Computing in C++
Part I : An introduction to C

Dr Robert Nürnberg

Introduction

This course will give an introduction to the programming language C. It runs in the autumn term and is addressed to students with no or little programming experience in C. Attending this course will enable you to write simple C/C++ programs and complete possible practical assignments of other courses that run alongside this course. In this term, there will be no separate coursework assignments for the course "Computing in C++".
The course will be followed in the second term by an introduction to Object Oriented Programming in C++. That is why from the start, we will be using a C++ compiler, even though the programs are really written in C or at least in "C-like C++". Concepts that are unique to C++ and do not form part of the C programming language will be marked with a footnote like this (*). This will help you adapt your code to a C compiler, if you ever have to.
Throughout this script, new concepts and ideas will be illustrated with easy to understand example programs. These examples can be compiled as they are and, if your pdf-viewer allows you to, you can even copy and paste the code into your editor without having to type it in yourself.
These lecture notes are organised as follows. Chapter 1 gives a short overview over the main techniques and features of C and is designed to enable you to start programming straight away. The remaining chapters will then cover each topic in more detail.

Recommended books and sites

- C only:
  - Tony Zhang and John Southmayd, *Teach Yourself C in 24 Hours*, 2000
  - Bradley Jones and Peter G. Aitken, *Teach Yourself C in 21 Days*, 2003
  - Brian W. Kernigham and Dennis M. Ritchie, *The C Programming language*, 1988
  - Richard Johnsonbaugh and Martin Kalin, *C for Scientists and Engineers*, 1996
  - Steven R. Lerman, *Problem solving and computations for scientists and engineers, an introduction using C*, 1993
  - `publications.gbdirect.co.uk/c_book` (The C Book online)
- C++:
  - Steve Oualline, *Practical C++ Programming*, 1995
  - Herbert Schildt, *Teach yourself C++*, 1992
  - Jesse Liberty, *Teach yourself C++ in 24 hours*, 1999
  - `www.cplusplus.org`, `www.cplusplus.com`, `www.cppreference.com`

Compilers

- Windows:
  - Microsoft Visual C++ .NET with Integrated Development Environment (IDE)
  - GNU C++ compiler (`g++`) as part of `Cygwin` or `MinGW`, without IDE
  - Dev-C++ – free compiler/IDE that is GNU compatible
- Linux:
  - GNU C++ compiler (`g++`) – part of any distribution
  - Intel C++ compiler (`icc`) – free for students
- Mac:
  - Xcode – free compiler/IDE that is GNU compatible
- Windows/Linux/Mac:
  - Code::Blocks – free compiler/IDE that is GNU compatible
  - NetBeans – free compiler/IDE that is GNU compatible

*A footnote to highlight parts that are not pure C.

Contents

1 Basics

A C/C++ program is nothing other than a text file with a very special syntax. In order to distinguish these files from other text files, they are usually given the ending `.cpp` (for windows) or `.cc` (for linux). These files are also called *source* files, as they are the source for the actual program.
A C/C++ compiler is then needed to translate the source files into an executable (or binary) file format, that can be executed by the operating system. This binary file is then the actual program. When working under windows in most cases you will not be aware of the executable file, as the compiler will allow you to run the code immediately after compilation.
The following syntax guidelines you need to know, before you can start to write your first C++ program. Each C/C++ program needs to have a `main()` program. Inside this `main()` program we define what the code is supposed to do. The commands that are to be executed are grouped together inside some "`{ }`" brackets. Each statement in C must be terminated by a semi-colon "`;`". See the first example programs in the next subsection.

1.1 "Hello World!"

There are two ways to output things on the screen. One uses the C++ stream `cout` (on the left) and the other uses the C function `printf()` (on the right). (*)

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

```cpp
#include <cstdio>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

During the course, I may make use of both of these possibilities and you are free to use whatever you prefer. Note, however, that in the second term we will almost exclusively be using the C++ stream `cout`.

1.1.1 Waiting for input

Under Windows, the terminal screen often disappears quickly after the last output of the program. To prevent this, you can make the program wait until you press Enter as follows.

```cpp
#include <iostream>
using namespace std;

int main() {
  cout << "Press Enter to continue" << endl;
  cin.get();
  return 0;
}
```

```cpp
#include <cstdio>

int main() {
  printf("Press Enter to continue\n");
  scanf("%*c");
  return 0;
}
```

1.2 Comments

Comments are parts of the code that will be ignored by the compiler. They are used to add descriptions and explanations to the source code. One can either comment parts or all of a single line with `//`, or comment bigger parts of the code by using `/*` at the beginning and `*/` at the end. Here an example. (†)

```cpp
#include <iostream>
using namespace std;

/**************************************
 * Example for using Comments in C/C++ *
 **************************************/
```

*The stream `cout` is unique to C++. Also note that when including header files in a pure C program, the syntax is e.g. `#include <stdio.h>`.

†Commenting code with `//` is not part of ANSI C. However, most C compilers accept it.

```
int main() {
  int i;                                  // an integer
  int sum;                                /* used to add up i's */

  sum = 0;
  for (i = 0; i<10; i++) {                // a for loop
    sum = sum + i;
  }

  cout << "The sum is " << sum << endl;
  return 0;
}
```

1.3 Variables and types

Variables are the very basis of every programming language. Without them you could not do computations and you could not program anything meaningful.
Variables allow you to store and recall numerical values, much like the memory register on modern handheld calculators. Only in C you can have almost infinitely many of these registers, and you can also name them whatever you like. All you have to do is to *declare* any variable you want to use, before actually using it.
In C there is a strong notion of separation of variables from different *types*. For instance, by default you can only assign to a variable values of the same type. So, when you introduce a variable, you have to declare its type.
A variable can be declared with the following syntax.

```
type variable_name;
```

Here `type` specifies the variable's type, and `variable_name` defines the name under which you can use the variable in your program.
The two most important types in C are `int` and `double`. The former is used to store integer numbers and is used for e.g. counters. The latter stores floating point numbers and is used for scientific computations. Here is an example of how to use them.

```
#include <iostream>
#include <cmath>                  // include math.h
using namespace std;

int main() {
  int i,j;                        // a declaration of two integers ...
  double x,y;                     // ... and two doubles

  i = 5;                          // an assignment
  j = 2*i + 3;                    // use value of i for assignment of j

  x = 3.14;                       // an assignment of a floating point number
  y = sin(x) * exp(-5*x);

  cout << "The value of i is " << i << ", while y = " << y << endl;
  return 0;
}
```

Note that before the variables are used they are declared. Once a variable has been declared with its name and its type, it can be assigned values, and these values can be recalled or updated later on in the program.

1.4 User input

The C++ input stream `cin` can be used to allow the user to type in values from the keyboard. Here is an example to illustrate this. (*)

*The `cin` stream is unique to C++.

```
#include <iostream>
using namespace std;

int main() {
  int i, number;
  double x;
  double sum = 0.0;                                    // initialize sum with 0.0

  cout << "How many numbers do you want to add up: ";
  cin >> number;                                       // input an integer

  for (i = 0; i < number; i++) {                       // a for loop
    cout << "Input next number: ";
    cin >> x;                                          // input a double
    sum = sum + x;
  }
  cout << "Final sum: " << sum << "." << endl;
  return 0;
}
```

If you do not want to make use of the C++ streams `cin` and `cout`, then the above program would have to look as follows.

```
#include <cstdio>

int main() {
  int i, number;
  double x;
  double sum = 0.0;                                    // initialize sum with 0.0

  printf("How many numbers do you want to add up: ");
  scanf("%d", &number);                                // input an integer

  for (i = 0; i < number; i++) {                       // a for loop
    printf("Input next number: ");
    scanf("%lf", &x);                                  // input a double
    sum = sum + x;
  }
  printf("Final sum: %6.4e.\n",sum);
  return 0;
}
```

Note that when using the C function `scanf()`, you have to pass the *address* of the variable you want to assign a value to. For example, you have to pass `&x` instead than simply `x`. We will learn more about addresses of variables when we discuss pointers in Chapter 6.

1.5 Control flow

Although the `while` loop is the simplest loop in C, the most frequently used loop is the `for` loop. The syntax for the two loops is

```
while (continuation) {
  instructions;
}
```

and

```
for (initialization; continuation; increment) {
  instructions;
}
```

respectively. Both loops repeat the given `instructions` over and over again, until the `continuation` evaluates to `false`. Note that if `continuation` is already `false` before the loop begins, then the `instructions` inside the loop will not be executed at all. In addition, the `for` loop allows the programmer to initialize a variable before the first execution of the loop and to increment a variable after it reaches the end of the loop, before the next evaluation of `continuation`. That is why the `for` loop is

used when the processing is sequential and when there is a well defined notion of an increment between one pass through the loop and the next.
The following example shows how to use the two types of loops. It also includes an example of another control flow construction, the `if-else` statement.

```
#include <iostream>
using namespace std;

int main() {
  double rate, pounds, euro, sum;
  int i;

  cout << "Please enter current exchange rate for GBP to EURO: ";
  cin >> rate;

  pounds = 1.0;                                          // any positive value
  while (pounds > 0.0) {                                 // while loop
    cout << "Enter amount in pounds to convert (0 to quit) : ";
    cin >> pounds;
    if (pounds > 0.0) {                                  // if - then
      euro = pounds * rate;
      cout << pounds << " GBP are " << euro << " in EURO." << endl;
    }
    else {                                               // else
      cout << "Now quitting this part." << endl;
    }
  }

  sum = 0.0;
  for (i = 0; i < 1000; i++) {                           // for loop
    sum = sum + 1.0 / (i + 1.0);
  }
  cout << "The sum 1 + 1/2 + 1/3 + ... + 1/1000 is " << sum << endl;
  return 0;
}
```

More details on control flow can be found in Chapter 3.

1.6 Basic operators

In our example programs, we have already come across the assignment operator `=`, the logical operator `<` and the arithmetic operators `*`, `/`, `+` and `-`.
The meaning of these operators is fairly clear. The assignment operator `=` evaluates the expression on the right hand side and assigns the value to the variable on the left hand side. Strictly speaking, the variable on the left and the expression on the right always have to be of the same type. For example,

```
int i;
i = 1.234;
```

is not allowed.
Any logical operator, like `==`, `<=`, `>=`, `>` or `!=`, compares the two operands and returns either `true` or `false`. This result can then be used in order to control the program flow (see the previous section for details). Note also that results from logical operators can be combined with the logical *and* and *or* statements. In C/C++ they are represented by `&&` and `||`, respectively.
Finally, some less obvious operators we have come across already are the so called inserter operator `<<`, that is used to direct output to the output stream `cout`, and the postfix increment operator `++`, as in the statement `i++`. The latter statement is equivalent to writing `i = i + 1`.
More information on operators can be found in Chapter 4.

1.7 Arrays

Arrays are a special form of variables in C and they are the natural representations of vectors in C. An array is declared with the syntax

```
type name[SIZE];
```

where `type` is any of the basic types and `SIZE` is a constant integer. You cannot declare an array with a variable size. The technical term for this is that arrays are statically allocated, and not dynamically. We will see more on this later.
For example, the statement

```
double x[100];
```

allocates in memory the space for 100 `double` numbers, and these are stored sequentially in memory. They are referred to as `x[0]` to `x[99]`, and inside the "`[ ]`" brackets any integer expression can be used. Note that `x[100]` refers to the first space in memory *after* the array, so it is not well defined. Trying to access this number can lead to catastrophic results when running the program. Unfortunately, the compiler will not warn you about this. So extra care has to be taken when using arrays in C.
The following program shows an example of how to use arrays.

```
#include <iostream>
using namespace std;

int main() {
  int i;
  double x[100], y[100];                          // declaration of two arrays
  for (i = 0; i < 100; i++) {
    x[i] = 2.0*i;
    y[99-i] = i;
  }
  for (i = 0; i < 100; i++)
    cout << i << ". x = " << x[i] << ", y = " << y[i] << endl;
  return 0;
}
```

1.8 Functions

Functions or subroutines are parts of the code, that perform a certain well defined task within the program. Defining functions allows you to re-use these parts of the code in other programs. Moreover, your program will be better structured and your source code will be easier to read.
Functions can have a return value. An example for this is the call to the `sin()` function in the example on page 4. The function returns a value of type `double` and this is used to compute the variable `y`.
If a function is not expected to return a value, then it needs to be declared as `void`. Moreover, each function takes a fixed number of arguments. The number and the type of these arguments has to be declared when defining the function. Here is a short example.

```
#include <iostream>
using namespace std;

void welcome() {
  cout << "Hi there." << endl;
}

int input() {
  int in;
  cout << "Please enter a number: ";
  cin >> in;
  return in;
}

void goodbye(int number) {
  cout << "You typed in: " << number << "." << endl;
  cout << "Thanks and goodbye!" << endl;
}

int main() {
  int i;
```

```
  welcome();                    // call the void function welcome()
  i = input();                  // get value for i from function input()
  goodbye(i);                   // call void function goodbye() with value i

  return 0;
}
```

Note that in this example the three functions `welcome()`, `input()` and `goodbye()` are defined. Note also, that two of the functions do not return any result value, and that two of the functions do not take a parameter.
You can also see that a function can be called simply by giving its name, followed by all the expected parameters inside round "`( )`" brackets. In case that a function does not take any parameters, these brackets are still necessary, although they are left empty.
Note furthermore that the statements to be executed in a function are again grouped together with the "`{ }`" brackets, as is the case for the `main()` program. It is worth mentioning that the `main()` program is simply a very special function inside every C/C++ program. In fact, the `main()` program is called by the operating system of your computer in much the same way that you call functions inside your C++ program.

1.8.1 Mathematical functions

C provides an extensive library of mathematical functions. The definitions can be found in the system header file `math.h`, which under C++ can be included with the line

```
#include <cmath>
```

at the beginning of your program, see the example in §1.3.
Having done that, you can access a wide selection of mathematical functions, such as `sqrt()`, `sin()`, `cos()`, `exp()`, `log()`, `pow()` etc. See `www.cplusplus.com/ref/cmath` for details.

1.9 Output control characters

There are several control characters, that you can use between quotes `""` in output via both `cout` and `printf()`. Here is a complete list.

character	meaning
`\n`	newline
`\t`	horizontal tab
`\v`	vertical tab
`\b`	backspace
`\r`	carriage return
`\f`	formfeed
`\a`	audible bell

Here is a small example program.

```
#include <iostream>
using namespace std;

int main() {
  cout << "Hello\t you\n\nWhy\t do you look\t like that?\n";
  cout << "Hello\v you\n\nWhy\v do you look\v like that?\n";
  cout << "Hello\b you\n\nWhy\b do you look\b like that?\n";
  cout << "Hello\r you\n\nWhy\r do you look\r like that?\n";
  cout << "Hello\f you\n\nWhy\f do you look\f like that?\n";
  cout << "Hello\a you\n\nWhy\a do you look\a like that?\n";
}
```

Note that using `cout << "\n"` is equivalent to writing `cout << endl;`.
In addition, when using the C++ output stream `cout`, you have several output manipulators that you can use. The following program shows the usage of some of them. Note that you have to `#include <iomanip>` in order to use these manipulators.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
  double x = 12.345678;
  cout << "Floating point number" << endl;
  cout << "Default: " << x << endl;
  cout << setprecision(10);
  cout << "Precision of 10: " << x << endl;
  cout << scientific << "Scientific notation: " << x << endl << endl;
  cout << setprecision(5) << fixed << "Back to fixed notation: " << x << endl;

  cout << "With setw(35) and right: " << setw(35) << right << x << endl;
  cout << "With width(20) and left: ";
  cout.width(20);
  cout << left << x << endl << endl;

  bool b = true;
  cout << "Boolean Number" << endl;
  cout << "Default: " << b << endl;
  cout << boolalpha << "BoolAlpha set: " << b << endl << endl;

  return 0;
}
```

1.10 File I/O

Input and output to files in C++ is very straightforward. You can write to a file in exactly the same way that you write to the output stream `cout`, and similarly you can read from a file as if you read from the input stream `cin`. Take a look at the following example. (*)

```
#include <iostream>
#include <fstream>                              // include fstream for file I/O
using namespace std;

int main() {
  ofstream fout("output.txt");                  // creates an ofstream called fout
  if (! fout.is_open()) {                       // test that file is open
    cout << "Error opening output file." << endl;
    return -1;
  }
  fout << "Hello World" << endl;
  fout.close();                                 // close ofstream fout

  ifstream fin("input.txt");                    // creates an ifstream called fin
  if (! fin.is_open()) {                        // test that file is open
    cout << "Error opening input file." << endl;
    return -1;
  }
  int number;
  fin >> number;
  fin.close();                                  // close ifstream fin
  cout << "Read the number: " << number << endl;
  return 0;
}
```

Note that we closed each file stream after its use, although C++ will close all open file streams automatically at the end of the program. But it is good programming practice to close your streams yourself, also

*File I/O in C is slightly more complicated. You need the variable type `FILE *f;` and can e.g. open files for writing with the command `f = fopen("output.txt","w");`. Then use `fprintf(f,"...")` instead of `printf("...")` and close the file with `fclose(f);`.

because the machine your program is running on will slow down if too many file streams are open at the same time.
Moreover, when dealing with variable length input data files, it is beneficial to note that the insertion operator `>>` itself evaluates to *true* if the last reading operation has been successful. In C++ this is the recommended way to check for correct input, rather than using e.g. the functions `fin.eof()` and `fin.good()` for the input file stream `fin`, which return *true* when the end of a file has been reached, and when the last input via `>>` was successful, respectively.

1.11 Namespaces

So far, we have used the command `using namespace std;` in each of our programs, without really knowing why.
Namespaces are a feature of C++ and roughly speaking they group together certain variables and functions under one name. We will learn more about namespaces in the second part of this course. For now, it suffices to know that the `namespace std` groups together all the objects needed for input and output that are provided by the system library `iostream`.
It is possible to not make use of the `namespace std`. For the sake of completeness, here is what e.g. the first program in §1.4 would then look like.

```
#include <iostream>
// no 'using namespace std' ! Watch out for 'std::'.

int main() {
  int i, number;
  double x;
  double sum = 0.0;                                    // initialize sum with 0.0

  std::cout << "How many numbers do you want to add up: ";
  std::cin >> number;                                  // input an integer

  for (i = 0; i < number; i++) {                       // a for loop
    std::cout << "Input next number: ";
    std::cin >> x;                                     // input a double
    sum = sum + x;
  }
  std::cout << "Final sum: " << sum << "." << std::endl;
  return 0;
}
```

2 Variables

2.1 Basic types

For each variable that is used inside a program, a suitable type has to be found. This choice will depend on the meaning of the variable, e.g. whether it is an integer or a floating point number, but also on the required accuracy or range of the variable.
When a variable is used by a program, it occupies some part of the main memory of the machine the program is running on. The amount of space it will occupy depends on the type of the variable. It is considered good programming practice to not "waste" the computer's memory by using variable types that use more memory than is actually needed for the purpose of the variable. However, this really only plays a role if huge amounts of data are used.
The amount of space a variable occupies is measured in bytes. One byte consists of 8 bits, and hence can store up to $2^8 = 256$ different states. The following table gives an overview of the basic types, what they can store and their range of values. The list is not exhaustive.
The size of any type can be assessed with the `sizeof()` function. Note that if for input and output you want to use the C functions `printf()` and `scanf()`, you will need different format descriptors, like `%d` for `int` and `%e`, `%g` for `double`. A full list can be found in Kernighan & Ritchie, p244. See also `www.cplusplus.com/ref/cstdio/printf.html` and `www.cplusplus.com/ref/cstdio/scanf.html`.

*The type `bool` is only defined in C++.

type	bytes	range	example
`int`	4	$-2147483648 \dots 2147483647$	i = -5;
`short int`	2	$-32768 \dots 32767$	i = 4;
`unsigned short int`	2	$0 \dots 65535$	i = 55555;
`long int`	8	$-9223372036854775808 \dots 9223372036854775807$	i = -1234567890;
`float`	4	$\pm 1.4013 \times 10^{-45}$ to $\pm 3.402823 \times 10^{38}$	x = -1.23e-2;
`double`	8	$\pm 4.94065 \times 10^{-324}$ to $\pm 1.797693 \times 10^{308}$	x = 3.1e-15;
`long double`	16	$\pm 6.4752 \times 10^{-4966}$ to $\pm 1.18973 \times 10^{4932}$	x = 0.12e1054;
`char`	1	characters like 'a','Z','@'	c = 'b';
`bool` (*)	1	$0 \dots 1$ or "true", "false"	read = false;

Table 1: Basic types and their representation.

2.2 Internal representation of variables

We have seen in the previous section that each variable that is declared for a certain type occupies some space in the memory of the computer. Now we will look at exactly how the values of a variable are stored in memory.

When memory is declared for a `short int`, 16 bits (or 2 bytes) are allocated in the memory of the machine. One of these bits is used to store the sign of the integer, and the remaining bits store the digits of the integer in binary representation. Thus the largest 16bit integer which can be stored is $2^{15} - 1 = 32767$. Note that the `unsigned` version does not need to store the sign of the number, and can hence represent integers up to $2^{16} - 1 = 65535$. Analogues hold true for the 32bit integer type `int`.

A `char` is very similar to an `int`, but since on most machines the character set consists of less than 256 characters, and $2^8 = 256$, 8 bits are sufficient to store a character.

Slightly more complicated is the representation of floating point numbers. The IEEE single-precision floating point standard states that such a number should be represented by 32 bits. The first bit stores the sign of the number, and we call it s. The next 8 bits are interpreted as an integer e, called the exponent. The last 23 bits, $b_1 b_2 \dots b_{23}$, are interpreted as $0.b_1 b_2 \dots b_{23}$ in binary expansion. This is called the mantissa, and we represent it with m. The entire floating point number x is then given by

$$x = (-1)^s \, 2^{e-127} \, (1.m)_2.$$

Note that zero is treated specially, and is given by $m = e = 0$. Apart from zero, no other number is allowed to have $e = 0$. Hence the smallest positive number that can be written in the above format is defined by $m = 0$ and $e = 1$, giving 2^{-126} or 1.175×10^{-38}. However, on most systems so called *subnormal* numbers are defined. These cover the case, where $e = 0$ and $m \neq 0$. Then the implicitly defined leading 1 is changed to a 0 and the formula is adapted to

$$x = (-1)^s \, 2^{e-126} \, (0.m)_2.$$

These subnormal numbers are machine representable but are less accurate in computation than the normalizable values discussed earlier.

Double-precision numbers (type `double`) are represented in a similar way, except that both the exponent and the mantissa are represented using a greater number of bits. In particular, a `double` element has a sign bit, an 11-bit exponent and a 52-bit mantissa. So the representation for a normalizable `double` number is

$$x = (-1)^s \, 2^{e-1023} \, (1.m)_2,$$

while a subnormal number (with $e = 0$ and $m \neq 0$) is interpreted as

$$x = (-1)^s \, 2^{e-1022} \, (0.m)_2.$$

2.2.1 Precision problems and rounding errors

As a consequence of this mechanism for representing floating point numbers, the number stored in memory is only an *approximation* to its true value in the real world. Only those floating point numbers with small binary expansions can be stored precisely, for instance $\frac{1}{2}$ and $\frac{1}{8}$.

Moreover, the value which is stored depends on the way the number was computed. For example

```
(1.0 / 3.0) * 2.5 == 5.0/6.0
```

is `false`, since the floating point representation of $\frac{5}{6}$ obtained by dividing the two floating point numbers 5.0 and 6.0 is not the same as the representation which is arrived at by first taking $\frac{1}{3}$ and then multiplying by 2.5.
This example illustrates that one should *never* test floating point numbers for equality, and one should *never* rely on equality for termination of a `while` or a `for` loop. For termination tests, a strict inequality like `<` should be used, while in place of an equality test between floating point numbers one should use something like

```
if (x - y < tol && x - y > - tol)          // x and y are equal up to tol
```

where `tol` is a given tolerance, e.g. 10^{-16}.
Another precision problem is the occurrence of so called *overflows* and *underflows*. The former are events, where the maximum range of a type is reached. This can cause unpredictable outcomes for your program. Underflows, on the other hand, are events where information that was stored in floating point numbers is lost, either because the lower end of the range of the type was reached, or because a much larger value was added, say, to a small number. Since only a finite number of digits can be stored for every number, some or all digits of the smaller number will be erased and lost in the process.
The number of significant decimal digits for the discussed types are 7 digits for the type `float` and 16 digits for `double`, respectively. For `long double` this value is approximately 33.

2.2.2 Type conversion

It is clear by now that if C/C++ in an assignment involving different types simply equated the values stored in memory, this would have undesired effects. Often the amount of memory needed to store the different types is not even the same. Hence a meaningful conversion has to take place, when e.g. an `int` value is assigned to a `double` variable, and vice versa.
This is done via type conversion, and in most cases, the programmer need not worry about them. That is because the compiler invokes these conversions automatically, wherever the desired conversion is obvious. Here is a program with a list of examples.

```
#include <iostream>
using namespace std;

int main() {
  int i = 7;
  unsigned int j = i;                 // from int to unsigned int
  short int k = i;                    // from int to short
  double x = i;                       // from int to double

  cout << i << " " << j << " " <<  k << " " << x << endl;
  return 0;
}
```

In some cases, the compiler is going to give a warning, when types in an assignment do not match. Then it is possible to enforce a conversion, by using a technique called casting. Here is an example, where a `double` is cast to an `int`, and similarly a `char` is converted to an `int`.

```
#include <iostream>
using namespace std;

int main() {
  double x = 1.999;
  int i = (int) x;                    // cast from double to int
  char c = 'a';

  cout << i                           // '1'
       << " " << x                    // '1.999'
       << endl;
  cout << c                           // 'a'
       << " " << (int) c              // '97'
       << endl;
  return 0;
}
```

2.2.3 Rounding functions

As one can see from the previous example, simple type casting is usually not enough when converting floating point values to integers. To do this scientifically, one can use one of the rounding functions that are available in the C system header file `math.h`. The following program demonstrates their use.

```
#include <iostream>
#include <cmath>                          // include math.h
using namespace std;

int main() {
  double i, j;
  double x = 3.14, y = -3.14;

  i = floor(x); j = floor(y);             // largest int <= x
  cout << i << " " << j << endl;          // '3 -4'
  i = ceil(x);  j = ceil(y);              // smallest int >= x
  cout << i << " " << j << endl;          // '4 -3'
  i = round(x); j = round(y);             // round to nearest int
  cout << i << " " << j << endl;          // '3 -3'
  i = trunc(x); j = trunc(y);             // nearest int i with |i| < |x|
  cout << i << " " << j << endl;          // '3 -3'
  i = (int) (x + 0.5); j = (int) (y + 0.5); // math.h not needed for this
  cout << i << " " << j << endl;          // '3 -2'
  return 0;
}
```

Note that all of the functions `floor()`, `ceil()`, `round()` and `trunc()` return a `double` value, even though the number itself is integral. It is for this reason, that the variables `i` and `j` in the example were declared as `double`. If you want to use `int`, then — in order to avoid a compiler warning message — you would have to cast the result from the rounding function to an integer type.

2.3 Names of variables

The following names are reserved by the C language. These have predefined uses and cannot be used for any other purpose in a C program.

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
const     float   short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

In addition, the C++ language reserves the following keywords.

```
asm           export     private           true
bool          false      protected         try
catch         friend     public            typeid
class         inline     reinterpret_cast  typename
const_cast    mutable    static_cast       using
delete        namespace  template          virtual
dynamic_cast  new        this              wchar_t
explicit      operator   throw
```

Other than these names, you can choose any names of reasonable length for variables and functions. The names must begin with a letter, and then can contain letters, numbers and underscores.
It is also possible to start names with an underscore, but because the compiler and many system libraries use this for their variables and functions, this should be avoided at all costs.

2.4 Initialization of variables

In contrast to some other programming languages, it is important to know that a variable that is declared in C/C++ is by default *not* given any value. This is the task of the programmer. All that happens in C/C++ is, that the appropriate amount of memory is reserved for the declared variable. But no value is written to that place in memory. You can test this behaviour with the following small program.

```
#include <iostream>
using namespace std;

int main() {
  double x;                                          // x is not initialized
  double y = 0.0;                                    // y is initialized with 0

  cout << "The value of x is " << x << endl;         // gibberish
  cout << "The value of y is " << y << endl;         // 0.0

  x = 0.1;                                           // first assignment to x
  cout << "Now the value of x is " << x << endl;     // 0.1
  return 0;
}
```

2.5 Scope of a variable

Each variable in a C/C++ program has its scope. The scope of a variable is — roughly speaking — the area within your source code, in which you are allowed to use the variable. This means that inside the scope of a variable, you can assign values to it and read values from it. Outside the scope the variable is not defined, and you can do neither of these things.
The compiler will always warn you if you try to use a variable outside of its scope. Be also careful with variables in different parts of the code, that you have given the same name. It will always depend on the scope of the variables, which of the two variables you are actually referring to.
Basically, the scope of a variable is the block or part of the program, in which it has been declared. Remember that commands or statements are always grouped together inside some "`{ }`" brackets. Once you leave such a grouped block of commands, the variables defined in this block run out of scope.
This concept sounds more complicated than it is. Here is an example program that illustrates the concept.

```
#include <iostream>
using namespace std;

void foo(double x) {
  cout << "foo() got the value x = " << x << endl;
}                                          // the scope of x ends here

int main() {
  int i;                                   // i's scope is the whole main() program

  for (i = 0; i < 5; i++) {
    char z;                                // z only defined within for loop
    cout << "Please enter a character: ";
    cin >> z;
  }                                        // the scope of z ends here

  z = 'c';                                 // this will result in compiler error!

  {
    short int k;                           // k's scope are these 3 lines
    k = 5;
    cout << "k = " << k << endl;
  }

  return 0;
}
```

Variables, whose scope is a single user defined function, are also called *local variables*, see Chapter 5. One consequence of the above concept is, that in none of your subroutines can you directly make use of variables that were defined in the `main()` program. Variables that you want to use inside of subroutines have to be passed to them as parameters. See Chapter 5 for more details.

2.5.1 Global variables

The exceptions to the rule are so called global variables. These are variables, that have as scope the whole source file in which they are defined. That means that all the subroutines inside the source file — including the `main()` program — have access to these variables.
However, the use of global variables is considered a bad programming practice and their use should be avoided if at all possible. The main reason being that programs that make use of global variables are difficult to debug and difficult to read, since it is not immediately clear where they have been changed last, for instance.
For the sake of completeness, here is an example for the use of global variables.

```
#include <iostream>
using namespace std;

int max_number;                         // this is a global variable

void initialize() {
  max_number = 10;                      // max_number is global
}

void print() {
  int i;
  for (i = 0; i < max_number; i++)      // max_number is global
    cout << "*";
}

int main() {
  initialize();
  print();
  cout << "Global variable max_number = " << max_number << endl;
  return 0;
}
```

2.6 Arrays and strings

As mentioned in §1.7, arrays are a special form of variables in C/C++. They are the natural implementation of fixed length vectors in C. When an array like `double x[100]` is declared in C, the space for 100 double variables is reserved in the memory of the machine. As we know from §2.1, in this case this amounts to 800 bytes. The space for the array is allocated sequentially in memory, and the elements of the array can be referred to as `x[0]`, `x[1]`, ..., `x[99]`. It is important to remember that the element `x[100]` no longer belongs to the array `x`. Instead, it points to the first 8 bytes in memory *after* the array `x`. This could be anything from other data of your program, your program code itself, or data from another program that is running on the machine at the time. Recall the program in §1.7 for an example on how to use arrays.

2.6.1 Initialization of arrays

As is the case with normal variables, by default arrays are not initialized when they are declared. The compiler only allocates the necessary space in memory, without checking or changing the contents of that part of the memory.
However, as with variables it is possible to initialize arrays with meaningful values, if this is what the programmer wants. Check the following example for the syntax.

```
#include <iostream>
using namespace std;
```

```
void print(double a[], int length) {
  int i;
  for (i = 0; i < length; i++)  cout << a[i] << "   ";
  cout << endl;
}

int main() {
  double a[5] = {1, 0.1, 3.5, 0.7, -0.5};
  double x[10];                                    // not initialized
  double y[10] = {1, 2, 3};                        // '1 2 3 0 0 0 0 0 0 0'
  double z[10] = {};                               // '0 0 0 0 0 0 0 0 0 0'
  cout << "a = "; print(a, 5);
  cout << "x = "; print(x, 10);
  cout << "y = "; print(y, 10);
  cout << "z = "; print(z, 10);
  return 0;
}
```

Note that you can specify every entry of the array when initializing it. If you give less values than the array is long, the remaining entries are initialized with 0. The arrays `y` and `z` in the above program are an example for this.
Finally, the function `print()` in the example above demonstrates how one can pass arrays to a function. More details on that will follow in §5.3.

2.6.2 Strings in C

A special type of arrays in C are arrays of type `char`. They are also called strings. They work exactly the same way as all the other arrays in C. In particular, you need to be careful to not write more characters to a string, than it was declared for. This would cause your program to terminate with an error.
There are many predefined functions in C that let you manipulate and work with strings. For example `strcat` to concatenate two strings, `strcpy` to copy a string, `strcmp` to compare two strings lexicographically, `strchr` to find the first occurrence of a character in a string and `strstr` to find the first occurrence of a string in another string. For some of these functions you need to have knowledge of *pointers*, see Chapter 6 for details.
Internally, C marks the end of a string with the `\0` termination character. This is necessary, because the length of the string and the memory that is allocated for it can be different. All of the above functions rely on this termination character to work properly. When initializing a string with a sequence of characters, C will automatically append the termination character `\0` to the end.
Note also that the format descriptor for the `printf()` function for a string is `%s`. See the example program below.

```
#include <iostream>
#include <cstring>                          // include string.h for C strings
#include <cstdio>                           // include stdio.h for printf
using namespace std;

int main() {
  char s[10] = "Hi there.", r[20];          // C strings
  char *t;                                  // pointer to char

  cout << "Second character of s is '"      // [] works as for other arrays
       << s[1] << "'." << endl;
  strcpy(r, s);
  cout << "r = " << r << endl;              // "Hi there."
  strcat(r, s);
  cout << "r = " << r << endl;              // "Hi there.Hi there."
  t = strstr(r, "re.");
  cout << "t = " << t << endl;              // "re.Hi there."

  printf("Value of string s = '%s'\n",s);   // "Hi there."

  strcpy(s, r);                             // causes program termination
```

```
                                          // because r is longer than s
  return 0;
}
```

We have seen in the previous example, that handling C strings can be difficult and that it can lead to an unwanted termination of the program. A safer way to use strings is offered by C++. We will look into this in more detail in Part II of this course. For now we provide an equivalent program to the above, that uses the C++ string class.

```
#include <iostream>
#include <string>                         // include string for C++ strings
#include <cstdio>                         // include stdio.h for printf
using namespace std;

int main() {
  string s = "Hi there.", r, t;                   // C++ strings

  cout << "Second character of s is '"            // [] works as for other arrays
       << s[1] << "'." << endl;
  r = s;
  cout << "r = " << r << endl;                    // "Hi there."
  r = r + s;
  cout << "r = " << r << endl;                    // "Hi there.Hi there."
  t = r.substr(r.find("re."), r.length());
  cout << "t = " << t << endl;                    // "re.Hi there."

  printf("Value of string s = '%s'\n",s.c_str()); // Pass s as a C string

  s = r;                                          // works fine with C++ strings
  return 0;
}
```

2.6.3 Multidimensional arrays

In the same way that a standard array is an array with entries from a basic type, one can also define an array, whose entries are made up of standard arrays. Similarly, one can define an array of an array of an array, and so on. These arrays are called multidimensional arrays. Two dimensional arrays are the natural implementation of mathematical matrices, whereas higher dimensional objects can be implemented with higher dimensional arrays.
The following program includes some examples.

```
#include <iostream>
using namespace std;

int main() {
  int i,j,k,l;
  double A[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};       // matrix A
  double x[3] = {10, 12, -6};                               // vector x
  double y[3];                                              // vector y
  double M[2][2][2][2];                                     // 4 dimensional
  for (i = 0; i < 3; i++)
    for (y[i] = j = 0; j < 3; j++)
      y[i] += A[i][j] * x[j];
  cout << "y = A*x = ";
  for (i = 0; i < 3; i++)
    cout << y[i] << "  ";
  cout << endl << "M = ";
  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
        for (l = 0; l < 2; l++) {
          M[i][j][k][l] = 1000*i + 100*j + 10*k + l;
          cout << M[i][j][k][l] << "   ";
        }
```

```
  cout << endl;
  return 0;
}
```

2.7 Constants

A constant is similar to a variable in the sense that it has a type and it represents a memory location. The difference is, of course, that it cannot be reassigned a new value after initialization. I.e. you cannot change its value after the constant was declared and initialized.
In general, constants are a useful feature that can prevent program bugs and logic errors. Unintended modifications to a supposedly constant value in your program are prevented from occurring. The compiler will catch attempts to reassign new values to constants.
In order to declare a constant, you only have to add the keyword `const` to the definition. This will turn a variable into a constant. For instance,

```
const double pi = 3.14159265;
```

will create the constant `double` value `pi`. You can now use the name `pi` anywhere in the scope of this constant, instead of the numerical value 3.14159265.
Note that in C++ you can use integer constants to define the length of an array. (*) However, if you work with C compilers, this does not work. That is because in many ways, the compiler still treats constants as variables. So while a constant is really only initialized once the program is executed and that line of code is reached, the dimension of an array, on the other hand, has to be known at compile time already.
An alternative way to have a constant expression in your source code that defines the length of an array, that also works in C, is to use the `#define` preprocessor directive. The preprocessor is a program that modifies your source file just prior to compilation. Another preprocessor directive that we have already seen is the `#include` directive. The `#define` directive is used to define macros, e.g. as follows.

```
#define LENGTH 100
```

Now, wherever the macro `LENGTH` appears in your source file, the preprocessor replaces it by its value. So, every "`LENGTH`" in your source code will be replaced by "100". The compiler will only see the value 100 in your code, not "`LENGTH`".
Here is an example program to demonstrate the use of the described techniques.

```
#include <iostream>
using namespace std;

#define LENGTH 100                                  // define macro LENGTH

int main() {
  int i;
  const double pi = 3.14159265;
  double x[LENGTH];                                 // LENGTH used here

  for (i = 0; i < LENGTH; i++) {                    // LENGTH used here
    x[i] = pi*i;
  }
  for (i = 0; i < 100; i++)
    cout << i << ". x = " << x[i] << endl;

  return 0;
}
```

3 Control flow

Most of the control flow statements we have already come across by now. See §1.5 for a short introduction. We will now look at all of the control flow constructs available in C/C++ in a bit more detail.
Any control flow construct enables you in some way to control how the execution of the program flows from one instruction to the next, hence their name.

*This only works in C++. In C you need to use macros. Some C++ compilers, e.g. `g++`, even allow you to use integer *variables* to define the length of an array. But this is a nonstandard addition. `g++ -pedantic` deactivates this.

3.1 The `while` loop

The `while` loop is the simplest loop in C/C++. The syntax is

```
while (continuation) {
  instructions;
}
```

and the `instructions` inside the loop are executed until the boolean expression `continuation` is false. Note that the `while` loop will not be executed at all, if this expression is `false` on encountering the `while` loop for the first time.
Here is a short example program for a `while` loop.

```
#include <iostream>
using namespace std;

void important_stuff() {
  // some important code
}

int main() {
  char answer;
  cout << "Do you want to start the program? (y/n) : ";
  cin >> answer;

  while (answer == 'Y' || answer == 'y') {
    important_stuff();
    cout << "Do you want to continue? (y/n) : ";
    cin >> answer;
  }
  return 0;
}
```

3.2 The `for` loop

The `for` loop can be viewed as a specialized version of the `while` loop. The syntax is

```
for (initialization; continuation; increment) {
  instructions;
}
```

and similar to the `while` loop, the given `instructions` are repeated, until the `continuation` evaluates to `false`. In addition, this loop allows the programmer to initialize a variable or several variables before the first execution of the loop. Furthermore, the programmer can execute certain statements after each execution of the loop, before the next evaluation of `continuation`.
Both `initialization` and `continuation` may include more than one statement. In this case, different statements are separated by commas.
Here is an example.

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  double sum;
  int i;

  for (i = 0, sum = 0.0; i <= 100; cout << i << endl, i++) {
    sum += pow(2, - (double) i);
  }

  cout << "The sum of 1/2^i for i = 0,...,100 is " << sum << endl;

  for (int j = 10; j > 0; j--) {
    cout << j << endl;
```

```
  }                                                    // scope of j ends here
  return 0;
}
```

The second example shows another important feature of the `for` loop in C++. It is possible, to declare a variable inside the `initializations`, that is only defined inside the `for` loop. That means, that the variable will run out of scope (see §2.5) as soon as the `for` loop is finished. This should be the preferred way of defining counter variables for `for` loops, as this prevents any confusion with other counter variables you might have defined elsewhere in your code. (*)

3.3 The `do-while` loop

The syntax for the `do-while` statement is

```
do {
  instructions;
} while (continuation);
```

There is little difference between `do-while` and `while`. The only difference is that in the `do-while` loop the `continuation` condition is evaluated at the end of the loop, rather than at the beginning. This means that the `instructions` enclosed between the braces are always executed at least once.
Here is an example, where a `do-while` loop makes sense.

```
#include <iostream>
#include <iomanip>
using namespace std;

double f(double x) { return x + (2-x*x) / 2.0 / x; }

int main() {
  double x_old, x_new, tol = 1e-8;
  cout << "Enter an initial guess : "; cin >> x_new;
  do {
    x_old = x_new;
    x_new = f(x_old);
    cout << setprecision(10) << x_new << endl;
  } while (x_new - x_old > tol || x_old - x_new > tol);

  cout << "The square root of 2 is approximately equal to " << x_new << endl;
  return 0;
}
```

This works well as a `do-while` loop, because one evaluation of the function `f()` is inevitable.

3.4 The `if-else` statement

This is perhaps the most common of all control flow statements. It is also one of the simplest. The syntax is

```
if (expression) {
  instructions;
}
else {
  alternative_instructions;
}
```

If `expression` evaluates to *true*, then `instructions` are executed, otherwise `alternative_instructions`. If the `else` part is not supplied, then no action is taken unless `expression` evaluates to *true*.
The `if-else` statement can be joined together with other `if-else` statement to form bigger constructs like

```
if (expression1) {
  first_instructions;
}
```

*This is not possible in C. In C you have to define the counter variable used in the `for` loop *before* the loop itself.

```
else if (expression2) {
  second_instructions;
}
else if (expression3) {
  third_instructions;
}
else {
  default_instructions;
}
```

Here `first_instructions` are executed if `expression1` is *true*. If it is *false*, however, and only if it is *false*, `expression2` is tested. If `expression2` is *true*, then `second_instructions` are executed, otherwise `expression3` is tested and so on. In any case, only one set of instructions will be executed. No `else` actions are investigated once an `expression` has evaluated to *true*.
Note here and throughout, that the boolean expression `true` and `false` are simple placeholders for the integer values 1 and 0. Moreover, the C/C++ compiler treats any integer value different from 0 as *true*.
Here is a small example program for the `if-else` statement.

```
#include <iostream>
using namespace std;

int main() {
  int a, b;
  cout << "Enter two integers : ";
  cin >> a >> b;
  if (a < b)
    cout << "The smaller number is a = " << a << endl;
  else if (b < a)
    cout << "The smaller number is b = " << b << endl;
  else
    cout << "The numbers are equal a = b = " << a << endl;
  return 0;
}
```

Note that if the instructions to be executed inside the `if` or `else` branch only consist of a single statement, the curly brackets can also be omitted.

3.5 The `switch` statement

In certain situation one can avoid heavily nested `if-else` statements as above with the `switch` statement. However, this applies only if one wants to test one integer expression for equality with several constant values. The syntax is as follows.

```
switch (expression) {
  case constant-expression1 : statements1;
  case constant-expression2 : statements2;
  ...
  case constant-expressionn : statementsn;
  default : default_statements;
}
```

The meaning is best explained with an example. Take the following `switch` statement, where the variable `grade` is of type `char`.

```
switch (grade) {
  case 'A' : cout << "Excellent" << endl;
  case 'B' : cout << "Good" << endl;
  case 'C' : cout << "OK" << endl;
  case 'D' : cout << "Mmmmm...." << endl;
  case 'F' : cout << "You must do better than this" << endl;
  default  : cout << "Cannot recognize grade." << endl;
}
```

Here, if the `grade` is `'A'` then the output will be

```
Excellent
Good
OK
Mmmmm....
You must do better than this
Cannot recognize grade.
```

This is because in the C `switch` statement, once a positive match has been made, execution continues on into the next case clause if it is not explicitly specified that the execution should exit the switch statement at that point. The correct syntax for that would be:

```
switch (grade) {
  case 'A' : cout << "Excellent" << endl;
             break;
  case 'B' : cout << "Good" << endl;
             break;
  case 'C' : cout << "OK" << endl;
             break;
  case 'D' : cout << "Mmmmm...." << endl;
             break;
  case 'F' : cout << "You must do better than this" << endl;
             break;
  default  : cout << "Cannot recognize grade." << endl;
             break;
}
```

The `break` command tells your program to leave the `switch` statement after executing the instructions for the relevant case. Note that the `break` command is not really necessary for the `default` clause (or the last clause in general), but it is good programming practice to put it in anyway.
There is an important instance, where it makes sense *not* to put a `break` command. That is, when you want to execute the same instructions for several of the constant expressions that you test your variable for.
The following program includes an example for that.

```
#include <iostream>
using namespace std;

int main() {
  int i;
  cout << "Enter a number < 10 : "; cin >> i;
  if (i > 0 && i < 10)
    switch (i) {
      case 2 :
      case 3 :
      case 5 :
      case 7 : cout << "You have entered a prime number." << endl;
               break;
      default: cout << "The number you entered is not prime." << endl;
    }
  else
    cout << "Your number was out of range." << endl;
  return 0;
}
```

3.6 `break` and `continue`

The main usage of the `break` statement, as we have already seen, is for the `switch` construct. But it can be used similarly in any of the loops that we have come across. It will cause the program flow to exit the loop immediately, and continue with the first statement after the loop.
It is almost always easily possible to avoid using `break`, and this will make your code easier to read and easier to understand. However, sometimes it might be the most natural choice, especially when dealing with very large loops.
The `continue` statement is similar to the `break` statement, in that it also terminates the body of a loop immediately. But, whereas the `break` statement causes the program flow to jump out of the loop,

`continue` simply finishes this cycle of the loop, ready to start the next one. You can think of this as jumping to just after the last statement inside the loop, but staying inside the loop itself.
So, `continue` in a `while` loop returns to the head of the loop and checks to see whether the test is still *true* and, if it is, resumes execution at the start of the loop. In a `do-while` loop, it jumps to the foot of the loop where the logical test is performed, and in a `for` loop, it jumps to the foot of the loop, in the sense that the `increment` expression is executed before the next logical test of `continuation`.
Also the `continue` statement should be used with care. It can often make sense in `for` loops, if there are some members in the processed sequence that are not required. In all other loops, however, using `continue` is not advisable, as it can lead to infinite loops.
We round this section off with another example program.

```
#include <iostream>
using namespace std;

int main() {
  double temp = 0.0;
  char answer;
  while (temp < 37.0) {
    cout << "Do you want to continue? (y/n) : ";
    cin >> answer;
    if (answer == 'n' || answer == 'N')
      break;                                                    // break
    cout << "Give me your temperature : ";
    cin >> temp;
    if (temp >= 37.0)
      cout << "You have temperature and should see a doctor." << endl;
  }

  cout << "Which number is missing here ... ?" << endl;
  for (int i = 0; i < 10; i++) {
    if (i == 5) continue;                                       // continue
    cout << i << "  ";
  }
  return 0;
}
```

4 Operators

4.1 Arithmetic operators

C/C++ supports five binary arithmetic operations, which are useful when doing scientific programming. They are addition (`+`), multiplication (`*`), subtraction (`-`), division (`/`) and modulus (`%`).
The operators `+,*,-` evaluate the sum, the product and the difference of two numbers, where the two operands can be of integer or floating point type. If the two operands are not of the same type, the compiler will still attempt to do something sensible. Usually this will involve one of the operands to be type cast to the type of the other operand, see §2.2.2. Although the compiler will perform type casts automatically to avoid conflicts, the only unambiguous way is to perform the type cast explicitly. This is done by explicitly stating the type which you want the variable to be cast to.
This is especially important when it comes to the devision operator `/`. The result of `a/b` depends on the types of `a` and `b`. If `a` and `b` are of floating point type, then `a/b` evaluates to `a` divided by `b`. However, if `a` and `b` are of integer type, then `a/b` evaluates to the integer part of `a` divided by `b`. For example, if `a`= 14 and `b`= 4, then `a/b` evaluates to 3, since $\frac{14}{4} = 3.5$. If we want `a/b` to evaluate to 3.5, we must explicitly cast `a` and `b` into e.g. `double`s before performing the devision.
The following program gives some examples of what can happen when using devision in C/C++.

```
#include <iostream>
using namespace std;

int main() {
  int a,b,i;
  double x[7];
```

```
  a = 17;
  b = 8;
  x[0] = 1 / 2;                    // '0'          (integer)
  x[1] = 1.0 / 2;                  // '0.5'
  x[2] = 1 / 2.0;                  // '0.5'
  x[3] = a / b;                    // '2'          (integer)
  x[4] = (double) (a / b);         // '2'          (integer result cast to double)
  x[5] = (double) a / b;           // '2.125'
  x[6] = a / (double) b;           // '2.125'
  for (i = 0; i < 7; i++)
    cout << x[i] << endl;
  return 0;
}
```

Note that type cast operations take precedence over binary operations, and hence `x[5]` is evaluated to the floating point number 2.125.
Finally, `a%b` evaluates to `a` modulo `b`, e.g. `14%4` evaluates to 2 since $14 \equiv 2 \pmod 4$. For this operator, `a` and `b` must be of integer type. It is an error for either of them to be of any other type. Sensible results are not guaranteed if either `a` or `b` are negative. An error will occur if `b` is zero.

4.2 Precedence of operators

We have seen in the example above that, just as in mathematics, the precedence of operators is important in determining the result, and brackets can be used to force a different precedence from the default. Table 2 gives a complete list of operators showing their precedence. Not all of the operators have been discussed yet.
Operators at the top of the table have a higher precedence than those lower down.
When in doubt, use brackets to indicate what you intend to happen. The *associativity* indicates in what order operators of equal precedence in an expression are applied. Operators of equal precedence appear between horizontal lines.

4.3 Assignment operator

In many of our examples we have already seen the assignment operator `=`. As discussed earlier, in general the types of the expression on the right hand side and the variable on the left hand side should be the same. When this is not the case, the compiler tries to use some form of type casting, see §2.2.2.
It is worth mentioning that the assignment operator in C/C++ is treated like any other operator, which means that in particular it evaluates to a result. For instance, the assignment

```
a = b;
```

in addition to setting `a` equal to `b`, actually evaluates to `b`. In this particular example, the result is discarded, but one can use this feature of C/C++ in multiple assignments like

```
a = b = c;
```

Here, `a` is set equal to the result of the assignment `b = c`, while the assignment `b = c` sets `b` to take the value of `c` and evaluates to `c`. Hence `a` is also set equal to `c`.

4.3.1 Special assignment operators

Arithmetic operations are often combined with an assignment, e.g. `a = a + b; x = 2 * x / y;`. C/C++ provides a special type of operator that combines arithmetic operations with some kinds of these types of assignments.
These new operators are `+=`, `-=`, `*=`, `/=` etc. The general rule is that statements of the form

```
x #= y;
```

where `#` is a binary operation such as `+`, are interpreted as

```
x = x # y;
```

so that, for example `x += 3` is equivalent to `x = x + 3`.
The result is slightly more efficient code, because using `x += 3` the machine simply evaluates the right hand side and adds it to the left, whereas `x = x + 3` involves making a copy of `x`.

Operator	Description	Associativity
`()`	Parentheses (grouping)	left-to-right
`[]`	Brackets (array subscript)	
`.`	Member selection via object name	
`->`	Member selection via pointer	
`++ --`	Unary preincrement/predecrement	right-to-left
`+ -`	Unary plus/minus	
`! ~`	Unary logical negation/bitwise complement	
`(type)`	Unary cast (change type)	
`*`	Dereference	
`&`	Address	
`sizeof`	Determine size in bytes	
`* / %`	Multiplication/division/modulus	left-to-right
`+ -`	Addition/subtraction	left-to-right
`<< >>`	Bitwise shift left, Bitwise shift right	left-to-right
`< <=`	Relational less than/less than or equal to	left-to-right
`> >=`	Relational greater than/greater than or equal to	
`== !=`	Relational is equal to/is not equal to	left-to-right
`&`	Bitwise AND	left-to-right
`^`	Bitwise exclusive OR	left-to-right
`|`	Bitwise inclusive OR	left-to-right
`&&`	Logical AND	left-to-right
`||`	Logical OR	left-to-right
`? :`	Ternary conditional	right-to-left
`=`	Assignment	right-to-left
`+= -=`	Addition/subtraction assignment	
`*= /=`	Multiplication/division assignment	
`%= &=`	Modulus/bitwise AND assignment	
`^= |=`	Bitwise exclusive/inclusive OR assignment	
`<<= >>=`	Bitwise shift left/right assignment	
`,`	Comma (separate expressions)	left-to-right

Table 2: Operator precedences

4.4 Logical operators

By now we have come across most of the logical operators in C/C++. You can check Table 2 for a complete list of relational and logical operators.
The following program shows some examples of these operators being used, as well as the usage of the variable type `bool`.

```
#include <iostream>
using namespace std;

int main() {
  bool small;
  double x = 1.2, y = 1.1;
  small = 2 < 3;                              // store boolean value

  if (small && x > y) {                       // logical AND
    if (y > 0.5 || x < -3)                    // logical OR
      cout << "Then." << endl;
  }
  if (!small) cout << "Not." << endl;         // logical NOT
  cout << small << endl;                      // '1'
  return 0;
}
```

The compiler converts any logical expression internally into an integer number, either 0 (`false`) or 1

(`true`). This means, that these predefined values for `bool` variables are simple placeholders for the two numbers 0 and 1. Moreover, any integer number different from 0 is also treated as `true`. Hence, one has to be careful when using the equality operator `==` and not confuse it with the assignment operator `=`. Here is an example for this classic mistake.

```
#include <iostream>
using namespace std;

int main() {
  int a = 1, b = 2;

  if (a == b)                                          // equality test
    cout << "This will not be printed." << endl;
  if (a = b)                                           // assignment !
    cout << "This will be printed." << endl;
  return 0;
}
```

Note that the last line will be printed because "`a = b`" evaluates to 2, which is interpreted as `true`.

4.5 Unary operators

In addition to the binary operators we have looked at so far, there are also some unary operators. A unary operator is an operator that takes only one argument or operand. A familiar example is the `-` operator, as in `x = - y;`, which simply negates the expression which follows it.
There are two other arithmetic unary operations defined in C/C++: the increment operator `++` and the decrement operator `--`. The former adds 1 to the operand and is often used in `for` loops, while the latter subtracts 1.
Like all the other operators, `++` and `--` also evaluate to a result after they have performed the addition/-subtraction. Now, the result of this evaluation depends on whether the operator is used in the postfix notation, as in `y = x++;` or in the prefix notation, as in `y = ++x;`. In the first case, 1 is added to `x` only *after* the value of `x` was assigned to `y`. In the prefix case, however, the increment takes place first, followed by the assignment to `y`.
The following program demonstrates this behaviour.

```
#include <iostream>
using namespace std;

int main() {
  int i = 0, j;
  j = i++;
  cout << i << " " << j << endl;          // '1 0'
  j = ++i;
  cout << i << " " << j << endl;          // '2 2'
  return 0;
}
```

Other unary operators include the type cast operator `(type)` and the logical "not" operator `!`, see Table 2.

5 Functions

In this chapter, we will take a closer look at how to use functions in C. Although many programs can be written with a single `main()` function, i.e. without defining separate functions, it is usually better to split up the code into a number of different parts, each one performing a well defined task. This will enable you to modify the code easily, adapt it to different problems and understand readily what it is doing.
Consider the following code to integrate the ordinary differential equation $\frac{dy}{dx} = x + \frac{\sin(y)}{x}$ from $x = 1$ to $x = 2$ with initial condition $y(1) = 1$.

```
#include <iostream>
#include <cmath>
using namespace std;
```

```
/* Solve ODE   dy/dx = f(x,y)   with forward Euler */

double f(double x, double y)
{ // Take x and y and return f(x,y)
  double val;
  val = x + sin(y) / x;
  return val;
}

int main() {
  int i, n = 1000;
  double x, y, dx;
  dx = 1.0 / n;
  x = y = 1.0;
  for (i = 0; i < n; i++) {
    y += dx * f(x,y);
    x += dx;
  }
  cout << "y(" << x << ")" << " = " << y << endl;
  return 0;
}
```

In this example, the user has defined the function `f()`, that represents the right hand side of the ODE. This makes it easy to identify the part of the code that needs to be changed, in order to adapt the program to a slightly different ODE. Similarly, one could define a function for the integration scheme itself and thus provide the possibility to adapt the code to other schemes like Euler backward, or Runge-Kutta methods.
Notice that before the user defined function `f()` can be used in the `main()` function, it has to be declared. In our example, we have combined the declaration of the function with the implementation of the function itself. There is another way of doing this, see §5.7 for details.
The declaration of the function comes down to the line

```
double f(double x, double y)
```

and is here followed be the implementation of `f()`. The declaration of a function has to state the number and the type of arguments it takes, as well as the type of the variable it returns. Functions that do not return a value have to be declared as `void`.
We must now examine in detail the behaviour of the program when the function is called. Although these details appear technical, their understanding is crucial when starting to do non-trivial C/C++ programming.
The line `y += dx * f(x,y);` is interpreted by the compiler as "pass the values of `x` and `y` to the function `f()`, give control of the program to the function `f()`, then take the value returned from function `f()`, multiply it by `dx` and add it to `y`".
When control passes to the function `f()`, the compiler interprets the function as "accept the values of two `double` numbers from the calling function, store these values in two *local variables* `x` and `y`, then introduce another local variable `val`, assign to this variable the value of `x + sin(y) / x`, and finally return the value of `val` to the calling function."
Note that the scope (refer to §2.5) of the variable `val` is the function `f()` itself. That means, this variable is local to that function. In particular, the `main()` function has no knowledge of that variable, and the only way that the function `f()` can pass the value of that local variable to the `main()` function is via the `return` statement. In a similar way, the variables inside the `main()` function are local to that function, and the only way the function `f()` can get information about them, is via the parameters that are passed to it.
There are two important concepts here that need to be well understood.

5.1 Passing by value

The parameters of the function `f()` in the previous example are passed *by value*. That means, that it is the values of `x` and `y` only that are passed from `main()` to the function `f()`. In particular, the function `f()` can only read the values of the two arguments, but it cannot change the values of the two variables `x` and `y` in the `main()` function.

To ensure this, the compiler forces the function `f()` to only work on *copies* of the variables `x` and `y`. Hence, upon execution of the function `f()`, the function creates a local copy for each of the parameter variables that are passed to it, then it assigns the values of the outside parameters to these local copies and henceforth works on these copies. As a consequence, the values of the variables in the function call from `main()` remain unchanged.

5.2 The `return` statement

The `return` statement has a few special features. When a function is called, execution starts at the first line of the function and continues until either a `return` statement is reached, or until the last line of the function is reached. This means that we can leave a function before the last line, if we use a `return` statement before.
Consider the following example.

```
#include <iostream>
using namespace std;

double sign(double x) {
  if (x > 0)  return 1.0;
  if (x < 0)  return -1.0;
  return 0.0;
}

int main() {
  double a = -1.349;
  cout << "The sign of a is: " << sign(a) << endl;
  return 0;
}
```

Note that the function `sign()` can be defined without using any `else` statement, simply because the `return` statement signifies the end of the function, and hence the remaining commands will only be executed if the condition in the `if` statement was `false`.
A function may return no value. In this case, the function must be declared to be of type `void`. For a `void` function, a `return` statement is optional. If it is used, the syntax is "`return;`".
Any function `foo()`, no matter whether it returns a value or not, can be called with the syntax

```
foo(var1,var2,...);
```

If you want to use the return value of the function, the corresponding syntax is

```
a = foo(var1,var2,...);
```

Here `var1,var2,...` are the arguments which `foo()` requires, and the return value is assigned to `a`.

5.3 Passing arrays

A notable exception to the concept of passing by value are arrays. When passing arrays to a function in a fashion that looks like they are passed by value, in reality a concept called *pointers* is used, see Chapter 6 for details. For now it suffices to know that when arrays are passed in this way, *no* local copy of the array is created and hence any change that is done to the elements in the array inside the function, will take effect outside the function as well. I.e. the function can write directly do the space in memory, that is allocated for the array.
The syntax for passing the array `double x[100]` to a function `void foo()`, say, is either

```
void foo(double x[100])
```

or

```
void foo(double x[])
```

In the first case, the function `foo()` can only be called for arrays of length 100, whereas in the second case arrays of any length can be used.
The following example program illustrates this. Note also that the array `y` is changed by the function call `add_to(y,x)`.

```
#include <iostream>
using namespace std;

void print(double a[], int length) {                    // variable length arrays
  for (int i = 0; i < length; i++)  cout << a[i] << "   ";
  cout << endl;
}

void add_to(double a[5], double b[5]) {                 // fixed length arrays
  for (int i = 0; i < 5; i++)  a[i] += b[i];
}

void print_matrix(double A[3][3]) {                     // dimensions have to be fixed
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++)  cout << A[i][j] << "   ";
    cout << endl;
  }
}

int main() {
  double x[5] = {1, 0.1, 3.5, 0.7, -0.5};
  double y[5] = {};
  print(y, 5);                                          // '0 0 0 0 0'
  add_to(y, x);
  print(y, 5);                                          // '1 0.1 3.5 0.7 -0.5'

  double M[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  // 2 dimensional matrix
  print_matrix(M);
  return 0;
}
```

Note that when passing multidimensional arrays to a function, all of the dimensions (apart from the first one) have to be fixed in the declaration of the function. You cannot pass multidimensional arrays of variable lengths to a function, as is the case for one dimensional arrays.
In Chapter 6 we will study a way to pass matrices of variable length to a function. Instead than using arrays, this technique will make use of pointers.

5.4 Passing by reference

Passing by reference is really a feature of C++ and is not supported by C. Nevertheless, the concept is close to "Passing by pointer" (see §6.2) and will play a major role in Part II of this course. (*)
A reference can only be defined for an existing variable and it basically holds the information, where in memory your variable lives. This then enables you and your functions to access that variable, without actually using the variable name itself. This is particularly important when passing variables to a function, that you want to have changed by the function. As we have seen, you cannot pass the variable by value to achieve this. But if you simply pass the reference of the corresponding variable, then the function will be able to change the value of the variable.
The syntax for passing a `double` by reference to the function `void foo()`, say, is

```
void foo(double &x)
```

Then, inside the function implementation, you can use the passed variable name as if it was passed by value. However, any change to the variable inside the function will take effect also outside the function.
The syntax for calling the above function `void foo()` is the same as calling a function where the variable is passed by value, i.e.

```
foo(x);
```

Here is an example program to illustrate the concept of passing by reference.

```
#include <iostream>
using namespace std;
```

*Passing by reference is not possible in C.

```
void add_to_byvalue(double x) { x++; }            // does not change x outside
void add_to_byreference(double &x) { x++; }       // does change x outside

int main() {
  double z = 4.5;
  add_to_byvalue(z); cout << z << endl;           // '4.5'
  add_to_byreference(z); cout << z << endl;       // '5.5'
  return 0;
}
```

Note that the function where `z` is passed by value does not change the value of `z`. Only the function where `z` is passed by reference does change the value, in this case adding 1 to it.

5.5 Keyword `const`

We have seen that both variables that are passed by reference and arrays that are passed to functions can be changed inside the function. In certain situations it might be desirable to prevent any changes to variables even though they are passed in this way.
Using the keyword `const` allows for this. Once placed before the type definition of a parameter to a function, the compiler will reject any statement inside the function that tries to change the value of this parameter. For example, on defining

```
void foo(const double &x)
```

the compiler will not allow statements like `x += 1.0;` inside the function `foo()`. The same holds true for arrays, that is

```
void foo(const double x[], int n)
```

will not allow statements like `x[1] = 4.1;` inside `foo()`.
The following example program demonstrates this.

```
#include <iostream>
using namespace std;

void print_array(const double a[], int length) {          // const array
  for (int i = 0; i < length; i++)  cout << a[i] << "   ";
  cout << endl;
  a[3] = 1.0;                                             // FORBIDDEN !
}

void print_by_ref(const double &x) {                      // const reference
  cout << x << endl;
  x -= 0.5;                                               // FORBIDDEN !
}

int main() {
  double x[5] = {1, 0.1, 3.5, 0.7, -0.5};
  double z = 3.1;
  print_array(x, 5);
  print_by_ref(z);
  return 0;
}
```

Note that the program as it stands will not compile.

5.6 Static variables

Up to now we have seen that local variables defined inside functions run out of scope once the function is left by the program. The next time the function is used inside the program, the local variables will be created and initialized again, as if it was the first time the function was called.
Sometimes it can be desirable to keep track of certain things from one function call to the next. This is where `static` variables come in. Any local variable inside a function can be declared as `static`. That means, that the function will remember the value of that variable from one function call to the next.

It also means that the variable is only initialized once, during the first call to the function. On any subsequent call the variable will have the value it had when the function was last terminated.
Here is an example.

```
#include <iostream>
using namespace std;

void foo() {
  int a = 0;                                    // a gets reset every time
  static int b = 0;                             // b is static, initialized only once
  a++; b++;
  cout << a << " " << b << endl;
}

void clever() {
  static bool first = true;                     // detects if function was called before
  if (first) {
    cout << "You have called me for the first time!" << endl;
    first = false;
  }
  else
    cout << "You have called me before." << endl;
}

int main() {
  foo();                                        // '1 1'
  foo();                                        // '1 2'
  foo();                                        // '1 3'
  clever();                                     // 'first time'
  clever();                                     // 'called before'
  return 0;
}
```

5.7 Declaration of functions

So far, we have always declared a function together with its definition or implementation. In general, this does not always have to be the case.
In order to compile your source code, the compiler only needs to know the declaration of your function, i.e. the name, return type and the parameters it takes. The implementation of the function is only needed once the executable is produced (this process is called *linking*) and can be given elsewhere, for instance in another file. We will discuss this in more detail when we learn about header files, see §8.4.
Another possibility is to give the implementation of the function in the same source file, but *after* the `main()` program. The following example demonstrates this.

```
#include <iostream>
using namespace std;

void welcome();                     // here the functions are only declared
int input();
void goodbye(int );                 // no variable name needs to be given

int main() {
  int i;

  welcome();
  i = input();                      // functions are called as usual
  goodbye(i);

  return 0;
}

/*************************************************/
/* The implementations of the functions follow here */
```

```
/************************************************/

void welcome() {
  cout << "Hi there." << endl;
}

int input() {
  int in;
  cout << "Please enter a number: ";
  cin >> in;
  return in;
}

void goodbye(int number) {
  cout << "You typed in: " << number << "." << endl;
  cout << "Thanks and goodbye!" << endl;
}
```

6 Pointers

We have seen so far that any variable that is declared in C/C++ occupies a certain amount of space in memory. In order to read from and write to that space in memory, we can use the variable name and assign values to it or read values from it.
On the other hand, every variable has its unique address in memory, where its values are stored. This address will only be fixed at run-time, at the exact time when the variable is declared and when the memory is allocated for it. As long as the program remains within the scope of the variable, this address will not change. Once the variable has run out of scope, the memory that was allocated to it will be freed, and the old address will be no longer valid. That is, the part of memory that the address points to is no longer guaranteed to hold meaningful information.
So another way of changing the variable's value is to write directly to the part of memory that is allocated to it. Because we can get hold of the variable's address, we can read from and write to that particular part of memory in the knowledge that this will always correspond to the original variable.
In the following we will learn exactly how to achieve that.

6.1 Pointer syntax

A pointer can hold the address of a variable. Since it is important in C/C++ to distinguish between variables of different types, and since it is even more important to know the variable's type before trying to write values to it, there is a unique kind of pointer for every variable type.
The syntax to declare a pointer for a certain type is

```
type *pointer_name;
```

This will create a pointer named `pointer_name` to `type`. You can also think of `pointer_name` as a variable of type `type*`.
A pointer can only hold the address of a variable of `type`. To assign the address of a variable to a pointer, you need the address operator `&`. Moreover, in order to read the value of a variable with the help of the pointer, you need the dereferencing operator `*`. Dereferencing means that instead than being interested in the address the pointer points to, you would rather want to know about the value that is stored at that address in memory.
Take a look at the following example program.

```
#include <iostream>
using namespace std;

int main() {
  int i;                                    // an integer
  int *ip;                                  // a pointer to an integer

  i = 5;                                    // assignment to i
  ip = &i;                                  // ip gets the address of i
```

```
  cout << "The value of i is " << *ip << endl;  // '5'

  *ip = 0;                                      // write 0 to i

  cout << "The value of i is " << i << endl;    // '0'
  return 0;
}
```

Note that the pointer `ip` is first declared. Then it is assigned the address of the variable `i`. Next the pointer is dereferenced to read the value of `i` from it. Finally, the dereferenced version of the pointer is used to write the value 0 to the position in memory that is occupied by `i`. This is evidenced by the last output of the program.

6.2 Passing by pointer

The first application of the pointer concept is to pass variables to functions in a new way. Passing variables by pointer to a function allows the function to change the variables' values. This is similar to passing by reference (see §5.4), but in contrast to passing by reference, passing by pointers is a concept supported by C. Recall that passing by reference is really only part of C++.
When using passing by pointer, you have to remember to pass the *addresses* of the variables that you want to give to the function, rather than the variables themselves. The function will not be able to change the value of the pointer itself, but this is perfectly fine and even necessary, as the address of the variable in memory is fixed. However, the function will now be able to change the value in memory that the pointer points to. To this end, inside the function you need to always dereference the pointer, in order to access the value of the variable that you have passed. This makes the usage of passing by pointer slightly cumbersome compared to passing by reference.
What follows is the example program of §5.4, this time adapted to passing the variable by pointer. Note the difference in the syntax. In particular, note that when you want to pass a standard variable by pointer, you have to give the address to the function using the address operator `&`.

```
#include <iostream>
using namespace std;

void add_to_byvalue(double x) { x++; }            // does not change x outside
void add_to_bypointer(double *x) { (*x)++; }      // does change x outside

int main() {
  double z = 4.5;
  add_to_byvalue(z);                              // pass z
  cout << z << endl;                              // '4.5'
  add_to_bypointer(&z);                           // pass &z, i.e. the address
  cout << z << endl;                              // '5.5'
  return 0;
}
```

Note carefully that we had to use brackets in the expression `(*x)++`, because of the precedences of the involved operators. Otherwise the meaning of the statement would have been `*(x++)`, something which will produce garbage and will probably cause your program to terminate with an exception. What that statement would mean in detail we will discover once we know about pointer arithmetic, see §6.4.1.

6.2.1 Keyword `const`

In §5.5 we have seen how changes to parameters that were passed by reference can be suppressed. The same holds true for passing by pointer. In particular, if an argument of type pointer was declared as `const`, then the compiler will disallow any statement that might change the variable that the pointer points to.
The syntax is

```
void foo(const double *x)
```

and possible examples can be derived as in §5.5.

6.3 The NULL pointer

Like any other type of variable, by default when a pointer is declared it is not initialized. That means that the pointer will hold an arbitrary value that could point to just about any address in memory. In order to be able to give pointers a value that signals that they have not yet been given a variable's address to point to, the NULL pointer was introduced. The actual numerical value of NULL is indeed 0, but this should never be used. Instead, you should always use NULL if you want to initialize pointers with an "empty" value.
Note also that many system routines use the NULL pointer to signal that a desired operation involving a pointer has failed, see §6.6.

6.3.1 if statement and pointers

One consequence of the above is that one can simply test whether a pointer holds an address or is defined as NULL by writing

```
if (p)
  cout << "Pointer p is defined." << endl;
else
  cout << "Pointer p is NULL." << endl;
```

If p is a NULL pointer, then the above if statement will fail because the numerical value for NULL is 0, which means false. Similarly, testing for (!p) means to check if the pointer p is NULL.
The same technique can be used for while and for loops.

6.4 Pointers and arrays

In C, there is a strong relationship between pointers and arrays. First of all, when an array is declared, its name serves as a pointer to the first element of the array. Thus

```
double x[100], *xptr;
xptr = x;                          // equivalent to xptr = &x[0];
```

declares an array of 100 elements of type double and then sets xptr, which is of type pointer to double, to point to the head of the array x. It is important to realize that since x itself is nothing other than the address of the first element of the array x, you can pass x directly to the pointer xptr.

6.4.1 Pointer arithmetic

Now, if xptr points to a particular element in an array, then xptr+1 always points to the next one. This feature of C is called pointer arithmetic. For instance, one can define another pointer and assign this new address to it.

```
double x[100], *xptr, *xp2;
xptr = x;
xp2 = xptr + 1;                    // pointer arithmetic
```

Now xp2 will point to the second element of the array x, i.e. xp2 is equal to &x[1].
Similarly, if xptr points to the first element of an array, then xptr+i points to element i in the array. Hence *(xptr+i) and x[i] are equivalent. Notice again that brackets are needed in the pointer version. Referring to the precedence table (Table 2), we see that dereferencing takes precedence over addition. As a consequence *xptr+i written without brackets means "dereference *xptr and then add i to the result", whereas *(xptr+i) means "add i to xptr and then dereference the result". For this reason, special care is needed when arithmetic operations and dereferencing are performed in the same statement.
We should also note that *(x+i) means the same as x[i]. Indeed, the compiler will always interpret x[i] as *(x+i). Similarly, xptr[i] is also a legal expression. (But even experienced programmers may be surprised to learn that also i[x] is legal, and is the same as x[i].)
Pointer arithmetic is not restricted to adding integer values to a pointer. Similarly, you can subtract integer values from a pointer. Furthermore, you can use operators like +=, -= and ++, -- to achieve the same effect. Finally, it is also possible to subtract two pointer values. As long as they both point to elements belonging to the same array, the result is the number of elements separating them. You can also check (again, as long as they point into the same array) whether one pointer is greater or less than another. A pointer is "greater than" another pointer if it points beyond where the other one points. You can also compare pointers for equality and inequality. Two pointers are equal if they point to the same

variable or to the same cell in an array, and are unequal if they don't. The latter testing can also be done for two pointers that do not point into the same array.
See the following program for some examples.

```
#include <iostream>
using namespace std;

int main() {
  double x[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8 , 9};
  double *xp, *xp_end;

  xp = &x[4];
  *(xp+2) += 10;                                  // '6' -> '16'
  *(xp-2) = 0;                                    // '2' -> '0'

  xp_end = &x[10];                                // points to after x[9]

  for (xp = x; xp < xp_end; xp++)                 // xp++ to increase pointer
    cout << *xp << " ";                           // '0 1 0 3 4 5 16 7 8 9'

  if (xp == xp_end) {
    cout << endl << "for loop was successful." << endl;
  }
  return 0;
}
```

6.4.2 Differences between pointers and arrays

So far, we have only looked at similarities between pointers and arrays. But there are two very important differences between the two.
Let's look at the previous example again.

```
double x[100], *xptr;
xptr = x;                          // equivalent to xptr = &x[0];
```

First, a pointer declaration such as `double *xptr;` allocates only the space required to store an address. It does not allocate any space that could be used to store an array of double precision numbers. This is only natural, since the compiler at this point does not know how big an array we want `xptr` to point to later.
Second, if we declare an array as `double x[100]`, for example, the variable `x` is what is called a *pointer constant*, recall §2.7. That means that once the array is declared, the value of the pointer `x` is fixed, and we cannot change it or assign anything to `x`. Therefore, although `xptr = x;` is legal, `x = xptr;` is not legal.

6.5 Pointers to pointers

We can declare pointers to any type of variable. A pointer is itself a variable and we can have another set of variables which point to the pointers.
This can be useful when dealing with a large list of pointers, or when using multidimensional arrays. Here is an example program.

```
#include <iostream>
using namespace std;

int main() {
  char *name = "John";                  // pointer to char
  char *surname = "Smith";
  char *list[2];                        // array of pointers to char
  list[0] = name;
  list[1] = surname;
  char **l = list;                      // pointer to pointer to char
  for (int i = 0; i < 2; i++)
    cout << l[i] << " ";
```

```
  return 0;
}
```

6.6 Dynamic memory allocation

The main reason why we need to know about pointers is the possibility to dynamically allocate memory at run time. When declaring arrays, we have to know their dimensions in advance. Very often we are not in a position to know in advance how much memory our algorithms and programs are going to need. For this reason, C/C++ support dynamic memory allocation, which is the ability to allocate memory at run time.
In C++, the operators `new` and `delete` are defined for this purpose. The operator `new` is used to allocate memory of a given size at run time. The syntax is either

```
pointer = new type;
```

to allocate memory for a single variable of `type`, or

```
pointer = new type[SIZE];
```

to allocate memory for `SIZE` elements of `type`. The operator `new` returns a pointer of type `type*` that points to the first element of the newly allocated memory. (*)
The most important difference between using an array and allocating memory with `new` is that the size of an array must be a constant value, which limits its size to what we decide at the moment of designing the program before its execution, whereas the dynamic memory allocation allows the allocation of memory during the execution of the program using any variable or constant expression or a combination of both as the size to be allocated.
The dynamic memory is generally managed by the operating system, and in multitask interfaces it can be shared between several applications. So there is a possibility that the memory of the machine exhausts. If this happens and the operating system cannot assign the memory that we request with the operator `new`, a `NULL` pointer will be returned. For that reason it is recommended to always check to see if the returned pointer is `NULL` after a call to `new`.
Once you have used the dynamically allocated memory in your program, you probably want to make it available again so that yours and other programs can use it for other things. The command to release dynamically allocated memory in C++ is `delete`. The syntax is as follows.

```
delete pointer;
```

or

```
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single variable, and the second one for memory allocated for multiple elements.
Here is a program that demonstrates the use of `new` and `delete`.

```
#include <iostream>
using namespace std;

int main() {
  double *x;                                          // pointer to double
  int n;
  cout << "How many numbers would you like to input : ";
  cin >> n;
  x = new double[n];                                  // allocate memory
  if (x == NULL) {
    cout << "Could not allocate memory" << endl;      // check if successful
    return 1;
  }
  for (int i = 0; i < n; i++) {
    cout << i+1 << ". number: ";
    cin >> x[i];                                      // usage like array
  }
  for (int i = 0; i < n; i++)
    cout << x[i] << " ";
```

*`new` and `delete` are unique to C++. In C you have to `#include <stdlib.h>` and then use the functions `malloc()` and `free()`.

```
  delete [] x;                                          // free memory
  return 0;
}
```

Note that it is not possible to use `new` and `delete` in C. The easiest way to allocate memory in C is to use the `malloc()` function. The general syntax is

```
pointer = (type *)malloc((unsigned long) SIZE*sizeof(type));
```

This will allocate memory for `SIZE` elements of `type`. Note the usage of the `sizeof()` operator that was briefly discussed in §2.1. Note also that the two type casts to `(type *)` and to `(unsigned long)` are crucial to make this approach work.
Again, `malloc()` will return a `NULL` pointer, if the allocation was not successful.
The following program shows how the previous example would have to be written in C.

```
#include <cstdio>
#include <cstdlib>                                      // include stdlib.h

int main() {
  double *x;                                            // pointer to double
  int n, i;
  printf("How many numbers would you like to input : ");
  scanf("%d", &n);
  x = (double *)malloc((unsigned long) n*sizeof(double)); // allocate memory
  if (x == NULL) {
    printf("Could not allocate memory\n");              // check if successful
    return 1;
  }
  for (i = 0; i < n; i++) {
    printf("%d. number: ", i+1);
    scanf("%lf", &x[i]);                                // usage like array
  }
  for (i = 0; i < n; i++)
    printf("%6.4e ", x[i]);
  free(x);                                              // free memory
  return 0;
}
```

Apart from `malloc()` you can also use `calloc()` to dynamically allocate memory and initialize the elements at the same time, or `realloc()` to dynamically change the size of the memory a pointer points to. See `www.cplusplus.com/ref/cstdlib` for more details.

6.6.1 Segmentation faults and fatal exception errors

Pointers are responsible for almost all of the program crashes that a programmer will encounter when developing a project. Depending on the operating system, these crashes will either be called *Segmentation fault* (Linux/Unix) or *Fatal exception errors* (Microsoft Windows).
These errors will occur whenever a pointer is used that does not hold a valid memory address. For instance, when a pointer is used without being properly initialized, or when a `NULL` pointer is used in trying to access a location in memory.
Most often, however, it will have to do with dynamically allocated memory and a violation of the allocated array's boundaries. I.e. the programmer tries to read from or write to an element which is no longer part of the allocated space. So special care has to be taken when working with pointers.

6.6.2 Dynamic allocation of multidimensional arrays

There is a very important difference between fixed sized multidimensional arrays and dynamically allocated arrays.
When declaring a fixed sized array like

```
type name[N1][N2]...[Nn];
```

then in effect, the compiler will allocate space for all the elements in one big vector and `name` will be a pointer to `type`. Take the following example.

```
int x[100][20];
```

Then the compiler will allocate space for $100 \times 20 = 2000$ sequential elements and `x[0]` will point to the first element of this vector. In particular, this means that `x` itself, which is nothing other than `x[0]`, is of type `int *` and not `int **`.
Furthermore, when addressing an element like `x[i][j]`, the compiler computes the position of the element as "take the address of `x`, add `i` times 20 + `j` to it". Notice that the length of the fastest varying index was necessary for the success of the operation. In general, the dimensions of all the indeces apart from the first one are needed.
On the other hand, if `x` is declared to be of type `int **`, then `x[i][j]` is obtained by "take the address of `x` and add `i` to it, then take the value stored at that address as a new address, add `j` to it and return the value thus addressed". This latter operation is closest to the spirit of matrix manipulation, and it is much preferable to the first for any scientific computation. Note also, that this method has to be used wherever variable sized multidimensional arrays are to be used.
When allocating memory for a multidimensional structure with the help of pointers, it is probable that several different pointer tables have to be set up. In order to allocate the above 100×20 array, one would first have to allocate a table of length 100 for the type `int*`. Then for each of the entries of that table, one needs to allocate an array of length 20 for the type `int`.
See also the following example program, in which an $n \times m$ matrix is dynamically allocated and initialized.

```
#include <iostream>
using namespace std;

double **new_matrix(int n, int m) {
  double **mat;
  mat = new double * [n];            // allocate space for n pointers to double
  if (!mat) return NULL;             // if unsuccessful, return NULL
  for (int i = 0; i < n; i++) {
    mat[i] = new double[m];          // now for each row, allocate m entries
    if (!mat[i]) {
      for (int j = i-1; j >= 0; j--) // if unsuccessful, free allocated space
        delete [] mat[j];
      delete [] mat;
      return NULL;                   // and return NULL
    }
  }
  return mat;                        // if successful, return pointer to space
}

int main() {
  double **matrix;
  int n, m;
  cout << "Enter dimensions for matrix: ";
  cin >> n >> m;
  matrix = new_matrix(n,m);                    // allocate matrix
  if (!matrix) return -1;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      cin >> matrix[i][j];
  // etc
  return 0;
}
```

For higher dimensional arrays, pointers of the types `double ***`, `double ****`, etc and similar functions to allocate the necessary structure have to be used.

6.7 Pointers to functions

The beauty of pointers is that they cannot only hold the address of variables, they can also hold the address of functions. Once your source code is compiled and run, all your functions live in some part of the memory. In particular, they also have an address, that can be used to call and execute them.
Cases where this might indeed be useful occur when you want to use a certain piece of code in many different situations. For instance, a routine to solve an equation using Newton's method, or a method to

integrate a given function over a specified range using the trapezoidal rule. In these cases, the algorithm is defined in terms of a general function. The user can then supply that function – via a pointer – when the code is called.
When declaring a pointer to a function, whether as a standalone variable or as a passed parameter to a function, the return type *and* the parameters the function takes have to be supplied. For example

```
double (*func)(double);
```

declares a pointer to a function that returns a `double` value and takes a single `double` parameter.
Note that the brackets in the declaration are necessary, since the function evaluation has a higher precedence than the dereferencing operator `*`, see Table 2. Hence, `double *func(double);` is interpreted as `double *(func(double));` which would mean to do the function evaluation first, and then return the result as a pointer to `double`. This clearly makes no sense. In order to override this default, we must write `double (*func)(double);` which now has the intended result.
Suppose that we have defined a function

```
double foo(double);
```

in our source code. Then the pointer `func` can be given the address of `foo()` by writing

```
func = &foo;
```

or

```
func = foo;
```

The latter syntax is valid, since it is impossible to actually assign the function itself to the pointer. Hence the C/C++ compiler knows that we really only want to have the address of `foo()`. Similarly, there are two ways to call the function with the help of the pointer `func`. Either you use the dereferencing operator and write

```
y = (*func)(x);
```

or you omit the dereferencing and simply write

```
y = func(x);
```

The following example demonstrates how pointers to functions can be used as parameters for other functions. It uses the trapezoidal rule to approximate the integral $\int_0^{\frac{\pi}{2}} \sin^{\frac{1}{3}}(x)\,\mathrm{d}x$.

```
#include <iostream>
#include <cmath>
using namespace std;

double my_f(double x) { return pow(sin(x), 1.0/3.0); }

double trapez(double a, double b, int n, double (*func)(double)) {
  // integrate func over [a,b] using n intervals
  double x = a;
  const double dx = (b - a) / (double) n;
  double val = 0.5 * func(x);
  x += dx;
  for (int i = 1; i < n; i++, x+= dx)
    val += func(x);
  val += 0.5 * func(x);
  return val*dx;
}

int main() {
  int n;
  cout << "Integrating sin^1/3 (x) over [0,PI/2]." << endl;
  cout << "How many steps to take for the integral: ";
  cin >> n;
  cout << "The integral using " << n << " steps is "
       << trapez(0.0, 0.5*M_PI, n, my_f) << ".\n";              // pass my_f
  return 0;
}
```

Now all that is needed to use the implemented trapezoidal rule for a different integrand, is to supply the new function definition and to call the function `trapez()` with a pointer to that new function. In particular, the implementation of the function `trapez()` need not be changed.

7 Structures

A structure is a collection of one or more variables grouped together under a single name for convenient handling. Here the variables that are grouped together can have different types. Structures help to organise complicated data, that is best described by more than a single variable or array, because they allow a group of related variables to be treated as a unit instead of as separate entities.
A natural example is an employee record that consists of the name, date of birth, address, salary, ID number etc. of the person involved. A structure allows the programmer to group all these properties into one unit, and treat that new structure as a new kind of variable type in the program.
There are also many naturally occurring examples of structures in scientific programming. Here are a few examples:

- Complex numbers: A complex number is naturally defined as a structure containing two `double`s.
- Vectors: A vector consists of a pointer to type and an integer to give the length of the vector.
- Matrices: A matrix is a pointer to pointer to type and two integers to give the dimensions of the matrix.
- Points: A point is a structure containing all its coordinates.
- Triangles: A triangle is a structure with three edges and three vertices.

7.1 Structure syntax

Consider first the structure required for complex numbers. Such a structure would be expected to have two members, say one called `re` and the other called `im`. The structure is then defined as follows.

```
struct complex {
  double re, im;
};
```

This definition does not declare any variables. It rather defines a new type of variables that can now be used within the program. Structure definitions usually come before function declarations in C/C++ programs, because if functions refer to structures then the structures must have been defined before the function declarations are reached.
Variables of type `struct complex` can now be declared as

```
struct complex z1, z2;
```

meaning that two variables, called `z1` and `z2` are now declared, i.e. space has been allocated for them in memory.
Members of a structure can then be referred to using the `struct` member operator "`.`". So, to set $z = 3 + 4i$, we can use the following syntax.

```
struct complex z;
z.re = 3.0;
z.im = 4.0;
```

Notice from the precedence table (Table 2) that the `struct` member operator is at the highest precedence. That means that names of structure members can be used as freely as any other variable name. For example,

```
struct complex z;
double *p = &z.re;
```

works without brackets, and the address of the real part of `z` is indeed passed to the pointer `p`.

7.2 Operations on structures

Although you can define any number of structures of all types, not all elementary operations are supported on structures in C. The only operations which are built in to the C language are:

- Accessing a member of a structure
- Taking the address of a structure

- Copying a structure for the purpose of: (i) assigning one structure to another one of the same type, (ii) passing a structure as an argument to a function, (iii) returning a structure a a result of a function call

This means that `z1 = z2;` is legal – it copies the value of every element of `z2` into the same place in `z1` – whereas `z1 = 2.0*z2;` is not legal. For the latter to make sense we have to use C++, and we will learn about that in part II of this course.
The reason why not more operations are defined in C is simply this. Structures can contain any mixture of variables of different types and there is no way that any operations other than direct copying could be guaranteed to make sense.
However, it is a simple matter to write your own functions for manipulating complex numbers. For example, to multiply two complex numbers, we can use the following code.

```
struct complex c_mult(struct complex z1, struct complex z2) {
  struct complex val;
  val.re = z1.re * z2.re - z1.im * z2.im;
  val.im = z1.im * z2.re + z1.re * z2.im;
  return val;
}
```

7.3 Pointers and structures

Pointers to structures can be defined as for any other variable type. For instance, the previous implementation of the function `c_mult()` can be changed to

```
struct complex c_mult(const struct complex *z1, const struct complex *z2) {
  struct complex val;
  val.re = z1->re * z2->re - z1->im * z2->im;
  val.im = z1->im * z2->re + z1->re * z2->im;
  return val;
}
```

i.e. the two parameters `z1` and `z2` are now passed by pointer.
Notice the use of the `->` operator. `z1->re` is entirely equivalent to `(*z1).re`, which means "dereference the address `z1` to give a structure, then find the element `re`". It is *not* equivalent to `*z1.re`, which assumes that `z1` itself is a structure, with a member `re` and this member is then treated as a pointer and dereferenced by the `*` operator. Obviously, this wouldn't make sense at all in our case.
To do away with the need for brackets when using pointers to structures, the `->` operator was introduced.

7.4 Passing structures

Recall that whenever a variable is passed by value to a function, and whenever a function returns a function value, an extra local copy of the variables involved is created and the necessary values of the variables are copied across. Clearly this should be avoided when large structures are involved.
Hence large structures should always be passed by pointer or – when using C++ – by reference. Similarly, large structures should not be returned as function values. Instead, the result should be returned as an argument that is passed by pointer or by reference and that is then changed by the function.

7.5 `typedef`

As we have seen, the syntax for structures is rather clumsy, so it is convenient to define `struct complex`, for example, to be its own type. That means we want to give this new type a name and then be able to refer to this `struct` type by its new name. In C++ this is possible straight away. For instance, after defining the `struct complex` variables of this type can be declared as (*)

```
complex z1, z2;
```

Note the omission of the keyword `struct`. In C, however, this is *not* possible. However, it is possible to give a structure a new name under which it can be referred to as a new type. This is done with the `typedef` command. So, to declare the new type `Complex`, one can write

```
typedef struct complex Complex;
```

*In C this is not possible and one has to use `typedef`.

This says that the type `struct complex` can now be referred to as a single keyword `Complex`. It is worth noting that `typedef` cannot only be used for structures, but also for any other type that you want to give a new name.
The `typedef` instruction can occur anywhere in the source file before it is used. It can even come ahead of the definition of the structure itself. Thus, in C we can re-write our complex number type as follows.

```
typedef struct complex Complex;
struct complex {
  double re, im;
};
```

But as indicated before, when using C++ the definition of a new name for a `struct` type is not necessary. Hence in C++ the following example program shows how to implement the types and functions discussed earlier.

```
#include <iostream>
using namespace std;

struct complex {
  double re, im;
};

complex c_mult(const complex &z1, const complex &z2) {
  complex val;
  val.re = z1.re * z2.re - z1.im * z2.im;
  val.im = z1.im * z2.re + z1.re * z2.im;
  return val;
}

int main() {
  complex z1, z2, z3;
  z1.re = 1.0; z1.im = 2.0;
  z2.re = -1.0; z2.im = 0.2;
  z3 = c_mult(z1, z2);
  cout << "Result: " << z3.re << " + " << z3.im << "i." << endl;
  return 0;
}
```

Note that the two arguments to `c_mult()` are passed by reference.

7.5.1 `typedef` and `#define`

You can think of `typedef` as a very special `#define` preprocessor directive. And in fact, most `typedef` instructions could be replaced by a corresponding `#define` directive. However, there are cases where using `#define` would give wrong results. The code below declares two variables of type pointer to char. Replacing the `typedef` with `#define MY_STRING char *` would cause the second variable to be of type `char`, rather than pointer to `char`.

```
typedef char * MY_STRING;
MY_STRING s, r;
```

In general, one should be very careful when using `typedef` for pointer types. This can easily lead to programming errors that will be hard to detect, as it will be unclear which level of dereferencing is necessary for a given object, especially when passed through a function. It is therefore advisable to only `typedef` pointer types in situations where this cannot lead to confusion.

7.6 Selfreferential structures

A very useful feature of structures is that they can refer to other members of their own type. This makes it possible to implement objects such as linked lists, where each member of the list contains a number of attributes and a pointer to the next member of the list, and binary trees, where each member has up to two pointers to children elements "below".
The syntax for this is very straightforward.

Suppose we have a number of airline passengers arriving for a flight. Each passenger possesses certain attributes, such as their check-in number, their final flight destination, the number of checked bags they have and so on. The structure for a list of passengers could hence look like this.

```
struct passenger {
  int checkin_id;          // check-in number
  char dest[3];            // 3 letter airport code
  int bags[2];             // bag label numbers for their 2 bags
  int f_class;             // 1 = 1st, 2 = business, 3 = economy
  string name;             // C++ string class for passenger name
  passenger *next;         // pointer to structure of next passenger
};
```

The whole passenger list will be represented by a pointer to `passenger`, which will hold the address of the first list element. Then each element points to the successor in the list, until the last element, where the pointer `next` should be set equal to `NULL`.
The beauty of this approach is that the number of passenger does not have to be known in advance. Furthermore, it is simple to add more passengers to the list as they arrive for the flight. Here we simulate adding a passenger treating the passenger list as a "stack", so that the last member added is on top of the stack.
The following function would be called when a new passenger arrived.

```
passenger *new_passenger(passenger *p) {
  passenger *new_p = new passenger;
  /* Ask check-in clerk to enter details, or get them from a file
     e.g.  new_p->name = ....; new_p->f_class = ...; etc              */
  new_p->next = p;
  return new_p;
}
```

The function `new_passenger()` would be called with

```
pass_list = new_passenger(pass_list);
```

since the function takes as an argument the head of the linked list of passengers and returns the new head of the linked list. Prior to the arrival of any passenger, the pointer `pass_list` must be set equal to `NULL`, so that the first call to `new_passenger()` assigns the value `NULL` to the `next` member of this passenger. This will subsequently mark the end of the linked list.
To see how many passenger are checked in for the flight in a given class, we would use a simple function such as the following.

```
int n_pass(passenger *p, int c) {
  int count = 0;
  while (p) {
    if (p->f_class == c) count++;
    p = p->next;
  }
  return count;
}
```

The function traverses the list from beginning to end. It reads through the list of passengers and after dealing with each passenger moves on to the next one. This happens for as long as the `while` is true. However, when we get to the last passenger, which will have the value `NULL` stored at the member `next`, this value will be assigned to `p` and the while loop will terminate.
It is clear that a certain amount of care is needed to ensure that the ends of linked lists always have their `next` member as `NULL`. If this is not the case, the program will keep traversing through memory, trying to interpret whatever it sees as a structure. It will almost certainly produce garbage, and it will probably cause the program to crash.

7.7 Examples

7.7.1 Sparse matrices

A very good example for the use of pointers in scientific computing is the implementation of sparse matrices. Sparse matrices have only very few non-zero entries, usually of order $\mathcal{O}(N)$, where $N \times N$ is

the dimension of the matrix. Hence the amount of data necessary to store the matrix can be greatly reduced, if only the non-zero elements of the matrix are stored. Furthermore, implementations of, say, matrix-times-vector multiplications will now only need $\mathcal{O}(N)$ operations, compared to $\mathcal{O}(N^2)$ if all the elements of the matrix were stored explicitly.
The idea for the implementation considered here is to store for each row of the matrix the non-zero elements in that row. So the matrix itself is an array of pointers, and for each row the pointer points to a list of non-zero entries.
A simple implementation can then look like this.

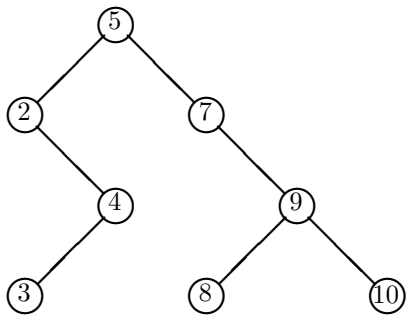
```
#include <iostream>
using namespace std;

struct entry {
  int j;                // column number
  double val;           // value of entry
  entry *next;          // pointer to next entry
};

entry **create_matrix(int dim) {
  entry **m;
  m = new entry * [dim];
  for (int i = 0; i < dim; i++) m[i] = NULL;
  return m;
}

void add_entry(entry **m, int dim, int i, int j, double v) {
  // adds entry A_ij to sparse matrix m
  entry *p;
  if (i < 0 || i >= dim) return;
  for (p = m[i]; p; p = p->next) if (p->j == j) break;
  if (!p) {
    p = new entry; p->j = j; p->val = 0.0;
    p->next = m[i]; m[i] = p;
  }
  p->val += v;
}

void matrix_vector(double *y, entry **m, int dim, const double *x) {
  // return y = A*x
  for (int i = 0; i < dim; i++) {
    y[i] = 0.0;
    for (entry *p = m[i]; p; p = p->next) y[i] += p->val * x[p->j];
  }
}

int main() {
  entry **sparse_matrix;
  double a, *x, *y;
  int dim, i, j;
  cout << "Dimension of matrix: "; cin >> dim;
  sparse_matrix = create_matrix(dim);
  x = new double[dim]; y = new double[dim];
  for (i = 0; i < dim; i++) add_entry(sparse_matrix, dim, i, i, 1.0);
  do {
    cout << "Add entries to matrix A. Give i,j and A_ij: ";
    cin >> i >> j >> a;
    if (i >= 0 && j >= 0) add_entry(sparse_matrix, dim, i, j, a);
    else cout << "Done." << endl;
  } while (i >= 0 && j >= 0);
  for (i = 0; i < dim; i++) { cout << i+1 << ". coordinate of x: "; cin >> x[i]; }
  matrix_vector(y, sparse_matrix, dim, x);
  for (i = 0; i < dim; i++) cout << i+1 << ". coordinate of y = " << y[i] << endl;
```

```
  return 0;
}
```

Of course, there is a more sophisticated way. You could define the rows of the matrix to be a list of pointers itself. In this way, you would not store an empty row, like you would in the previous implementation. However, as this case occurs only for singular matrices, in practice the above implementation is usually good enough.

7.7.2 Binary trees

Binary trees can be used for binary sorting and for binary search in data. The idea is simple. A binary tree is made of nodes, where each node contains a *left* pointer, a *right* pointer, each of them pointing to a child node, and a data element that holds the data to be stored. The root pointer points to the topmost node in the tree, and hence defines the tree itself, similarly to the head pointer defining a linked list. The left and right pointers recursively point to smaller subtrees on either side. A `NULL` pointer represents a binary tree with no elements, i.e. the empty tree. A node with two `NULL` pointers is called a leaf.
Here is a simple sketch of a tree with 8 nodes, of which 3 are leaves.



A binary search tree is a type of binary tree where the nodes are arranged in a certain order. This allows the tree to be used for sorting data and for searching in a data base. The order is defined as follows. Each node holds a search key that determines its position with respect to other nodes. This key can be a date of birth, a name or an internal id number. Then for each node, all elements in its left subtree hold a key that is less or equal to the node's key, and all the elements in its right subtree hold a key greater than the node. The tree shown above is a binary search tree, the root node is 5, and its left subtree nodes $(2, 3, 4)$ are smaller, while its right subtree nodes $(7, 8, 9, 10)$ are bigger. Moreover, each of the subtrees below the root also obeys the binary search tree constraint.
A natural implementation of a binary tree in C would use the following structure.

```
struct node {
  Data *data;                  // pointer to some important data
  int  key;                    // the search key
  node *left, *right;          // pointers to left and right child
};
```

The tree itself will be defined by its root, i.e. by a pointer

```
node *root;
```

Now all the methods for a binary search tree can be implemented with the help of recursive algorithms. For instance, to print the elements of the tree in ascending sorted order one can use the following function.

```
void print_tree(node *node) {
// prints binary search tree in ascending order
  if (node == NULL) return;
  print_tree(node->left);
  cout << node->key; print_data(node->data);
  print_tree(node->right);
}
```

This way of traversing the tree is also called "in-order", because the node itself is inspected in between its two subtrees. Other ways of traversal are pre-order and post-order. The above function can be invoked with `print_tree(root);` where `root` is the previously defined pointer to the root node of the tree.
Similarly, an algorithm to search for a certain entry can be implemented as follows.

```
node *find_node(node *node, int target) {
  // searches binary search tree for node with key == target
  if (node == NULL) return NULL;                          // target not found
  else {
    if (target == node->key) return node;                 // target found
    else {
      if (target < node->key) return find_node(node->left, target);
      else return find_node(node->right, target);
    }
  }
}
```

Finally note that sorting with the help of a binary search tree simply amounts to inserting the to be sorted elements into an initially empty tree. Of course, the crucial point is that all the elements are inserted in such a way, that the binary search tree criterion is not violated.

8 Advanced material

8.1 `enum` and `union`

Enumerations can be used by the programmer to define new types. They can make the code more readable and provide extra type checking. Very often integer constants are used in programs to differentiate between different situations. Consider a situation where you want to assign a colour as an attribute to a certain object, say a car. One way of implementing this would be as follows.

```
const int white = 0;
const int blue = 1;
const int red = 2;

int car_colour = red;
```

Although this works perfectly fine, there is no intrinsic type checking taking place. In particular, the compiler does not know that you want to restrict the values for the integer variable `car_colour` to only the few valid colours. Hence assignments like `car_colour = -123;` are accepted by the compiler, although they might not make sense in your program.
But you can use an `enum` to have the compiler enforce this sort of type checking. In this example, one would define

```
enum colour {white, red, green, blue, black};
colour car_colour = red;
```

The compiler will now reject assignments like `car_colour = 4`. Depending on your compiler you may just get a warning, or you may get an error.
Internally, the `enum` values will be represented by some integer values. Usually, they will start with 0 for the first item, 1 for the second one and so on. The programmer does not really need to know about them. But in some cases, the programmer might want to assign special numerical values to each item so that they can switch between computations involving the numerical value of the enumerations and their name definitions. In this case, one can define the enumeration as follows.

```
enum colour {white = 0, red = 13, green = 27, blue = 31,  black = 32};
colour car_colour = red;
```

Of course, the numerical values have to be distinct.
A `union` is a very technical feature of C/C++ and you will probably never have to use it. A `union` is similar to a `struct` in that it can hold several different data types. But in contrast to a `struct`, for a union at any one time only one of the data entries is active. Moreover, the entry that was assigned a value last is the only one that is valid, and it deletes any previous information held by the union.
Declaring a `union` is like declaring a `struct`.

```
union robot {
 char *name;
 int ammo;
 int energy;
};
```

The size of a `union` is the size of its largest member. This is because a sufficient number of bytes must be reserved for the largest sized member.
The unusual behaviour of a `union` is demonstrated by the following example.

```
#include <iostream>
using namespace std;

union robot {
  char *name;
  int ammo;
  int energy;
};

int main() {
  robot r;
  r.ammo = 20;
  r.energy = 100;
  cout << "Ammo = " << r.ammo << endl;              // '100'
  cout << "Energy = " << r.energy << endl;          // '100'
  return 0;
}
```

Note that the assignment to `r.energy` overrides any previous information held by the union. There are not many practical applications of the `union` concept. They mostly appear in source codes for system libraries, so you are unlikely to ever come across them.

8.2 Conditionals

A very efficient way to write some conditionals is to use the `?` operator. The `?` operator is a ternary operator, i.e. it takes three operands. The syntax is

```
expression1 ? expression2 : expression3
```

and this defines a conditional expression. If `expression1` is true, then it evaluates to `expression2`, otherwise it evaluates to `expression3`. That means, some `if-else` statements can be written in a much shorter and more efficient way. For instance, the expression

```
z = a > b ? a : b;
```

is the same as

```
if (a > b)
  z = a;
else
  z = b;
```

The operand `expression2` in the conditional expression may be omitted. Then if `expression1` is nonzero, its value is the value of the conditional expression. Therefore, the expression `x ? : y` has the value of `x` if that is nonzero, otherwise the value of `y`. This example is equivalent to `x ? x : y`, of course. The only useful application of this feature is when the `expression1` itself is a function call, and this function should not be called twice. An example could be

```
z = count_students() ? : no_one_there();
```

8.3 Preprocessor directives

8.3.1 `#define`

We have already seen how to use the `#define` preprocessor directive in order to define simple macros. But it can also be used to generate macro functions. For instance

```
#define MAX(a, b)   ((a) > (b) ? (a) : (b))
#define SQR(a)      ((a)*(a))
```

defines a macro function that will return the maximum of two given value, and a macro that will return the square of a given number. E.g.

```
double x = 3.1;
double y = MAX(x, 5.4);
double z = SQR(x);
```

Like any other preprocessor directive, these macro function definitions are expanded by the preprocessor just prior to the compilation of the source code. The `#defined` macros are active in the source file from the line where they were defined, and there they will be replaced with their intended meaning. It is important to enclose with brackets any appearance of a parameter for the macro in the definition of the macro. Otherwise the parameter name is going to be interpreted literally, and it is not going to be replaced by the value of the argument in the macro function call.
Note that you cannot use the same name for both a simple macro and a macro with arguments. Furthermore, note that by convention all macro names are capitalised. This makes it easier to identify them within the source code, and once again it makes debugging easier.
In the definition of a macro with arguments, the list of argument names follows in brackets and this must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. For example,

```
#define FOO(x) - 1 / (x)
```

defines `FOO` to take an argument and expand it into minus the reciprocal, whereas

```
#define BAR (x) - 1 / (x)
```

defines `BAR` to take no argument and always expand into "`(x) - 1 / (x)`". Of course, the latter will not compile unless `x` is a valid variable wherever the macro `BAR` is used.
If you want to define a macro that is longer than a single line, you can use the "\" character to mark the end of a line. Here is an example

```
#define PRINT(a) { cout << "In what follows, we will output a parameter " \
                        << "to a macro function." << endl; \
                   cout << "The parameter is: " << (a) << endl; }
```

8.3.2 `#undef`

It is also possible to undefine a macro, i.e. to cancel its definition. This is done with the `#undef` directive. Here are two examples for an undefinition.

```
#undef PI
#undef MAX
```

Like definitions, an undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name. I.e. the macro name is always taken literally.
The main use for `#undef` is when redefining macros. Macros can simply be redefined with another `#define` directive, but this will usually result in a compiler warning. Using `#undef` before the redefinition will prevent that error message. For example,

```
#define LENGTH 100
double x1[LENGTH];
#undef LENGTH
#define LENGTH 200
double x2[LENGTH];
```

8.3.3 `#if, #endif, #else` and `#elif`

These directives allow to discard part of the code of a program if a certain condition is not fulfilled. Conditionals in the preprocessor are similar to `if-else` statements in C/C++, only that here the conditionals are evaluated only at compile time. Depending on their outcome, different versions of the program are compiled.
Typically preprocessor conditionals are used to allow a program to be compiled for different machine architectures and operating systems, or to be able to switch on/off time consuming consistency checks and output of debugging information.
The `#if` directive in its simplest form consists of

```
#if expression
  // some C code
#endif
```

Of course, `expression` may not contain any variable as it will have to be evaluated at compile time. Furthermore, it is restricted to a C expression of integer type and it may contain previously defined macros and all the known relational operators, like `<`, `>`, `==` etc..
The `#else` directive can be added to a conditional to provide alternative text to be used if the condition is false. This is what it looks like:

```
#if expression
  code-if-true
#else
  code-if-false
#endif
```

If `expression` is nonzero, and thus the `code-if-true` is active, then the `code-if-false` is ignored. Otherwise, the opposite is true.
Finally, the `#elif` directive can be helpful in nested conditionals. It simply means `#else #if` and is usually used to check for more than two possible alternatives. For example, you might have

```
#if DIM == 1
  // code for 1d
#elif DIM == 2
  // code for 2d
#elif DIM == 3
  // code for 3d
#else
 cout << "Error: not implemented for dimension " << DIM << endl;
#endif
```

8.3.4 `#ifdef` and `#ifndef`

`#ifdef` allows that a section of a program is compiled only if the macro name that is specified as the parameter has been defined, independently of its value. Its operation is:

```
#ifdef name
  // code here
#endif
```

For example:

```
#ifdef LENGTH
double x[LENGTH];
#endif
```

In this case, the line `double x[LENGTH];` is only considered by the compiler if the defined constant `LENGTH` has been previously defined, independently of its value. If it has not been defined, that line will not be included in the program.
`#ifndef` serves for the opposite: the code between the `#ifndef` directive and the `#endif` directive is only compiled if the macro name that is specified has not been defined previously. For example:

```
#ifndef LENGTH
#define LENGTH 100
#endif
double x[LENGTH];
```

Here, if when arriving at this piece of code the macro `LENGTH` has not yet been defined it would be defined with a value of 100. If it already existed it would maintain the value that it had before.

8.4 Header files

Another preprocessor directive is the `#include` directive that we have already used in order to use system libraries. The syntax is either

```
#include <filename>
```

or

```
#include "filename"
```

In both cases the compiler looks for a file called `filename` to be included in the source code. The only difference between both expressions is the directories in which the compiler is going to look for the file. In the first case, the compiler searches for the file in the directories that it is configured to look for for the standard header files. In the second case, the file is looked for in the directory of the source file itself. Only in case that the header file cannot be found there, the compiler looks for the file in the default directories where it usually searches for standard header files.
Including a header file is used in order to be able to call functions that were defined elsewhere. A typical example are system libraries like `<iostream>` and `<fstream>`, or `<cmath>` for mathematical functions. If you did not `#include` these header files, you would have to implement the necessary functions yourself. But header files can also be used to organise medium to large scale projects of your own. They enable you to define and implement certain necessary functions and structures in one source file, while your main source file simply `#include`s the necessary header file.
When the preprocessor comes across an `#include` directive, it simply inserts the contents of the given header file into the source file where it finds the directive. Usually a header file contains function declarations and macro definitions that you want to share between different projects, or it contains all the definitions belonging to a certain module of your project.
Here is an example. Let the file `"matrix.h"` contain the following declarations.

```
double **new_matrix(int n, int m);                    // allocate matrix
double *new_vector(int n);                            // allocate vector
double *read_vector(int n);                           // read in vector
void y_A_x(double *y, int n, double **A, double *x);  // y = Ax
void print_vector(int n, double *x);
```

Then, assuming that the source file `matrix.cc` implements these functions appropriately, the header file can be used in the following fashion.

```
#include <iostream>
#include "matrix.h"                    // use header file
using namespace std;

int main() {
  int n, m;
  double **A, *x, *y;
  cout << "Dimension of matrix A : "; cin >> n >> m;
  A = new_matrix(n,m);
  x = read_vector(n);
  y = new_vector(n);
  y_A_x(y, n, A, x);
  print_vector(n, y);
  return 0;
}
```

The advantages of using header files are obvious. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them. We have seen that including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time consuming and error prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when they are compiled next. The header file eliminates the labour of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.
Note that when using different source and header files within your projects that only one of the source files may define a `main()` program.

8.5 Command line parameters

In C/C++ it is possible to accept command line arguments. Command line arguments are parameters that are passed to a program when it is called by the user. This is standard practice in linux/unix, but it is also possible under Windows. In oder to use this feature in C/C++, you need to know that the ANSI definition for declaring the `main()` function in C/C++ is either

```
int main()
```

or

```
int main(int argc, char *argv[])
```

This second version allows arguments to be passed from the command line. The integer parameter `argc` is an argument counter and contains the number of parameters passed from the command line, including the name of the program. The parameter `argv` is the argument vector which is an array of pointers to C-strings that represent the actual parameters passed. Here `argv[0]` is the name of the program itself, or an empty string if the name is not available. After that, every element number up to `argc-1` are command line arguments. You can use each `argv` element just like a C-string. Note that `argv[argc]` is a `NULL` pointer.
The following example simply prints the name of the program followed by all the given arguments.

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  cout << "You called program " << argv[0] << " with arguments " << endl;
  for (int i = 1; i < argc; i++)
    cout << argv[i] << " ";
  cout << endl;
  return 0;
}
```

Finally, here is a common example for how command line arguments can be used. The given program takes the name of a (text) file and outputs the contents of the file onto the screen.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  if (argc != 2)
    cout << "Usage: " << argv[0] << " <filename>" << endl;
  else {
    ifstream f(argv[1]);
    if (!f.is_open())
      cout << "Could not open file " << argv[1] << endl;
    else {
      char x;
      while (f.get(x))  cout << x;
    }
    f.close();
  }
  return 0;
}
```