MH 1400 Introduction to Scientific Programming

Lecture Notes by Assoc. Prof. Bernhard Schmidt

July 23, 2011

Contents

1 Introduction

1.1 Goals of this course

Why do we need to learn programming in Mathematical and Natural Sciences? The answer is that these sciences are *experimental* and we need programming to perform and evaluate the experiments. Yes, believe it or not, Mathematics is also an *experimental science*. Most mathematical theorems are not found by "pure thinking" — they are results of extensive computational experiments and efforts to find the patterns behind the results.

Why do we need ***C++*** in Mathematical and Natural Sciences? Because it is the most efficient programming language for performing computational experiments! C++ is very fast and many efficient C++ libraries exist (a *library* is a ready-to-use program for a standard task, like solving systems of equations). Forget Java, for instance, — too slow, no efficient libraries. C++ enhances your scientific skills substantially and is essential for your further study. It is used, in particular, in the modules MAS 331 (Undergraduate Research Experience in Mathematical Sciences), MAS 491 (Honours Project), MAS 492 (Industrial Internship), PAP 929 (Undergraduate Research).

At most universities, students of Mathematical and Natural Sciences are required to take courses in Java or C++ together with the students of Computer Science. But these two groups of students have different interests: Computer Science students usually need to be able to "program from the scratch" while students of Mathematical and Natural Sciences only need to know how to use programming to do computations in their areas of interest.

At NTU we have a different approach — MAS 110 is an introduction to Scientific Programming designed *especially for students of Mathematical and Natural Sciences*. This makes it very different, for instance, from an introduction to C++ for Computer Science students. What you learn are the *most useful* aspects of C++ from the viewpoint of Mathematical and Natural Sciences.

Some topics we study in this course may sound quite advanced, e.g. classes, templates or the C++ Standard Template Library. Don't worry — we only use these features to simplify programs and to *make life easier*. We will not worry much about the theoretical background ("object oriented programming") or the process of creating classes and templates (this can be difficult), but concentrate on *using existing classes and templates* to create simpler, more powerful programs.

Bottom line: The goal of this course is to give you the power to perform scientific computations efficiently. These skills will be very helpful in your studies and professional career.

1.2 The first C++ program

The following is a complete C++ program.

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

int main()
{
    cout << 7*11*13;
    system("PAUSE");
    return 0;
}
```

The most significant part of this program is `cout << 7*11*13;` Here, `7*11*13` computes the product of 7, 11 and 13 (=1001). The command `cout << x` is used to print a number `x` on the screen. The semicolon denotes the end of the command. In summary, `cout << 7*11*13;` prints the number 1001 on the screen.

The rest of the program is only "framework". *All* C++ programs have a very similar framework. Explanations:

- `<iostream>` This is a library of commands including the command `cout` for printing to the screen.
- `#include <iostream>` This is a so-called "include directive". Here, it is necessary because we want to use command `cout` which is part of `<iostream>`.
- `using namespace std;` The "real name" of the command `cout` is `std::cout`. By including `using namespace std;` in the program, we can drop `std::`. This works also for other commands like `std::cin`.
- `int main(){...}` This is called the `main` function of the program. The dots must be replaced by C++ commands. The program executes exactly the commands which are inside the curly braces of the `main` function.
- `system("PAUSE");` When we start a C++ program (using the Dev-C++ environment), a new window opens up. Usually, the results of the program are printed inside this window. The command `system("PAUSE");` makes sure that the window *is not immediately closed* after the program has been executed. So, it makes sure that we can see the results.
- `return 0;` The `main` function must return an integer when it terminates.
  The command `return 0;` means normal program termination. Returning a nonzero integer would indicate an abnormal termination. This could be used, for instance, if an error is discovered during the execution of the program.

1.3 More examples

The following are some more example C++ programs. Try them out!

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

int main()
{
    cout << "2+2 = " << 2+2 << endl;
    cout << "2*3 = " << 2*3 << endl;
    cout << "5/2 = " << 5/2 << endl;
    system("PAUSE");
    return 0;
}
```

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

int main()
{
    int a,b,c;
    cout << "Enter 3 integers:  << endl;
    cin >> a >> b >> c;
    cout << "The product of your integers is "
         << a*b*c << endl;
    system("PAUSE");
    return 0;
}
```

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

int main()
{
    double x;   // declaration
    int y=10;   // declaration + initialization
    x = 3.14;   // assignment
    cout << "x: " << x << " y: " << y << endl;
    system("PAUSE");
    return 0;
}
```

2 Some terminology used in programming

In order to develop programming skills, it is necessary to be able to understand written and verbal explanations on programming issues. For these explanations, we need some basic terminology which is introduced in this section.

2.1 Textfiles

Every C++ program is saved in a *textfile* or several textfiles. A textfile is a simple file format to save text in a computer. To create a textfile, you can use any text editor like `Notepad`, `Wordpad` or `Word` under Windows. Just open the text editor, type some text, go to `Save as` and save the file as a `Text Document`. Textfiles often have the file name extensions `.txt`. However, the textfiles we use for C++ will have file name extension `.cpp` and `.h`. These extensions indicate that the files contain C++ instructions.

We will not need the above mentioned text editors since our programming environment `Dev-C++` already supplies a text editor which is very convenient for C++ programming.

2.2 What is "C++ code"?

A C++ program consists of *instructions*. The instructions determine the actions that the program has to perform. All these instructions are saved in textfiles. The content of these textfiles is called *C++ code*. The following is the C++ code of a program we did in Lab 1.

```
//Program 1
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    cout << "1111111111111111111 is a prime number!" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

2.3 Executable files and compiling

Computer programs are saved on the hard disc as *executable files*. This means that if we open the file, for instance for double-clicking on the icon of the file, the program automatically starts to run. Executable files usually have the file name extension `.exe` (`exe` stands for executable). For instance, the executable file belonging to the program `Word` under Windows has the file name `winword.exe`. When you double-click on the icon of the program `Word`, then the file `winword.exe` is opened and the program `Word` launches.

However, when you double-click on the textfile containing the code of `Progam 1` above, then a text editor opens and shows the content of this file. The C++ program is *not running at all*. In order to make it run, we need one more step. We need to create an executable file from the textfile. This is done by a `C++ compiler and linker`. In other words, a C++ compiler and linker translates the C++ code contained in text files into a C++ program contained in an executable file. We can then start the program by double-clicking on the icon of the executable file. Under Dev-C++, we can do compiling, linking and starting the program just by clicking on a single button.

2.4 Errors

Even the best programmers make lots of errors. Errors are simply unavoidable in programming. However, there is no reason to get frustrated since errors always can be removed. Even if the compiler tells you your program contains 100 errors, there is no need for desperation, you can remove the errors step by step. Some basic knowledge on possible errors, and of course experience, will help you getting rid of errors.

2.4.1 Compiler errors

As we will learn, C++ code (i.e. the C++ instructions we save in textfiles) has to be written according to very strict rules. The rules are called *C++ syntax*. If these rules are violated, the compiler will inform us that there is a mistake and will tell us exactly where the mistake occurs. Such an error is called *compiler error*. Compiler errors are *usually easy to remove* since the compiler gives us valuable information on the error.

2.4.2 Linker errors

The code of big C++ programs (say, with more than 200 lines of code) is usually not written into one single textfile, but is distributed into several different textfiles. This is part of well structured programming. However, sometimes it happens that the code contained in different files *does not fit together*. The *linker*, which links these different files together, will then produce an error message. This is called a *linker error*. Linker errors can be quite difficult to correct.

2.4.3 Runtime errors

If there are no compiler or linker errors, then the compiler and linker produce an executable program. However, there can still be errors! When you run the program, it can happen that it is terminated by Windows and you get an error message like "program1.exe has encountered a problem and needs to close. We are sorry for the inconvenience." The reason for this is usually that the program tried to perform an *forbidden operation* on the computer. In the worst case, such an operation could even destroy the computer, so the program really needs to be stopped. Such an error is call a *runtime error*. Runtime errors can also be quite difficult to correct.

2.4.4 Logical errors

There were no compiler or runtime errors; the program is running smoothly - there is also no runtime error. Does this mean that the program is correct? No!!! Often the worst of all errors is still lurking inside - the *logical error*. This means that there is a severe mistake in the *logic of the program*. It runs, but produces wrong results! Even worse, it often happens that the programs produces correct results in 99% and errors only in 1% of the cases. This means that logical errors often remain *undetected*. For commercial programs, the worst case is that the error is only detected after delivery to the customer (there is a world famous software company which constantly has this problem...).

How can logical errors be removed? The answer is *testing, testing, testing*. Programs usually can be run with many kinds of different input. If the program runs correctly for 1000 kinds of input, then there is some hope it is correct.

3 Basic C++ syntax

The instructions of C++ programs are written into textfiles. The content of these textfiles is called C++ code. C++ code must we written according to very strict rules. These rules are called *C++ syntax*. In this section, we study the most important parts of C++ syntax.

3.1 Case sensitivity and typos

C++ is *case sensitive*. This means that it matters if letters are in lower or upper case. For instance, `cout << 123;` is correct while `Cout << 123;` is wrong. Fortunately, such mistakes are usually compiler errors (dectected by the compiler) and easy to correct since the compiler will tell you the exact location of the error.

Mixing up lower and upper case is actually a special case of a *typo* (typing mistake). C++ is extremely "typo sensitive". Every typo will produce an error. Usually typos are detected by the compiler, but they can also cause more severe errors.

3.2 C++ comments

It is often useful to include comments (explanations) in C++ code. The comments are not part of the program and are ignored by the compiler. There are two ways to include comments in C++ programs:

3.2.1 Double slash method

After a double slash (`//`) everything until the end of the line is a comment. Each double slash only works *for one line*. Example:

```
cout << 123 << endl;
// this prints the number 123 on the screen
cout << 321 << endl; // prints the number 321 on the screen
```

The second line only contains a comment. The third line contains a C++ command and a comment.

3.2.2 Slash-star method

Between `/*` and `*/` everything is considered as a comment. In contrast to double slash comments, a slash-star comment can comprise several lines. Example:

```
/*
   This is
   a slash-star comment
   comprising
   six lines
*/
```

3.3 Commands, statements and statement blocks

A *C++ command* instructs the program to "do something". Important: every C++ command *must be ended by a semicolon*. Examples of C++ commands:

```
cout << 123; //command to print 123 on the screen.
int x;       //command to allocate memory for an integer x
cin >>  x; //command to read the value of x from the keyboard
```

A *statement block* is a sequence of commands enclosed by *curly braces*. Example:

```
{                          // start of statement block
    cout << 123 << endl;
    int x;
    cin >> x;
}                          // end of statement block
```

This statement block comprises three commands. The number of commands in a statement block is the same as the number of semicolons contained in it.

A *C++ statement* is either a single C++ command or a statement block. When studying the syntax diagrams later, remember that *both* a single C++ command and a statement block are called C++ statements! Example:

```
cout << 444;                // statement (single command)

{                           // start of statement
    int x;
    cin >> x;
}                           // end of statement (statement block)
```

3.4 Syntax diagrams

Syntax diagrams are a concise and efficient way to describe how C++ features are used. The following is a syntax diagram for printing an integer on the screen.

`cout <<` *integer* `;`

Note that *integer* appears in typeface italics. Italics in syntax diagrams mean that in C++ code *you must replace this part by something suitable*. Here you have to replace *integer* by an actual integer, for instance, `24`, `13042` or `-1234`.

The rest, i.e. `cout <<  ;`, appears in typeface typewriter (this is like `this is typewriter`). The parts in typewriter style must be included in the C++ code *literally*. In other words, you must not change the parts in typewriter style at all.

Examples of correct C++ code according to the above syntax diagram:

```
cout << 24;
cout << 13042;
cout << -1234;
```

Examples of *incorrect* C++ code according to the above syntax diagram:

```
cout << integer;    // italics part has not been replaced
cout << 24          // typewriter part has been changed
                    // (semicolon missing)
Cout << 24;         // typewriter part has been changed
                    // (Cout instead of cout)
```

3.5 Variables

During the execution of C++ programs, intermediate results must be stored in the memory (RAM) of the computer. For this purpose, *variables* are used. Hence a variable is a *name for a region of computer memory*. The *value* of a variable is the information stored in the memory region belonging to the variable. The *type* of a variable determines how this information is interpreted (e.g. as an integer, a floating point number or a string).

3.5.1 Variable types

The type of a variable determines what kind of values a variable can take. The most common and useful types in C++ are `char` (character type), `int` (integer type) and `double` (floating point type).

The value of a variable of type `char` is any one-letter symbol, for instance, `1` or `e` or `E` or `%`.

The value of a variable of type `int` is any integer in the interval $[-2^{31}, 2^{31} - 1]$. Values outside this interval are not allowed and lead to *integer overflow*.

The value of a variable of type `double` is a floating point number whose absolute value is 0 or in the interval $[2.22507 \cdot 10^{-308}, 1.79769 \cdot 10^{308}]$. Values outside this interval are not allowed and lead to *double overflow*. The precision of variables of type `double` is limited to 15 significant decimal digits. Hence the use of `double` variables can lead to round-off errors. However, in most cases these errors are insignificant or can be controlled by appropriate methods.

3.5.2 Variable declaration

Syntax

type variableName`;`

Before a variable can be used in C++, it must be *declared*. In a variable declaration, the type and the name of the variable must be specified. The variable name has to be chosen by you. It can be any sequence of letters, numbers and underscores (`_`) beginning with a letter. Declarations can be concatenated to declare several variables by one single command. Examples:

```
int x;              // type integer, name x
int x34234;         // type integer, name x34234
int a,b,c;          // declares 3 integers a,b,c (concatenation)
double y1;          // type double, name y1
double 1y;          // wrong! name must begin with letter
char a;             // type char, name a
```

3.5.3 Variable assignment and initialization

Syntax

variableName `=` *value*`;`

After a variable declaration, the variable has a name, but no *value* yet. Before we can do any computations with a variable, we must specify a value for it. This is done by an *assignment*. The *first* assignment of a value to a variable is called an *initialization*. Examples:

```
int x,y,z;              // declarations
x=10;                   // initialization of x
x=10*10;                // assignment (not initialization)
y=x*x;                  // initialization of y
z=x*y;                  // initialization of z
```

3.5.4 Fundamental data types

The types that are built-in in C++ and are ready to use are called *fundamental data types*. The following specifications are valid for the Dev-C++ environment we are using and for many other C++ compilers. The by far most important fundamental data types are `char`, `int` and `double`. The other types are listed for completeness, but will not be important in this course. However, for advanced programming one should make full use of all the built-in data types.

Type	**Bytes**	**Range**	**Comment**
`char`	1	$[-128, 127]$	Characters are stored according to their ASCII-Code, see Hubbard, Appendix A. For instance, `char x='A';` is equivalent to `char x=65;`
`short`	2	$[-32768, 32767]$	integer
`int`	4	$[-2^{31}, 2^{31}-1]$	integer
`long`	4	$[-2^{31}, 2^{31}-1]$	identical with `int`
`unsigned short`	2	$[0, 65535]$	integer
`unsigned int`	4	$[0, 2^{32}-1]$	integer
`unsigned long`	4	$[0, 2^{32}-1]$	identical with `unsigned int`
`signed short`	2	$[-32768, 32767]$	identical with `short`
`signed int`	4	$[-2^{31}, 2^{31}-1]$	identical with `int`
`signed long`	4	$[-2^{31}, 2^{31}-1]$	identical with `int`
`float`	4	$\pm[1.17e{-}38, 3.4e38]$	Floating point type with a precision of 7 decimal digits. However, don't use `float`! The type `double` is much more precise with practically the same efficience.
`double`	8	$\pm[2.2e{-}308, 1.79e308]$	Floating point type with a precision of 15 decimal digits. Usually it is advisable to use `double` for all floating point computations. Floating point numbers like 324.343 are automatically interpreted as of type `double`.
`long double`	8	$\pm[2.2e{-}308, 1.79e308]$	identical with `double`
`bool`	1	`true, false`	`true` is identical with 1, `false` with 0. Attention: *All* values $\neq 0$ are converted into `true` by the compiler when interpreted as `bool`.
`void`	0	-	Return type for function without a return value (must not be ommitted!).

3.6 Operators

An *operator* is a symbol that performs an action. For instance, the operator `+` is used to add two numbers. Further examples of operators are
`-` (subtraction),
`*` (multiplication),
`/` (division),
`<<` (left shift operator used in connection with `cout`),
`>>` (right shift operator used in connection with `cin`).

Warning: A C++ operator for exponentiation does not exist! In particular, the operator `^` cannot be used for exponentiation. If you try `cout << 2^3;`, for instance, you will get a compiler error.

3.6.1 Binary Arithmetic Operators

Binary operators take two values as input and produce a new value from it. For example, if we write `3+10` in C++ then the operator `+` takes the numbers `3` and `10` as input and produces the new value `13`.

Operator	Explanations
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division. For division of integers, the fractional part is discarded. For instance, `4/3` yields `1` and not `1.333...`
`%`	Modulus. The remainder of a division. For instance `20%5` yields `0` while `50%7` yields `1`.

3.6.2 Arithmetic assignment operators

In C++ it is possible to combine an arithmetic operation with an assignment: Instead of `a=a+b;` we can just write `a+=b;`. Similarly, `a=a*b;` is equivalent to `a*=b;` and likewise for the other three arithmetic operations.

3.6.3 Comparison operators

These operators compare two values and return either `1` for `true` or `0` for `false`.

Operator	Explanations
`<`	`a<b` returns `true` if `a` is strictly less than `b` and `false` otherwise.
`<=`	`a<=b` returns `true` if `a` is less than or equal to `b` and `false` otherwise.
`>`	Similar to `<`
`>=`	Similar to `<=`
`==`	Test for equality. Returns `true` if the left and ride side have the same value and `false` otherwise. Using the assignment operator `=` instead of `==` is a common mistake.
`!=`	Test for "not equal". Returns `true` if the left and the right side are not equal and `false` otherwise.

3.6.4 Logical operators

The comparison operators are provide us with "logical expressions", i.e., expressions whose value is `true` or `false`. For instance, `4<5` is a logical expression with value `true`.

Logical operators are used to combine logical expressions:

Operator	Explanations
`&&` (and-operator)	Let `A` and `B` be logical expressions. Then `A&&B` is also a logical expression and `A&&B` has value `true` if and only if `A` *and* `B` both have value `true`.
`||` (or-operator)	Let `A` and `B` be as above. Then `A||B` is a logical expression and `A||B` has value `true` if and only if at least one of `A` *or* `B` has value `true`.

The operator `&&` has higher priority than `||`, i.e. `&&` is evaluated first. To force `||` to be evaluate first, we can use parenthesis. Actually, it is never a bad idea to put parenthesis around logical expressions - makes the program easier to read and less error prone. Examples:

```
cout << ((4<5) || (4<3) && (2<1)) << endl;
cout << (((4<5) || (4<3)) && (2<1)) << endl;
cout << (4<5) || (4<3) && (2<1) << endl;  // compiler error
```

The first line gives `1` (`true`) and the second line gives `0` (`false`). Why? The third line causes a compiler error - the missing parenthesis around the whole logical expression lead to a conflict with the output operator `<<`. Errors like this always can be easily avoided by using enough parentheses. No use in trying to be clever and minimize the number of parentheses...

3.6.5 Unary operators

Unary operators take only one value as input and produce a new value from it. For example, if we write `a++` in C++ then the operator `++` adds `1` to the variable `a`.

Operator	Explanations
`!`	Logical negation. If `a` is `true`/`false` then `!a` returns `false`/`true`, i.e. the negated boolean value of `a`.
`++`	Increment operator. `a++` (postfix) increases the value of `a` by 1, but returns the original value of `a` (!). On the other hand, `++a` (prefix) also increases the value of `a` by 1, but returns the new value of `a`.
`--`	Decrement operator. Similar to `++`, but decrements by 1.

3.7 Literals, scientific notation, expressions

A C++ *literal* is a part of C++ code which takes its value "literally". This may sound confusing, but is easy to understand by example:

```
cout << 1324;          // 1324 is an integer literal
cout << 12.334;        // 12.334 is a double literal
cout << "hello";       // "hello" is a string literal
```

In other words, literals are values which we can type directly into C++ code. By the way, for `double` literals we can use the so-called *scientific notation* in C++. Examples:

```
1e10       // means 10^10
0.34e4     // means 0.34 · 10^4
0.34e-200  // means 0.34 · 10^-200
```

A C++ *expression* is any sequence of C++ literals, variable names and C++ operators which does not violate the C++ syntax rules. Expressions can be very simple, but also extremely complicated. Actually, in our C++ code, we should always try to keep the expressions simple. Complicated expression often lead to mistakes. Examples of expressions:

```
23423                 // every C++ literal is an expression
a*b*c-2*b             // expression involving variable names
                      // and operators
a=34.340340           // expression involving a variable name,
                      // an operator and a double literal
```

Note that the last two expressions are only correct if the variables `a,b,c` have been declared previously (variables always must be declared before using their names!).

In C++ *every expression has a value*. This value is called the *return value* of the expression and is computed automatically by the program. How can we find out the return value of an expression? This is very easy - we just have to use `cout <<` *expression*`;`. Then the return value of the expression is printed to the screen. Example:

```
int a=5, b=10;
cout << a*b-10;
```

The return value of the expression `a*b-10` in this example is `40`. Hence `40` is printed to the screen. You can easily do some experiments to find out the return values of other expressions. What is the return value of an assignment like `a=5`, for instance? (do not forget to declare `a` first!)

4 Loops

Scientific computations often involve performing an operation *repeatedly*. For example, when we compute

$$\sum_{n=1}^{1000000} \frac{1}{n^2}, \tag{1}$$

we need to do $1,000,000$ multiplications $(n \cdot n)$, $1,000,000$ divisions $(1/n^2)$ and $999,999$ additions (actually, we could save some, but that's marginal). For such repeated operations we use *loops* in C++. There are three kinds of loops in C++: for-, while- and do-while loops. These three types of loops are the subject of this section.

Remark The computation of the huge sum (1) in C++ using type `double` only takes around 15 milliseconds on a common PC. The value of the sum (1) is very close to $\pi^2/6$, a fact that was discovered by the great mathematician Euler.

Remark *Remarks* in the lecture notes contain some additional information which is not tested in the exams.

4.1 for-loops

When we do operations repeatedly, each step is called an *iteration*. For example, we need $1000,000$ iterations for the computation of the sum (1) and each iteration consists of one division, one multiplication and one addition. When we compute the sum (1), we *know the number of necessary iterations in advance*. In other cases, for example, if we repeatedly add random numbers until the sum exceeds 100,000, we do *not* know the number of iterations in advance.

In case we *know the number of iterations in advance*, the most suitable type of loop is a for-loop.

Syntax

`for(`*initialization*`;` *condition*`;` *increase*`)`
    statement

Explanations

- The *statement* is executed repeatedly as long as the *condition* is true.
- Each execution of the *statement* is called an *iteration*.
- Recall that a statement can be a single command or a statement block.

- The *initialization* and *increase* can be any valid statements. Note that the *increase* statement is *not* ended by a semicolon.
- In almost all cases the *initialization* is used to initialize a *counting variable* which simply counts the number of completed iterations.
- In almost all cases the *increase* statement is used to increase the value of the counting variable.

Example 1

```
for(int i=0;i<100;i++)
    cout << i << endl;
```

Explanations

- This for-loop prints the numbers `0`,`1`,...,`99` to the screen, each number on a new line.
- The *initialization* statement of this for-loop is `int i=0;`. This sets the counting variable `i` to `0`.
- The *condition* is `i<100`. Hence the for-loop will be executed until `i` has the value `100`. In particular, this means that the value of `i` after completion of the for-loop is `100` (not `99`!).
- However, the number `100` will *not* be printed to the screen. The for-loop is terminated *immediately* when the condition `i<100` is false, hence the `cout`-statement will not be reached anymore when `i` has value `100`.
- The *increase* statement of this for-loop is `i++`. This increases the value of `i` by `1` in each iteration. In other cases, it can be better to use a different step size. Using `i+=2` instead, for example, we would only print every second number.
- The *statement* of this for-loop is `cout << i << endl;` This prints the number `i` to the screen, followed by a new line.
- Note that it is allowed (and most times necessary) to use the counting variable in the statement of the for-loop.

Example 2

```
for(int i=99;i>=0;i--)
    cout << i << endl;
```

Explanations

- Similar the Example 1. The difference is that the numbers `0`, `1`,...,`99` are printed to the screen *in reverse order*.
- Recall that `i--` *decreases* the value of `i` by 1.

Example 3

```
1 int x=0;
2 for(int i=0;i<10;i++)
3 {
4     x += rand();
5     x -= rand();
6 }
7 cout << x << endl;
```

Explanations

- In this for-loop, random numbers are alternatingly added to and subtracted from a variable `x`. Here we have the rare case where the statement of a for-loop does not contain the counting variable. The reason for this is that we can create a random number without knowing the number `i` of the current iteration.
- On the left side, line numbers are indicated. These line numbers *do not belong to the C++ code*, they are only given for the purpose of explanation.
- This for-loop performs operations on the variable `x`. If we want to perform operations in a for-loop on a variable which is not a counter variable, then *we must declare and initialize it before the for-loop*. This is done in line 1.
- `rand()` produces a random integer in the interval `[0,32767]`.
- In each iteration of the for-loop, one random number is added to `x` (line 4) and one random number is subtracted from `x` (line 5). Since each execution of `rand()` gives a *new random number*, `x` will not become `0` in general!
- Lines 3-6 comprise the statement of the for-loop. Here the for-loop statement is a *statement block* consisting of two commands (lines 4 and 5).
- Line 7 does *not* belong to the for-loop since a for-loop ends at the same spot where its statement ends (here line 6).

Example 4

```
1  double sum=0;
2  for(double n=1;n<=1000000;n++)
3     sum = sum + (1/(n*n));
4  cout << sum << endl;
```

Explanations

- By this for-loop, the sum (1) is computed.
- When we compute a sum by a for-loop, *we need a variable to store the intermediate results*. This variable has to be declared and initialized *before the for-loop*. This is done in line 1.
- We use a counter variable `n` of type `double` here. It would be wrong to use the type `int`. If `n` was of type `int`, then `1/(n*n)` would have value `0` for all `n>1` (integer division discards the fractional part).
- Line 3 contains the statement of the for-loop. Like in Examples 1 and 2, it is only one single command.
- Line 4 *does not belong to the for-loop*!! If we do not use a *statement block* (enclosed by curly braces) as the for-loop statement, then *only the single command* following the for-loop belongs to the for-loop.
- Since line 4 does not belong to the for-loop, only the final result will be printed to the screen.

Remark How long does your computer need to compute the sum (1)? Using some C++ commands you can actually find out how much time the computation needs on your processor ("CPU time"). This can be done as follows.

- Put `#include<ctime>` at the beginning of your program.
- Directly before the for-loop, insert the command `double start=clock();` this measures the start time.
- Directly after the for-loop, insert the command `double end=clock();` this measures the time after the completion of the for-loop.
- The CPU time in seconds you can print to the screen by

  ```
  cout << (end-start)/CLOCKS_PER_SEC << endl;
  ```

  The division by the built in constant `CLOCKS_PER_SEC` is necessary to convert the time to seconds.

4.2 while-loops

When we do an operation repeatedly, sometimes we know the number of iterations in advance and sometimes not. If we know the number of iterations in advance, we should use a for-loop. If we do *not* know the number of iterations in advance, the best choice usually is a *while*-loop.

Syntax

`while(`*condition*`)`
statement

Explanations

- The *statement* is executed repeatedly as long as the *condition* is true.
- Each execution of the *statement* is called an *iteration*.
- Recall that a statement can be a single command or a statement block.
- Make sure that the *condition* becomes false after finitely many iterations! Otherwise the execution of the while-loop will never stop. This is called an *infinite loop*.
- A while-loop usually contains no variable declarations or initializations Hence variables used in a while-loop must be declared and initialized *before the while-loop*.

Example 5

```
1 int x=0;
2 while(x<100)
3 {
4     cout << x << endl;
5     x++;
6 }
```

Explanations

- This while-loop prints the numbers `0`,`1`,...,`99` to the screen.
- The condition of this while-loop is `x<100`.
- The statement of this while-loop is the statement block contained in lines 3-6.
- Note that the command `x++` is crucial. Without it, the value of `x` would never change and we would have an infinite loop.
- Note that `x` must be declared and initialized *before the while-loop*.
- If we interchange lines 4 and 5, then the numbers `1`,`2`,...,`100` are printed to the screen. Can you explain why?
- Actually, in this example we know the number of iterations in advance: it is 100. So it would be better to use a for-loop (but to learn the syntax of a while-loop, the example is suitable).

Example 6

```
int x=0;
while(x<=1000000)
     x+= rand();
cout << x << endl;
```

Explanations

- This while-loop adds random numbers to the variable `x` until it becomes larger than a million.
- The statement of this while-loop is the single command `x+= rand();` which adds a random number to `x`.
- In this example, we do *not* know the number of iteration in advance, so we *should not use a for-loop* instead of the while-loop.

Example 7

```
int x=0;
int counter=0;
while(x<=1000000)
{
     x+= rand();
     counter++;
}
cout << "The number of iterations is " << counter << endl;
```

Explanations

- Like in Example 6, this while-loop adds random numbers to a variable `x` until it becomes larger than a million.
- However, this example also demonstrates the use of a *counter variable* in a while-loop: the variable `counter` is increased by `1` in each iteration of the while-loop. Hence, after completion of the while-loop, the value of `counter` is the number of iterations which were done.

Example 8

```
while(1)
   cout << "Please stop me!" << endl;
```

Explanations

- This is an example of what must be avoided: an infinite loop. It prints `Please stop me!` on the screen forever, each message on a new line.
- The condition of this while-loop is `1`. Recall that anything different from `0` is interpreted as `true`. Hence the condition `1` is always true.

4.3 do-while-loops

It can happen that the condition of a while-loop is false right at the beginning. This means that *no iteration is performed at all*. If you want to make sure that at least one iteration is performed, then you can use a do-while-loop instead.

Syntax

```
do
    statement
while(condition);
```

Explanations

- The statement is performed at least once in any case.
- The statement is repeated until the condition becomes false.
- Note the semicolon at the end. It is necessary.

Example 9

```
1 int x=rand();
2 do
3     x+= rand();
4 while(x<=20);
5 cout << x << endl;
```

Explanations

- This do-while-loop adds random numbers to a variable `x` until it becomes bigger than `20`.
- After line 1, the value of `x` is a random number.
- *In any case*, the do-while-loop will add another random number to `x`. In other words, the first iteration of the loop is done *in any case*.
- However, the *second* and subsequent iterations are only performed if `x` is at most `20`.

4.4 `continue` and `break`

Often it is useful or convenient to interrupt the execution of loops.

To interrupt *just the current iteration of a loop*, we can use the command `continue`. This command terminates the current iteration of a loop immediately. However, the execution of the loop *continues* (thus the name!) with the next iteration.

On the other hand, to stop the execution of a loop immediately and *completely*, we can use the command `break`. After `break`, no further iterations of the loop are executed at all.

The commands `continue` and `break` only make sense in combination with if-conditions which we will study in the next section. Therefore, we postpone the example for `continue` and `break` until then.

5 Conditions

Imagine a program which requires user input; for instance, the program may ask the user to enter a date. After the input, the program will perform some computations which depend on the input. The output of the program, for example, the weekday corresponding to the date, will also depend on the input. The dependence on the input implies that *we do not know in advance which values the variables in the program will have during the execution of the program.* Say, for example, that a variable `w` represents weekdays and `0` represents Saturday. If the programs computes that `w` is `0`, then we should print `Saturday` to the screen. If the result is that `w` is `5`, then we should print another weekday.

All this means that we must be able to *make decisions* in a program *depending on conditions*. In C++ this is done by using if- or if-else-conditions.

5.1 if-conditions

Syntax

`if(`*condition*`)`
statement

Explanations

- If the condition is true, then the statement will be executed.
- If the condition is false, then the statement will *not* be executed.
- Recall that a statement can be a single command or a statement block.

Example 10

```
1 int x= rand();
2 if(x<100)
3     cout << "x is quite small" << endl;
4 cout << "End of program" << endl;
```

Explanations

- In line 1, the variable `x` is set to a random number. Note that we *do not know the value of* `x` *in advance*. This is the reason why a if-condition can be useful here.
- Line 3 contains the statement of the if-condition. It is a single command.
- The message `x is quite small` is only printed to the screen of `x` is smaller than `100`.
- Note that line 4 *does not belong to the if-condition*. Hence the message `End of program` will be printed *in any case*.

Example 11

```
1 int color;
2 cout << "Enter a color (0=red, 1=blue, 2=yellow): " << endl;
3 cin >> color;
4 if(color==0)
5     cout << "You entered red" << endl;
6 if(color==1)
7     cout << "You entered blue" << endl;
8 if(color==2)
9     cout << "You entered yellow" << endl;
```

Explanations

- The user has to determine the value of the variable `color`. Hence we do not know the value of `color` in advance.
- Depending on the value of `color`, a confirmation message is printed.
- Recall that we must use `==` to test for equality. It would be wrong to use `=`.
- If the user input is valid (one of the numbers `0`, `1` or `2`), then exactly one of the statements in lines 5,7 and 9 will be executed.

5.2 if-else conditions

Syntax

```
if(condition)
   statement1
else
   statement2
```

Explanations

- If the condition is true, then *statement1* will be executed.
- If the condition is false, then *statement2* will be executed.
- There actually can be more than one else part. However, this can be quite confusing and always can be avoided. If you want to use more than else part, please consult the textbook.

Example 12

```
1 int x= rand();
2 if(x<10000)
3 {
4     x++;
5     cout << 10000-x << endl;
6 }
7 else
8    cout << "x is large" << endl;
```

Explanations

- The variable `x` is set to a random number.
- Lines 2-5 comprise *statement1* of the if-else-condition. This statement is executed if and only if `x` is smaller than `10,000`. Hence, if `x` is smaller than `10,000`, it is increased by `1` and then `10,000-x` is printed to the screen.
- If and only if `x` is larger or equal to `10,000`, the message `x is large` is printed to the screen.

5.3 switch

In C++ there is one more method to make decisions depending on conditions: the switch structure. However, switch structures always can be replaced by if-else conditions and many programmers do not use it all. If you want to use it, please consult the textbook.

5.4 continue and break: examples

Example 13

```
1 int x=0;
2 while(1)
3 {
4      x+= rand();
5      if(x>1000000)
6         break;
7  }
8  cout << x << endl;
```

Explanations

- In this example, random numbers are added to a variable `x` until it becomes larger than a million.
- Note that the condition of the while-loop is just `1` which is always true. The only way to stop such a loop is to use the `break` command.

- The crucial part here is lines 5 and 6. The while loop is stopped completely as soon as `x` is larger than a million.

Example 14

```
1  int number, counter=0, sum=0;
2  cout << "Enter two even integers: "<< endl;
3  while(counter<2)
4  {
5      cin >> number;
6      if(number%2==1)
7      {
8          cout << "This number is odd." << endl;
9          cout << "Enter an even integer: " << endl;
10         continue;
11      }
12      sum+=number;
13      counter++;
14  }
15  cout << "The sum of your even integers is " << sum << endl;
```

Explanations

- This program asks the user to input two even integers and outputs the sum of these integers.
- The difficulty of this program is that we want to reject odd numbers. In case the user enters an odd number, the program asks the user to input another integer.
- The condition `number%2==1` in true if and only if the value of `number` is odd. Hence lines 8-10 are executed if and only if the value of `number` is odd.
- If the value of `number` is odd, then the `continue` statement is executed. This means that the program does not proceed with lines 12 and 13, but immediately goes to the next iteration (line 5).
- Note that the statement `counter++` is executed if and only if the value of `number` is even. Hence the while-loop terminates only when two even integers where entered.
- It does not matter how many odd integers were entered. Even if 1000 odd integers were entered, but only 1 even integer, the while-loop still runs.

6 Arrays and vectors

Imagine we are provided with 10,000 floating point values which describe the outcome of a statistical experiment. Our goal is to process these data with C++, for instance, compute the mean, variance etc.

How will we enter the data into our program? If we just use variables of type `double`, then we have to type 10,000 commands like `double x1 = 234.34;` This is certainly unacceptable. We need methods to process *collections of values of the same type* more efficiently. Exactly this is the purpose of arrays and vectors in C++.

In C++, arrays and vectors are both *collections of values of the same type*. The number of entries of an array or vector is called its *length* or *size*. Vectors are the "modern form" of arrays and are better than arrays in any respect: much easier to use and sometimes more efficient. *If possible, use vectors, not arrays!*. There are only some rare cases when you need to use arrays, for example, if you use programming libraries which are written in the language C and need arrays as input.

6.1 Arrays

Arrays have been superseded by vectors, but they are still fundamental for understanding of C++ programming.

6.1.1 Declaration

Syntax

type arrayName`[`*size*`];`

Explanations

- This is a declaration of an array.
- The name of the array is *arrayName* (you have to choose a name).
- The number of entries of the array is *size*. This must be a positive integer constant.
- The type of the entries of the array is *type*.

Example 15

```
1 int A[10];
2 double B[100];
```

Explanations

- Line 1: declaration of an array with name `A`, length `10` and entries of type `int`.
- Line 2: declaration of an array with name `B`, length `100` and entries of type `double`.

6.1.2 Accessing the entries

The entries of a C++ array are accessed by using the array name followed by the index of the entry we want to access in square brackets. For example, `A[4]` is the entry with index `4` of an array `A`.

There is one important fact we must always be aware of: in C++, array indices start with `0`, not with `1`. Hence the entries of an array `A` of length `n` are `A[0]`, `A[1]`,...,`A[n-1]`, not `A[1]`, `A[2]`,...,`A[n]`. Erroneously using `A[n]` can cause a nasty problem: a runtime error.

Using indices in square brackets, we can not only access the entries of an array, but also *set their values*.

Example 16

```
1 int A[3];
2 A[0]=23;
3 A[1]=234;
4 A[2]=934;
5 cout << A[0] << endl;
6 cout << A[1] << endl;
7 cout << A[2] << endl;
```

Explanations

- Line 1: Declaration of an array `A` of length `3` and entries of type `int`.
- Lines 2-4: The entries of `A` are set (initialized).
- Lines 5-7: The entries of `A` are accessed for printing to the screen.

Usually the arrays we use are quite big and we cannot deal with their entries one by one. That's why we use for-loops most of the time when we do computations with arrays.

Example 17

```
1 int A[100000];
2 for(int i=0;i<100000;i++)
3     A[i]=rand();
4 for(int i=10000;i<10010;i++)
5     cout << A[i] << endl;
```

Explanations

- Line 1: Declaration of an array `A` of length `100000` and entries of type `int`.
- Lines 2-3: The entries of `A` are set to random numbers.
- Lines 4-5: The entries with indices `10000`,...,`10009` of `A` are accessed for printing to the screen.

6.1.3 Arrays are dangerous!

As explained in a previous section, runtime errors are among the worst errors which occur in programming. When using arrays, we should keep the following fact in mind.

Wrong use of arrays causes almost all runtime errors.

This means that arrays can cause a lot of trouble if we do not use them properly. However, if we are careful we can avoid these problems or at least remove them without too much effort. The following examples show some typical mistakes in using arrays which must be avoided.

Example 18

```
1 double A[10];
2 cout << A[2] << endl;  // error!
```

Explanations

- Line 1 only declares the array `A`. The entries are *not initialized yet.*
- In line 2, a not initialized value is accessed. The program will run, but the result is unpredictable.

Example 19

```
1 double A[10];
2 A[10] = 1232;     // error!
```

Explanations

- In line 2, a non-existing entry of `A` is accessed (the last entry of `A` is `A[9]`).
- The result is unpredictable. Sometimes, this will cause a runtime error. It also can happen that the mistake will remain unnoticed and may cause errors later which are extremely hard to detect.

Example 20

```
1 double A[1000000];  // error!
```

Explanations

- On a common PC, this causes a runtime error since the array is too big for the memory C++ uses for arrays.

Example 21

```
1 int n=50;
2 double A[n];  // error!
```

Explanations

- This causes a compiler error. The size of an array must be an integer constant (e.g. integer literal).
- If we want to use variables for array sizes, we must use *dynamic memory allocation* (done later in this course).

6.2 Vectors

C++ vectors are *not* part of the built-in types of C++. They are part of an extension of C++ called the *Standard Template Library*. Some people consider the Standard Template Library as "advanced" and do not include any part of it in introductory courses for C++. However, vectors are much easier to use than arrays and make life easier - especially for beginners. That's the reason why we will prefer to use vectors.

6.2.1 Declaration and accessing the entries

Syntax

`vector<`*type*`>` *vectorName*`(`*size*`);`

Explanations

- This describes the declaration of a vector.
- At the beginning of program, we have to put `#include<vector>` in order to use vectors.
- The name of the vector is *vectorName* (you have to choose a name).
- The number of entries of the vector is *size*. This must be a nonnegative integer.
- The type of the entries is *type*.
- Contrary to arrays, the entries of a vector are *automatically initialized*. If the entries are numbers (integer or floating point), then all entries will automatically be set to `0`.
- The entries of vectors are accessed in the same way as entries of arrays: by the name of the vector followed by an index in square brackets.
- Contrary to arrays, the size of a vector *can be a variable*. The variable must be of integer type and must have a nonnegative value.

Example 22

```
1 #include<vector>
2 ...
3 vector<int> A(3);
4 A[0]=23;
5 A[1]=234;
6 A[2]=934;
7 cout << A[0] << endl;
8 cout << A[1] << endl;
9 cout << A[2] << endl;
```

Explanations

- This example is very similar to Example 16. The only differences are in lines 1 and 3.
- The include statement in line 1 is necessary since the type vector is contained in a library which is included in this way.
- In line 3, a vector `A` of length `3` with entries of type `int` is declared.
- After the declaration, all `3` entries of `A` are `0`.
- In lines 4-6, the values of the entries are changed.
- In lines 7-9, the values of the entries are accessed for printing to the screen.

Example 23

```
1  #include<vector>
2  ...
3  int size;
4  cout << "Size of your vector: " << endl;
5  cin >> size;
6  vector<int> A(size);
7  for(int i=0;i<size;i++)
8      A[i]=rand();
9  cout << "vector of size " << size
10      << " has been initialized with random numbers"
11      << endl;
```

Explanations

- This example demonstrates that the size of a vector can be a variable. Here, the value of the variable is determined by the user.
- Lines 3-5: the size of the vector is read from the keyboard (user input).

- Line 6: a vector `A` of this size is declared.
- Lines 7-8: the entries of the vector are initialized to random numbers.
- Line 9-11: a confirmation message is printed to the screen.

6.2.2 Useful functions for vectors

For vectors several useful functions are available. The most important are the following.

`size()`	returns the size of the vector (the number of its entries)
`push_back(value)`	appends the `value` at the end of the vector
`resize(n)`	changes the length of the vector to `n`
`clear()`	removes all elements of the vector and changes the size to `0`

To apply these functions to a vector, we use the name of the vector, a dot, and then the function name. For instance, `A.size()` returns the size of a vector `A`. Note that the brackets `()` are necessary.

Example 24

```
1 vector<int> A(1);
2 A.clear();
3 A.push_back(2);
4 A.resize(10);
5 cout << A.size() << endl;
6 for(int i=0;i<A.size();i++)
7       cout << A[i] << " ";
```

Explanations

- Line 1: a vector of size `1` with integer entries is created. The entry is automatically set to `0`.
- Line 2: all entries are removed. The size of the vector becomes `0`.
- Line 3: the entry `2` is appended to the vector. The size becomes `1`.
- Line 4: the size of the vector is changed to `10`. The entries `A[1]`, ...,`A[9]` are automatically set to `0`. The entry `A[0]` remains `2`.
- Line 5: output will be `10`.
- Lines 6-7: all entries of the vector are printed to the screen, separated by spaces. The output will be `2 0 0 0 0 0 0 0 0 0`.

7 Input and Output

Why do we need to write computer programs in the first place? Because many scientific problems involve way *too many data* to be handled by hand computations. This means that useful computer programs commonly must be able to deal with *huge amounts of data*.

There are two important types of data used in computer programs: *input data* and *output data*. On the one hand, the input data are the data which are *known already* and which need to be *read into the program* as a basis for the computation it performs. On the other hand, the output data contain the *results* which were obtained by the program.

For example, consider a program for evaluating the outcome of a physical experiment. Imagine that 10,000 measurements are taken and give us 10,000 floating point values. These 10,000 floating point values comprise the *input data* and have to be read into the program. The task of the program will be to determine the pattern behind the data. This is usually is done by computing certain parameters like mean, variance and other "estimators". These results comprise the *output data* of the program.

Input data can be read into a program from the keyboard *if the data set is small*. This is called *keyboard input*. If the data set is big, the input data must be saved in computer files and have to be read into the program from these files. This is called *file input*.

The situation for output data is similar. If the size of the output is moderate, we simply can print it to the screen. This is called *screen output*. If the size of the output is large, we should write it into computer files. This is called *file output*.

Summary:

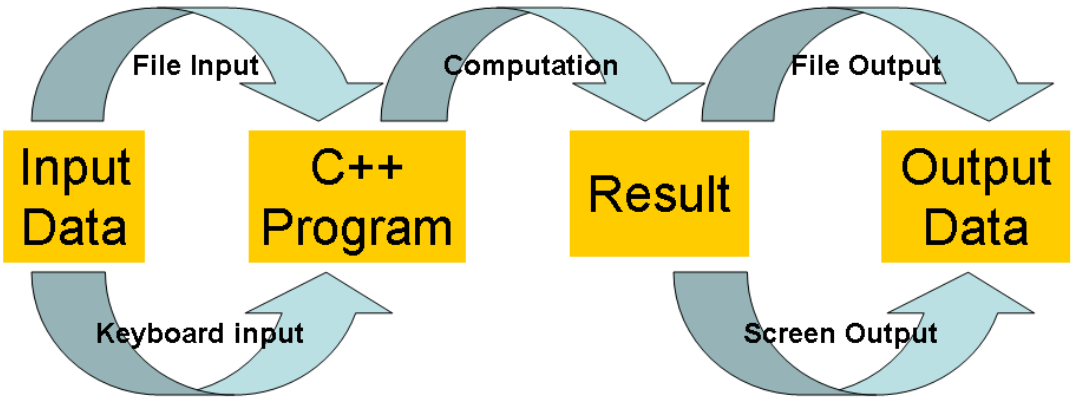


Figure 1: Data Flow for a C++ Program

7.1 Keyboard input

Syntax

`cin >>` *variableName*`;`

Explanations

- This allows the user to set the value of the variable with name *variableName* during the execution of the program. The program will stop at this point and wait for the input of the user.
- The variable with the name *variableName* must have been declared declared already *before the* `cin`*-command*.
- The input is done by the user by typing a suitable value on the keyboard.
- The user must end the input by pressing `Enter`.
- The use of `cin` requires `#include<iostream>`.
- Input by `cin` can be concatenated. For instance, we can use `cin >> x >> y;` to input the values of two variables `x` and `y`.
- Two or more values can be entered in one step from the keyboard; they just have to be separated by spaces.
- Spaces or tabs are interpreted by `cin` *only as seperators*. Ten spaces have the same effect as one space.
- Keyboard input is *error prone* since user input often is incorrect. With `if(cin)` we can check weather the input was successful. This condition will true if and only if the input was successful.

Example 25

```
1 int x,y;
2 cout << "Enter two integers: " << endl;
3 cin >> x >> y;
4 if(!cin)
5     cout <<  "input failed!" << endl;
6 if(cin)
7     cout << "sum: " << x+y << endl;
```

Explanations

- Line 1: variables whose values we want to read from the keyboard must be initialized first.
- Line 2: it is advisable to explain to the user which kind of input is expected.

- Line 3: this is the crucial step where the values of `x` and `y` have to be entered by the user.
- Lines 4-5: the return value of `cin` *after the input* will be nonzero if and only if the input was successful. Hence `!cin` will be true if and only if there was an error. So, in this case, an error message is printed.
- Lines 6-7: in case of successful input, the sum of the entered values in printed.

7.2 Screen output

Syntax

`cout <<` *expression*`;`

Explanations

- This prints the return value of the expression to the screen.
- If the expression is a literal or variable, this prints its value to the screen.
- A string (piece of text) is printed to the screen by putting it between quotation marks, for instance, `cout << "hello";`
- A space is printed by `cout << " ";`
- A new line is printed by `cout << endl;`
- Screen output can be concatenated, for instance, `cout << x << " " << y << endl;`
- `cout << x;` works for all variables whose type is a *fundamental data type*. Variables of other types, for instance, arrays or vectors have to be output differently.
- The use of `cout` requires `#include<iostream>`.

7.3 File Input

We have learned how to use `cin` to read data from the keyboard. This - at the moment somewhat mysterious - `cin` is called an *input stream object* since it sends a "stream of data" from the keyboard to the C++ program.

File input means reading data from a computer file into a C++ program. File input is *very similar* to keyboard input with `cin`. All we have to do to read data from a file, is to *declare another input stream object* which reads from the file instead of the keyboard. Such an input stream object will have a name different from `cin`, but can be used in *almost exactly the same way as cin.*

Syntax

`ifstream` *streamName*`(`*fileName*`);`

Explanations

- This declares an input stream object with name *streamName* which reads from the file *fileName*.
- After the declaration, *streamName* can be used completely similarly to `cin`.
- The input is not from the keyboard, but from the file *fileName*.
- The *fileName* can be an absolute path, for instance, `"C:\\test.txt"` for the file with absolute path `C:\test.txt`. Note that the file name must be put in quotation marks and that a backslash in a path must be replaced by a double backslash.
- The *fileName* also can be just the *name of a file* without the full path. In this case, the file *must be contained in the same folder as the executable C++ program.*
- The use of `ifstream` requires `#include<fstream>`.

Example 26 Assume we have saved a textfile `test.txt` in the folder `C`. Then the absolute path of this file is `C:\test.txt`. Assume the content of the file is the following.

```
1001 71
3.14 2.18
```

We can use the following C++ code to read the values contained in the file `test.txt` into a C++ program.

```
1 int a,b;
2 double x,y;
3 ifstream in("C:\\test.txt");
4 in >> a >> b >> x >> y;
5 cout << a << " " << b << endl;
6 cout << x << " " << y << endl;
```

Explanations

- Lines 1-2: Before we can read data into our program, we need to declare variables to store the values.
- Line 3: this is the crucial part. An input stream with name `in` is declared which reads from the file with absolute path `C:\test.txt`. After line 3, the input stream `in` can be used completely similarly to `cin`.
- Line 4: this reads the four values from the file into the variables `a,b,c,d`. Note that the *types must match*; otherwise there could be an input error or a loss of information. A loss of information can occur, for example, if a `double` value is read into a variable of type `int`: the fractional part would be lost.

- Spaces, tabs and newlines in the file are *all interpreted only as seperators*. For instance, it would not matter at all if we included another 1,000 newlines in the file `test.txt`. The result would be the same.
- Lines 5-6: the values which were read are printed to the screen to be sure that everything worked.

7.4 File Output

We have learned how to use `cout` for printing to the screen. This - at the moment slightly mysterious - `cout` is called an *output stream object* since it sends a "stream of data" from the C++ program to the screen.

File output means writing data from a C++ program into a computer file. File output is *very similar* to printing to the screen with `cout`. All we have to do to write data to a file, is to *declare another output stream object* which writes to the file instead of the screen. Such an output stream object will have a name different from `cout`, but can be used in *exactly the same way as cout*.

Syntax

`ofstream` *streamName*`(`*fileName*`);`

Explanations

- This declares an output stream object with name *streamName* which writes to the file *fileName*.
- Note that the only difference to declaring an input stream object is the use of `ofstream` instead of `ifstream`.
- After the declaration, *streamName* can be used completely similarly to `cout`.
- The output is not to the screen, but to the file *fileName*.
- If it does not exist, the file *fileName* is automatically created by the C++ program. If it exists, it is deleted (!!) and a new empty file with this name is created. Note that this requires some caution since otherwise we may loose important data in some cases.
- The *fileName* can be an absolute path, for instance, `"C:\\test.txt"` for the file with absolute path `C:\test.txt`. Note that the file name must be put in quotation marks and that a backslash in a path must be replaced by a double backslash.
- The *fileName* also can be just the *name of a file* without the full path. Then the file is created in the same folder as the executable C++ program.
- The use of `ofstream` requires `#include<fstream>`.

Example 27

```
1 ofstream out("C:\\test.txt");
2 for(int i=0;i<1e6;i++)
3      out << rand() << endl;
```

Explanations

- This writes one million random numbers to the file `C:\test.txt`.
- Line 1: an output stream object with name `out` is declared which writes to the file `C:\test.txt`.
- Caution: the declaration in line 1 already deletes the file `C:\test.txt` if it existed before!
- After the declaration, `out` can be used exactly like `cout`.
- Lines 2-3: one million random number are written the the file `C:\test.txt`, each number on a new line.

8 Functions

In Mathematics, a function is a mapping which has a unique value for each of its arguments. For instance, consider the function $f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$. The *value* of f at the *argument* x is $f(x) = x^2$. Mathematical functions also can have several arguments, for instance, we can define a function by $g(x, y, z) = x^2 + y^2 + z^2$.

In C++, there are also functions and they behave quite similarly. The arguments of C++ functions are called *function parameters*. Like mathematical functions, C++ functions can have have one or more parameters. However, C++ functions can also have *no parameter at all*. The values of C++ functions are called *return values*. A C++ function *can* have a return value, but there are also C++ functions without any return value.

C++ functions have a *much broader purpose than mathematical functions*. The idea behind C++ functions is simple, but extremely powerful: break up a complex task into small steps.

> *The purpose of C++ functions is to divide programs into small independent blocks. Each block (=function) gets a name and can be executed - repeatedly if necessary - just be typing its name.*

Advantages of using functions

- The C++ commands contained in a function can be executed as often as we like - just by typing the name of the function. This makes programs shorter and easier to write.

- Functions provide *structure* to our programs. Like a text is easier to read if it is structured into sections, subsections and paragraphs, a C++ program is easier to understand if it is structured into functions.
- Since functions are *independent* blocks of a program, we can understand the program step by step by understanding its functions one by one.
- A crucial task in programming is ensuring the *correctness* of programs. This is done by repeated *testing*. Testing a complete program can be very hard if its parts are interdependent. Functions are *independent* parts of the program and can be tested one by one.

8.1 Using built-in C++ functions

8.1.1 Why we should use built-in functions

Let x be a positive real number. In Calculus you can learn about Newton's method which shows that we can obtain the numerical value of $\sqrt{x}$ in the following way: start with $y = 1$ and repeatedly replace y by $(y + \frac{x}{y})/2$. Then y will rapidly approach $\sqrt{x}$. For instance, when we compute $\sqrt{2}$, we get the following values for y.

1,
$(1 + 2/1)/2 = 3/2 = 1.5$,
$(3/2 + 2/(3/2))/2 = 17/12 \approx 1.4167$,
$(17/12 + 2/(17/12))/2 = 577/408 \approx 1.414216$, ...

If we continue some steps further, the result will be very precise already. So, we can use the following C++ code to compute $\sqrt{2}$.

Example 28

```
double y=1;
for(int i=0;i<10;i++)
    y=(y+2/y)/2;
cout << "square root of 2: " << y << endl;
```

Now imagine we want to write a program that computes the square roots of 2, 3, 26 and 41. It could look like this:

Example 29

```
int main()
{
    double y=1;
    for(int i=0;i<10;i++)
        y=(y+2/y)/2;
    cout << "square root of 2: " << y << endl;
```

```
    y=1;
    for(int i=0;i<10;i++)
        y=(y+3/y)/2;
    cout << "square root of 3: " << y << endl;

    y=1;
    for(int i=0;i<10;i++)
        y=(y+26/y)/2;
    cout << "square root of 26: " << y << endl;

    y=1;
    for(int i=0;i<10;i++)
        y=(y+41/y)/2;
    cout << "square root of 41: " << y << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

The C++ code to compute the square root has been repeated four times. This is not only ugly, but also tedious and makes the program difficult to read. In such a case it is high time to use a *function*. To compute square roots, we can actually use the built-in C++ function `sqrt` contained in the library `cmath`. Then the program looks much better:

Example 30

```
#include<cmath>
...
int main()
{
    cout << "square root of 2: "  << sqrt(2)  << endl;
    cout << "square root of 3: "  << sqrt(3)  << endl;
    cout << "square root of 26: " << sqrt(26) << endl;
    cout << "square root of 41: " << sqrt(41) << endl;

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

8.1.2 How to use built-in functions

What do we need to know to use the built in function for computing square roots?

- First of all we need to know its name - `sqrt`.
- Then we need to know the number and the type of its parameters - it has one parameter of type `double`. Hence we can use it like `sqrt(2.34)` or `sqrt(x)` where `x` is a variable of type `double`.
- We can also use `sqrt(x)` if `x` is of type `int` since type `int` is automatically converted to type `double`.
- Finally, we need to know what the function returns - `sqrt(x)` returns the square root of `x` in type `double`.

The same is true for using most other built-in C++ functions: we just need to know the name, parameters and return value of the function to use it correctly. In the following, this information is given for the most important built-in mathematical functions.

8.1.3 Table of most important built-in mathematical functions

To use the following functions, we need `#include<cmath>`. Aside from the function `abs`, *all these functions have return values of type* `double`. This is true even for the functions `ceil` and `floor`.

`abs(x)`	absolute value of `x` where `x` is of type `int`; the return value is of type `int`
`acos(x)`	arc cosine of `x` where `x` is of type `double`
`asin(x)`	arc sine of `x` where `x` is of type `double`
`atan(x)`	arc tangent of `x` where `x` is of type `double`
`ceil(x)`	the smallest integer not less than `x` where `x` is of type `double`
`cos(x)`	cosine of `x` where `x` is of type `double`
`cosh(x)`	hyperbolic cosine of `x` where `x` is of type `double`
`exp(x)`	returns $e^{\texttt{x}}$ where e is the Euler number and `x` is of type `double`
`fabs(x)`	absolute value of `x` where `x` is of type `double`
`floor(x)`	largest integer not greater than `x` where `x` is of type `double`
`log(x)`	natural logarithm of `x` where `x` is of type `double`
`pow(x,y)`	$\texttt{x}^{\texttt{y}}$ where `x` and `y` are of type `double`; here `x` *must* be of type `double` while, for `y`, type `double` or `int` is possible. Using type `int` for `x` causes a compiler error.
`sin(x)`	sine of `x` where `x` is of type `double`
`sinh(x)`	hyperbolic sine of `x` where `x` is of type `double`
`sqrt(x)`	square root of `x` where `x` is of type `double`
`tan(x)`	tangent of `x` where `x` is of type `double`
`tanh(x)`	hyperbolic tangent of `x` where `x` is of type `double`

Example 31

```
1 #include<cmath>
2 ...
3 double x=1.11;
4 cout << log(exp(x))-x << endl;
5 cout << pow(sin(x),2)+pow(cos(x),2)-1 << endl;
6 cout << cosh(x) - (exp(x)+exp(-x))/2 << endl;
7 cout << sinh(x) - (exp(x)-exp(-x))/2 << endl;
```

Explanations

- Line 4: the output should be 0 since $\log(e^x) = x$ for all $x \in \mathbb{R}$.
- Line 5: the output should be 0 since $\sin^2 x + \cos^2 x = 1$ for all $x \in \mathbb{R}$.
- Line 6: the output should be 0 since $\cosh x = (e^x + e^{-x})/2$ for all $x \in \mathbb{R}$.
- Line 7: the output should be 0 since $\sinh x = (e^x - e^{-x})/2$ for all $x \in \mathbb{R}$.
- Usually, the output will *not be exactly* 0 because of the limited precision of the type `double`. But the error should be less than 10^{-15}.

8.1.4 Other useful built-in functions

To use the following functions we need `#include<cstdlib>`.

`abort()`	Immediately terminates the program.
`exit(x)`	Immediately terminates the program. Here `x` is a value of type integer. Usually `exit(0)` indicates normal termination while `exit(1)` means termination because of a failure.
`rand()`	Returns a random number; each use of `rand()` generates another random number. However, `rand()` always generates the same sequence of random numbers unless we define a "seed" with `srand`.
`srand(x)`	Sets a "seed" `x` for the sequence of random numbers generated by `rand()`. Here `x` must be an nonnegative integer.

To use the following functions we need `#include<ctime>`.

`clock()`	Returns the processor time the program has used up so far. To convert this time to seconds, first convert it to type `double`, for instance, by `double time1=clock();` and then divide it by the built-in constant `CLOCKS_PER_SEC`.
`time(0)`	Returns the time in seconds which has elapsed since 00.00 hours, January 1, 1970. The `time` function is particularly useful for defining a random number seed which is different for each execution of the program.

Example 32

```
1  srand(time(0));
2  int randomNumber1, randomNumber2;
3  double startTime=clock();
4  while(1)
5  {
6     randomNumber1=rand();
7     randomNumber2=rand();
8     if(randomNumber1==13 && randomNumber2==13)
9     {
10        double endTime=clock();
11        cout << "Two unlucky numbers in a row!" << endl;
12        cout <<  "That's too much - stop." << endl;
13        cout << "Time used up: "
14             << (endTime-startTime)/CLOCKS_PER_SEC << endl;
15        system("pause");
16        exit(1);
17    }
18 }
```

Explanations

- Line 1: the random number seed is set to the current time. Hence the seed will be different each time we start the program.
- Line 4: a while-loop with condition `1` is started. Since `1` is always true, the loop will run until it is interrupted by `break` or by using one of the function `abort` or `exit`.
- Line 5: `startTime` measures the time before the start of the while loop.
- Line 8: if we encounter two unlucky numbers in a row, we print a message to the screen and stop the program immediately.
- Line 15: before `exit`, we should use `system("pause")` since otherwise the screen would immediately close and we could not see the output of the program.
- Since the sequence of random numbers generated by `rand()` will be different each time we run the program, the number of iterations performed and thus the execution time will also be different.
- On a common PC using the Dev-C++ compiler, the program usually will run for 5-60 seconds.

8.2 Creating your own functions

C++ has only a few built-in functions. They are very useful, but by far not sufficient for most programs. One of the most powerful features of C++ is that we can create *our own functions* and we can adapt them to fulfill exactly the task we need in our program. Such functions are called *user defined functions*.

Right at the beginning, it is very important to understand that *creating a function and using a function are totally different things*. We have *used* several functions already. For instance, `cout << sqrt(2) << endl;` *uses* the function `sqrt` of the library `cmath`. When we *create functions*, we do something completely different: we have to specify *how the function works*.

Of course, each user defined function *must have a task*. Otherwise it would be useless... When we create a user defined function, we have to accept the *full responsibility for its correct "functioning"*. Correct functioning means that the function always performs its task successfully. When we define a function, we have to supply the following information in our C++ code.

- The function *name*.
- The number and types of the function *parameters*.
- The type of the *return value* of the function. This type is called the *return type* of the function.
- The C++ commands which fulfill the *task* of the function.

The name, parameters and return type of a function are all provided in the so-called *function head*. The C++ commands which fulfill the *task* of the function are contained in the so-called *function body*.

8.2.1 Function head

Syntax

returnType FunctionName(parameters)

Explanations

- *returnType* is the return type of the function.
- *FunctionName* is the name of the function. We have to choose this name. It is always a good idea to choose a name that *describes the purpose of the function.*
- *parameters* are the parameters of the function. They are given as a *comma separated list of variable declarations.*
- A function can have many parameters, but only one return type and only one return value.
- A function can have *no parameters.*
- A function can have *no return value.* Then the return type is `void`.

Example 33

```
1 // examples of functions heads
2 int Function1(int a)
3 double Function2(int a,double b,char c)
4 int Function3()
5 void Function4(int a,int b,double c,double d)
```

Explanations

- Line 2: function with name `Function1`, return type `int` and one parameter of type `int`.
- Line 3: function with name `Function2`, return type `double` and three parameters of types `int`, `double` and `char`.
- Line 4: function with name `Function3`, return type `int` and no parameters.
- Line 5: function with name `Function4`, return type `void`, two parameters of type `int` and two parameters of type `double`. Return type `void` means that the function *does not return a value.*

8.2.2 Function body

Syntax

```
{
    statements
}
```

Explanations

- The function body consists of a sequence of C++ statements enclosed by curly braces.
- The statements in the function body must fulfill the task of the function.
- The curly braces *cannot be dropped* - even if the function body only consists of a single statement.

Example 34

```
1 void PrintSomething()
2 {
3       cout << "Hello!" << endl;
4 }
```

Explanations

- Line 1: This is the function head. The function name is `PrintSomething` and the function has no parameters. The return type is `void` which means that the function has no return value.
- Lines 2-4: This is the function body. The only purpose of the function is to print `Hello!` on the screen, followed by a newline.

8.2.3 Function definition

Syntax

```
returnType
FunctionName(parameters)
{
    statements
}
```

Explanations

- A function definition consists of the function head followed by the function body.
- Once we have provided a complete function definition, the function is "ready to use", i.e. we can use it in any other part of the program.

- Function definitions *must be placed outside of any other function.* Such places outside of any function are called *at global scope.* So, functions must be defined at global scope.
- In particular, any function definition *must be outside of the main function.*

Example 35

```
1  #include <cstdlib>
2  #include <iostream>
3  using namespace std;
4
5  void PrintSomething()
6  {
7       cout << "Hello!" << endl;
8  }
9
10 int main()
11 {
12    ...
13 }
```

Explanations

- Lines 5-8: the function `PrintSomething` is defined. It is correctly defined at global scope: outside of any function, in particular, outside of the main function.
- It would be wrong to put the definition of the function `PrintSomething` inside the main function. This would produce a compiler error.
- Lines 1-3: note that the include-statements and `using namespace std;` are also at global scope.

8.2.4 Return value, return type and return statement

Most functions compute a value as a result and return the value to the program. This value is called the *return value* of the function. The type of the return value is specified in the function head and is called the *return type* of the function.

How does a function *return a value to the program*? This is done by a *return statement* inside the function body.

Syntax

`return` *expression*`;`

Explanations

- This is a *return statement.* Return statements are only allowed inside of function bodies.

- A return statement terminates the execution of a function immediately. It is comparable to a `break` in a loop.
- The return value of the expression must match the return type of the function.
- If the return type of a function is different from `void`, then we must make sure that the function returns a value of the correct type *under all circumstances*.
- In particular, if a function contains `if`- or `if-else`-conditions, then we must make sure that the function returns a value of the correct type in *all branches of the conditions*.

8.2.5 Function parameters

C++ functions are quite similar to complete C++ programs: they usually require some input data, use these data as a basis to perform computations and obtain a result. The input data of a function are its *parameters*.

When we use function parameters, we face a kind of paradoxical situation which can be quite confusing: when we write the C++ code for the function definition, we do *not know the values of the function parameters, but we have to assume that they are known*.

For instance, if we write a function `Sum` which computes the sum of two integers, we certainly do not know the values of the two integers since the function *must work for any two integers as input*. For example, `Sum(3,5)` must return `8` and `Sum(11,11)` must return `22`. But when we write the definition of the function `Sum`, we cannot just initialize the two integers to `3` and `5` - they also could be `11` and `11`. We have to assume that the two integers are known, since they are specified *later* when the function is *used*.

Since we do not know the values of the function parameters when we write a function, we must *consider the function parameters as variables*. However, the values of the parameters will be specified only when the function *is used*, not in the function definition! In particular, it is usually *totally wrong to assign values to function parameters inside the function body*. This would destroy the purpose of function parameters which is to specify values *when the function is used, not when the function is defined*.

Difficult? It all should become clear if you study the following example carefully.

Example 36

```
1  int Sum(int a, int b)
2  {
3      return a+b;
4  }
5
6  int main()
7  {
8      cout << Sum(5,10) << endl;
9      system("PAUSE");
10 }
```

Explanations

- Lines 1-4: this is the definition of a function `Sum` which takes two integers as parameters, computes their sum and returns the result.
- Note that in the function definition no values for the parameters `a` and `b` are specified. They are considered as variables.
- Line 8: this is where the function `Sum` is *used*.
- When the command in line 8 is executed, the program automatically substitutes `5` for `a` and `10` for `b`.
- It would be totally wrong to specify the values of `a` and `b` inside the function definition, for instance, with `a=5`, `b=10`. In this case, the function would always give the result `15`, no matter what the input is.

8.2.6 Function call

Syntax

FunctionName`(`*parameterValues*`)`

Calling a function simply means using a function. A function call is done by typing the name of the function followed by a comma separated list of values for the function parameters in parenthesis. The number and type of the parameter values in a function call *must match with the function definition.*

Example 37

```
int Sum(int a, int b)
{
    return a+b;
}
```

We can call this function by `Sum(5,10)`, but not by `Sum(5,10,15)` - the number of parameters would be wrong. A function call `Sum("hi","hi")` is also incorrect - the type of the parameters would by wrong.

Function calls as commands or part of commands

- If a function returns a value, then the function call usually is *part of a command* which uses the return value. For instance, `cout << Sum(5,10) << endl;` prints the return value to the screen.
- If a function does not return a value (return type void), then a function call usually is a *complete command* and has to be ended by a semicolon. For instance, `PrintSum(5,10);` is a function call of the function `PrintSum` (see below) which has no return value, but prints the result to the screen.

Example 38

```
void PrintSum(int a,int b)
{
    cout << a+b << endl;
}
```

It is important to remember that through a function call the *values of the parameters are automatically passed to the function body*. This is the reason why we do not need to (and must not!) specify the values of the parameters inside the function body.

Another important rule: a function call can be *treated in the same way as a value of the return type of the function*. What we can do with a value of the return type, we can also do with a function call! For instance, recall that the function `Sum` from above has return type `int`. Hence a function call can be treated in the same way as a value of type `int`. Since we can print an integer to the screen, we can also do it with the function call: `cout << Sum(5,10) << endl;`. We can do arithmetic operations with integers. Hence we can also do arithmetic operations with with a function call: `cout << Sum(5,10)+10;` will print `25` to the screen.

8.2.7 Function variables are local

Inside a function body, we can declare and use as many variables as we like. Usually, we do this to store intermediate results. Such variables which are declared inside a function body *only belong to the function and do not have any meaning outside the function body*. For this reason, variables declared inside functions bodies are called *local variables*. They are only valid *locally*, inside the function body.

Example 39

```
1  int Sum(int a, int b)
2  {
3      int sum=a+b;
4      return sum;
5  }
6
7  int main()
8  {
9      cout << Sum(5,10) << endl;
10     cout << sum << endl;       //error!
11     int sum =234;              //correct!
12     ...
13 }
```

Explanations

- Line 3: `sum` is a local variable since it is declared inside the body of the function `Sum`.
- Line 9: the function `Sum` is called. During the execution of the function, the local variable `sum` will temporarily get the value `15`. However, after the function call, this value will be gone.
- Line 10: this tries to access a local variable from outside the function body. This produces a compiler error.
- Line 11: this is correct since outside the body of the function `Sum`, the name `sum` has no meaning. So, we are allowed to declare and use a variable with this name. This variable will be completely independent of the local variable `sum`.

8.2.8 Function parameters are local

Unless we use "pass by pointer" or "pass by reference" (if you are interested in this, see textbook), all function parameters also will be local. This means that the function parameters only have a meaning inside the function body, not in any other part of the program.

Example 40

```
1  int Sum(int a, int b)
2  {
3      return a+b;
4  }
5
6  int main()
7  {
8      cout << Sum(5,10) << endl;
9      cout << a << endl;          //error!
10     int a =234;                 //correct!
11     ...
12 }
```

Explanations

- Line 1: the parameters `a` and `b` of the function sum are only valid inside the function body.
- Line 8: the function `Sum` is called. During the execution of the function, the local variable `a` will temporarily get the value `5`. However, after the function call, this value will be gone.
- Line 9: this tries to access a local variable from outside the function body. This produces a compiler error.

- Line 10: this is correct since outside the body of the function `Sum`, the name `a` has no meaning. So, we are allowed to declare and use a variable with this name. This variable will be completely independent of the local variable `a`.

Example 41

```
1  void FailedIncrease(int a)
2  {
3      a++;
4  }
5
6  int main()
7  {
8      int x = 10;
9      FailedIncrease(x);
10     cout << x << endl;
11 ...
12 }
```

Explanations

- The function `FailedIncrease` is designed to increase the value of its parameter by `1`. However, this cannot work since it only increases the value of the *local variable* `a`.
- Line 9: when the function is called the following happens. The value of `x` is *copied* into the local variable `a`, then the local variable `a` (not `x`!!) is increased by `1`. After the function call, the local variable `a` is destroyed.
- Line 10: the output will be `10` since `x` was not increased.

8.2.9 Function declaration

We can put function definitions (at global scope!) at many different places of a program. Sometimes, it is even convenient to put the function definition *after the function calls*, for instance, after the main function. The function definition can also be put into a different source file than the function calls. However, in such a cases we must make sure that the compiler knows at the points of function calls that the *function exists*. This is the purpose of a *function declaration*.

Syntax

returnType FunctionName`(`*parameters*`);`

Explanations

- This tells the compiler that a function `FuntionName` exists and informs the compiler about the return type and the number and types of its parameters.
- A function declaration just consists of the function head followed by a semicolon.

- Contrary to the function definition, we do not need to specify the names of the parameters in a function declaration (but we can).

Example 42

```
1  int Sum(int,int);
2
3  int main()
4  {
5      cout << Sum(5,10) << endl;
6      ...
7  }
8
9  int Sum(int a, int b)
10 {
11     return a+b;
12 }
```

Explanations

- Line 1: this is a declaration of the function `Sum`. The declaration is necessary here since the definition of the function comes after the function call.
- Note that the names of the parameters (`a` and `b`) are not specified in the function declaration. However, it would also be correct to specify them: `int Sum(int a,int b);` is also fine.
- Line 5: the function is called before its definition. This is fine since it has been declared already.
- Lines 9-12: the function is defined after the main function. It even could be defined in another source file.

9 Classes

9.1 Purpose of Classes and Objects

In the last section, we learned about functions which are the most important feature of C++. Until the 1970s/80s, functions were almost the only tool used for structuring programs. But it turned out, that the use of functions can become very inconvenient for solving complex problems. For a complex problem, there usually will be a lot of input data of different kinds. A function which does computations on these input data *must take all the input through parameters*. This can give rise to functions will a lot of parameters, sometimes even more than 20 parameters! Imagine you have to call such a function: you have to know what all the parameters mean and you have to get everything in the right order. This is tedious and error prone. In fact, this problem, among others, led to so many software errors that people spoke of a "software crisis".

What is the solution? The problem is that a usual C++ function is "clueless" of the problem we are working on; we have to provide all the information on the problem by parameters. We should have "smarter functions" which "know" the input data of the problem already without telling them by parameters! And this is exactly the the point where *classes* come into play. A class is a framework for the data of a problem, the *data members*, and the functions which work on the problem the *member functions*. Member functions of a class are "smart functions" which know the data of a problem automatically, with no need to pass the data by parameters.

Actually, by using a class we should not only be able to solve one single problem, we should be able to solve *all* problems which have a suitable structure. For instance, if we use a class to solve quadratic equations, then the class should be written such that we can solve *any* quadratic equation with it, not just only one. This is the reason why the data members of a class must be *variables*: the values of the variables then can be chosen to describe any particular *instance* of the problem we are working on.

Since a class is used to solve many problems, a class can only be a *framework* for solving the problem. A particular instance of the problem is obtained by specifying the values of the data members of the class. This is done by creating an *object* of the class and initializing the data members in this object. So, an object of a class describes a *particular instance* of a problem which is solved by using the class.

A class is a framework for solving a problem; an object of a class describes a particular instance of the problem.

9.2 Use of Classes

The use of classes in C++ is very similar to the use of built-in types. A class is just a *new, user-defined* type. For every built-in type, we can declare and use *variables* of this type; for each class we can declare and use *objects* of this class. To declare a variable of a built-in type, we write down the type name, followed by the variable name and a semicolon; to declare an objects of a class we write down the class name, followed by the object name and a semicolon.

A class is new, user-defined type. An object is a variable of this type.

9.3 Class Declaration

Almost everything we use in C++ must be *declared* - classes are no exception. A class consists of variables, called *data members*, and functions, called *member functions*. All these variables and functions must be declared inside the class declaration.

Concerning the data and function members of a class, there is a somewhat peculiar feature: they can be declared as `private`, `public` or `protected`. We certainly will not worry much about this since the "optimal" use of these attributes is irrelevant for almost all scientific programming projects. We will simply declare all data members as `private` and all function members as `public`. This is the best choice in most of the cases anyway.

Why do we declare the data members as `private`? This essentially means that *only member functions of the class are allowed to access the values of data members*. This is good programming style since it makes sure that the data members will "not be visible" outside the class. Yes, we view the data members as "dirty details" which we do not want to see. We only want to see the "nice" member functions which solve our problems.

Syntax

```
class ClassName
{
private:
    dataMembers
public:
    memberFunctions
};
```

Explanations

- This declares a class with name *ClassName*.
- The data members are a (possibly empty) sequence of variable declarations.
- The member functions are a (possibly empty) sequence of function declarations or function definitions.

- There also could be `private` or `protected` member functions or `public` or `protected` data members. However, this is not necessary for our purposes.

Example 43

```
1 class Box
2 {
3 private:
4      double height,length,breadth;
5 public:
6      void SetDimensions(double h,double l,double b);
7      double Volume();
8 };
```

Explanations

- The name of this class is `Box`. Its purpose is to store the dimensions of a box and to provide a function which computes the volume of a box.
- This is only a rudimentary example for demonstrating the syntax of a class declaration. A "real life" class `Box` could have many further features, for instance, a member function `Draw` which draws a picture of a box on the screen.
- The data members of the class box are the variables `height`, `length` and `breadth` of type double.
- The member functions of the class `Box` are `SetDimensions` and `Volume`.
- The purpose of the member function `SetDimensions` is to initialize the data members `height`, `length` and `breadth`. This is necessary, since, when we create an object of a class, the data members are uninitialized unless we use a member function to initialize them.
- The purpose of the member function `Volume` is to compute the volume of a box and return the result.

9.4 Providing the Member Function Definitions

Recall that a *function definition* comprises all details concerning a function, namely, the return type, the function name, the parameters, and the function body. In a class declaration, member functions usually are only *declared*, i.e. the function body is not provided inside the class declaration. However, though not as usual, it is allowed to put member function definitions inside the class declaration. This can be convenient if the function body is short.

Member functions are very similar to ordinary C++ functions. However, there is one very important difference we must never forget:

Inside the function body of a member functions, the data members of the class are automatically available. The data members can be accessed and their value can be changed just by referring to their name which is defined in the class declaration.

For instance, if we want to return the volume of a box inside the function body of the member function `Volume` of the class `Box`, then we just can use `return height*length*breadth;` The variables `height`, `length` and `breadth` are automatically available inside the function body of member functions of the class `Box`. Note that for an ordinary C++ function, the command `return height*breadth*length;` only would make sense if `height`, `length` and `breadth` were *parameters* of the function. For the member function `Volume` we do not need these parameters since it is a "smart" function which "knows" the necessary values automatically.

There is a second, merely technical, difference between ordinary C++ functions and member functions:

For a member function definition outside the class declaration, we must put the class name followed by a double colon in front of the function name.

Note that this is "logical" since otherwise the compiler would have no chance to know that the function belongs to a class. For instance, if we provide the definition of the member function `Volume` of the class `Box` outside the class declaration, then the function head will be

```
double Box::Volume()
```

and *not* `double Volume()`.

In summary, the steps for providing the definition of a member function usually are as follows.

- Put the member function definition *after* the class declaration.
- The function head for the member function definition is the function head from the class declaration, but with class name followed by a double colon in front of the function name.
- In the function body, put the commands which fulfil the purpose of the member function.
- Keep in mind that the data members of the class are automatically available inside member functions.

Example 44

```
1 void Box::SetDimensions(double h,double l,double b)
2 {
3    height=h;
4    length=l;
5    breadth=b;
6 }
```

Explanations

- This is the definition of the member function `SetDimensions` of the class `Box`.
- The purpose of this function to initialize the data members of the class `Box` by a functions call. For intance, `...SetDimensions(1,2,3);` should set `height` to `1`, `length` to `2` and `breadth` to `3`.
- Note that `h,l,b` are *parameters* of this member functions. They are *not* data members of the class `Box`.
- Line 3: the data member `height` is set to `h`. This is possible since the data members are automatically available inside member functions. In an ordinary C++ function this would cause a compiler error "variable undeclared", but here it is perfectly fine.
- Note that `h=height;` would be totally wrong. Imagine we do a function call `...SetDimensions(1,2,3);` for a box whose data members have not been initialized yet. Then the command `h=height;` would attempt to assign the undefined value of `height` to `h`. It simply must be the other way round, `1` must be assigned to `height` by the command `height=h;`
- It would be wrong to use

  ```
  void Box::SetDimensions(double height,double length,double breadth)
  ```

  as the function head. Then, inside the function body, the data members `height`, `length` and `breadth` would suddenly *not* be available any more since they would be "overshadowed" by the parameters `height`, `length` and `breadth` which have the same name. The absurdity of this function head is also clear from the commands inside the function body we would have to use:

  ```
  height=height; length=length; breadth=breadth;
  ```

Example 45

```
1  double Box::Volume()
2  {
3      return height*length*breadth;
4  }
```

Explanations

- This is the definition of the member function `Volume` of the class `Box`.
- The purpose of this function is to return the volume of a box.
- Note that the function does not need any parameters since the data members of the class `Box` are automatically available inside member function definitions.

Example 46

```
1  class Box
2  {
3       private:
4       double height,length,breadth;
5       public:
6       void SetDimensions(double h,double l,double b)
7       {
8           height=h;
9           length=l;
10          breadth=b;
11      }
12      double Volume()
13      {
14          return height*length*breadth;
15      }
16 };
```

Explanations

- This example demonstrates that member functions can also be defined inside the class declaration.
- If a member function is defined inside the class declaration, we need not put the class name followed by a double colon in front of the function name (but we could if we like to).
- If the member function definition is provided inside the class declaration, then there must not be a member function declaration inside the class declaration. So, for instance, it would be wrong, and unnecessary anyway, to add the function declaration `double Volume();` to Example 46.

9.5 Creating Objects of a Class

Once we have provided a class declaration and have defined all the member functions, the class is ready-to-use. In order to utilize the class, we have to create objects and perform computations with them. The first step, creating objects, is done by an *object declaration*. An object declaration is essentially the same as a variable declaration since classes can be considered a types and objects as variables.

Syntax

ClassName ObjectName;

Explanations

- This creates an object *ObjectName* of the class *ClassName*.
- Note that the syntax is the same as for a variable declaration - with *typeName* replaced by *ClassName* and *variableName* replaced by *ObjectName*.
- After such an object declaration, the data members of the class are *not initialized yet* for this object. The initialization has to be done by calling member functions.

Remark (Constructors) A class can contain member functions whose name is the same as the class name. Such member functions are called *constructors*. The purpose of a constructor is to initialize the data members of the class. The values of the data members are specified as parameters of the constructor. A constructor has *no return type*, not even `void`. If a class `ClassName` has a constructor with function head

```
ClassName(type1 parameter1, type2 parameter2,...)
```

then we can specify values for the parameters of the constructor in an object declaration by putting these values in parenthesis after the object name:

```
ClassName ObjectName(value1, value2,...);
```

Example 47 (Constructors)

```
1  class Box
2  {
3       private:
4       double height,length,breadth;
5       public:
6       Box(double h,double l,double b)
7       {
8            height=h;
9            length=l;
10           breadth=b;
11      }
```

```
12      double Volume()
13      {
14              return height*length*breadth;
15      }
16  };
```

Explanations

- In this example, the member function `SetDimensions` has been replaced by the constructor `Box`.
- Note that the constructor has the same name as the class and has no return type, not even `void`.
- If, for instance, we declare an object of the class by `Box B(1,2,3);` then the constructor will automatically be called and will initialize `height` to `1`, `length` to `2` and `breadth` to `3`.

For our purposes, we should keep in mind that we can use constructors to initialize the data members in objects. Recall that we are mainly interested in the use of existing classes. For an existing class, we can easily check if constructors are available by consulting the documentation of the class or the class declaration. Actually, "behind our back", constructors have been working all the time already:

Example 48

```
vector<int> v(10);
```

Explanations

- This declares on object of the class `vector<int>` and calls a constructor which initializes the size of the vector to `10`.

9.6 Calling Member Functions

Now we assume that a complete class declaration is available, all the member functions have been defined and an object, say `X`, of the class has been declared already. The last and crucial step in utilizing the class is to *call the member functions of the class on* `X`.

Why do we say "on `X`"? Because we need an object of the class to be able to do computations with the values of data members. Recall that the class is only a framework which *does not contain any data values* (!). So, if we want to do computations with a class, we need to create an object of the class and initialize the data members in the object first.

So, all the member functions of a class will be called *on an object of the class*. For calling member functions on an object, there is a designated mechanism, namely, the *dot operator*:

Syntax

ObjectName.FunctionName(value1,value2,...)

Explanations

- This is the syntax of calling a member function *FunctionName* on an object *ObjectName*.
- *value1*, *value2*,... are the values for the parameters of the member function. If the function has no parameters, then the parenthesis are left empty, of course.
- The call of a member function is like the call of an ordinary C++ function, but with the object name and a dot in front of it.
- All the rules for calling ordinary C++ functions also apply to calls of member functions.

Example 49 Now we are finally able to admire the class `Box` in its full functionality:

```
1  #include <cstdlib>
2  #include <iostream>
3  using namespace std;
4
5  class Box
6  {
7       private:
8       double height,length,breadth;
9       public:
10      void SetDimensions(double h,double l,double b)
11      { height =h; length=l; breadth=b;}
12      double Volume()
13      {
14              return height*length*breadth;
15      }
16 };
17
18 int main()
19 {
20     Box B;
21     B.SetDimensions(1,2,3);
22     cout << B.Volume() << endl;
23     system("PAUSE");
24 }
```

Explanations

- Line 20: an object `B` of class `Box` is declared. Its data members are unitialized at this point.
- Line 21: the member function `SetDimensions` is called on `B` and initializes the data members in `B` to `1,2,3`.
- Line 22: the member function `Volume` is called on `B` and returns the volume of `B` which is `6`.

Example 50 We often have called member functions already:

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      vector<int> v;
9      for(int i=0;i<100;i++)
10        v.push_back(rand());
11     v.resize(50);
12     cout << v.size() << endl;
13     system("PAUSE");
14 }
```

Explanations

- Line 10: the member function `push_back` of the class `vector<int>` is called to push random numbers to the back of the vector `v`.
- Line 11: the member function `resize` of the class `vector<int>` is called to set the size of `v` to `50`.
- Line 12: the member function `size` of of the class `vector<int>` is called to access the size of `v`.

10 C++ and C Strings

In C++, a *string* is a sequence of one-letter symbols. Any symbol which has an ascii-code (see Lecture 10) can be part of a string. In particular, any one-digit number, any letter and any symbols like `! @ $ % ^ & * ( )` are allowed in a string. The number of symbols in a string is called its *length* or *size*. To obtain a *string literal*, we put a string in quotation marks. Examples of string literals:

`"abc"`, `"12345"`, `"&*-+=[]"`

10.1 C++ Strings

The standard C++ library contains a very useful and convenient class `string` which provides many helpful functions for string operations.

Syntax of C++ String Declaration

`string` *StringName*;

Explanations

- This declares an empty string with name *StringName*.
- The use of strings requires `#include<string>`.

The individual symbols in a C++ string can be accessed by indices. These indices run from `0` to the length of the string minus one. The following table describes how some of the most important operations with strings are executed. For this table, we assume that `S` and `S1` are C++ strings.

Command	**Explanation**
`S = "abc";`	Assigns the string `abc` to `S`.
`S += "zzz";`	Appends the string `zzz` to the end of `S`.
`S+S1`	Returns the string resulting from the concatenation of `S` and `S1`.
`cout << S;`	Prints `S` to the screen.
`cin >> S;`	Reads the value of `S` from the keyboard.
`S.size()`	Returns the length of `S`
`S[i]`	Returns the `(i+1)`st symbol in `S`.
`S<S1`	Returns true if and only if `S` is lexicographically smaller than `S1`. The comparison is based on the ascii-codes of the symbols.

Example 51

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8      string S,S1;
9      S = "12345abcdef";
10     cout << S << endl;
11     cout << S[4] << endl;
12     S1 = S + "&&&";
13     cout << "Length of S: " << S.size() << endl;
14     if(S<S1)
15        cout << "S is lexicographically smaller than S1."
16             << endl;
17     system("PAUSE");
18 }
```

Explanations

- Line 8: two empty strings `S` and `S1` are created.
- Line 9: the string `12345abcdef` is assigned to `S`.
- Line 10: `S` is printed to the screen.
- Line 11: the fifth symbol of `S` is printed to the screen.
- Line 12: the value of `S1` is set to the string `S`, followed by `&&&`.
- Line 13: the length of `S` is printed to the screen.
- Lines 14-15: the strings `S` and `S1` are compared lexicographically based an the ascii-codes of their symbols.

10.2 C Strings

C++ strings were not available in the older programming language C. Since libraries written in C are very useful for Scientific Programming, we have to know a little bit about the kind of strings used in C.

A *C string* is an array with entries of type `char`. The size of such an array is the number of symbols in the string plus one (!). The reason for this is that each C string contains an additional terminating character with ascii-code `0`.

Syntax of C String Declaration and Initialization

`char` *StringName*`[] =` *StringLiteral*`;`

Explanations

- This declares a C string *StringName* and initializes its value to the *StringLiteral*.
- Alternatively, we can use

  `char*` *StringName*`=` *StringLiteral*`;`

  which has the same effect.

Since C++ strings are much more convenient to use than C strings, I recommend to always use C++ strings. Only if we must use C strings, for instance, to call functions from a C library, we create the necessary C strings by converting C++ strings to C strings. Hence, for our purposes, the most important aspect of C strings is their conversion to C++ strings and vice versa. The following table shows how this can be done.

Command	**Explanation**
`char* Cstring = "data.txt";`	Creates a C string whose value is a file name. This is a typical application of C strings since they are often used in C libraries to represent file names.
`string CPPstring = Cstring;`	This converts the C string to a C++ string.
`char* test = CPPstring; //error`	Compiler error! Converting a C++ string to a C string just by assignment is impossible.
`char* test1 = CPPstring.c_str();`	This correctly converts the C++ string to a C string by using the member function `c_str` of the class string.

Actually, if we want to create an `ofstream` which writes to a file and we want to represent the filename by a C++ string, then we need to convert the C++ string to a C string in the `ofstream` declaration. The following is a typical example.

Example 52

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <string>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9      string filename = "data.txt";
10     ofstream out(filename.c_str());
11     out << rand() << endl;
12     system("PAUSE");
13 }
```

Explanations

- Line 10: an output file stream is created which writes to the file `data.txt`. It would be wrong to use `ofstream out(filename);` instead. In an output stream declaration, a C string is necessary for the file name, not a C++ string. So, if we want to use a C++ string to specify the file name, we need to convert it to a C string by using the member function `c_str` of the class `string`.
- Line 11: a random number is written to the file `data.txt` to check if the program works.

11 Graphs and Networks

We now start to apply our knowledge of C++ to solve some interesting, relevant problems. Of course, we have to work quite hard for this, but without *application to substantial problems*, our knowledge of C++ would remain too theoretical and "lifeless".

11.1 Basic Notions

We will solve so-called "graph optimization problems" with C++. *Graphs* are mathematical objects which are used to represent and solve many real life application problems.

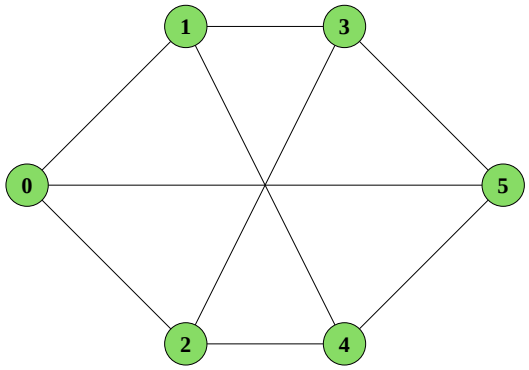
11.1.1 Graphs

A graph $G = (V, E)$ consists of a finite set V and a set E of 2-subsets of V. The elements of V are called *vertices* and the elements of E are called *edges*.

Note: By a "2-subset" we mean a subset with exactly 2 elements.

We can view the vertices of a graph as "locations" and the edges as "connections". It is easy to draw pictures of graphs. A vertex v is represented by a small circle with the label v inside. If two vertices v and w form an edge, i.e. $\{v, w\} \in E$, then we connect them by a line segment.

Example 53



$V = \{0, 1, 2, 3, 4, 5\}$

$E = \{\{0, 1\}, \{0, 2\}, \{0, 5\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\}$

Often it is quite tedious to write down the edges using the set notation. Hence, we often simply write ab instead of $\{a, b\}$. In this notation, we have $E = \{01, 02, 05, 13, 14, 23, 24, 35, 45\}$ for Example 53.

11.1.2 Adjacency and degree

Let $G = (V, E)$ be a graph. If $\{v, w\} \in E$, i.e. if v and w form an edge, we call v and w *adjacent*. Adjacent vertices are called *neighbors*. In Example 53, the vertex 0 has three neighbors, namely 1, 2 and 5. The vertices 1 and 2 are not adjacent, hence they are not neighbors.

The *degree* of a vertex v is the number of its neighbors. In Example 53, all vertices have degree 3.

11.1.3 Paths and cycles

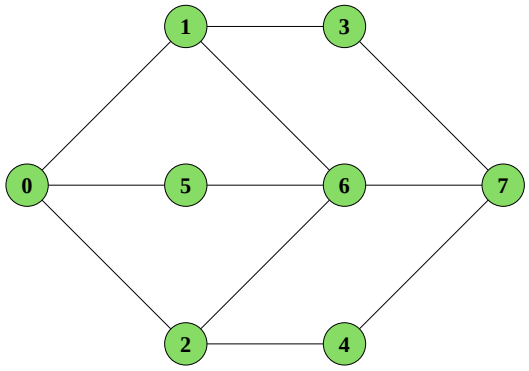
Imagine the vertices of a graph are locations in Singapore and the edges are roads. If we walk from location A to location B and use no road twice, we say we have walked along a *path*. If, moreover, A is identical with B, we say we walked along a *cycle*.

Formally: A *path* in a graph is a sequence v_1,...,v_r of vertices such that any two consecutive vertices form an edge, i.e. $\{v_i, v_{i+1}\} \in E$ for $i = 1, ..., r-1$, and all these edges are pairwise distinct, i.e. $\{v_i, v_{i+1}\} \neq \{v_j, v_{j+1}\}$ for $i \neq j$.

We often denote a path also by $v_1 - v_2 - ... - v_r$.

A *cycle* in a graph is a path $v_1 - v_2 - ... - v_r$ with $v_1 = v_r$.

Example 54



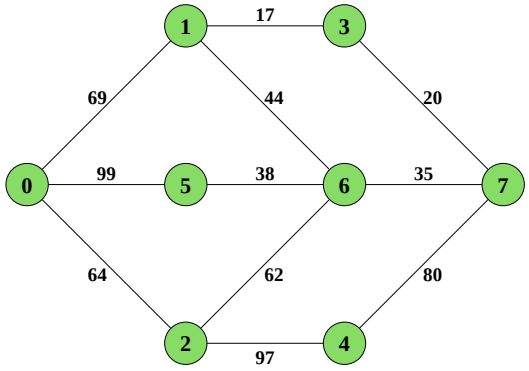
$0-1-3-7-4-2-6-1$ is a path in this graph, but $0-1-6-2-0-5-6-1-3$ is *not* a path since the edge $\{1, 6\}$ is used twice.

$0-1-6-2-4-7-6-5-0$ is a cycle in this graph, but $0-2-6-1-0-5-6-1-0$ is *not* a cycle since the edges $\{1, 6\}$ and $\{0, 1\}$ have been used twice.

11.1.4 Networks

A *network* is a graph $G = (V, E)$ in which each edge has a nonnegative number assigned to it. For an edge e, this number is denoted by $L(e)$ and is called the *length* of *weight* of e.

Example 55 (A network)



The numbers against the edges are the weights. For instance $L(01) = 69$, $L(37) = 20$, $L(47) = 80$.

11.1.5 Length of a path

Let $G = (V, E)$ be a network with lengths $L(e)$, $e \in E$. Let $P = v_1 - v_2 - ... - v_r$ be a path in G. Then length $L(P)$ of P is the sum of the weights of all edges contained in P, i.e.

$$L(P) = \sum_{i=1}^{r-1} L(v_i v_{i+1}).$$

In Example 55, the length of the path $0 - 5 - 6 - 7$ is $99 + 38 + 35 = 172$.

11.2 The shortest path problem

Given two vertices v and w in a network $G = (V, E)$, the *shortest path problem* is to find a path from v to w of *minimum length* (if such a path exists). The vertex v is called the *start vertex*. The length of a shortest path from v to w is called the *distance* between v and w.

In Example 55, the path $0 - 5 - 6 - 7$ is *not* a shortest path from 0 to 7. The path $0 - 1 - 3 - 7$ has length 106 and is shorter. In fact, $0 - 1 - 3 - 7$ is a shortest path from 0 to 7. Thus, the distance between 0 and 7 is 106.

For small networks, it usually quite easy to guess a shortest path, but for large networks which come from real world applications, only a *systematic search* can discover a shortest path. Our goal is to understand how such a systematic search can be performed and to implement it in C++. Once this is done, we will be able to solve shortest path problems with thousands of edges in milliseconds!

11.2.1 Concept of Dijkstra's Algorithm

The most common search strategy to solve the shortest path problems is called *Dijkstra's Algorithm*. We will learn how to implement this method in C++.

What are the basic ideas of Dijkstra's Algorithm? Imagine we want to find a shortest path from v to w. The first idea is to more: we actually will compute shortest paths from v to *any other vertex*, not only to w.

Temporary distances
Recall that the distance between v and w in a network is the length of a shortest path from v to w. We denote this distance by $d(v, w)$.

In Dijkstra's Algorithm, each vertex u will be assigned a "temporary distance", denoted by $Distance(u)$. These values change during the execution of the algorithm. The meaning of $Distance(u)$ is the length of the shortest path from the v to u constructed so far. This means, at any moment during the execution of the algorithm, $Distance(u)$ is the length of the shortest path from the v to u constructed until this moment. So, *during the algorithm*, $Distance(u)$ in general will *not* be the distance from v to u, but only the best value found so far. However, *after the algorithm is completed*, $Distance(u)$ will be the distance from v to u, i.e. $Distance(u) = d(v, u)$.

At the start of the algorithm, no paths have been constructed at all. Hence, we will set $Distance(u) = \infty$ for all $u \neq v$. During the execution of the algorithm, the values $Distance(u)$ will be improved step by step until the correct distances $d(v, u)$ are obtained. The values $Distance(u)$ are called *temporary distances* since their values change during the execution of the algorithm.

Improvement step starting from a vertex u
Imagine we are in the middle of the execution of the Dijkstra's Algorithm and, for some vertex u, we have $Distance(u) = r$. This means the shortest path from v to u we have found so far has length r. Now let s be any neighbor of u. Then there is a path of length $r + L(us)$ from v to s, namely, the path from v to u of length r followed by the edge us. Now it can happen that this path from v to s is shorter than any path which had been found so far. This will be the case if and only if $Distance(u) + L(us) < Distance(s)$. So, if this happens, we know that there is a shorter path from v to s and we have to update the temporary distance of s by setting it to $Distance(u) + L(us)$. Such a step is called an *improvement step*.

Predecessors
Imagine we have just done an improvement step as above. Then the shortest path from v to s we know so far consists of a path from v to u followed by the edge us. Hence, u is the *predecessor* of s on this path. This is the reason why we will set $Predecessor(s) = u$ after such an improvement step.

Marking process
If we look for improvements starting from a vertex u, we will check *all neighbors* of u for possible improvements. Once this is done, u will be *marked* which means that we will *not* search for improvements starting from u any more. This is to avoid searching from the same vertex repeatedly which would make the algorithm very inefficient.

Order of the search
It turns out that we cannot choose the vertices u for the improvement steps arbitrarily. Wrong choices of u can have the effect that shortest paths are not found. The key to Dijkstra's Algorithm is to always choose an u which is not marked and which has minimum temporary distance $Distance(u)$ among the unmarked vertices. It can be proved by mathematical induction that the algorithm will compute correct shortest paths if we use this rule for the choice of u. However, we will not study this proof here; it is part of Graph Theory.

11.2.2 Pseudocode of Dijkstra's Algorithm

Notation used in Dijkstra's Algorithm

- The input will be a network with vertices $1, 2, ..., n$. The start vertex is 1.
- The goal is to construct a shortest path from 1 to w where w is any given vertex.
- $L(a, b)$ denotes the length of the edge ab.
- We will use a set M that contains the vertices which still can be chosen as u for improvement steps.
- $Predecessor(a)$ denotes the predecessor of a on a shortest path from 1 to a found so far.

We will describe Dijkstra's Algorithm by a *pseudocode*. A pseudocode is a description of an algorithm which is similar to computer program, but much more informal. So, the following is *not* C++ code, but only a informal description similar to C++ code.

Pseudocode of Dijkstra's algorithm

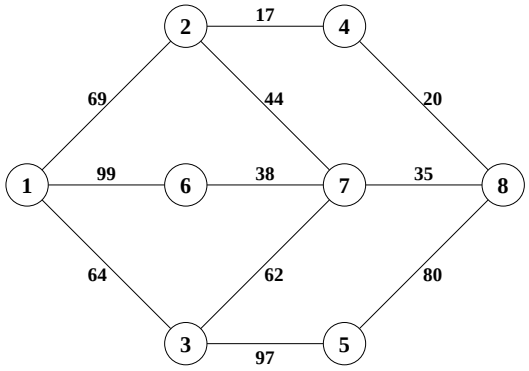
```
1. Initialization
for v=2,...,n set Distance(v) = infinity;
Distance(1) = 0 ;
Predecessor(1) = 0;
M={1};

2. Main step
while M is not empty do
{
    find an element u of M with minimum Distance(u);
    remove u from M;
    if u is not marked as used do
    {
        mark u as used;
        for each neighbor s of u do
        {
           newDistance = Distance(u)+L(u,s);
           if(newDistance<Distance(s))
           {
              Distance(s) = newDistance;
              Predecessor(s) = u;
              insert s into M;
           }
        }
    }
}

3. Construction of shortest path
start with empty path
set v=w
while(v!=0)
{
   push v to the front of path;
   v = Predecessor(v);
}
```

11.2.3 Example of the application of Dijkstra's Algorithm

Example 56



The following tables contain the details of application of Dijkstra's algorithm to Example 56. We use the letter Q here instead of M. An entry "+" in the row "Q" means that the vertex in this column is contained in Q, and an entry $-$ means it is not contained in Q. An entry "+" in the row "$Marked?$" means that the vertex is marked already, and entry "$-$" means it is not marked yet.

After initialization:

v	1	2	3	4	5	6	7	8
Q	+	−	−	−	−	−	−	−
$Distance(v)$	0	∞	∞	∞	∞	∞	∞	∞
$Predecessor(v)$	0	0	0	0	0	0	0	0
$Marked?$	−	−	−	−	−	−	−	−

After search from u= 1:

v	1	2	3	4	5	6	7	8
Q	−	+	+	−	−	+	−	−
$Distance(v)$	0	69	64	∞	∞	99	∞	∞
$Predecessor(v)$	0	1	1	0	0	1	0	0
$Marked?$	+	−	−	−	−	−	−	−

After search from u= 3:

v	1	2	3	4	5	6	7	8
Q	−	+	−	−	+	+	+	−
$Distance(v)$	0	69	64	∞	161	99	126	∞
$Predecessor(v)$	0	1	1	0	3	1	3	0
$Marked?$	+	−	+	−	−	−	−	−

After search from u= 2:

v	1	2	3	4	5	6	7	8
Q	−	−	−	+	+	+	+	−
$Distance(v)$	0	69	64	86	161	99	113	∞
$Predecessor(v)$	0	1	1	2	3	1	2	0
$Marked?$	+	+	+	−	−	−	−	−

After search from u= 4:

v	1	2	3	4	5	6	7	8
Q	−	−	−	−	+	+	+	+
$Distance(v)$	0	69	64	86	161	99	113	106
$Predecessor(v)$	0	1	1	2	3	1	2	4
$Marked?$	+	+	+	+	−	−	−	−

After search from u= 6:

v	1	2	3	4	5	6	7	8
Q	−	−	−	−	+	−	+	+
$Distance(v)$	0	69	64	86	161	99	113	106
$Predecessor(v)$	0	1	1	2	3	1	2	4
$Marked?$	+	+	+	+	−	+	−	−

After search from u= 8:

v	1	2	3	4	5	6	7	8
Q	−	−	−	−	+	−	+	−
$Distance(v)$	0	69	64	86	161	99	113	106
$Predecessor(v)$	0	1	1	2	3	1	2	4
$Marked?$	+	+	+	+	−	+	−	+

After search from u= 7:

v	1	2	3	4	5	6	7	8
Q	−	−	−	−	+	−	+	−
$Distance(v)$	0	69	64	86	161	99	113	106
$Predecessor(v)$	0	1	1	2	3	1	2	4
$Marked?$	+	+	+	+	−	+	+	+

After search from u= 5:

v	1	2	3	4	5	6	7	8
Q	−	−	−	−	−	−	−	−
$Distance(v)$	0	69	64	86	161	99	113	106
$Predecessor(v)$	0	1	1	2	3	1	2	4
$Marked?$	+	+	+	+	+	+	+	+

Now the main step is completed and we can easily construct shortest paths from 1 to any other vertex. To construct shortest paths from 1 to w, we need to start at w and follow the path *backwards* using the predecessors. Take $w = 8$, for instance. We get $8-4-2-1$, hence path in a shortest path from 1 to 8 is $1 - 2 - 4 - 8$.

11.2.4 Finding a vertex with minimum temporary distance with C++

The pseudocode for Dijkstra's Algorithm contains the following instruction.

```
find an element u of M with minimum Distance(u);  (*)
```

Since this step has to be repeated many times, it should be done efficiently. Surprisingly, this step is by far the most difficult in Dijkstra's Algorithm - if we want to do it properly. We also could do it "improperly" by just iterating over the whole set M, but this makes the algorithm much slower and is also bad style.

Priority queues

The key to doing step `(*)` efficiently is to use a *priority queue*. A priority queue is a collection of values like a vector or list, but with a special distinctive feature: the elements of a priority queue are *automatically sorted according to a criterion we can specify*. The first element of a queue is called the *top*.

Now, we always want to find a vertex u with minimum $Distance(u)$. So, we will use the temporary distances as a sorting criterion. We will arrange the queue such that a vertex u with minimum $Distance(u)$ is always at the top. Then we just have to retrieve this top vertex.

The crucial commands for dealing with priority queues are contained in the following example.

Example 57

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <queue>
4  using namespace std;
5
6  int main()
7  {
8      priority_queue<int, vector<int>, greater<int> > Q;
9      for(int i=0;i<10;i++)
10        Q.push(rand());
11     while(!Q.empty())
12     {
13         cout << Q.top() << endl;
14         Q.pop();
15      }
16 }
```

Explanations

- Line 3: to use priority queue, we need `#include <queue>`
- Line 8: this creates an empty priority queue with elements of type `int` which are sorted from top to bottom in *ascending order*.
- If we would use `less<int>` instead of `greater<int>`, then the elements would be sorted in the reverse order.
- The first "`int`" in line 8 specifies the type of the elements.
- The `vector<int>` tells the compiler how the elements should be saved "internally" (here, in a vector). This is of no concern for us. We simply always put `vector<`*typeOfEntries*`>` in this position.
- The `greater<int>` specifies the sorting criterion.
- Line 10: inserting an element into a priority queue is done by using the `push` function.
- Line 11: `Q.empty()` returns true if and only if `Q` is empty.
- Line 13: retrieving the top element is done by using the `top` function. `Q.top()` returns the top element, but will *not* remove it from the queue.
- Line 14: `Q.pop()` deletes to top element of `Q`.

Now imagine we have three vertices, say 1, 2, 3 with

$$Distance(1) = 50,\ Distance(2) = 15,\ Distance(3) = 40.$$

We want to insert them into a priority queue such that they will be sorted from top to bottom according to increasing distance. The problem is that they will be sorted according to their numbers (1, 2, 3) and not according to the distance if we simply insert them into a priority queue.

The solution is to insert each vertex *together with its distance* as a pair into the queue. Pairs of integers are sorted lexicographically in C++, so we should take the distance as the first member of the pair and the vertex as the second member (do a Google search "lexicographic order" if you are not familiar with this order). For pairs, there is a ready-to-use data structure in STL. See Lab Manual 9 for instructions how to use STL pairs. The following program solves our problem, i.e. inserts 1, 2, 3 into a priority queue such that they will be sorted from top to bottom according to increasing distance.

Example 58

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <queue>
4  using namespace std;
5
6  int main()
7  {
8      priority_queue<
9          pair<int,int>,
10         vector<pair<int,int> >,
11         greater<pair<int,int> > >Q;
12     Q.push(pair<int,int>(50,1));
13     Q.push(pair<int,int>(15,2));
14     Q.push(pair<int,int>(40,3));
15
16     while(!Q.empty())
17     {
18        cout << "vertex: " << Q.top().second;
19        cout << " distance: " << Q.top().first;
20        cout << endl;
21        Q.pop();
22     }
23 }
```

Explanations

- Lines 8-11: a priority queue `Q` is created whose elements are pairs of integers which are sorted from top to bottom in increasing lexicographic order.
- It is not necessary to understand all the details of this quite complicated declaration. It is more important to understand what the queue does. In Dijkstra's Algorithm, we can use exactly the same declaration.
- Line 12: vertex 1 is inserted in `Q` with distance `50`.
- Lines 16-22: the vertices and distances are printed as they appear in `Q` in order to verify that they were sorted correctly.

12 The C++ Library NTL and the RSA Cryptosystem

12.1 C++ Libraries

A *C++ library* is a collection of classes and functions which is ready-to-use in your own programs. Why do we need C++ libraries? The reason is that with the help of such libraries we can write much better, i.e. simpler and more efficient, programs.

When we solve scientific problems by programming, there will be *frequently recurring tasks* which occur as subproblems. A typical example is the solution of systems of linear equations, a task which occurs in thousands, if not millions, of scientific applications. Since these frequently recurring tasks are so important, extremely efficient programs *already have been written* to solve them. It would be very unwise to write our own programs for solving such standard problems. First of all, this would waste a lot of time. But more importantly, even if we spent a whole year to write a program for solving systems of linear equations, for instance, we would have no chance to reach the efficiency of the best available C++ libraries.

Bottom line: if there is a good C++ library available for a task we have to solve, then we should use it.

12.1.1 C++ Static Libraries

We have used a C++ library already: the Standard Template Library (STL). The STL is included in the Dev-C++ environment; hence we do not need to compile the STL before we use it. However, if we want to use other C++ libraries, we have to *compile them first.*

Since most libraries are quite big, it can take minutes or even hours to compile them. If we would have to compile a library each time we use it, we would waste time. The solution is to compile the library *only once* and then "link" the compiled library to our programs when necessary.

A C++ library usually contains a number of source files, say `L1.cpp`, `L2.cpp`,... When we compile the library, object files `L1.o`, `L2.o`,... are created. Since there can be hundreds of such object files, it is convenient to *bundle them together into a single file.* Such a file which contains a bundle of object files is called a *static library* or *archive.* For a static library, usually the file name extension `.a` is used (for "archive").

There are also "dynamic libraries" with file name extension `.dll`, but we will not study them in this course.

12.1.2 Header Files, Source Files, Include Directory

Most C++ libraries comprise *header files* and *source files.* Header files usually have file name extension `.h` and contain only variable, function and class *declarations.* The header files of a C++ library usually are contained in a single directory, called the *include directory* of the library. Source files usually have file name extension `.cpp` and contain

function and member function definitions. They often also are contained in a single directory with name `source` or `src`.

12.1.3 Compiling C++ Static Libraries

Before we can use a C++ static library, we have to compile it. The following are the main steps which usually have to be executed. A more detailed description can be found in the instructions for Lab 10.

- When creating the Dev-C++ project, choose "Static Library" (not "Console Application").
- Specify the include directory of the library in the project options.
- Add all source files of the library to the project.
- Compile the project.

If the compilation is successful, a file with file name extension `.a` will be created. This file is the static library associated with the library.

12.1.4 Using a C++ Static Library in Our Own Programs

We now assume that we successfully have compiled a C++ static library. To use this library in our own programs, we usually ave to proceed as follows.

- Create a new Dev-C++ project, "Console Application" (not "Static Library").
- Specify the include directory of the library in the project options (this step is identical with the second step for creating the static library, see above).
- Add the static library to the project. In Dev-C++, this is done by going to `Project->Project Options->Parameters->Add Library or Object`.
- In the source files that use features of the library, we have to include the header files of the library in which these features are declared. This is done by include statements of the form `#include<Path>` where `Path` is the path of the required header file *relative to the include directory of the library.*
- Compile and test the program.

12.2 The Number Theory Library (NTL)

The Number Theory Library (NTL) is a highly efficient C++ library developed by Victor Shoup (`http://www.shoup.net`). It has been used to set world records in breaking cryptosystems. The following are the main features of NTL.

- Arbitrary precision integers
- Arbitrary precision floating point numbers

- Vectors and matrices
- Polynomials
- Modular Arithmetic
- Finite Fields

All these features are implemented in a highly efficient way. For many large scale problems, NTL is thousands or even millions times faster than common mathematical software like Maple, Mathematica or Matlab.

12.2.1 Arbitrary Precision Integers

In C++, the range of the type `int` is usually limited to roughly $\pm 2 \cdot 10^9$. However, for many mathematical problems, this range is by far not sufficient. NTL provides a class `ZZ` which represents "arbitrary precisions integers", i.e. integers whose range is practically unlimited (only limited by the memory available on the computer).

The class `ZZ` has been written such that it is very easy to use. Almost everything which can be done with variables of type `int` also can be done with objects of the class `ZZ`. The following table explains the main aspects of the use of `ZZ`s.

Command	**Explanation**
`ZZ x;`	Declaration of an arbitrary precision integer `x`.
`ZZ x = 11;` `// error!`	Attention: this would produce an error. Combining assignment and initialization in this way it not possible for `ZZ`s.
`ZZ x = to_ZZ(11);`	This is fine. The function `to_ZZ(...)` converts its parameter to a `ZZ`.
`ZZ y =` `to_ZZ("1234567890123");`	When creating a huge integer, we have to specify it as a string as a parameter of the function `to_ZZ`
`cout << y;`	Output and Input of `ZZ`s is done in the same way as for `int`s
`power(a,b)`	NTL function which returns a^b. Here, `a` must be of type `ZZ` and `b` of type `int`.
`ZZ i;` `for(i=z;i<z+10;i++)...`	We can use `ZZ`s as counter variables in loops. But we have to declare them *before* the loop. `for(ZZ i=z,...` would be wrong.
`ProbPrime(z)`	Tests if `z` is a prime number.
`NextPrime(z)`	Returns the smallest prime `<= z`.
`RandomBnd(z)`	Returns a random number in the range `0,...,z-1`.

Example 59

```
#include<iostream>
#include<NTL/ZZ.h>
NTL_CLIENT

int main()
{
    ZZ x;
    x = 123;
    x = to_ZZ("12345678901234567890");
    ZZ y = to_ZZ("2747239842");
    cout << "x*y: " << x*y << endl;
    cout << "gcd of x and y: " << GCD(x,y) << endl;
    ZZ z = power(to_ZZ(2),65)-1;
    ZZ i;
    cout << "print 10 consecutive ZZs: " << endl;
    for(i=z;i<z+10;i++)
       cout << i << endl;
    cout << "test if z is a prime: " << ProbPrime(z) << endl;
    ZZ next = NextPrime(z);
    ZZ big = power(to_ZZ(10),100);
    ZZ largeRand = RandomBnd(big);
    cout << "large random number: " << largeRand << endl;
    system("pause");
}
```

Example 60 (Euler's fourth power conjecture) In 1769, the great mathematician Euler proposed the conjecture that the sum of three fourth powers of positive integers never can be a forth power itself, i.e. he conjectured that

$$a^4 + b^4 + c^4 = d^4$$

is impossible for positive integers a, b, c, d. For more than 200 years, nobody was able to prove or disprove the conjecture. In 1988, Noam Elkies (`http://www.math.harvard.edu/~elkies`) found a counterexample. It is contained in the following program.

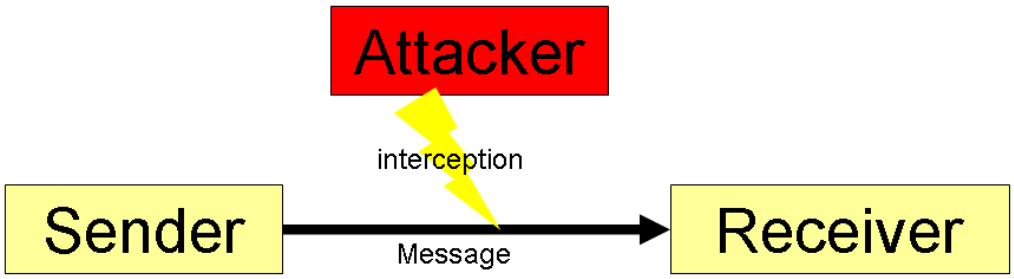
```
1  #include<iostream>
2  #include<NTL/ZZ.h>
3  NTL_CLIENT
4
5  int main()
6  {
7      ZZ a = to_ZZ(2682440);
8      ZZ b = to_ZZ(15365639);
9      ZZ c = to_ZZ(18796760);
10     ZZ d = to_ZZ(20615673);
11     if((power(a,4)+power(b,4)+power(c,4))==power(d,4))
12        cout << "Sorry, Euler." << endl;
13     system("pause");
14 }
```

Explanations

- Line 11: The comparison operator `==` can be used for `ZZ`s in the same way as for `int`s.

12.3 The RSA Cryptosystem

A *cryptosystem* is a method to send "secret messages". This means that no unauthorized person should be able to read the messages. Such an unauthorized person is called an *attacker*. It assumed that the attacker is quite powerful: he can *intercept all sent messages*, i.e. he has full access to all details of the messages.



Under these unfavorable circumstances, it seems quite hard to imagine that it is possible to send truly secret messages. However, it *is perfectly possible* and even by a very simple method. This is due to the epoch-making discovery of the `RSA cryptosystem` in 1977 by Ron **R**ivest, Adi **S**hamir and Len **A**dleman.

The only mathematical tool necessary for RSA is computations with large integers. Hence, RSA can easily be implemented in C++ using NTL. In RSA, a message has to be represented by an *integer*. Thus, if we want to send a message which contains text, we first need a method to convert a string to an integer.

12.3.1 Converting a String to an Integer and Vice Versa

To convert a string to an integer and vice versa, we can use *ascii-codes*. In C++, each symbol which can occur in a string is of type `char` and is represented by its ascii-code. The ascii-codes of commonly used symbols, numbers and letters are contained in the following table.

33	!	34	"	35	#	36	$	37	%	38	&	39	'	40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3	52	4	53	5	54	6
55	7	56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?	64	@	65	A
66	B	67	C	68	D	69	E	70	F	71	G	72	H	73	I	74	J	75	K	76	L
77	M	78	N	79	O	80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_	96	`	97	a	98	b
99	c	100	d	101	e	102	f	103	g	104	h	105	i	106	j	107	k	108	l	109	m
110	n	111	o	112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w	120	x
121	y	122	z	123	{	124	|	125	}	126	~										

If `c` is a variable of type `char`, then we can use `(int) c` to obtain the ascii code of the symbol corresponding to `c`.

Example 61

```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6      string s= "RSA";
7      for(int i=0;i<s.size();i++)
8        cout << (int) s[i] << endl;
9      system("pause");
10 }
```

Explanations

- Line 8: `s[i]` is the `(i+1)`st symbol in the string `s`.
- `(int) s[i]` returns the ascii-code of this symbol.
- The ascii-codes of the letter `R`, `S` and `A` will be printed to the screen, i.e. `82`, `83` and `65`.

The basic ideas to convert a string to an integer are the following.

- Modify the ascii-codes so that they all have exactly three digits.
- To do this, fill up the ascii-codes to three digits by adding leading zeros if necessary.
- To represent a string `s` as an integer, concatenate the ascii-codes of `s[size()-1]`,...,`s[0]`.
- If the resulting integer has leading zeros, delete them.
- Note that the ascii-codes are concatenated in *reverse order*.
-

Example 62

- The symbols in the string `"RSA"` have ascii-codes `82`, `83`, `65`.
- Fill them up to three digits: `082`, `083`, `065`.
- Concatenate them in reverse order: `065083082`.
- Delete leading zeros: `65083082`.
- Hence `"RSA"` is represented by `65083082`.

Actually, we can write down a formula for the integer corresponding to a string `s`:

$$\sum_{\texttt{i}=0}^{\texttt{s.size()}-1} 10^{3\texttt{i}}((\texttt{int})\texttt{s[i]})$$

Here, as above, `(int) s[i]` is the ascii-code of the `(i+1)st` symbol in `s`.

Example 63 The following functions convert strings to integers and vice versa. Since the integer we get usually will be huge, we use arbitrary precision integers from NTL.

```
1  ZZ StringToZZ(string s)
2  {
3      ZZ M=to_ZZ(0);
4      for(int i=0;i<s.size();i++)
5          M+= s[i]*power(to_ZZ(10),3*i);
6      return M;
7  }
8
9  string ZZToString(ZZ M)
10 {
11         string result;
12         result.resize(50);
```

```
13          int counter=0;
14          while(M>to_ZZ(0))
15          {
16              result[counter]= (M%1000);
17              M=M/1000;
18              counter++;
19          }
20          return result;
21  }
```

Explanations

- The ideas behind the function `StringToZZ` have been explained above.
- Line 12: we only allow strings of size at most 50.
- `ZZToString` is the "inverse function" of `StringToZZ`. The idea behind `ZZToString` is same as for the solution of Question 4 of the Midterm Exam. Think about it!

12.3.2 Encrypt an Integer

We have seen how to represent a message (string) as an integer, say M. Now we need a method to *encrypt* M. This means we construct another integer M' from M such that no attacker can reconstruct M from M'. The encryption of works as follows.

- Choose a huge integer $n > M$ and an integer e with $1000 \leq e < n$.
- Set $M' = M^e \bmod n$ (see Lab Manual 2 for the mod operator).

It is a fact (even if n and e are known!) that M cannot be recovered from M' with present day technology if n is large enough.

12.3.3 Decryption

The *receiver* needs to recover M from M'. This is called *decryption*. Decryption can be achieved if a *secret key* is available to the receiver. In our case, the secret key is an integer d with $1 < d < n$ such that $M = (M^e)^d \bmod n$ for all possible messages M. Since $M' = M^e \bmod n$, the receiver can reconstruct M from M' by computing $(M')^d \bmod n$.

Can such an integer d be found. The answer is yes, but only if the *prime divisors of n are known*. Hence, if the receiver knows the prime divisors of n and keeps them secret, he can construct d, but nobody else can!

12.3.4 Construction of Keys

As indicated in the last paragraph, the *receiver* is responsible for constructing the decryption key and keeping it secret. In a transmission process, usually the *sender* is first one to become active. However, for the RSA cryptosystem, it is different: the *receiver* first

has to take an action, namely, he has to construct the keys: the public key (n, e) as well a the secret key d.

To construct the keys, the receiver executes the following steps.

- Find two distinct random prime numbers p, q, both with around 150 digits.
- Compute $n = pq$.
- Compute a number e with $1000 < e < n$ which has no nontrivial common divisor with $(p-1)(q-1)$, i.e., g.c.d.$(e, (p-1)(q-1)) = 1$.
- Compute the number d with $1 < d < (p-1)(q-1)$ and $ed \bmod (p-1)(q-1) = 1$.
- Make n and e publicly available and keep d, p, q secret.

The crucial fact here is that the choice of n, e and d implies

$$M = (M^e)^d \bmod n \tag{2}$$

for all integers M. This makes sure that the receiver obtains the original message M when he computes $(M')^d \bmod n$, where $M' = M^e \bmod n$ is the encrypted message he received. We do not discuss the proof of (2) here. It is part of elementary Number Theory.

12.3.5 Sending and Receiving RSA Encrypted Messages

Sending and receiving an RSA encrypted message is done by executing the following steps.

- The receiver constructs the keys n, e and d, makes (n, e) publicly available, and keeps d secret.
- The sender converts the message (string) into an integer $M < n$.
- The sender encrypts M by computing $M' = M^e \bmod n$, and sends M' to the receiver.
- The receiver reconstructs M from M' by computing $M = (M')^d \bmod n$.
- The receiver converts M to the original message (string).

Summary of RSA transmission process