.rationalNumber = %ld/%ld\n",
         scalar.rationalNumber.numerator,
         scalar.rationalNumber.denominator);
  printf("scalar.complexNumber = %f + i%f\n",
         scalar.complexNumber.real,
         scalar.complexNumber.imaginary);


  return 0;
}
```

The output shows that the size of the union is the size of its largest member. Changing the value of the scalar in one representation, changes it in others too.

Warning

The output also clearly demonstrates that unions are intrinsically unsafe. One has to continuously keep track of what representation was used large. Using the incorrect representation (e.g. using rational or int instead of complex) leads to incorrect interpretation of the data.

```
sizeof(scalar) = 16 bytes
initial values:
scalar.intNumber = 1075047544
scalar.rationalNumber = 1075047544/1075043472
scalar.complexNumber = 4.965394 + i0.000000
After initializing with rational number (fraction) 2/3:
scalar.intNumber = 2
scalar.rationalNumber = 2/3
scalar.complexNumber = 0.000000 + i0.000000
After initializing with complex number (fraction) 1.5 - i2.3:
scalar.intNumber = 0
scalar.rationalNumber = 0/1073217536
scalar.complexNumber = 1.500000 + i2.300000
```

Note

A union is an *OR* construct. It holds one of a number of possible of data fields.

The compiler reserves enough memory to hold the largest of the optional data types. In the Scalar union of Example 3.6, “scalar.c” this is the Complex structure which holds 2 doubles and hence 8 bytes. The output listing of Example 3.6, “scalar.c” shows that the size of the Scalar union is indeed 16 bytes.

Exercises

1. Write a little application which replaces all occurances of a particular word with some substitute word. The application should first read in the text, then ask for the word to be substituted and the replacement word and then show the procecessed text.

2. Write an application which calculates the dot product of two vectors.

3. Write a program which reads in text as command line arguments and prints out the number of words of length 1 character, 2 characters and so on. You should be able to run

   ```
   wordlengths I am happy because the weather has been good this weekend and th
   and I could spend the day at the sea
   ```

   Here **wordLengths** represents the program name and the words are the command line arguments. Use an internal array for accumulating the number of words of a particular length and write a function which prints out this array in a formatted way. Skip word length for which there were no words. The output should look something like

   ```
   wordLength  noOfWords
     1           3
     2           4
     3           5
   ```

```
   4            2
   3            3
   6            2
   8            1
  11            1
```

Chapter 4. Pointers

Introduction to pointers

A pointer variable holds as value a memory address. They are the cornerstone around which flexible computing in C is achieved. Using pointers can result in efficient, compact code and ultimately form the basis for object-oriented programming in *C*.

Stack versus Heap

The stack is used to keep track of local variables and function pointers at all levels of the calling hierarchy. The stack should be viewed as a scarce resource (in some operating systems the stack per application is constrained to a maximum of `64kB`) and large memory structures should typically not be created on the stack.

When grabbing and releasing memory dynamically (via **malloc** and **free**) one grabs memory from the heap. The heap is a virtually unlimited resource constrained only by the virtual memory limit which may be the sum total of your physical RAM and the available disk space. In *C*, when using memory on the heap one has to use pointers.

Declaring and using pointers

A pointer points to a block of memory of a certain size sitting at a particular position in memory

- The position in memory is determined by the value of the pointer variable.
- The size of the memory the pointer points to is determined by its type, (e.g. a double pointer points to 8 bytes).

Example 4.1. simplePointerDemo.c

In the following simple example we declare an integer variable, `k` and initialize it to the value `3`. We then declare a pointer to an integer, `int *pk`, and initialize its value to the address of the variable `k`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int k = 3;

  int *pk = &k; /* Declares a pointer and initializes */
                /* it to point to the address of k.    */

  printf("pointer value, pk= %x\n", pk);
  printf("derefencing the pointer: *pk=%d\n", *pk);

  *pk = 5;
  printf("after *k=5: k=%d\n", k);

  return 0;
}
```

Dereferencing the pointer gives us the value of the 4 bytes (the pointer being an int*) interpreted as an integer to be 3.

Setting the value of the variable to which the pointer points to 5 has the consequence that the value of k changes to 5. The output of the program is shown below:

```
pointer value, pk= bffff384
dereferencing the pointer: *pk=3
after *k=5: k=5
```

Typed versus untyped pointers

For untyped pointers no data type is specified. The size of the memory block they are pointing to is thus not known. Neither is it known how the memory content should be interpreted.

Void pointers are type compatible with any other pointer, i.e. we can assign the value of a void pointer to that of any other pointer. In the other direction we have to, however, explicitly cast:

```
double x = 2.3;
double* p_x = &x;

void* p = p_x;  /* no casting required */

double* p2_x = (double*)p;  /* have to cast */
```

To clarify the latter, recall for that example, both int and float typically occupy 4 bytes. Interpreting the same data as an int or as a float will, however, yield very different results.

Warning

Pointers are intrinsically unsafe and a lot of care has to be taken that no errors occur. Void-pointers are even more dangerous and should be used with even greater care.

Example 4.2. voidPointer.c

```
#include <stdio.h>

int main (int argc, char *argv[])
{
  int k = 3;
  float f = 3;

  printf("k = %d\n", k);
  printf("f = %f\n", f);

  void *void_p = &k;

  printf("after setting void_p = &k:\n");
  int intValue = *((int*)void_p); /* dereferencing void pointer  */
                                  /* after casting it to an int* */
  printf("*((int*)void_p) = %x\n", intValue);
  float floatValue = *((float*)void_p); /* dereferencing void pointer   */
                                        /* after casting it to a float* */
  printf("*((float*)void_p) = %f\n", floatValue);

  void_p = &f;
  printf("after setting void_p = &f:\n");
  printf("*((int*)void_p) = %d\n", *((int*)void_p));
  printf("*((float*)void_p) = %f\n", *((float*)void_p));

  return 0;
}
```

The listing above defines a void* which is made to first point to an integer and then to a floating point variable. The data is now open to interpretation, i.e. interpreting it as an int or as a float produces vary different results.

```
k = 3
f = 3.000000
after setting void_p = &k:
*((int*)void_p) = 3
*((float*)void_p) = 0.000000
after setting void_p = &f:
*((int*)void_p) = 1077936128
*((float*)void_p) = 3.000000
```

Dynamic memory allocation

C enables one to dynamically allocate and release memory. This enables one to create dynamic data structures like linked lists, dynamically resizing arrays and so on.

Malloc and free

The two core functions used when allocating memory dynamically: `malloc` and `free`.

- **void* malloc(size_t numBytes).** Malloc grabs a contiguous memory block of `numBytes` bytes from the heap
- **void free(void* p).** The function `free` releases the memory allocated to that pointer by a previous call to `malloc`.

Warning

You have to call free on the same pointer variable for which you called malloc. If you use pointer assignment to define a new pointer pointing to the same memory and then call free on the new pointer, you are likely to get memory leaks.

Example 4.3, “polygon.c” illustrates the use of functions `malloc` and `free`.

Example 4.3. polygon.c

In this example we define a structure Point as before (and register the structure as a type via typedef). We then ask the user for the type of polygon (i.e. the number of points, 3 for triangle, 4 for quadrangle, ..., 10 for pentangle and so on).

Once we have the number of points, we request memory for `numPoints` of type Point. For example, if the user requested 3 points for a triangle, our call to `malloc` will request `3 * sizeof(Point) = 3 * 16 = 48` bytes.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

typedef struct PointStruct
{
  double x, y;
}
Point;

int main(int argc, char* argv[])
{
  int numPoints = 0;

  /* Query the number of point for the polygon (e.g. 3 for */
  /* triangle and 7 for septangle)                          */
  while (numPoints < 3)
  {
    printf("Polygon: numPoints = ");
    scanf("%d", &numPoints);

    if (numPoints < 3)
      printf("Error: a polygon must have at least 3 points.\n");
  }

  /* Now grab the memory for the number of points.*/
  Point* polygon = (Point*)malloc(numPoints * sizeof(Point));

  /* Check whether the operating system could supply */
  /* us the memory we requested.                     */
  if (polygon == NULL)
  {
    printf("ERROR: ran out of memory.");
    return -1;
  }

  /* Construct the polygon */
  int i;
  for (i=0; i<numPoints; ++i)
  {
    printf("Enter point %d: x y = ", i+1);
    scanf("%lf %lf", &polygon[i].x, &polygon[i].y);
    /* note that we directly use the array element */
    /* access on our pointer variable.             */
  }

  /* Print the polygon */
  printf("\nYou have no created the following polygon:\n{");
  for (i=0; i<numPoints; ++i)
    printf(" (%f,%f) ", polygon[i].x, polygon[i].y);
  printf("}\n\n");

  /* Reporting the memory positions of the polygon points: */
  for (i=0; i<numPoints; ++i)
    printf("Start of memory position for point %d: %x\n", i, &polygon[i]);
  printf("\n\n");

  /* Release the memory allocated for the polygon */
  free(polygon);

  printf("After freeing the memory: polygon = %x\n", polygon);

  polygon = NULL;

  return 0;
}
```

We then initialize the memory with the points for the polygon. For this we simply use the array element access operator, **[]**. Arrays as pointers discusses why we can simply use the array element access operator and what happens behind the scene. In the section called “Pointer arithmetic” we show how we could have used pointer arithmetic to access the individual points of our polygon.

Arrays as pointers

Arrays are internally represented as pointers and we can use the array elements access operator directly on a pointer. Hence

```
polygon[3]
```

is equivalent to

```
&(polygon + 3*sizeof(Point))
```

In the last expression we add 3 times the size of a Point to the address held by the pointer, `polygon`. This will be the address of the 4'th point in the array of points. We then dereference the address, interpreting the 16 bytes following that memory address as a Point.

Next we print out the points of our polygon followed by the memory positions for the individual points. Finally we release the memory allocated for the polygon.

Calloc

The function `calloc` is similar to `malloc` in that it requests blocks of memory from the operating system. Unlike `malloc`, `calloc` also initializes the memory to zero. Hence it is a little less efficient than `malloc`, but has the benefit that you may not need to initialize the elements you created dynamically yourself.

The syntax for `calloc` is different to that of `malloc`:

```
void* calloc(size_t numElements, size_t elementSize)
```

Hence, for our point example, we would call it as

```
Point* polygon = (Point*)calloc(numPoints, sizeof(Point));
```

Realloc

The function `realloc` is used to modify the size of a block of memory allocated to a pointer, i.e. to either increase or decrease the amount of memory assigned to the pointer.

The latter is, of course, always possible. The former, i.e. to increase the amount of memory allocated to a pointer, may or may not be possible, depending on whether there is sufficient memory available directly after the memory occupied currently. If t is not possible, `realloc` will allocate a new block of memory and copy the contents of the original block of memory into the start of the new block of memory.

The syntax for `realloc` is

```
void* realloc(void* p, size_t newSize);
```

Pointer arithmetic

C supports a number of arithmetic operations including

- incrementing and decrementing pointers in order to point to the next or previous element of the pointer data type and
- subtracting one pointer from another returning the number of elements of the pointer type between the two pointers.

Incrementing and decrementing pointers

When incrementing a pointer the number of bytes by which the pointer is shifted is equal to the size of the data type of the pointer. Thus incrementing or decrementing a double* shifts the address by 8 bytes.

Subtracting one pointer from another

You can only subtract pointers of the same data type from one another. Subtracting one pointer from another returns the number of elements of that data type between the two pointers, i.e. it returns the number of bytes between the two pointers divided by the size of the data type.

Example 4.4. pointerMaths.c

The following application illustrates pointer arithmetic operations:

```
#include <stdio.h>
#include <malloc.h>

int main(int argc, char* argv[])
{
  int numElements = 20;

  double* vec = (double*)malloc(numElements*sizeof(double));

  double* iter = vec;

  printf("vec:\n");
  int i;
  for (i=0; i<numElements; ++i)
  {
    *iter = i; /* set value of element to which pointer points */
    printf("%f ", *iter);
    ++iter;    /* adding sizeof(double) to address of pointer */
  }
  printf("\n\nvec = %d\n", vec);

  printf("\niter = %d\n", iter);

  printf("vec - iter = %d\n", vec-iter);

  free(vec);

  return 0;
}
```

The output of the application is shown below:

```
vec:
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.00000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000

vec = 134518304

iter = 134518464
vec - iter = -20
```

Pointers to structures

A pointer can point to an instance of a structure, for example

```
struct Point
{
double x, y;
};

struct Point *p_point = malloc(sizeof(struct Point));
```

C provides in this case two ways in which the structure elements can be retrieved

1. We can dereference the pointer and use the standard element access operator as in **(*p_point).x**,
2. or we can use a pointer-specific element access operator as in **p_point->x**. This second notation is often more readable.

A linked list example

Below we show an implementation of a double-linked circular list which makes heavy use of pointers, structures and dynamic memory allocation:

```
#include <stdio.h>
#include <malloc.h>

struct Node
{
  struct Node *next;
  struct Node *previous;
  double data;
};

struct Node* createNode(double data)
{
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->next = newNode;
  newNode->previous = newNode;
  return newNode;
}

void appendNode(struct Node *head, struct Node *newNode)
```

```
{
  struct Node *oldTail = head->previous;
  oldTail->next = newNode;
  head->previous = newNode;
  newNode->previous = oldTail;
  newNode->next = head;
}

void append(struct Node* head, double data)
{
  struct Node* newNode = createNode(data);
  appendNode(head, newNode);
}

double removeNode(struct Node* nodeToBeRemovedFromList)
{
  struct Node *previousNode = nodeToBeRemovedFromList->previous;
  struct Node *nextNode = nodeToBeRemovedFromList->next;
  previousNode->next = nextNode;
  nextNode->previous = previousNode;
  double data = nodeToBeRemovedFromList->data;
  free(nodeToBeRemovedFromList);
  return data;
}

double removeNodeAt(struct Node* head, int index)
{
  struct Node *node = head;
  int i;
  for (i=1; i<index; ++i)
  {
    node = node->next;
  }
  return removeNode(node);
}

void deleteList(struct Node* head)
{
  struct Node *lastNode = head->previous;
  while (lastNode != head)
  {
    removeNode(lastNode);
    lastNode = head->previous;
  }
  removeNode(head);
}

void insertNodeBefore(struct Node *node, struct Node *newNode)
{
  struct Node *nextNode = node->next;
  nextNode->previous = newNode;
  node->next = newNode;
  newNode->next = nextNode;
  newNode->previous = node;
}

void insertBefore(struct Node *node, double data)
{
  struct Node* newNode = createNode(data);
  insertNodeBefore(node, newNode);
}

void insertIntoList(struct Node* head, int index, double data)
{
  struct Node *node = head;
  int i;
  for (i=1; i<index; ++i)
    node = node->next;
  insertBefore(node, data);
}
```

```
void printList(struct Node *head)
{
  struct Node *iter = head;
  while (iter->next != head)
  {
    printf("%f, ", iter->data);
    iter = iter->next;
  }
  printf("%f\n", iter->data);
}

int main(int argc, char *argv[])
{
  struct Node *head = createNode(11.0);

  append(head, 22.1);
  append(head, 33.2);
  append(head, 44.3);

  printList(head);

  insertIntoList(head, 2, 999.99);

  printList(head);

  printf("now deleting list\n");
  deleteList(head);

  return 0;
}
```

Pointers to pointers

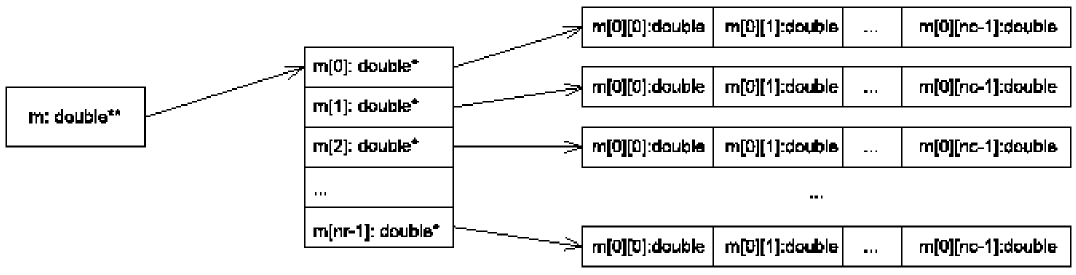
A pointer may point to a variable which itself is a pointer. This is the basis on which multi-dimensional arrays is built.

Multi-dimensional arrays

When we construct a multi-dimensional array we

- either create it as a single large memory block and resolve the correct element by mapping between the multi-dimensional representation used by users and the single-dimensional internal representation or
- build up the multidimensional structure in memory using pointer to pointers as is illustrated in Figure 4.1, “Constructing a 2-dimensional array dynamically.”.

Figure 4.1. Constructing a 2-dimensional array dynamically.



Example 4.5. matrixProduct2.c

In this example we revisit the matrix product discussed in Example 4.5, “matrixProduct2.c”. But this time we make use of pointers to make our code a lot cleaner and more re-usable:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

double** readMatrix(int* nRows, int* nCols)
{
  printf("Enter numRows numCols: ");
  scanf("%d %d", nRows, nCols);

  double** mat = malloc((*nRows) * sizeof(double*));

  int nr, nc;
  for (nr=0; nr<*nRows; ++nr)
  {
    mat[nr] = malloc((*nCols) * sizeof(double));

    for (nc=0; nc<*nCols; ++nc)
    {
      printf("mat[%d][%d] = ", nr, nc);
      scanf("%lf", &mat[nr][nc]);
    }
  }
  return mat;
}

double** matrixProduct(const double * const * const mat1,
                       const int nRowsMat1, const int nColsMat1,
                       const double * const * const mat2,
                       const int nRowsMat2, const int nColsMat2,
                       int* nRowsProduct, int* nColsProduct)
{
  if (nColsMat1 != nRowsMat2)
  {
    printf("Error: invalid dimensions in matrix multiplication.");
    exit(-1);
  }

  *nRowsProduct = nRowsMat1;
  *nColsProduct = nColsMat2;

  double** product = malloc((*nRowsProduct) * sizeof(double*));

  int nr, nc;
  for (nr=0; nr<(*nRowsProduct); ++nr)
  {
    product[nr] = malloc((*nColsProduct) * sizeof(double));

    for (nc=0; nc<(*nColsProduct); ++nc)
    {
      double sum = 0;
      int i;
      for (i=0; i<nRowsMat2; ++i)
        sum += mat1[nr][i] * mat2[i][nc];

      product[nr][nc] = sum;
    }
  }
  return product;
}

void printMatrix(const double * const * const mat,
                 const int nRows, const int nCols)
{
  int nr, nc;
  for (nr=0; nr<nRows; ++nr)
  {
    for (nc=0; nc<nCols; ++nc)
      printf("%15.6f", mat[nr][nc]);

    printf("\n");
  }
}

void delete(double** mat, int nRows)
{
  int nr;
  for (nr=0; nr<nRows; ++nr)
    free(mat[nr]);
  free(mat);
}

int main(int argc, char *argv[])
{
  int *nRowsA = malloc(sizeof(int));
  int *nColsA = malloc(sizeof(int));
  double** A = readMatrix(nRowsA, nColsA);

  int *nRowsB = malloc(sizeof(int));
  int *nColsB = malloc(sizeof(int));
  double** B = readMatrix(nRowsB, nColsB);

  int *nRowsC = malloc(sizeof(int));
  int *nColsC = malloc(sizeof(int));
  double** C = matrixProduct((const double * const * const)A,
                             (const int)*nRowsA, (const int)*nColsA,
                             (const double * const * const)B,
                             (const int)*nRowsB, (const int)*nColsB,
                             nRowsC, nColsC);

  printf("Product matrix:\n");
  printMatrix((const double * const * const)C,
              (const int)*nRowsC, (const int)*nColsC);

  delete(A, *nRowsA);
  delete(B, *nRowsB);
  delete(C, *nRowsC);
  return 0;
}
```

Note how we construct our matrix row-for-row and note also how we free the pointers in reverse order, i.e. that we first release the memory held by the double*s which reference the matrix rows and then we release the double** pointing to the array of row-pointers.

Pointers and constants

In Example 4.5, “matrixProduct2.c” we used constant pointers to lay down a contract that a method will not change certain information. For example, the function `matrixProduct` will take two matrices as arguments and return the product matrix. Neither of the two operand matrices will be modified by the function. In order to state this in our interface specification we used the following interface:

```
double** matrixProduct(const double * const * const mat1,
                     const int nRowsMat1, const int nColsMat1,
                     const double * const * const mat2,
                     const int nRowsMat2, const int nColsMat2,
                           int* nRowsProduct, int* nColsProduct)
{ ... }
```

On first sight this may look rather intimidating. How should one interpret const double * const * const. In order to understand this data type, take another look at Figure 4.1, “Constructing a 2-dimensional array dynamically.”. Let us now go from right to left. The right-most const specifies that the double** may not be modified, i.e. the address it points to.

The second const specifies that each of the double*, i.e. each of the row pointers is will not be modified by the `matrixProduct` function and the left-most const specifies that the doubles they point to (i.e. the actual matrix elements) will not be modified by the function.

Function pointers

One of the very powerful aspects of *C* is that of function pointers, i.e. we can call a function by simply having a pointer to a function and we do not need to know the function name itself or which function the pointer is currently pointing to. A function pointer thus holds the address of the entry point of a function.

Function pointers and polymorphism

Function pointers directly provide a mechanism for working at a higher level of abstraction, for polymorphism. We can write code calling functions without knowing which function will be executed at run time. All we need to know is the interface for that function, i.e. which parameters it gets and what type it returns.

Declaring function pointers

The syntax for declaring function pointers is

```
returnType (*functionPointerName)(arg1Type arg1, ..., argNType argN);
```

For example

```
double (*d_f_d)(double x)
```

declares a function pointer with variable name `d_f_d` which can point to any function which gets a double as argument and returns a double.

Having declared a variable of type pointer-to-function, we can now assign this pointer to point to any function with compatible function header. For example,

```
f = sin;
```

or

```
f = sqrt;
```

Example 4.6. simpleFunctionPointer.c

```
#include <math.h>
#include <stdio.h>

double myFunc(double x)
{
  return 3*x + sqrt(x);
}

int main(int argc, char *argv[])
{
  double (*f)(double x); /* declaring a function pointer varable, f */

  double x = 2;

  f = myFunc;
  double y = f(x);

  printf("f(%f) = %f\n", x, y);

  f = sqrt;
  y = f(x);

  printf("f(%f) = %f\n", x, y);

  return 0;
}
```

Running the program yields the following output:

```
f(2.000000) = 7.414214
f(2.000000) = 1.414214
```

Simpson integration

Simpson integration is one of the simplest integration techniques. In this example we provide a basic implementation of Simpson integration which can integrate any bounded function over a specified interval.

Note that the function, `simpsonIntegrate`, takes as first argument a function pointer. The function which will be integrated will be that function to which the pointer points.

Example 4.7. simpsonIntegration.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

double simpsonIntegrate(double (*f)(double x),
                        const double a, const double b,
                        const int numIntervals)
{
  double dx = (b-a)/numIntervals;
  double x = a+dx;

  double sumEven = 0, sumOdd = f(x);

  int n;
  for (n=2; n<numIntervals; n+=2)
  {
    x += dx;
    sumEven += f(x);
    x += dx;
    sumOdd += f(x);
  }
  return (f(a) + f(b) + 2*sumEven + 4*sumOdd) * dx/3;
}

double myFunc(double x)
{
  return 2.3*sin(x/5) + x*x;
}

int main(int argc, char *argv[])
{
  printf("Simpson integration:\n");

  int numIntervals;
  printf("number of intervals = "); scanf("%d", &numIntervals);

  double a, b;
  printf("integration boundaries: a, b = ");
  scanf("%lf %lf", &a, &b);

  double integral_a_b = simpsonIntegrate(myFunc, a, b, numIntervals);

  printf("Integral_%f^%f f(x) dx = %f\n", a, b, integral_a_b);

  return 0;
}
```

An example output is shown below:

```
Simpson integration:
number of intervals = 100
integration boundaries: a, b = -7.6 9
Integral_-7.600000^9.000000 f(x) dx = 392.522064
```

Functions with variable number of arguments

At times it may be useful to define functions which take a variable number of arguments. The `main` is an example. There may be a variable number of command-line arguments. Other examples are `scanf` and `printf`.

Function which receive a variable number of arguments need to be told how many arguments there are. This is typically done by supplying an int with the number of arguments as first argument (in the case of `printf` and `scanf` it is inferred from the number of format specifiers).

Declaring functions with variable number of arguments

The syntax for declaring a function with a variable number of arguments is

```
returnType func(int numArgs, ...);
```

We thus simply use 3 dots to indicate that there are a variable number of arguments.

Implementing a function with variable number of arguments

Within the function implementation we need to access the individual arguments. To this end *C* supplies the following variable argument support constructs:

- **va_list.** This is a type for a pointer for the arguments list.
- **va_start(va_list argsPointer, int numArgs).** This macro initializes the arguments list.
- **dataTypeOfArguments va_arg(va_list argsPointer, dataTypeOfArgument).** This macro return the argument to which the arguments-list pointer currently points to and then increments the pointer to point to the next element.
- **va_end(va_list argsPointer).** This macro should be called after all arguments have been retrieved to clean up the memory and free the arguments-list array.

Example 4.8. varArgsAverage.c

The following defines a function, `average`, which takes a variable number of arguments and calculates the average across them:

```
#include <stdio.h>
#include <stdarg.h>

double average(int numArgs, ...);

int main()
{
  printf("average(2, 1.1, 2.2) = %f\n", average(2, 1.1, 2.2));

  printf("average(5, 1.1, 1.5, 2.0, 2.1, 7.3) = %f\n",
        average(5, 1.1, 1.5, 2.0, 2.1, 7.3));

  return 0;
}

double average(int numArgs, ...)
{
  va_list arg_ptr;            /* declare a variable of type */
                              /* arguments-list pointer */

  va_start(arg_ptr, numArgs); /* initialize arguments pointer */

  double sum = 0;
  int i;
  for (i=0; i<numArgs; ++i)
    sum += va_arg(arg_ptr, double);

  va_end(arg_ptr);            /* request clean-up. */

  return sum/numArgs;
}
```

If you run this application your output will be:

```
average(2, 1.1, 2.2) = 1.650000
average(5, 1.1, 1.5, 2.0, 2.1, 7.3) = 2.800000
```

Pointers and constants

Pointers are intrinsically unsafe. In order to increase the comfort zone when working with pointers we can use constants to prevent either the pointer itself from changing or the memory the pointer points to or both.

Constant pointers

A pointer can, like any other variable, be declared a constant, in order to prevent the modification of the pointer variable itself, i.e. the address to which the pointer points. For example, if we have a **int *** then we can declare it constant via

```
int i = 17;

int * const i_p = &i;
```

which declares **i_p** as a constant which holds the address of the variable **i**. Now we cannot say, at a later stage,

```
i_p = &k;
```

because we may not change the value of the pointer, but we can say

```
*i_p = -5;
```

changing the value of the variable to which the pointer points to **-5**.

Pointers to constants

On the other hand, we can also specify that the pointer itself is not constant, but that the memory to which it points is. This is done via

```
int i = 17;

const * i_p = &i;
```

which declares **i_p** as a pointer variable to a memory position which may not be modified (at least not through the pointer **i_p**).

Now we cannot say, at a later stage,

```
*i_p = -5;
```

because we may not change the value of the variable to which the pointer points through this pointer, but we can say

```
i_p = &k;
```

changing the value of the pointer variable itself such that it now points to another memory position occupied by the variable **k**.

Constant pointers to constants

Finally we can also have constant pointers to constants. In this case one may neither change the pointer itself, nor what the pointers are pointing to:

```
const int * const i_p = &i;
```

Pointers to pointers and constants

Pointers to pointers are not uncommon in *C*. For starters, they are required when constructing multi-dimensional arrays. For example, we used

```
double * * mat;
```

to specify a data structure for a matrix. We can now insert constants at any level. For example

```
const double * const * const mat;
```

Here

- the first **const** specifies that the **double** variables to which the **double ***s point are constant,
- the second **const** specifies that the **double ***s themselves are constant and
- the third **const** specifies that the pointers to the **double ***s may not change either.

Exercises

1. Write a function,`swap`, which swaps two doubles and test the function with a suitable `main` program.
2. Write a function which approximates the derivative (slope) of a function by

   ```
   (f(x) + f(x+h))/h
   ```

   The function should get as arguments

   - the function whose derivative is to be approximated,
   - the value `x` at which the derivative of the function is to be approximated and
   - the step size, `h`.

   Write also a main program which allows the user to select between one of two functions, supply `x` and `h` and which prints out the derivative of the selected function at the requested -`x`value.
3. Write a implementation of a dynamically resizing array of contact details where users can add,

remove and query array elements. Each contact should be stored in a structure containing a name, a telephone number and an e-mail address.

4. Provide the functionality of a doubly-linked circular list functionality in C.

Chapter 5. Stream I/O

Opening a file

Opening a file, resembles opening a stream linked to a disk file. The `stdio.h` file defines a function, `fopen`, which opens a file for reading and or writing and returns a pointer to the file. The interface is

```
FILE * fopen(const char * filename, const char * mode);
```

where the mode can is one of the modes listed in Table 5.1, “File modes.”

Table 5.1. File modes.

Mode	Description
r	Opens an existing file for reading. Returns NULL if the specified file does not exist.
w	Opens a file for writing. Creates the file if it does not exist and overwrites file if it does.
a	Opens a file for appending, i.e. the file pointer is initialized to point to the end of the file. The file is created if it does not exist.
r+	Opens the file for reading and writing. The file is created if it doesn't exist. If the file exists writing the file pointer is set to the start of the file. Writing at that position will result in overwriting any existing data.
w+	Opens a file for reading and writing. The file is created if it doesn't exist. If it does, the file is overwritten (i.e. deleted).
a+	Opens a file for reading and appending. If the file does not exist, it is created. If it does, it does

Closing files

Files are simply closed via

```
int fclose(FILE *fp);
```

which returns 0 on success and -1 if the file could not be closed successfully.

You can also close *all non-standard streams* (i.e. except `stdin`, `stdout`, `stdprn`, `stderr` and `stdaux`) via

```
int fcloseall()
```

which returns the number of streams closed.

What happens with open streams when the program

ends?

If the application ends normally (by reaching the end of `main` or via the `exit()` function), then all buffers are flushed and all streams are closed.

However, if the application crashes, files may not be closed and buffers may not be flushed.

Flushing buffers

To flush the buffer associated with a particular file pointer, you can use

```
int flush(FILE *fp);
```

which returns 0 on success or EOF upon error.

Alternatively you can flush all non-standard streams via

```
int flushall()
```

which returns the number of open streams.

Standard Streams

C supplies a number of predefined streams. These standard streams are listed in Table 5.2, Stan-“dard I/O streams in C”.

Table 5.2. Standard I/O streams in *C*

stream variable	**description**
stdin	The standard input stream which, if not redirected, is the keyboard.
stdout	The standard output stream which, if not redirected, is the screen.
stderr	The standard error stream which, by default, is the screen.
stdprn	The default print stream.

The standard streams cannot be closed.

Formatted file I/O

For formatted I/O one simply uses `printf` and `scanf` with the formatting/conversion specifiers listed in Table 2.7, “Text conversion specifiers.”. See Chapter 2, *C Fundamentals* for a discussion of the `printf` and `scanf` functions.

Character I/O

The functions

```
int getc(File * fp);
```

and

```
int fgetc(File * fp);
```

both get a character from a file stream (they are interchangeable) and similarly, the functions

```
int putc(int ch, File * fp);
```

writes a single character (only the lower byte is used) to the file, `fp` and, if successful, returns the character just written. Otherwise, it returns an EOF character (`-1` integer or `FF` in hex).

To write a line of text one can use the function

```
char fputs(char * str, File * fp);
```

supplying a null-terminated character string and a file pointer. The function returns a positive number upon success and an EOF character or a negative number upon failure.

Binary I/O

The functions

```
int fread(void * buffer, int size, int count, File * fp);
```

reads `count` byte blocks each of size `size` into the memory area pointed to by the pointer `buffer` and returns the number of blocks read. This may be less than `count` if the end of the stream has bean reached or if an I/O error occurred prior to having read `count` blocks.

For example, to read 5 fixed length client records (specified as a structure type client, say) we could say

```
int numClientsRead = fread(clientArray, sizeof(client), 5, clientFile);
```

Similarly, writing a number of occurrences of a data block from memory into a file is done via

```
int fwrite(void * buffer, int size, int count, File * fp);
```

IO via random access

Random access files are particularly useful for reading and writing fixed length records (structures). As such they can be seen as rapid access files.

C provides 3 functions supporting random access:

- **int rewind(File * fp)** for resetting a file pointer to the beginning of the file,
- **int fseek(File * fp, long offset, int origin)** which moves the file pointer by the specified offset (positive or negative) relative to the `origin`. The latter has one of the values described in Table 5.3, "Possible values of the origin for fseek"

Table 5.3. Possible values of the origin for fseek

Constant	Value	Description
SEEK_SET	0	beginning of file
SEEK_CUR	1	current position in file (file pointer position)
SEEK_END	2	end of file

- **long ftell(File * fp)** for querying the position of a file pointer relative to the beginning of the file.

Detecting the end of a file

One way to detect the end of a file is simply to read a character and check whether the EOF character is returned.

In addition to this, *C* also supplies a function,

```
int feof(File * fp)
```

which returns `0` if the end of the file has not been reached and a non-zero value if the end of file has been reached.

File Management Functions

C supplies two utility functions,

```
int remove(const char * fileName);
```

to deletes a file and

```
int rename(const char * oldName, const char * newName);
```

which renames a file respectively. Both functions return 0 on success and -1 (EOF) otherwise.

Exercises

- Write a **copy** program which copies a file onto another. The source and destination files should be supplied as command-line arguments.
- Write a contacts manager which maintains a contacts with names, telephone number and e-mail addresses in a database stored as a random access file.

Chapter 6. Managing larger *C* projects

Splitting you application into multiple source files

So far we have had our application all within a single source file. Particularly for large projects, this is not sustainable.

Advantages of localizing functions in separate source files

There are several advantages in separating the program source into separate source files including

- Reuse of functionality across applications and projects.
- Simpler and localized maintenance.
- Several developers can work simultaneously on a project or application, each busy with a different file.
- Component-based or even object-oriented development with each object and component having its own data and functionalities.
- Separation of the interface (which are part of the requirements) and the implementation leads to a more contract/requirements-centered software development approach.
- More efficient recompilation of the application where only the files which have been modified need to be recompiled.

Basic separate-file compilation

In *C* any language element must be declared before it is used. To this end *C* introduces function prototypes which we discussed in the section called “Function prototyping”.

These functions prototypes are all that is needed to compile code which makes use of these functions. As such one can

- define an implementation of a function, say `someFunc` in one file,
- compile that file separately,
- insert the function prototype for `someFunc` into the top of the file which has code making use of `someFunc`,
- compile that one too and
- finally link the different compiled code elements (object files) together.

Header files

In the above approach the developer of the client code (which makes use of a function defined in a separate file) has to still look at the source file of the function. This may or may not be available and is as such anyway not desirable.

To this end *C* introduces the concept of a header file which provides the prototypes of the elements which are to be exported in order to be used in other files. A typical header file is shown below:

```
#ifndef integrationRoutines_h
#define integrationRoutines_h

double simpsonIntegrate(double (*f)(double x),
                        const double a, const double b,
                        const int numIntervals);

#endif
```

Note

We make certain that each header file is read only once by defining a pre-compiler variable (a macro constant) the first time and including the file only if that variable has not yet been defined. This prevents multiple declarations of the same elements due to multiple include statements.

The implementation file

The implementation file looks in many ways no different to what it looked previously. We would typically want to include the header file so that the compiler validates that there are no conflicts between the interface (the contract) and the implementation. We also have the benefit of having then forward declarations of the prototypes.

Below we show the implementation file:

```
#include "integrationRoutines.h"

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

double simpsonIntegrate(double (*f)(double x),
                        const double a, const double b,
                        const int numIntervals)
{
  double dx = (b-a)/numIntervals;
  double x = a+dx;

  double sumEven = 0, sumOdd = f(x);

  int n;
  for (n=2; n<numIntervals; n+=2)
  {
    x += dx;
    sumEven += f(x);
    x += dx;
    sumOdd += f(x);
  }
  return (f(a) + f(b) + 2*sumEven + 4*sumOdd) * dx/3;
}
```

Client-side files

The client files will include the header file. It can be compiled independently from the server-side implementation file (even before the latter exists).

```
#include "integrationRoutines.h"

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>


double myFunc(double x)
{
  return 2.3*sin(x/5) + x*x;
}

int main(int argc, char *argv[])
{
  printf("Simpson integration:\n");

  int numIntervals;
  printf("number of intervals = "); scanf("%d", &numIntervals);

  double a, b;
  printf("integration boundaries: a, b = ");
  scanf("%lf %lf", &a, &b);

  double integral_a_b = simpsonIntegrate(myFunc, a, b, numIntervals);

  printf("Integral_%f^%f f(x) dx = %f\n", a, b, integral_a_b);

  return 0;
}
```

Building an application using multiple source files

To compile multiple source files depends to some extend on the compiler you use. Many Unix targeting compiler use the same command-line arguments as the GNU C compiler. For example

```
cc -ansi -Wall -lm -o integrationRoutinesTest integrationRoutines.c integration
```

will compile and link the two source files into an executable `integrationRoutinesTest`.

Compiling and linking separately

In the previous section we still compiled all source files in a single step. We could have compiled them separately, and once all application components have been compiled we would link them. This could be done by the following sequence of commands:

```
cc -c -ansi -Wall -o integrationRoutines.c
```

This produces a compiled object file, `integrationRoutines.o`.

Linking the compiled object files

We can now link the compiled object files via cc which will simply call the linker, **ld**, under the hood:

```
cc -lm -o integrationRoutinesTest integrationRoutinesTest.o integrationRoutines
```

Specifying external resources via extern

When splitting source code up into multiple files certain variables may need to be accessed across different files, i.e. certain variables or structures defined in one file may need to be accessed from another. To inform the compiler that the variable will exists and will be available at link time, we use the **extern** keyword.

Example 6.1. Simple external variables example

In the following example we define a string array for the month names in one compilation unit and use it in another. The file `monthNames.c` looks as follows:

```
char * monthNames[] = {"January", "February", "March", "April",
                        "May", "June", "July", "August", "September",
                        "October", "November", "December"};
```

We publish the publicly available elements in a header file, `monthNames.h`, declaring, to any *C* source code file which includes this header, that `monthNames` is an externally defined array of character pointers which will be available at link time:

```
#ifndef monthNames_h
#define monthNames_h

  extern char* monthNames[];

#endif
```

Finally, the code which makes use of the month names includes the header:

```
#include "monthNames.h"

int main(char * args[])
{
  int n = 1;
  printf("Enter month no: ");
  scanf("%d", &n);
  char* monthName = monthNames[n-1];
  printf("Month no %d is %s\n", n, monthName);
}
```

We can compile the files individually via

```
cc -c -ansi monthNamesTest.c
```

and

```
cc -c -ansi monthNames.c
```

(i.e. we can compile `monthNamesTest.c` even before the implementation file, `monthNames.c` has been written) and we link the resources via

```
cc -o monthNamesTest monthNames.o monthNamesTest.o
```

Restricting the scope of variables and functions via static

In the previous section we showed how to publish variables so that they can be used from outside the file in which they are declared. At other times one wants to do the converse, i.e. to restrict access to certain variables or functions to within the file in which they are declared.

To this end *C* supplies the **static** keyword. Recall that we used **static** previously to define local function variables whose value had to survive across function calls. The option of being able to declare functions local to the file in which they are declared via **static** will be important once we get to object-oriented software development in ANSI-C where it will represent the backbone around which encapsulation is ensured.

Example 6.2. integrationRoutinesStaticFunction.c

In the example below we have modified our Simpson integration routine to make use of a *"private"* function which can be used only from within the file in which it is declared.

```
#include "integrationRoutines.h"

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

static double simpsonResult(double f_xMin, double f_xMax,
                            double sumEven, double sumOdd, double stepSize)
{
  return (f_xMin + f_xMax + 2*sumEven + 4*sumOdd)*stepSize/3;
}

double simpsonIntegrate(double (*f)(double x),
                        const double a, const double b,
                        const int numIntervals)
{
  double dx = (b-a)/numIntervals;
  double x = a+dx;

  double sumEven = 0, sumOdd = f(x);

  int n;
  for (n=2; n<numIntervals; n+=2)
  {
    x += dx;
    sumEven += f(x);
    x += dx;
    sumOdd += f(x);
  }
  return simpsonResult(f(a),f(b),sumEven,sumOdd,dx);
}
```

Implementation hiding

Note that the above implementation implements the same header file and hence the same interface or contract. The client code does not have to be modified at all and the `simpsonResult` is totally hidden and no dependencies can be built onto that function except from within that particular file.

Using the C Precompiler

The compilation of a *C* program first goes through a precompilation step. This enables one to provide pre-compiler directives for a number of useful features including

- Inclusion of header files and libraries.
- Conditional compilation including, conditionally, debug information or different blocks of code for different compilation targets
- Definition of constants.
- Definition of in-line functions.

As we shall explain below, the latter two should be used with caution.

Macros

Macros are segments of code which are to be expanded by the precompiler inline within the code every time a macro reference is encountered. They are used define

- constants and
- inline functions

The general syntax for a macro is

```
#define <macro> <replacement text>
```

Using macros for constants

In this case the constant name becomes the macro name and the constant value becomes the replacement text:

```
#define FALSE 0
#define TRUE !FALSE
```

```
#define PI 3.1415926

#define eps 1e-8
```

Every time a macro is encountered by the precompiler it literally substitutes it with the replacement text. The disadvantage of using macros for constants is that the constants are never entered into the symbol table and hence no debugger will know about them by name, only by value.

Instead of using macros for constants we could just use constants directly. They will not be replaced by the precompiler and hence will be entered into the symbol table. Furthermore, we can scope our constants to within methods, structures or files.

Macros for inline functions

Macros are not great for constants but they can be outright dangerous for inline functions. Consider, for example, the following commonly used macro

```
#define SQR(x) x*x
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

One needs the parenthesis around `x` and `y` in order to allow the user to use the macro with expressions. The macros look harmless enough. Consider, however, the following pitfalls:

```
y = SQR(1+2);      // result: 5 instead of 9
z = MAX(x++,y);
```

In the first case the macro expansion results in

```
y = 1+2 * 1+2;
```

which evaluates to 5 (since multiplication has higher precedence than addition). In the second case `x` will be incremented either once or twice, depending on whether `x+1` is greater than `y`. or not. It should be clear that macros like the above are not only bad style but also very dangerous.

Inline functions

Inline template functions are a very elegant alternative to macros in *C++*. In *C* we can still get the performance benefit of in-lining, but the generics is more difficult to achieve.

Example 6.3. Inline functions

The ANSI-*C* standard does not define a **inline** keyword. Inline functions are, however, supported by most *C* compilers. The body of an inline function is expanded inline every time the function is called and hence the overheads of a function call are avoided at the cost of a potentially larger executable.

Whether a function is to BB expanded in-line or not is really an optimization issue and one may feel that this should be left to the optimization algorithm of compilers.

Whether a function is declared **inline** or not is an implementation details and is thus not published in the header file:

```
              #ifndef inlineFunctions_h
```

```
#define inlineFunctions_h

double sqr(const double x);

double * max(const double * const x, const double * const y);

#endif
```

The implementation now specifies that the function should be expanded in-line.

```
inline double sqr(const double x) {return x*x;}

inline double * max(const double * const x, const double * const y)
{
  (*x) > (*y) ? x : y;
}
```

Note

The **inline** keyword is, however, solely a compiler request and a compiler can, for whatever reasons, decide to ignore the request.

Exercises

1. Define a

   ```
   swap(T, x, y)
   ```

   preprocessor macro which will swap two arguments, `x` and `y` of a data type T.

Part II. Object-Oriented Programming in ANSI C

Abstract

Even though *C* is a procedural programming language, it is sufficiently powerful to enable one to provide full object-oriented framework supporting features like *objects* and *classes*,*encapsulation*, *inheritance* and even polymorphism and exception handling.

In this part of the book we proviide an interface-drivven approach to object-oriented progrramming in ANSI-*C*.

Table of Contents

Chapter 7. Objects and Classes in ANSI-*C*

Introduction

The last 20-odd years have seen the transition of software development to object-oriented software development using object-oriented programming languages like *Smalltalk*, *C++*, *Java*, *Delphi* and *C#*. In this context it may be surprising that after 20 years of object-orientation, a procedural language like *C* has still a relatively strong following with even new projects sometimes still being developed in ANSI-*C*.

Some of these are relatively low-level operating-system related projects. Others are in the telecommunication field leveraging off large libraries available and the speed benefits which can be obtained from *C* code. Other areas may be forced to use *C* for security reasons (e.g. many defense projects), having the benefit of being able to say at any stage what information sits where in memory - often defense projects are contractually barred from using object-oriented programming languages.

Nevertheless, developing in *C* does not prevent one from following object-oriented principles and, as we shall see, it is possible to do full object-orientated programming in ANSI-*C* supporting

- objects,
- classes,
- encapsulation,
- inheritance,
- polymorphism,
- interfaces, and even
- exception handling.

Benefits of object-orientation

The benefits of object orientation can be summarized as

- Better responsibility localization.
- Well designed systems absorb changes to the requirements readily and cost-effectively.
- Localized maintenance.
- The ability to work at various levels of abstraction.
- Higher level of re-use.
- Object-oriented systems are usually simpler to test.

What is an object?

An object is a *identifiable* conceptual of physical unit which

- may have attributes,
- may offer services,
- and may have message paths to lower-level service providers.

On a conceptual level objects can be identified by identifying all the nouns of a description of the system.

What is a class?

A class is a template from which objects are created. It encapsulates the commonalities across those objects which are instances of the class.

Implementing objects and classes

In order to implement objects and classes we have to encapsulate all implementation details such that

- the object attributes can only be accessed through the object and
- that the object services (methods or functions) can only be accessed through the object interface.

The core *C* language features which will enable us to achieve this are

1. structures and pointers to structures.
2. function pointers,
3. static functions and
4. external variables.

The class interface as published in a header file

We use a structure to define the interface for our class. The structure contains only function pointers for the class and instance methods.

The pointers in the structure are as yet not populated. This will be done in the implementation file which encapsulates all the hidden information. The populated structure is published as a constant pointer to a constant instance of the structure (neither the pointer value nor the structure it points to may be changed). This constant pointer to a constant structure will be set in the implementation file, i.e. the variable is an **extern**al variable.

Example 7.1. Simple Account: Account.h

```
#ifndef string_h
#define string_h

extern const void * Account;

#endif
```

Class versus instance methods

In object-orientation one distinguishes between services offered by a class and those offered by instances of a class. In languages like *C++* and *Java*, the former are **static** methods while the latter are normal methods (with the exception of constructors which should really be declared **static** to be consistent).

For example, you are not going to credit or debit or even request the balance for the class of accounts. Instead you are going to send the **getBalance**, **credit** and **debit** messages to particular accounts (i.e. specific instances of the **Account** class).

On the other hand, you may want to assign the responsibility of keeping track of the number of instances of a class to the class itself. After all, the class is the template from which instances are created.

Instance services in object-oriented ANSI-*C*

To request an instance service through the interface we have to provide as argument the instance for which the service is requested. We have given that instance the customary variable name, **this**.

Class services in object-oriented ANSI-*C*

A class service is offered by the interface as a method which does not get a reference to an instance on which it operates - after all we are not requesting the service from an instance, only from the class itself.

The class implementation

In the class implementation we have to

1. define a data structure of the object's state,
2. define the functions which resemble the class and instance methods and restrict their access to within the implementation file,
3. populate an instance of the structure resembling the class interface with the function pointers pointing to the functions realizing the class and instance methods and
4. publish the populated class interface as a constant pointer to a constant structure.

Example 7.2. Simple Account: Account.c

```
#include <string.h>

#include "Account.h"

struct AccountState
{
  char * accountNo;
  double balance;
};

const struct AccountClass * const Account;

/* NOTE: All methods declared static (private to file) and can hence
         not be accessed directly, only through the interface, Account
         defined above which has method pointers assigned to these
         methods. */
```

```
static double getBalance_Account(const void * const this)
{
  return ((struct AccountState*)this)->balance;
};

static const char * const getAccountNo_Account (const void * const this)
{
  return ((struct AccountState*)this)->accountNo;
};

static void credit_Account(const void * const this, const double amount)
{
  ((struct AccountState*)this)->balance += amount;
};

static int debit_Account(const void * const this, const double amount)
{
  if (amount  > ((struct AccountState*)this)->balance)
    return -1;
  else
    {
      ((struct AccountState*)this)->balance -= amount;
      return 0;
    }
};

static void * create_Account(const char * const accountNo)
{
  struct AccountState * newInstance
    = (struct AccountState *)calloc(1, sizeof(struct AccountState));

  newInstance->accountNo = (char *)malloc(strlen(accountNo)+1);

  strcpy(newInstance->accountNo, accountNo);

  newInstance->balance = 0;

  return newInstance;
};

static void delete_Account(void * * this)
{
  free (*this);
  *this = 0;
};

static const struct AccountClass theAccountClass =
{
  create_Account,       /* initializes function pointer for constructor */
  getAccountNo_Account,
  getBalance_Account,   /* initializes function pointer to getBalance method */
  credit_Account,       /* initializes function pointer to credit method */
  debit_Account,        /* initializes function pointer to debit method */
  delete_Account        /* initializes function pointer to destructor */
};

const struct AccountClass * const Account = &theAccountClass;
```

Destructors

Note that the destructor gets a pointer to the `this`-pointer, enabling it to make the pointer safe (i.e. set it to null) after deleting the object it pointed to.

Example using objects and classes

Below we show an example of an object-oriented ANSI-*C* application:

Example 7.3. Simple Account: AccountTest.c

```
#include "Account.h"

int main()
{
  void * account1 = Account->create("6542376");

  printf("Created account with object reference %d\n", account1);

  printf("Balance of account # %s is %lf\n",
    Account->getAccountNo(account1), Account->getBalance(account1));

  Account->credit(account1, 500);

  if (Account->debit(account1, 200) != 0)
    printf("debit failed.");

  printf("After crediting with 500 and debiting with 200 the balance of account
    Account->getAccountNo(account1), Account->getBalance(account1));

  Account->delete(&account1);

  printf("After deleting the account our object reference is %d\n", account1);

  return 0;
};
```

We first create an account, `account1`, supplying the account number. Then we request, through the `Account` interface the `getBalance` and `getAccountNo` services.

Next we credit `account1` though the interface with R500.00. This is followed by a request for a `debit` where we check whether the service requested was indeed supplied or not.

Next we query the balance and print it out. Finally we delete `account1`. This will ensure that all memory grabbed for that object is released again.

We can compile the client application via

```
cc -ansi -Wall -c AccountTest.c
```

When compiling the client application we are completely decoupled from the class implementation (it need not even exist at this stage). The latter can be compiled independently via

```
cc -ansi -Wall -c Account.c
```

We can link the components together, creating an executable, via

```
cc -o AccountTest AccountTest.o Account.o
```

Finally we can run the generated executable to yield the following output:

```
Created account with object reference 134519160
Balance of account # 6542376 is 0.000000
After crediting with 500 and debiting with 200 the balance of account # 6542376
After deleting the account our object reference is 0
```

Exercises

1. Write a counter class. Counters can be created with or without specifying an initial count. In the former case the initial count is assumed to be 0. Users should be able to

   - increment and decrement counters,
   - add an integr count to a counter,
   - reset a counter to zero,
   - query the current counter value of a counter and
   - delete counters.

   Write a little test application which makes use of 2 counters.

Chapter 8. Specialization: Inheritance and Polymorphism in ANSI-*C*

Introduction

So far we have not yet tapped the core power of object-orientation - that of being able to work at different levels of abstraction. For this we need *subclassing*, *substitutability* and *polymorphism*.

In order to support these we will have to extend the framework developed in the previous section significantly. However, our interfaces will be in the same spirit as before and hence the client code will not be affected by the rather significant changes to the underlying machinery.

As an example we will again use an Account class, but this time we will introduce a subclass, ChequeAccount, through which we will demonstrate how the core object-oriented concepts of

- inheritance,
- substitutability and
- polymorphism

can be obtained in object-oriented ANSI-*C*.

The header files publishing the class interfaces

As before, the header files publish the class interface via a structure of function pointers which will point to the class and instance services and constant pointers to constant structure instances which users of the provide a safe class handle.

Example 8.1. Specialization: Account.h

The account interface is virtually unchanged. We only add a `print` service which will be realized in different ways for different types of accounts (i.e. different specializations of Account).

```
#ifndef Account_h
#define Account_h

struct AccountInterface
{
  void *             (*create)       (const char * const accountNo);
  const char * const (*getAccountNo) (const void * const this);
  double             (*getBalance)   (const void * const this);
  void               (*credit)       (const void * const this, const double amo
  int                (*debit)        (const void * const this, const double amo
  void               (*print)        (const void * const this);
  void               (*delete)       (void * * this);
};

extern const struct AccountInterface * const Account;
#endif
```

The subclass, ChequeAccount must supply at least the same services as Account (otherwise we will violate substitutability). For our *C* implementation we will also require that the subclass interface has the same function pointer variables as the superclass in the same order as is specified in the superclass interface.

Of course, a subclass may provide additional services (for example our cheque account provides a `getChequeFee` service which is not supplied by Accounts). The function pointers for these additional services must be appended to the services supplied by the superclass.

Implementing specialization with substitutatibily and polymorphism

Of course, we always have a price to pay for using a language which does not directly support object-orientation. For starters, the implementation is more complex and more error prone.

A second consequence is that we are forced to construct dependencies between implementation files of a class and its subclasses. The reason for this is that we need to be able to do the method mappings to **static** functions in the subclasses (recall that we declare the instance and class methods **static** in order to prevent users from using these methods directly bypassing the object-oriented interface).

Defining an object explicitly

In our more sophisticated implementation which supports inheritance, substitutability and polyorphism, it will be useful to introduce the concept of an object explicitly. An object represents an *Abstract Data Type* where the data and the operations applicable to the data are encapsulated within a single unit, an object.

The data we will encapsulate within a state structure while the instance and class methods will be encapsulated, as before, in a structure containing function pointers to the functions representing the class functions and instance methods.

Example 8.2. Specialization: Object.c

```
#ifndef Object_c
#define Object_c

/*================== Objects link state and methods =====================*/

struct Object
{
  void * state;
  void * class;
};
#endif
```

Representing object state

The state of an object is represented by a structure which is not exported - i.e. which is local to the implementation file(s). For example, the state of an account could be represented via

```
struct AccountState
{
  char * accountNo;
  double balance;
};
```

Specifying the mapping for the class and instance services

As before, we use function pointers to specify the class and instance services. The latter receive as first argument the instance whose service is requested. However, this time the mapping is not done directly at the interface level but at a hidden class level, i.e. in the implementation file, we specify the structure, initialize and instance and provide a constant handle to it:

```
struct AccountClass
{
  void *             (*create)       (const char * const accountNo);

  const char * const (*getAccountNo) (const void * const this);
  double             (*getBalance)   (const void * const this);
  void               (*credit)       (const void * const this, const double amo
  int                (*debit)        (const void * const this, const double amo
  void               (*print)        (const void * const this);

  void               (*delete)       (void * * this);
};

/*===================== Account Methods ================================*/

/* NOTE: All methods declared static (private to file) and can hence
         not be accessed directly, only through the interface, Account
         defined above which has method pointers assigned to these
         methods. Implementation is encapsulated. */

static double getBalance_Account(const void * const state)
{
  return ((struct AccountState*)state)->balance;
};

...

/*============== Setting the pointers for the Account Methods ===========*/

/* Here we initialize the function pointers for the instance and class
   methods. Instance methods get the instance reference (object reference)
   as argument, class services (static methods) do not. */

static const struct AccountClass theAccountClass =
{
  create_Account,       /* initializes function pointer for constructor */
  getAccountNo_Account, /* initializes function pointer for getAccountNo method
  getBalance_Account,   /* getBalance method */
  credit_Account,       /* credit method */
  debit_Account,        /* debit method */
  print_Account,        /* print method */
  delete_Account        /* initializes function pointer to destructor */
};

/* Specifying a non-modifiable handle to the class with the function pointers
   fixed. */
const struct AccountClass * const accountClass = &theAccountClass;
        </programlisting>
```

```
      </para>
    </section>

    <section><title>Implementing dynamic binding to support polymorphism</title
      <para>
       In order to support polymorphism we have to identify the actual class
       of an object and link its method dynamically. In our framework we can
       easily delegate that functionality to the interface which
       <itemizedlist>
         <listitem><para>
           extracts the objects state and class and then
         </para></listitem>
         <listitem><para>
           sends the service request message to the class supplying the object
           state as argument.
         </para></listitem>
       </itemizedlist>
      </para>
      <para>
        For example, the implementation of the interface service, <function>deb
        is
        <programlisting>
static int debit_AccountInterface(const void * const this, const double amount)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  return class->debit(state, amount);
}
```

Implementing constructors

Since constructors are class services, not instance services, we do not need dynamic binding for them. At the interface level we simply have

```
static void * create_AccountInterface(const char * const accountNo)
{
  return create_Account(accountNo);
}
```

At the class level we provide the implementation for the constructor. We have to

- create and initialize the instance state,
- create an object and initialize the state structure and class for that object and
- return a handle to the object.

```
static void * create_Account(const char * const accountNo)
{
  struct AccountState * newInstanceState
    = (struct AccountState *)calloc(1, sizeof(struct AccountState));

  newInstanceState->accountNo = (char *)malloc(strlen(accountNo)+1);

  strcpy(newInstanceState->accountNo, accountNo);
  newInstanceState->balance = 0;

  struct Object *object = (struct Object *)malloc(sizeof(struct Object));

  object->state = newInstanceState;
  object->class = (void *)accountClass;

  return (void *)object;
};
```

Implementing destructors

While constructors are class services, destructors, on the other hand are instance services. We

- free the memory for the state fields and the state structure itself and
- set the self pointer (`this` pointer) to NULL.

```
static void delete_Account(void * * this)
{
  struct AccountState * * accountState = (struct AccountState * *)this;

  free ((*accountState)->accountNo);
  free (*accountState);

  *(struct Object * *)this = 0;
};
```

Example 8.3. Specialization: Account.c

The complete listing of the Account class:

```
#ifndef Account_c
#define Account_c

#include <string.h>

#include "Account.h"
#include "Object.c"

/*====================== Account State Structure ========================*/

struct AccountState
{
  char * accountNo;
  double balance;
};

const struct AccountInterface * const Account;

/*========================== Account Class =============================*/

struct AccountClass
{
  void *             (*create)       (const char * const accountNo);

  const char * const (*getAccountNo) (const void * const this);
  double             (*getBalance)   (const void * const this);
  void               (*credit)       (const void * const this, const double amo
  int                (*debit)        (const void * const this, const double amo
  void               (*print)        (const void * const this);

  void               (*delete)       (void * * this);
};

const struct AccountClass * const accountClass;

/*===================== Account Methods ================================*/

/* NOTE: All methods declared static (private to file) and can hence
         not be accessed directly, only through the interface, Account
         defined above which has method pointers assigned to these
         methods. Implementation is encapssulated. */

static double getBalance_Account(const void * const state)
{
  return ((struct AccountState*)state)->balance;
};

static const char * const getAccountNo_Account (const void * const state)
{
  struct AccountState * accountState = (struct AccountState*)state;
  return (accountState)->accountNo;
};

static void credit_Account(const void * const state, const double amount)
{
  ((struct AccountState*)state)->balance += amount;
};

static int debit_Account(const void * const state, const double amount)
{
  if (amount  > ((struct AccountState*)state)->balance)
    return -1;
  else
    {
      ((struct AccountState*)state)->balance -= amount;
      return 0;
    }
};

static void print_Account (const void * const state)
{
  struct AccountState * accountState = (struct AccountState*)state;

  printf("%s , balance = %10.2lf", accountState->accountNo, accountState->balan
};

/* Constructor of Account Class */

static void * create_Account(const char * const accountNo)
{
  struct AccountState * newInstanceState
    = (struct AccountState *)calloc(1, sizeof(struct AccountState));

  newInstanceState->accountNo = (char *)malloc(strlen(accountNo)+1);

  strcpy(newInstanceState->accountNo, accountNo);

  newInstanceState->balance = 0;

  struct Object *object = (struct Object *)malloc(sizeof(struct Object));

  object->state = newInstanceState;
  object->class = (void *)accountClass;

  return (void *)object;
```

```
};

/* Destructor of Account Class */

static void delete_Account(void * * this)
{
  struct AccountState * * accountState = (struct AccountState * *)this;

  free ((*accountState)->accountNo);
  free (*accountState);

  *(struct Object * *)this = 0;
};

/*============== Setting the pointers for the Account Methods ===========*/

/* Here we initialize the funnction pointers for the instance and class
   methods. Instance methods get the instance reference (object reference)
   as argument, class services (static methods) do not. */

static const struct AccountClass theAccountClass =
{
  create_Account,       /* initializes function pointer for constructor */
  getAccountNo_Account, /* initializes function pointer for getAccountNo method
  getBalance_Account,   /* getBalance method */
  credit_Account,       /* credit method */
  debit_Account,        /* debit method */
  print_Account,        /* print method */
  delete_Account        /* initializes function pointer to destructor */
};

/* Specifying a non-modifiable handle to the class with the function pointers
   fixed. */
const struct AccountClass * const accountClass = &theAccountClass;

/*======= Account Interface implements Virtual Method Table ========== */

/* The methods query the instance (object) for its class which is only known
   at run-time and then request the service from the class supplying the
   instance pointer as argument. This facilitates polymorphism. */

static const char * const getAccountNo_AccountInterface(const void * const this
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;
  const char * const accountNo = class->getAccountNo(state);
  return accountNo;
}

static double getBalance_AccountInterface(const void * const this)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  return class->getBalance(state);
}

static void credit_AccountInterface(const void * const this, const double amoun
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  class->credit(state, amount);
}
```

```
static int debit_AccountInterface(const void * const this, const double amount)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  return class->debit(state, amount);
}

static void print_AccountInterface(const void * const this)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  class->print(state);
}

/* destructor is a dynamically bound innstance method */

static void delete_AccountInterface(void * * this)
{
  struct Object * * instance = (struct Object * *)this;

  struct AccountClass * class = (struct AccountClass *)(*instance)->class;
  struct AccountState * state = (struct AccountState *)(*instance)->state;
  struct AccountState * * state_p = &state;

  class->delete((void * *)state_p);
}

/* Constructor statically bound */

static void * create_AccountInterface(const char * const accountNo)
{
  return create_Account(accountNo);
}

/*============== Setting the pointers for the Account Interface ===========*/

/* Setting now the function pointers for the interface through which Account
   objeccts are accessed. */

static const struct AccountInterface theAccountInterface =
{
  create_AccountInterface,

  getAccountNo_AccountInterface,
  getBalance_AccountInterface,
  credit_AccountInterface,
  debit_AccountInterface,
  print_AccountInterface,

  delete_AccountInterface
};

/* The type users are working with, Account, is actually an interface. Below we
   specify a non-modifiable handle to the interface which is exported in the
   header file. This public interface is used by users to access account
   objects (as  well as possibly subclass instances of Account). */

const struct AccountInterface * const Account = &theAccountInterface;

#endif
```

Defining a subclass

The subclass, ChequeAccount, looks in many ways very similar to the superclass, Account with the following features to be noted

- The state of the subclass fully contains the state of the superclass.
- The subclass supplies all the services of the superclass and potentially adds additional services.
- A subclass may map a service onto a superclass implementation, thereby inheriting the service from the superclass.
- A subclass may override a service of the superclass. In the implementation it may often still call the superclass service.
- Because the subclass implementation depends explicitly on the superclass implementation we have to import not only the header, but also the implementation of the superclass, Account

The header file of the subclass

Here we must make certain that the subclass contains up-front all the methods of the superclass in the same order as what the superclass supplies them.

We then add the subclass-specific services.

Example 8.4. Specialization: ChequeAccount.h

```
#ifndef ChequeAccount_h
#define ChequeAccount_h

struct ChequeAccountInterface
{
  void *             (*create)       (const char * const accountNo, const doubl
  const char * const (*getAccountNo) (const void * const this);
  double             (*getBalance)   (const void * const this);
  void               (*credit)       (const void * const this, const double amo
  int                (*debit)        (const void * const this, const double amo
  void               (*print)        (const void * const this);
  void               (*delete)       (void * * this);

  double             (*getChequeFee) (const void * const this);
};

extern const struct ChequeAccountInterface * const ChequeAccount;
#endif
```

Implementing a subclass

The implementation file contains the implementations of the subclass-specific methods (both, the new methods and the overridden methods) and specifies the method bindings (also to the inherited methods of the superclass).

Example 8.5. Specialization: ChequeAccount.c

The complete listing of the Account class:

```
#ifndef Account_c
#define Account_c

#include <string.h>

#include "Account.h"
#include "Object.c"

/*===================== Account State Structure =======================*/

struct AccountState
{
  char * accountNo;
  double balance;
};

const struct AccountInterface * const Account;

/*========================== Account Class ============================*/

struct AccountClass
{
  void *             (*create)       (const char * const accountNo);

  const char * const (*getAccountNo) (const void * const this);
  double             (*getBalance)   (const void * const this);
  void               (*credit)       (const void * const this, const double amo
  int                (*debit)        (const void * const this, const double amo
  void               (*print)        (const void * const this);

  void               (*delete)       (void * * this);
};

const struct AccountClass * const accountClass;

/*===================== Account Methods ================================*/

/* NOTE: All methods declared static (private to file) and can hence
         not be accessed directly, only through the interface, Account
         defined above which has method pointers assigned to these
         methods. Implementation is encapssulated. */

static double getBalance_Account(const void * const state)
{
  return ((struct AccountState*)state)->balance;
};

static const char * const getAccountNo_Account (const void * const state)
{
  struct AccountState * accountState = (struct AccountState*)state;
  return (accountState)->accountNo;
};

static void credit_Account(const void * const state, const double amount)
{
  ((struct AccountState*)state)->balance += amount;
};

static int debit_Account(const void * const state, const double amount)
{
  if (amount  > ((struct AccountState*)state)->balance)
```

```
    return -1;
  else
    {
      ((struct AccountState*)state)->balance -= amount;
      return 0;
    }
};

static void print_Account (const void * const state)
{
  struct AccountState * accountState = (struct AccountState*)state;

  printf("%s , balance = %10.2lf", accountState->accountNo, accountState->balan
};

/* Constructor of Account Class */

static void * create_Account(const char * const accountNo)
{
  struct AccountState * newInstanceState
    = (struct AccountState *)calloc(1, sizeof(struct AccountState));

  newInstanceState->accountNo = (char *)malloc(strlen(accountNo)+1);

  strcpy(newInstanceState->accountNo, accountNo);

  newInstanceState->balance = 0;

  struct Object *object = (struct Object *)malloc(sizeof(struct Object));

  object->state = newInstanceState;
  object->class = (void *)accountClass;

  return (void *)object;
};

/* Destructor of Account Class */

static void delete_Account(void * * this)
{
  struct AccountState * * accountState = (struct AccountState * *)this;

  free ((*accountState)->accountNo);
  free (*accountState);

  *(struct Object * *)this = 0;
};

/*============== Setting the pointers for the Account Methods ===========*/

/* Here we initialize the funnction pointers for the instance and class
   methods. Instance methods get the instance reference (object reference)
   as argument, class services (static methods) do not. */

static const struct AccountClass theAccountClass =
{
  create_Account,       /* initializes function pointer for constructor */
  getAccountNo_Account, /* initializes function pointer for getAccountNo method
  getBalance_Account,   /* getBalance method */
  credit_Account,       /* credit method */
  debit_Account,        /* debit method */
  print_Account,        /* print method */
  delete_Account        /* initializes function pointer to destructor */
};

/* Specifying a non-modifiable handle to the class with the function pointers
   fixed. */
const struct AccountClass * const accountClass = &theAccountClass;

/*======= Account Interface implements Virtual Method Table ========== */
```

```
/* The methods query the instance (object) for its class which is only known
   at run-time and then request the service from the class supplying the
   instance pointer as argument. This facilitates polymorphism. */

static const char * const getAccountNo_AccountInterface(const void * const this
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;
  const char * const accountNo = class->getAccountNo(state);
  return accountNo;
}

static double getBalance_AccountInterface(const void * const this)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  return class->getBalance(state);
}

static void credit_AccountInterface(const void * const this, const double amoun
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  class->credit(state, amount);
}

static int debit_AccountInterface(const void * const this, const double amount)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  return class->debit(state, amount);
}

static void print_AccountInterface(const void * const this)
{
  struct Object * instance = (struct Object *)this;

  struct AccountClass * class = (struct AccountClass *)instance->class;
  struct AccountState * state = (struct AccountState *)instance->state;

  class->print(state);
}

/* destructor is a dynamically bound innstance method */

static void delete_AccountInterface(void * * this)
{
  struct Object * * instance = (struct Object * *)this;

  struct AccountClass * class = (struct AccountClass *)(*instance)->class;
  struct AccountState * state = (struct AccountState *)(*instance)->state;
  struct AccountState * * state_p = &state;

  class->delete((void * *)state_p);
}

/* Constructor statically bound */
```

```
static void * create_AccountInterface(const char * const accountNo)
{
  return create_Account(accountNo);
}

/*============== Setting the pointers for the Account Interface ===========*/

/* Setting now the function pointers for the interface through which Account
   objeccts are accessed. */

static const struct AccountInterface theAccountInterface =
{
  create_AccountInterface,

  getAccountNo_AccountInterface,
  getBalance_AccountInterface,
  credit_AccountInterface,
  debit_AccountInterface,
  print_AccountInterface,

  delete_AccountInterface
};

/* The type users are working with, Account, is actually an interface. Below we
   specify a non-modifiable handle to the interface which is exported in the
   header file. This public interface is used by users to access account
   objects (as  well as possibly subclass instances of Account). */

const struct AccountInterface * const Account = &theAccountInterface;

#endif
```

Calling superclass methods from subclass methods which override them

This is not uncommon because, in the spirit of *design by contract* a subclass method must provide all the postconditions (deliverables) of the superclass method and may add additional responsibilities. The reason for this is once again the core of specialization, substitutability.

Substitutability and polymorphism in practice

The following example program demonstrates

- inheritance of the `credit` method,
- polymorphism on the `debit`, `print` and `delete` methods and
- substitutability of ChequeAccounts for Accounts.

Example 8.6. Specialization: AccountTest.c

```
#include "Account.h"
#include "ChequeAccount.h"
```

```
void processAccount(void * acc)
{
  printf("Processing account -> ");
  Account->print(acc);  /* done polymorphically */
  printf("\n");

  Account->credit(acc, 100); /* cheque accounts inherit credit */
  printf("After crediting with 100: ");
  Account->print(acc);  /* done polymorphically */
  printf("\n");

  Account->debit(acc, 50);   /* done polymorphically */
  printf("After debiting with 50: ");
  Account->print(acc);  /* done polymorphically */
  printf("\n");
}

void testInheritanceAndPolymorphism()
{
   void * account = Account->create("Acc1");
   processAccount(account);
   Account->delete((void * *)&account);  /* done polymorphically */

   account = ChequeAccount->create("ChAcc1", 6);
   processAccount(account);
   Account->delete((void * *)&account);  /* done polymorphically */
}


int main()
{

  /* Testing inheritance */

  testInheritanceAndPolymorphism();

  return 0;
};
```

Exercises

1. Define an Employee class and a Developer subclass. Employees have a name and a salary. Developers also have a computer allowance. Both provide a `getTotalSalary` service, which in the case of employees simply returns the salary, but in the case of developers returns the salary plus the computer allowance. Supply also a polymorphic print service. Finally write a little test application which tests

   - inheritance of the `getName` method,
   - polymorphism on the `getSalary`, `print` and `delete` services and
   - the substitutability of developers for employees.

2. Define an interface Sortable which specifies that any class which implements Sortable+ must be comparable to another Sortable. The `compareTo` method should return `-1` if the argument object is less than the `this` object, `0` if they are equal and `+1` otherwise. Now, let Person implement Sortable such that the comparison is done lexicographically first on the surname and then on the first names (the Person class should have the appropriate attributes). Define also an Account class with `accounts` being sorted on their balance. Now write a little BubbleSort

class which implements the Sorter interface specifying that every Sorter must provide a service, `sort` which can sort an array of Sortables. Test your sorter by sorting an array of persons and an array of accounts respectively.

Note

You are now in a position to select any implementation of a Sorter to sort anything which is Sortable.