

Chapter 2

C++ basics

C combines the power of Assembler
with the portability of Assembler...

Anonymous,
alluding to Bill Thacker.

2.1 The structure of program

The best way to start learning a programming language is probably by writing a program. Lets start writing our first C++ program by modifying the *wxDev-C++* sample - see Table 2.1. Table 2.2 represents output of program.

This program is one of the simplest programs, that can be written in C++, but it already contains the basic components, that every C++ program has got. Look line by line at the program code.

```
// a simple program
```

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behaviour of the program. The programmer can use them to include short explanations of the program code. Only one line (or part of line) could be commented using double slash (//). */**/* signs could be used to comment several code lines (*/** beginning of the comments, **/* the end of comments). It is important to remember, that we can not put one multiline comment into another.

```
/*This is /*a wrong type of*/ comments*/  
  
#include <cstdlib>  
#include <iostream>
```

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compilers preprocessor. In this case the directives: first `#include <cstdlib>` and second `#include <iostream>` tell the preprocessor to include the appropriate libraries. This specific files (*cstdlib* and *iostream*) include the declarations of the input-output libraries in C++. Later we will use one more directive for the preprocessor (`#define`).

Table 2.1.	Simple program.
<pre>// a simple program #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { cout << "Hello World! This is my first C++ program" << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; }</pre>	

Table 2.2.	Output of program.
<pre>Hello World! This is my first C++ program Press the enter key to continue ...</pre>	

```
using namespace std;
```

Namespaces allow to group entities like classes, objects and functions under a name. In this way the global scope can be divided in "sub-scopes", each one with its own name. All the elements of the standard C++ library are declared within the namespace called `std`. So, in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library. [45]

Global variables can be declared after the specification of namespace.

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

This line corresponds to the beginning of the definition of the `main()` function. The `main()` function is the point where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For the same reason, it is essential that all C++ programs have a `main()` function.

The word `main` is followed by expression `(int argc, char *argv[])` because it is a function declaration. In C++ each function has brackets after its name. It does not matter, weather function has any parameters or not. Parameters `argc` and `argv[]` are simply naming conventions. They have meaning **argument count** and **argument vector**. These arguments are used to parse command line parameters to a program. Argument `argc` contains the amount of arguments, separated by spaces, that were entered when the program was invoked. Argument `argv` contains the actual arguments themselves.

```
cout << "Hello World! This is my first C++ program" << endl;
```

After the name of the function and its parameters we can find the body of the function enclosed in brackets `{...}` . The information contained within these brackets is what the function does when it is executed.

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program. Name `cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the *Hello World! This is my first C++ program* sequence of characters) into the standard output stream (`cout`, which usually corresponds to the screen).

Stream `cout` is declared in the `iostream` standard file within the `std` namespace. That is why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Statement `endl` in the end of the line moves the cursor to the next line of the screen. We put semicolon in the end of each C++ program line. It is also important to remember that C++ compiler is case-sensitive. So, if one write `Cout` instead of `cout` he will have a mistake.

```
cout << "Press the enter key to continue ...";
```

This statement prints the sentence *Press the enter key to continue ...* to the program window.

```
cin.get();
```

This statement is used to stop the execution of the program (program will wait ENTER button to be pressed).

```
return EXIT_SUCCESS;
```

The return statement causes the main function to finish. Return may be followed by a return code (in our example is followed by the return code with a value of `EXIT_SUCCESS`). In general, the return code can be any integer number. This code can be used for program debugging.

2.2 Variables and data types

The variable is a PC memory area assigned to store particular data type. Using variables it is possible to store different values of the same data type in the same memory area at different times. It is important to understand what types of variables are used in C++ and how they are stored.

Each variable needs an identifier that distinguishes it from the others. A valid identifier is a sequence of one or more letters, digits or underscore characters (`_`). Neither spaces nor punctuation marks or symbols can be a part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (`_`), but in some cases these may be reserved for compiler specific keywords or external identifiers. Identifier can not begin with a digit. Another important rule is that identifier name can not match any keyword of the C++

language or your compiler's specific ones, which are reserved keywords. It is very important to remember that the C++ language is a case sensitive language. It means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters.

By creating a variable programmer assigns memory area to store its value. It is important to specify what kind of data will be stored. The computer memory is organized in bytes. A byte is the minimum amount of memory that can be managed in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In order to store more complex data we must have additional data types that use groups of bytes to store the information. [45]

The main C++ variable types are following: `char`, `int`, `float`, `double`. The characteristics of each data type are presented in Table 2.3. The size and range depends on the compiler and on the system the program is compiled for. The most common values are presented.

There is also a `void` (empty) data type. It is used to create the function which does not return any value. Unsigned data type shifts the range of values. For example, if we know that we will use only positive integers, we can use the unsigned type, and the potential number range will be from 0 to 4294967295. Floating-point, i.e., real number storage in memory is different from integer numbers, because the first ones are stored in two parts: separate stored decimal and exponent value.

In order to use a variable in C++, firstly we should declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like `int`, `bool`, `float`...) followed by a valid variable identifier. For example:

```
float a; int number;
```

These are two valid declarations of variables. The first one declares a variable of type `float` with the identifier `a`. The second declares a variable of type `int` with the identifier `number`.

Table 2.3. Characteristics of the main C++ data type.			
Name	Description	Size, B	Range for signed and unsigned
<code>char</code>	character or short integer	1	(-128 ÷ 127); (0 ÷ 255)
<code>short int</code> <code>short</code>	short integer	2	(-32768 ÷ 32767); (0 ÷ 65535)
<code>int</code>	integer	4	(-2147483648 ÷ 2147483647); (0 ÷ 4294967295)
<code>long int</code> <code>long</code>	long integer	4	(-2147483648 ÷ 2147483647); (0 ÷ 4294967295)
<code>bool</code>	boolean	1	TRUE (1) or FALSE (0)
<code>float</code>	floating point	4	± 3.4e ± 38 (7 digits)
<code>double</code>	double precision, floating point	8	± 1.7e ± 308 (15 digits)
<code>long double</code>	long double precision, floating point	8	± 1.7e ± 308 (15 digits)
<code>wchar_t</code>	wide character	2/4	one wide character

Once declared, the variables `a` and `number` can be used within the rest of their scope in the program. Variables can be declared anywhere in the program code, but before the first use of the variable.

Usually most variables are declared at the beginning of program, outside all functions (global variables) or at the beginning of function (local variable). Declaring more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

The integer data types `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name. By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed. Variable value can be set just after the declaration. It is called variable initiation. Initiation (initialization) is the assignment of variable value, declaring it. For example, variables can be described as follows:

```
int      k = 3;
float length = 2.18;
```

Initialization allows to shorten the program.

2.3 Constants

The sample "Hello World" program has just printed several sentences on the screen. Quotes were used to mark the text we would like to print:

```
cout << "Hello World! This is my first C++ program" << endl;
```

The text in quotes is string constant in this case. We use quotes to mark strings in C++. Single quotation marks are used to mark a separate symbol (`'a'`). String and symbolic constant, also Boolean constants are non-numeric constants.

Numeric constants are also used in C++ language. There are integer and floating point constants. Integer constants identify integer decimal values. In addition to decimal numbers, C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (a zero character). In order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
const int e1 = 2012; // decimal
const int e2 = 03734; // 0, octal
const int e3 = 0x7DC; // 0x, hexadecimal
const int e4 = 0x7dc; // 0x, hexadecimal
```

Note that we can use both lower-case or capital letters typing x and corresponding letters in hexadecimal number. Printing the constant its value is automatically converted into decimal number.

Floating point constants express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character.

```
const double e5 = 9.87654;      // 9.87654
const double e6 = 5.97219e24;   // Earth mass, kg
const double e7 = 1.674e-27;    // mass of the hydrogen atom, kg
const double e8 = 1e100;       // gugol
```

Real numbers corresponding Earth mass, mass of the hydrogen atom (both in kg), and famous abstract number *gugol* are presented in usual notation:

$$e6 = 5.97219 \cdot 10^{24} ; \quad e7 = 1.674 \cdot 10^{-27} ; \quad e8 = 1 \cdot 10^{100} \quad (2.1)$$

We can make the program more readable by changing numeric values to identifiers. Besides, the use of constants enables us to change the parameter value easily. So, constant in programming is the identifier associated with fixed values. In C++ language we use two ways to define constants.

We can use macros to define constants. The `#define` preprocessor directive is used in this case. For example, we write:

```
# define VILNIUS 1323
# define EUR 3.45
# define ERROR "My error..."
```

We do not put the semicolon at the end of line, because this is not a C++ statement but a preprocessor directive. Compiling a program the defined constant name in program text will be replaced with the corresponding value. Once the constants are defined, you can use them in the rest of the code instead of writing their values. Constant names are usually written in capital letters.

We can use `const` command to define declared constants. The profile of the formal syntax is as follows:

```
const <data type> <constant name> = <constant value>
```

For example:

```
const char FIRST_LETTER = 'A';
const int MONTH = 30;
const float PI = 3.14;
```

2.4 Basic input and output

Using the standard input and output library, programmer is able to interact with the user by printing messages on the screen and getting the user's input from the keyboard. C++ uses a convenient abstraction called streams to perform input and output operations in sequential

media, such as the screen or the keyboard. A stream is an object, where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know, will it accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

By default, the standard output of a program is the screen, and the C++ stream object defined to access is `cout`. Function `cout` is used in conjunction with the insertion operator, which is written as `<<` (two "less than" signs).

```
cout << "My text";    // prints "My text" on screen
cout << 12.56;        // prints number 12.56 on screen
int x=7; cout << x;    // prints number 7 (the content of x) on screen
```

The insertion operator (`<<`) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

The last statement would print the message *Hello, I am a C++ statement* on the screen. The utility of repeating the insertion operator (`<<`) is useful when we want to print out a combination of variables and constants or more than one variable.

It is important to notice that `cout` does not add a line break after its output unless we indicate it explicitly, therefore, the following statements:

```
cout << "This is a first sentence. ";
cout << "This is a second sentence. ";
```

will be shown on the screen one following the other without any line break between them:

```
This is a first sentence. This is a second sentence.
```

even though we had written them in two different insertions into `cout`. In order to perform a line break on the output we must explicitly insert a new-line character into `cout`. In C++ a new-line character can be specified as `\n` (backslash, n):

```
cout << "First line.\n";
cout << "Second line.\nThird line.";
```

This produces the following output:

```
First line.
Second line.
Third line.
```

(`\n`) - combination of backslash and n is called *escape code*. These are special characters that are difficult or impossible to express otherwise in the source code of a program. All of them are preceded by a backslash (`\`). Here you have a list of the most recent escape codes.

<code>\n</code>	newline
<code>\b</code>	backspace
<code>\'</code>	single quote (')
<code>\?</code>	question mark (?)

<code>\t</code>	tab
<code>\a</code>	alert (beep)
<code>\"</code>	double quote (")
<code>\\</code>	backslash (\)

In order to add a newline, you may also use the `endl` manipulator. For example:

```
cout << "First line." << endl;
cout << "Second line." << endl;
```

would print out:

```
First line.
Second line.
```

Statement `cout` has two useful methods, which allow formatted printing.

Method `width()` defines the width of the output field. We must use this method before the output statement. We can set the width of the output field only for one output.

Method `precision()` defines the number of significant digits for real numbers (floating-point numbers). Precision remains valid for all output sentences used after its declaration, until we set a new value. The syntax of these methods is as follows:

```
cout.width      (<number of positions>);
cout.precision (<digit count>);
```

A sample program of formatted printing is presented in Table 2.4. The output of the program is shown in Table 2.5. As you can see, `cout.width(9)` reserves 9 positions for output and affects only one element (the second `cout` is not affected). The statement `cout.precision(4)` makes program to print only 4 decimal digits of `x` value.

Analogically, `cout.precision(3)` makes program to print only 3 decimal digits of `x` value. The value of `x` remains the same in both cases, only output is changed (the value of `x` is rounded). Note, that precision method affects the mantissa (decimal) part of a floating-point real number and do not take into account the value of exponent. We have different number of decimal digits while working with larger `x` values using the same precision.

We use system command `system("PAUSE")` at the end of the program to hold on the program window before exit.

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (`>>`) on the `cin` stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int day;
cin >> day;
```

The first statement declares a variable of type `int` called `day`, and the second one waits for an input from `cin` (the keyboard) in order to store it in this integer variable.

Table 2.4.	Formatted output.
<pre> //Formatted printing #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { float x = 0.12345; // initialization of x cout << 123456789 << endl; // first line cout << x << endl; // output of x cout.width(9); // width is set to 9 positions cout << x << endl; // x value is printed to the reserved field cout << x << endl; // width value is no longer valid cout.precision(4); // precision is set to 4 digits cout << x << endl; // x value with 4 decimal digits (rounded) cout.precision(3); // precision is set to 3 digits cout << x << endl; // x value with 3 decimal digits (rounded) x=1.2345; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) x=12.345; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) x=123.45; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.5.	Output of program.
<pre> 123456789 0.12345 0.12345 0.12345 0.1235 0.123 1.23 12.3 123 Press any key to continue ... </pre>	

Function `cin` can only process the input from the keyboard once the `RETURN` key has been pressed. Therefore, even if you request a single character, the extraction from `cin` will not process the input until the user presses `RETURN` after the character has been introduced.

We must always consider the type of the variable that is used as a container with `cin` extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters. So the user must provide a valid data to the program. The input of invalid data may cause your program to misoperate.

We can also use `cin` to request more than one datum input from the user:

```
cin >> x >> y;
```

is equivalent to:

```
cin >> x;
cin >> y;
```

In both cases the user must give two data, one for variable `x` and another one for variable `y` that may be separated by any valid blank separator: a space, a tab character or a newline.

We can also use input and output functions inherited from the C language. These functions require the inclusion of `stdio.h` header file to the program. Function `scanf()` is used for data entry. The syntax is as follows:

```
scanf ("%f", &x);
```

Here is the address operator `&`, i.e., `&x` means the address of the variable `x` in the computer memory. Specifier `%f` means that floating-point number will be entered. Similarly, we can define another format, for example:

<code>%c</code>	character	<code>%s</code>	string
<code>%d</code>	integer	<code>%ld</code>	long
<code>%lf</code>	double		

If we specify `%s` the address operator (`&`) is not required because the string is an array and the array name represents its address.

Function `printf()` is often used for formatted printing. This function offers more possibilities for formatted printing than the output stream `cout`. The first parameter of `printf()` function is a string-constant, indicating the desired text and format settings for parameters we want to print, after this string we specify the variables and values to be printed. We use the same format specifiers as with `scanf()` function.

To define the format of floating-point number we use `%n.mf` structure where `n` is the width of the output field and `m` is the number of decimal digits. In this case the number of decimal digits shown do not depend on the number as it was with `precision` method. We can insert `-` before `n` to enable left alignment. It is also possible to use escape codes to improve the formatting.

Table 2.6 represents the program, where the possibilities of `printf()` function are shown. Table 2.7 is devoted for program output.

Table 2.6.	Formatted printing.
	<pre> //Formatted printing #include <cstdlib> #include <iostream> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { float x = 0.12345; // initialization of x printf("123456789123456789\n"); // first line printf("%f\n",x); // output of x printf("%9.5f\n",x); // 9 positions, 5 decimal places printf("%-9.5f%9.5f\n",x,x); // left and right alignment printf("%9.4f\n",x); // x value with 4 decimal digits printf("%9.3f\n",x); // x value with 3 decimal digits x=1.2345; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits x=12.345; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits x=123.45; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits printf("1:%f\t2:%f\n3:%f\t4:%f\n",x,x,x,x); //tabulation system("PAUSE"); return EXIT_SUCCESS; } </pre>
Table 2.7.	Output of program.
	<pre> 123456789123456789 0.123450 0.12345 0.12345 0.12345 0.1235 0.123 x= 1.235 x= 12.345 x=123.450 1:123.449997 2:123.449997 3:123.449997 4:123.449997 Press any key to continue ... </pre>

2.5 Operators

The purpose of most programs is data entry and manipulation with data. For that purpose, C++ integrates operators. Operators are symbols or words that make the program to do something with the variables or expression. C++ has many different operators. The most important of them are the arithmetic (+ - * / %), compound assignment (+= -= *= /= %=), increase and decrease (++ --), relational and equality (> >= < <= == !=), logical (&& || !), and conditional operator (? :). The assignment operator (=) assigns a value to a variable.

```
y = 7;
```

This statement assigns the integer value 7 to the variable `y`. The left part of the assignment has to be a variable whereas the right part may contain either a constant, a variable, the result of an operation or any combination of those.

The most important rule while assigning is the right-to-left rule: **the assignment operation always takes place from right to left, and never the other way.**

```
x = y;
```

This statement assigns to variable `x` (the left value) the value contained in variable `y` (the right value). The value that was stored until this moment in `x` is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of `y` to `x` at the moment of the assignment operation. Therefore a later change of `y` will not affect the new value of `x`.

The five arithmetical operations supported by the C++ language are: + addition; - subtraction; * multiplication; / division; % modulo. Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable `a` will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Writing program in C++ we can shorten the expression `x=x+a` like this: `x+=a`. This operator is called *compound assignment operator*. There are more of them:

+=	<code>x=x+a;</code>	<code>x+=a;</code>
-=	<code>x=x-a;</code>	<code>x-=a;</code>
*=	<code>x=x*a;</code>	<code>x*=a;</code>
/=	<code>x=x/a;</code>	<code>x/=a;</code>
%=	<code>x=x%a;</code>	<code>x%=a;</code>

Shortening some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (`++a`) or after it (`a++`). Although in simple expressions like `a++` or `++a` both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In

the case that the increase operator is used as a prefix (`++a`) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (`a++`) the value stored in `a` is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. [45]

In order to evaluate the comparison between two expressions we can use the relational and equality operators (`==`, `!=`, `>`, `<`, `>=`, `<=`). The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

It is important to remember, that the operator `=` (one equal sign) is not the same as the operator `==` (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (`==`) is the equality operator that compares whether both expressions in the two sides of it are equal to each other.

Logical expressions are those expressions that contain logical operators, logical variables and/or the relational operators. The result of logical expression is true or false. We can use the following logical operators: `&&` (*AND*), `||` (*OR*), `!` (*NOT*).

Operator `!` (*NOT*) has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false.

The logical operators `&&` (*AND*) and `||` (*OR*) are used evaluating two expressions to obtain a single relational result. `&&` (*AND*) operation results true if both its two operands are true, and false otherwise. `||` (*OR*) operation results true if either one of its two operands is true, thus being false only when both operands are false themselves.

The conditional operator evaluates an expression returning a `value1` if that expression is true and a `value2` if the expression is evaluated as false. Its format is:

```
condition ? value1 : value2
```

Conditional operator enables us to shorten the program.

Type casting operators allow to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (`()`):

```
int i;
float f = 3.14;
i = (int) f;
```

Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

Note, that in C++ the type of expression result is determined according to the types of arguments. For example, if an integer is divided by integer, the answer is truncated and remains integer. Such program line:

```
cout << 5/2 <<endl;
```

will print out the answer 2. If we want to see the right answer, we must change the type of arguments:

```
cout << 5.0/2 <<endl;
```

or

```
cout << (float)5/2 <<endl;
```

First time we changed the data type of first argument (5.0 is a real number). Second time we used type casting. Using type casting we can also convert char to integer and integer to char. For example, such two program lines:

```
cout << (char) 97 << endl;
cout << (int) 'b' << endl;
```

will print out letter 'a' and number 98 (ASCII code of letter 'b'), respectively.

It is useful to remember the priority of operator. From greatest to lowest priority, the priority order is as follows:

- a) increase (increment) (++), decrease (decrement) (--) and logical *NOT* (!);
- c) type casting;
- c) multiplicative operators (*, /, %);
- d) additive operators (+, -);
- e) relational operators (>, >=, <, <=);
- f) equality operators (==, !=);
- g) logical *AND* (&&);
- h) logical *OR* (||);
- i) conditional operator (?:);
- j) assignment operators (=, +=, -=, *=, /=, %=).

We can change the priority of evaluation by using brackets.

2.6 Macros

In order to create simple functions it is also possible to use **define** preprocessor directive. These functions are usually called *macros*.

A macros is a finite fragment of the code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. It is possible to define any valid identifier as a macro, even if it is a C++ keyword. The preprocessor does not know anything about keywords. This can be useful if you want to hide a keyword such as `constant` from an older compiler that does not understand it.

We have already mentioned the possibility of using **define** directive to describe constants. For example:

```
#define LT 3.45
```

defines the macro named `LT` as an abbreviation for the value `3.45`. If somewhere after this **define** directive a C++ statement of the form comes,

```
money = eur * LT;
```

the C++ preprocessor will recognize and expand the macro `LT`. The C++ compiler will see this command line as if you had written

```
money = eur * 3.45;
```

By convention, macro names are written in uppercase. It is easier to read a program when it is possible to tell at once which names are macros.

The macro's body ends at the end of the **define** line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will come out on one line. For example:

```
#define NUMBERS 1, \
                2, \
                3
int x[ ] = { NUMBERS };
```

It corresponds to the statement:

```
int x[] = { 1, 2, 3 };
```

It is also possible to define macros which use looks like a function call. These are called function-like macros. Use the same **define** directive to define a function-like macro. But you should put a pair of parentheses immediately after the macro name. A function-like macro is only expanded if its name appears with a pair of parentheses after it. If you put spaces between the macro name and the parentheses in the macro definition, that does not define a function-like macro. We can even pass the parameters to macro. For example:

```
#define SUM(a, b) a+b
```

Using this macro, the output statement

```
cout << SUM(5, 3) << endl;
```

will print out value 8. Note, that output statement

```
cout << SUM(5, 3)*7 << endl;
```

will print out value 26, as we have no brackets in macro definition. So, we have an expression like this $5+3*7$.

It is also possible to use Assembler code in C++ program. The usage of Assembler allows to speed up the program, or to create specific programmer implemented functionality. For example, we can create function **add()** using Assembler. Function **asm** contains Assembler code.

```
int add(int a, int b)
{
    int c;
    asm(
        "addl %%ebx, %%eax;"
        : "=a" (c)
        : "a" (a),
          "b" (b)
        );
    return c;
}
```

This function will implement addition operation for integer numbers. The output sentence

```
cout << add(5, 3) << endl;
```

will print out 8.

2.7 Control structures

A program is usually not limited to a linear sequence of instructions. During the process it may bifurcate, repeat the code or take decisions.

For that purpose C++ provides control structures, that serve to specify, what has to be done by our program, when and under which circumstances.

In this section we are going to study such control structures as conditional structures, loops and some others.

In real life we have to change our activities according to the circumstances. For example, if it is raining outside, we choose proper clothes, perhaps, we will not go to the picnic. While programming with C++ we must also consider different conditions. C++ language, as many others programming languages, offers two main conditional structures: `if()` structure and `switch()` structure. So, the program can choose, what action to perform according to certain conditions.

The `if` keyword is used to execute a statement or a block of commands only if a condition is fully filled. Its form is:

```
if (condition) statement
```

where *condition* is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure. Note, that condition should always be in brackets. The statement may be a single command or a block of commands (if we want to execute more than one command).

Block of commands or compound statement is a group of statements, which are separated by semicolons (;), like all C++ statements, but grouped together in a block enclosed in braces: `{ }`:

```
{ statement1; statement2; statement3; }
```

For example, the following code fragment prints `y` value if `x` is positive:

```
if (x > 0)      cout << "y = " << y << endl;
```

We can specify, what must be done, if the condition is not fully filled by using the keyword `else`. This keyword is appended to `if()` as follows:

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

For example:

```
if  (x % 2 == 0)    cout << "x is an even number";  
else                cout << "x is an odd  number";
```

We can add several `if-else` statements one to another if we want to evaluate several conditions. For example:

```
if      (x < 0)      cout << "x is a negative number";  
else if (x > 0)      cout << "x is a positive number";  
else                cout << "x is 0";
```

If we want to check several possible constant values of an expression, we can use `switch()` statement. Its form is the following:

```
switch (expression)
{
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    ...
    default:
        default statement;
}
```

Statement `switch()` evaluates expression and checks if it is equivalent to `value1`. If it is, it executes `statement1` (here can be several statements) until it finds the `break` statement. When it finds this break, the program jumps to the end of the `switch()` selective structure. If expression was not equal to `value1`, it will be checked against `value2`. If it is equal to this, it will execute `statement2`, until a `break` keyword is found, and then will jump to the end of the `switch()` selective structure. Finally, if the value of expression does not match any of the previously specified values (you can include as many case labels, as many values you want to check), the program will execute the statements included after the `default` label, if it exists (since it is optional).

In `switch()`, each `case` must be ended with `break`, otherwise the program will not stop after executing the statements, that must be executed for a specific condition, and will check out other values. We must also remember, that we can use only discrete expression in `switch` statement, and all the values also must be discrete.

As an example we will present a simple calculator (see Table 2.8). This program allows to enter two numbers and an operation sign. Afterwards it prints the result.

We often need to repeat some actions several times, perhaps changing only some details. Cyclic control structures enable us to repeat some of the steps many times and to organize cyclic computations.

C++ programming language has three types of cyclic structures (loops): `while()`, `do-while()` and `for()`.

The `while()` loop repeats a statement or a block while the condition set in expression is true. Its format is:

```
while (expression) statement
```

Note, that if the expression is false from the beginning, the `while` loop statement will not be executed at all. Its functionality is only to repeat the statement, while the condition set in expression is true.

For example, this program finds the first three-digit number which is divided by 17 (Table 2.9). The idea of the program is to multiply the number 17 by different numbers (we start with 1 and increase the value of second multiplier during each loop iteration) until we will get a number which is greater or equal to 100.

Table 2.8. Simple calculator.

```

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

int main(int argc, char *argv[])
{
    double x,y;  char s;
    cout << "Enter x: " ; cin >> x;
    cout << "Enter the operation sign (+, -, *, /): " ;
    cin >> s;
    cout << "Enter y: " ;
    cin >> y;

    switch (s)
    {
        case '+':
        {
            cout << x << "+" << y << "=" << x+y << endl;  break;
        }
        case '-':
        {
            cout << x << "-" << y << "=" << x-y << endl;  break;
        }
        case '*':
        {
            cout << x << "*" << y << "=" << x*y << endl;  break;
        }
        case '/':
        {
            if(y != 0)  cout << x << "/" << y << "=" << x/y << endl;
            else      cout << "Can not divide by 0\n";
            break;
        }
        default:
            cout<<"Invalid operation sign\n";
    };
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Creating a `while` loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. If we want to have the eternal loop, we can write `while(true)...` or `while(1)....`

Table 2.9.	Program.
	<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int a, b, c; a=17; b=1; c = a * b; while (c < 100) { b++; c = a * b; } cout << b << " " << c << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>

Sometimes we need to check the condition after the loop statement (in order to execute the loop statement at least once). We can use the **do-while** loop in this case. The functionality of this loop is exactly the same as the **while** loop, except that condition in the **do-while** loop is evaluated after the execution of statement instead of before. Its format is:

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

The following example program will ask for the year of Vilnius establishment until the right answer will be entered (Table 2.10).

The last, but not the least is **for()** loop. The main function of this loop is to repeat the statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So, this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration. Its format is:

```
for (initialization; condition; increase) {statement;}
```

It works in the following way [45].

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. Condition is checked. If it is true the loop continues, otherwise the loop ends and the statement is skipped (not executed).
3. The statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Table 2.10.	Vilnius establishment.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int a=0; do { cout << "Enter the year of Vilnius establishment \n"; cin >> a; } while (a != 1323); cout << "You are right! Vilnius was established in 1323" << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example, we could write:

```
for (; n<10; ) {...}
```

if we wanted to specify no initialization and no increase; or

```
for (; n<10; n++) {...}
```

if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before). Even the condition could be refused. In this case we will have open loop

```
for (;;) {...}
```

This is the infinite loop.

Optionally, using the comma operator (,) we can specify more than one expression in initialization and increase fields of **for** loop. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. Note, that if we use a condition in **for** loop, it should be only one.

For example, we have a deposit in a bank and we want to know how much money we will have after a year, two, etc, 10 years. A program using a **for** loop is presented in Table 2.11.

It is possible to insert one loop into another. For example, this program will print out the multiplication table (see Table 2.12).

The inner loop prints one line, while the outer loop repeats new line printing with another value of *i*. Sometimes, working with loops or other structures we need to break the operation, or to skip the iteration or even to end the program.

Table 2.11.	Deposit.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { float s=1000; // the value of the deposit float p= 3; // the interest rate for(int y=1; y<=10; y++) { s *= 1.0 + p/100; cout << "After " << y << " year(s) we will have: " << s << endl; } system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.12.	Multiplication table.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { for(int i=1; i<10; i++) { for(int j=1; j<10; j++) { cout.width(3); cout<<i*j; } cout<<endl; } cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Statements `break`, `continue` and `exit` are used in these cases. It is possible to leave a loop even if the condition for its end is not fulfilled using a `break` statement. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we can use infinite loop to give the user an opportunity to perform some actions as many times as he wants (Table 2.13).

Table 2.13.	Infinite loop management.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x; char y; for(;;) { cout << "Enter x value:\n"; cin >> x; cout << x << "^2=" << x*x << endl; cout << "Do You want to continue? (y/n)\n"; cin >> y; if (y == 'n') break; } system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.14.	Infinite loop.
<pre> // Infinite loop with continue and break statements #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x; char y; for(;;) { cout << "Enter x value:\n"; cin >> x; if (x==0) continue; cout << "1/" << x << "=" << 1.0/x << endl; cout << "Do You want to continue? (y/n)\n"; cin >> y; if (y == 'n') break; } system("PAUSE"); return EXIT_SUCCESS; } </pre>	

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, this program will skip the iteration if 0 will be entered (see Table 2.14).

The purpose of exit is to terminate the current program with a specific exit code. This

function is defined in the `cstdlib` library. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means, that the program finished normally and any other value means that some error or unexpected results happened.

2.8 Functions

With the help of functions we divide program into smaller parts. Thus we can structure programs in a more modular way, accessing all the potential that structured programming can offer us in C++.

A *function* is a group of statements, that is executed when it is called from some point of the program. The format of function definition is as follows:

```
type name ( parameter1, parameter2, ...) { statements }
```

where following parameters are used:

- type* - data type specifier of the data returned by the function;
- name* - identifier by which it will be possible to call the function;
- parameters* - created as many as needed. Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas;
- statements* - function's body. It is a block of statements surrounded by braces `{ }`.

Let us create a first program with function as presented in Table 2.15. The result is presented in Table 2.16.

Let us analyze the program code. A C++ program always begins its execution by the `main` function. The first line of the `main()` function initializes integer variables `k` and `l` (`k=2`, `l=7`) and declares integer variable `m`. The second line calls function `multi()`. Being attentive we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself:

```
int multi (int x, int y);
m = multi(5,3);
```

The parameters and arguments have clear correspondence. Within the `main` function we called to function `multi` passing two values: 5 and 3, that correspond to the `int x` and `int y` parameters declared for function `multi`. Calling the function the value 5 is assigned to functions local variable `x` and the value 3 is assigned to functions local variable `y`.

At the point where the function is called from within `main`, the control is lost by `main` and passed to function `multi`. Function `multi` declares another local variable (`int r`), and by means of the expression `r=x*y`, it assigns to `r` the result of `x` multiplied by `y`, because the actual parameters passed for `x` and `y` are 5 and 3 respectively, the result is 15.

Statement `return r` finalizes the work of function `multi`, and returns the control back to the function that called it in the first place (in this case, `main`).

Table 2.15.	Multiplication function
<pre> #include <cstdlib> #include <iostream> using namespace std; int multi (int x, int y) // multiplication function { int r; r = x * y; return r; } int main(int argc, char *argv[]) { int k=2, l=7, m; m = multi(5,3); cout << "5*3=" << m << '\n'; cout << "8*4=" << multi(8,4) << "\n"; cout << "2*7=" << multi(k,l) << "\n"; m = 5 + multi(k,l); cout << "5+2*7=" << m << "\n"; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.16.	Output of program.
<pre> 5*3=15 8*4=32 2*7=14 5+2*7=19 Press the enter key to continue ... </pre>	

At this moment the program follows its regular course from the same point at which it was interrupted by the call to `multi`. In addition, because the return statement in function `multi` specified a value: the content of variable `r` (**r**eturn `r`), which at that moment had a value of 15. This value becomes the value of evaluating the function call. This value is assigned to the variable `m` in our program. The line

```
cout << "5*3=" << m << "\n";
```

outputs `m` value to the screen (`5*3=15`).

Then we call function `multi()` several more times:

```
cout << "8*4=" << multi(8,4) << '\n';
cout << "2*7=" << multi(k,l) << '\n';
m= 5 + multi(k,l);
cout << "5+2*7=" << m << '\n';
```

We have used some different calling methods. Calling function `multi()` for the second time, we do not assign the function value to a variable. We just printed the returned value. Calling function `multi()` for the third time, we used variables `k` and `l` to pass the parameters. In this case the values passed to function `multi()` are the values of `k` and `l`, that are 2 and 7 respectively. The fourth call to function `multi` is made from the expression

```
m= 5 + multi(k,l);
```

The value of function is used to evaluate the value of the whole expression $(5+2*7)$. Sometimes we need to create a function with no parameters, or the function which returns nothing. In first case we simply live the parameter list of the function empty:

```
int myfunction()
{
    return 2012;
}
```

In the second case we should use the `void` type. This is a special specifier that indicates absence of type (see Table 2.17).

It is necessary to remember that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement.

Table 2.17.

Program.

```
// Sample program with void type function
#include <cstdlib>
#include <iostream>
using namespace std;

void line() // line printout
{
    cout << "-----\n";
}

int main(int argc, char *argv[])
{
    line();
    cout << "    Hello, World\n";
    line();
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}
```

What you should know about C++ functions?

1. Do not insert one function into another. All functions are equivalent and accessible to all. Only the function `main()` is required for each program. The only difference of `main()` function is that the program begins execution from it.
2. We should define the type of function before the function description.
3. The parentheses are always necessary calling the function, even if the parameter list is empty.
4. Function parameters are declared as follows: (*type1 parameter1, type2 parameter2, ...*), i.e. types can not be grouped together, as declaring variables.
5. If we add `const` before the parameter type, it is the order to the compiler that the function can not modify the parameter value.
6. The keyword `return` transmits function return value, if it exists.
7. If the return parameter type `void`, you can not use the keyword `return`. The exception is when you need to leave the function before it is fully implemented. In this case the `return` keyword without any parameters is used.
8. Using `return` keyword we can transmit only one value from the function. If we need to return more values, we must use other mechanisms.
9. Calling a function, the types of the parameter values must be compatible.
10. Functions (or its prototype) must be defined before the first call of it. The function prototype is similar to the first line of the function. We can omit the parameter names defining the function prototype.

Form of function prototype is:

```
type name ( argument_type1, argument_type2, ... );
```

Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function `multi` using the following code:

```
int k=5, l=3, m;
m = multi(k, l);
```

What we were doing in this case was to call to function `multi()` passing the values of `k` and `l`, i.e. 5 and 3 respectively, but not the variables `k` and `l` themselves.

In this way, when the function `multi()` is called, the value of its local variables `x` and `y` become 5 and 3 respectively, but any modification to either `x` or `y` within the function `multi` will not have any effect in the values of `k` and `l` outside it, because variables `k` and `l` were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function `triple` of the following example (see Table 2.18). The output of the program is presented in Table 2.19.

The first thing that should call your attention is that in the declaration of function `triple()` the type of each parameter was followed by an ampersand sign (`&`). This ampersand is what specifies that their corresponding arguments are to be passed by reference instead of by value.

Table 2.18. Passing parameters by reference.

```

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

void triple(int& x, int& y, int& z)
{
    x *= 3; y *= 3; z *= 3;
}

int main(int argc, char *argv[])
{
    int k=1, l=3, m=7;
    triple(k, l, m);
    cout << "k=" << k
         << ", l=" << l
         << ", m=" << m << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

Table 2.19. Output of program.

```

k=3, l=9, m=21
Press the enter key to continue ...

```

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

In other words, we associate **x**, **y** and **z** with the arguments passed on the function call (**k**, **l** and **m**) and any change that we do on **x** within the function will affect the value of **k** outside it. Any change that we do on **y** will affect **l**, and the same with **z** and **m**.

That is why our program's output, that shows the values stored in **k**, **l** and **m** after the call to function `triple()`, shows the values of all the three variables of `main` tripled.

If declaring the following function `void triple(int& x, int& y, int& z)` we had declared it this way: `void triple(int x, int y, int z)`; i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of **k**, **l** and **m** without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the sum and difference for two numbers (see Table 2.20). The output of the program is presented in Table 2.21.

Table 2.20.	Passing by reference.
-------------	-----------------------

```
// Function returns more than one value
#include <cstdlib>
#include <iostream>
using namespace std;

void count(int x, int y, int& sum, int& dif)
{
    sum = x + y;
    dif = x - y;
}

int main(int argc, char *argv[])
{
    int k=9, l=5, m, n;
    count(k, l, m, n);
    cout << "k+l=" << m << "; k-l=" << n << endl;
    cout << "Press the enter key to continue ... "; cin.get();
    return EXIT_SUCCESS;
}
```

Table 2.21.	Output of program.
-------------	--------------------

```
k+l=14; k-l=4
Press the enter key to continue ...
```

Declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example, see sample program in Table 2.22.

As we can see in the body of the program there are two calls to function `divide()`. In the first one:

```
cout << divide (12) << endl;
```

we have only specified one argument, but the function `divide()` allows up to two.

So the function `divide()` has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is 6 (12/2).

In the second call:

```
cout << divide (20,4) << endl;
```

Table 2.22.

Ignored default value.

```

// Default values for function arguments
#include <cstdlib>
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
    int r;
    r = a / b;
    return r;
}

int main(int argc, char *argv[])
{
    cout << divide (12)    << endl;
    cout << divide (20,4) << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

there are two parameters, so the default value for **b** (`int b=2`) is ignored and **b** takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4). [45]

Describing the default values of the arguments, we should follow following rules.

1. If we set default value to some parameter, we have to set default values to all the parameters after it.
2. We should not set default values to the functions parameters in random order.
3. The list of functions parameters must be divided into two parts: parameters, that do not have default values, and parameters, that have the default values.
4. While calling the function, we must assign the values to all the parameters, that do not have default values.
5. We can omit parameters with default values.
6. If we omit the value of the parameter, we have to omit all the rest of the parameters after it.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. These functions are called *overloaded* functions.

For example, let us define two functions with the same name, **triangle**, but one of them accepts two parameters of type float and calculates triangle area using formula

$$S = \frac{1}{2} \cdot a \cdot h \quad (2.2)$$

Table 2.23. Overloaded functions.

```

#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

float triangle (float a, float h)
{
    return 0.5*a*h;
}
float triangle (float a, float b, float c)
{
    float p=0.5*(a+b+c);
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
int main(int argc, char *argv[])
{
    cout << triangle(7,4) << endl;
    cout << triangle(3,4,5) << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

while another one accepts tree parameters of type float and calculates triangle area using Herons formula:

$$S = \sqrt{p * (p - a) * (p - b) * (p - c)} \quad ; \quad p = 0.5 * (a + b + c) \quad (2.3)$$

The compiler knows which one to call in each case by examining the number of arguments passed when the function is called (see Table 2.23).

Notice, that a function can not be overloaded only by its return type. At least one of its parameters must have a different type.

We can use the `inline` specifier before the function definition. This specifier indicates the compiler that the inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behaviour of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in the running time. Most compilers optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

2.9 Global, local and static variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them - so called local variables.

Table 2.24.

Local variables.

```

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

void modify (int x, int y)
{
    x *= 2;  y += 5;
    cout << "x=" << x << ", y=" << y << endl;
    x++;  y++;
}

int main(int argc, char *argv[])
{
    int x=3, y=7;
    cout << "x=" << x << ", y=" << y << endl;
    modify(x,y);
    cout << "x=" << x << ", y=" << y << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

Table 2.25.

Output of program.

```

x=3, y=7
x=6, y=12
x=3, y=7
Press the enter key to continue ...

```

For example, in a sample program (see Table 2.24) we use two variables `x` and `y` in `main()` function. The same variables are used in function `modify()`. Function `main()` sets the values of its local variables `x` and `y` (`x=3, y=7`) and prints out these values. Then the `modify()` function is called. The values of the variables `x` and `y` are assigned to the functions `modify()` local parameters `x` and `y`. The names of the variables in `main()` and in `modify()` functions are the same, but these variables are different. Function `main()` has its own local `x` and `y` and `modify()` has its own `x` and `y`. Function `modify()` changes the values of the parameters and prints them out to the screen.

After calling `modify()` function, `main()` function outputs the values of `x` and `y` once more (the values are not changed, see Table 2.25).

There is the possibility to declare global variables. These variables are visible from any point of the code, inside and outside all functions. In order to declare global variables we simply have to declare the variable outside any function or block; that means, directly in the body of the program (just after `namespace` definition).

Table 2.26.	Global variables.
<pre> #include <cstdlib> #include <iostream> using namespace std; int x, y; void modify() { x *= 2; y += 5; cout << "x=" << x << ", y=" << y << endl; x++; y++; } int main(int argc, char *argv[]) { x=3; y=7; cout << "x=" << x << ", y=" << y << endl; modify(); cout << "x=" << x << ", y=" << y << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.27.	Output of program.
<pre> x=3, y=7 x=6, y=12 x=7, y=13 Press the enter key to continue ... </pre>	

Let us implement previous program using global variables - see Table 2.26. Now two global variables `x` and `y` are used. Function `main()` sets the values of variables `x` and `y` (`x=3, y=7`) and prints out these values - see Table 2.27.

Then the `modify()` function is called. Function `modify()` changes the values of the parameters and prints them out to the screen. After calling `modify()` function, the `main()` function outputs the values of `x` and `y` once more (the values are changed).

In some cases the local value of a variable must be preserved between the function calls. The static variables are used in these cases. These variables can be accessed only within the function where they were defined, but the value of the variable remains after the function execution.

The memory for static variables is reserved at the beginning of the program and they are kept there during the program running time, although they are available only in the function in which they are created.

Table 2.28.	Static variables.
<pre> #include <cstdlib> #include <iostream> using namespace std; double average(double x) { static double sk = 0; static double suma = 0; ++sk; suma += x; return suma/sk; } int main(int argc, char *argv[]) { cout << "average=" << average(1) << endl; cout << "average=" << average(2) << endl; cout << "average=" << average(4) << endl; cout << "average=" << average(10) << endl; cout << "average=" << average(11) << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.29.	Output of program.
<pre> average=1 average=1.5 average=2.33333 average=4.25 average=5.6 Press the enter key to continue ... </pre>	

After the function completes its work, static variables retain their values.

For example, let us create an average counting program using static variables (see Table 2.28). The values of the variables `sk` and `suma` are preserved between function calls and enables us to calculate the average easily. Table 2.29 shows output of program.

2.10 Arrays

An array is a series of elements of the same type placed in the contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

Like a regular variable, it is necessary to declare an array before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where *type* is a valid type (like `int`, `float`, etc), *name* is a valid identifier and the *elements* field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain. The elements of the array are numbered from 0. So, if we want to declare an array of 4 integer numbers, we can do it as follows:

```
int numbers [4];
```

NOTE: The elements field within brackets `[]` which represents the number of elements the array is going to include, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length a dynamic memory is needed.

When declaring a local array its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, are automatically initialized with their default values, that means they are filled with zeros (working with numeric data types).

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each element of the array by enclosing the values in braces `{ }`. For example:

```
int numbers [4] = { 7, 15, 22, 19 };
```

The amount of values between braces must not be larger than the number of elements that we declare for the array between square brackets `[]`.

Initializing an array, C++ makes the possibility of leaving the square brackets empty `[]`. In this case, the compiler will assume a size for the array that matches the number of values included between braces `{ }`:

```
int numbers [] = { 7, 15, 22, 19 };
```

It is possible to access the value of any of the elements of array by specifying its index. It enables us to use array elements as if it was a normal variable. The format of accessing array element is as follows:

```
name [index];
```

Also it is possible to access third element of array `numbers` like this:

```
numbers[2];
```

If we want to set the value 37 to the second element of array `numbers`, we should write:

```
numbers[1] = 37;
```

Notice that the third element of `numbers` is specified `numbers[2]`, since the first one is `numbers[0]`, the second one is `numbers[1]`, and therefore, the third one is `numbers[2]`. The last element will be `numbers[3]`. If we write `numbers[4]`, we will access the fifth element of `numbers` and therefore exceed the size of the array.

Table 2.30.

The sum of array's elements.

```

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

int main(int argc, char *argv[])
{
    int mas[7] = {5, 9, 12, 3, 7, 2, 18};
    int    sum = 0;
    for(int i=0; i<7; i++)
    {
        sum += mas[i];
    }
    cout << "sum=" << sum << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

It is important to understand that in the C++ language we do not receive notification, if the array indexes exceed the specified array length. The value will be placed in the memory, which is not reserved for it and can cause runtime errors.

Brackets `[]` perform two different tasks while working with arrays: one is to specify the size of arrays when they are declared; and the second one is to specify indexes for concrete array elements.

We can operate with arrays, working with their elements like with stand alone variables. For example, we can calculate the sum of the arrays elements (see Table 2.30), or we can find the minimal element of the array (see Table 2.31).

Array input and output are also presented in this program.

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of which are of the same uniform data type.

The way to declare bidimensional array in C++ is:

```
int matrix[4][3];
```

This will declare a matrix with 4 rows and 3 columns. To access the second element of the third row of this array, we should write:

```
matrix[2][1];
```

Multidimensional arrays are not limited to two indexes (i.e., two dimensions). They can contain as many indexes as needed. But be careful! The amount of memory needed for an array increases rapidly with each dimension.

For example:

```
char century [100][365][24][60][60];
```

Table 2.31. The minimal element of the array.

```

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

int main(int argc, char *argv[])
{
    int mas[20];
    int n,min;
    cout << "How meny elements do you want to enter?\n";
    cin >> n;
    for(int i=0; i<n; i++)          // array input
    {
        cout << "mas[" << i << "]=";
        cin >> mas[i];
    }
    cout << "Entered array:\n";
    for(int i=0; i<n; i++)          // array output
    {
        cout << mas[i] << "  ";
    }
    cout << endl;
    min = 0;
    for(int i=1; i<n; i++)          // search of minimum element
    {
        if (mas[i] < mas[min]) min = i;
    };
    cout << "Minimal array element: " << mas[min] << endl;
    cout << "Its index is: " << min << endl;
    system("PAUSE"); return EXIT_SUCCESS;
}

```

declares an array with a char element for each second in a century, that is more than 3 billion characters. So this declaration would consume more than 3 GB (gigabytes) of memory!

At some moment it is necessary to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but it is possible to pass its address. In practice that has almost the same effect and this operation is much faster and more efficient. [45]

In order to accept arrays as parameters the only thing that is necessary to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

```
void procedure (int arg[])
```

accepts a parameter of type "array of int" called **arg**. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type [] [depth] [depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray [] [3] [4])
```

Notice that the first brackets [] are left empty while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension. [45]

2.11 Character sequences

C++ Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters. And, as strings are in fact sequences of characters, they can be represented as plain arrays of char elements. The difference between char sequence and other types of arrays is that the '\0' (backslash, zero) symbol is always placed at the end of the symbol sequence. This symbol is known as a null character and is used to signal the end of the valid sequence.

For example, the following array:

```
char myword [30];
```

It can store character sequences up to 29 characters long. But it is also possible to store shorter sequences. Sample program using character sequence is presented in Table 2.32. We use character sequence to enter the name and then the program prints out greetings. Note, that runtime errors can occur if the length of entered character sequence will be bigger than specified array size.

It is possible to initiate character sequence in several forms using string literal:

```
char myword[80] = "Some text here";
char sentence[ ] = "Vilnius is the capital of Lithuania.";
```

It is possible to work with character sequences as with ordinary arrays, but also specific functions can be used to manage them. Header file `string.h` must be included to operate with string functions. To copy the contents of one character sequence into another it is necessary to call the function `strcpy (str1, str2)`. This function copies the contents of one character sequence `str2` into another `str1`:

```
char planet[50];
strcpy (planet, "Earth");
```

Unfortunately it is impossible to write sentence `planet="Earth"` working with character sequences. We can use such assignment only initializing the value.

Table 2.32.	Character sequence.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { char name[30]; cout << "Enter your name: "; cin >> name; cout << "Hello, " << name << "!" << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

It is possible to delete specified character using `strcpy()` function:

```
strcpy (&planet[2], &planet[3]);
```

This command will delete the third character (the result will be **Eath**).

It is possible to determine the length of the character sequence using `strlen()` function, for example,

```
cout << strlen(myword);
```

will print out 14 (spaces are included, but null character does not).

We can compare two character sequences using `strcmp()` function:

```
int strcmp (str1, str2);
```

This function compares character sequences and returns a negative number if the `str1` is less than the `str2`, returns 0 if the character sequences are equal, and returns a positive number if the `str1` is greater than the `str2` (numerical codes of characters are compared).

Function `strlwr(str1)` changes in capital letters into small. Function `strupr(str1)` changes small letters into capital letters.

Note, that `cin` is able to input character sequence until the first space. We can use

```
cin.getline(str1,sizeof(str1));
```

to enter whole string. We can also use `gets(str1)` for input and `puts(str1)` for output. Both these functions are inherited from C. We must include `stdio.h` header file to use these functions.

It is also possible to create string array:

```
char names [20][10];
```

This array can store up to 20 words of 9 characters. Finally, sequences of characters, stored in char arrays, can easily be converted into string objects just by using the assignment operator:

```

string mystring;
char myntcs[]="some text";
mystring = myntcs;

```

2.12 Pointers

We must not care about the physical location of our data within memory while using variables. We simply use its identifier whenever we want to refer to our variable. The memory of the computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (1B, one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number, as the previous one, plus one.

In this way, each cell can be easily located in the memory, because it has the unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776, we know, that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be. Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address, where our variable is being stored during runtime in order to operate with relative positions to it [45].

The address, that locates a variable within memory, is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example, see Table 2.33. Table 2.34 shows output of program.

Sometimes it is convenient to store the address of one variable in another variable. The variable, that stores the reference to another variable, is what we call a pointer. Pointer is a very powerful feature of the C++ language, that has many uses in advanced programming. Now we will see how this type of variable is used and declared.

Table 2.33.

Variable address.

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

int main(int argc, char *argv[])
{
    int x=7;
    cout << "x=" << x << ", &x=" << &x << endl;
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 2.34. Output of program.

```
x=7, &x=0x22ff4c
Press any key to continue ...
```

Due to the ability of a pointer to refer directly to the value, that it points to, it becomes necessary to specify in its declaration, which data type a pointer is going to point to. It is not the same thing to point to a char, as to point to an int or a float. The declaration of pointers follows this format:

```
type * name;
```

where type is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int    * number;
char   * character;
float  * greatnumber;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact, all of them are pointers and all of them occupy the same amount of space in the memory (the size in memory of a pointer depends on the platform, where the code is going to run).

Nevertheless, the data to which they point to, do not occupy the same amount of space, nor are of the same type: the first one points to an `int`, the second one to a `char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type, they point to.

Using a pointer, we can directly access the value, stored in the variable, which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

It is necessary to emphasize, that the asterisk sign (*), that we use when declaring a pointer, means only that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator which is also written with an asterisk (*). They are simply two different things, represented with the same sign.

Let us analyze the code in Table 2.35. Table 2.36 shows output of program. In this program we initialize the variable `x` of type `int` as (`int x=7`) and declare integer type pointer `p` as (`int *p`). Then the address of variable `x` is saved in `p` as (`p=&x`). Finally, we perform the output. Note, that `x` and `*p` outputs the same value, and also `&x` and `p` are the same. We can also find the address of pointer itself (`&p`).

It is important to clarify the difference between the address (`p`) and the value, stored in the memory under this address (`*p`).

Notice the difference between the reference and dereference operators: `&` is the reference operator and can be read as "address of", `*` is the dereference operator and can be read as "value pointed by".

Thus, they have complementary (or opposite) meanings. A variable referenced with `&` can be dereferenced with `*`. Since we have got the address of the variable, we can use pointer to access it.

```
int  x;
int *p;
    p = &x;
    *p = 12;
    *p-= 7;
```

Table 2.35.	Pointer.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x=7; int *p; p = &x; cout << "x=\t" << x << ", &x=\t" << &x << endl; cout << "*p=\t" << *p << ", p=\t" << p << ", &p=\t" << &p << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.36.	Output of program.
<pre> x= 7, &x= 0x22ff4c *p= 7, p= 0x22ff4c, &p= 0x22ff48 Press any key to continue ... </pre>	

The meaning of an expression which used the dereference operator (*) is different from one, that does not. When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so, in fact, they are of the same concept.

For example, supposing these two declarations:

```
int numbers [20];
int *p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a constant pointer. We can use pointer `p` to access the elements of `numbers` array:

```
numbers[index] == *(numbers + index) == *(p + index)
&numbers[index] == (numbers + index) == (p + index)
```

Working with multidimensional arrays we can use such dependences:

```
int matrix[n][m];
matrix[index1][index2] == (*(matrix + index1) + index2)
&matrix[index1][index2] == *(matrix + index1) + index2
```

Only addition and subtraction operations are allowed to be conducted with pointers, the others make no sense in the world of pointers. But both addition and subtraction have a different behaviour with pointers according to the size of the data type to which they point.

Some data types occupy more memory, others require less memory space. For example, let's assume that in a given compiler for a specific machine, `char` takes 1 byte, `short` takes 2 bytes and `long` takes 4 bytes.

Suppose, that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively. So if we write:

```
mychar++;
myshort++;
mylong++;
```

variable `mychar`, as you may expect, would contain the value 1001. But not so obviously, `myshort` would contain the value 2002, and `mylong` would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer. [45]

This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Both the increase (`++`) and decrease (`--`) operators have greater operator precedence than the dereference operator (`*`), but both have a special behaviour when used as suffix (the expression is evaluated with the value, it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++ ;
```

Because `++` has greater precedence than `*`, this expression is equivalent to `*(p++)`. Therefore, what it does is to increase the value of `p` (so it now points to the next element), but because `++` is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased). Notice the difference with:

```
(*p)++ ;
```

Table 2.37.	Variable address.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x[] = {7, 9, 13, 19, 8, 2, 11}; int *p; p = x; for(int i=0; i<7; i++, p++) cout << *p << '\t'; cout << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.38.				Output of program.		
7	9	13	19	8	2	11
Press any key to continue ...						

Here, the expression would have been evaluated as the value pointed by `p` increased by one. The value of `p` (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer). [45]

It is very convenient to use pointer arithmetic with arrays - see Table 2.37, output - Table 2.38. Increase operator enables us to address all the elements of the array.

We can use addresses and pointers working with functions. We have already discussed the possibility of passing parameters by value or by reference. Let us review the material adding the pointers. Table 2.39 represents three different function styles. Table 2.40 shows output of program.

We will create three different functions `int cube1()`, `void cube2()`, `void cube3()` for calculating cube of a number according to simple arithmetic expression:

$$y = x^3 \quad (2.4)$$

Function `int cube1(int x)` is presented in traditional function style, one parameter passed by value, result will be received using `return` statement. Function `void cube2(int &x)` is written in reference style: parameter passed by reference, and result will be returned through the function parameter. Function `void cube3(int *x)` is written in pointer style: parameter passed using pointer, and result will be returned using pointer. Reference style is the most preferable style in C++ programming, in comparison to pointer style which is typical in C programming.

Table 2.39.	fun.cpp
<pre> #include <cstdlib> #include <iostream> using namespace std; int cube1 (int x); void cube2 (int &x); void cube3 (int *x); int main(int argc, char *argv[]) { // It is possible to pass // a number cube1(2) int a = 2; cout << a << " "; cout << cube1(a) << endl; // working with cube2 // we must pass a variable a = 3; cout << a << " "; cube2(a); cout << a << endl; // working with cube3 // we must pass an address a = 4; cout << a << " "; cube3(&a); cout << a << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	<pre> int cube1(int x) { // parameter passed by value int y = x*x*x; return y; // return to get the result } void cube2(int &x) { // parameter passed by reference int y = x*x*x; x = y; // return the result // through the function parameter } void cube3(int *x) { // parameter passed using pointer int y = *x * *x * *x; *x = y; // return the result // using pointer } </pre>

Table 2.40.	Output of program fun.cpp.
<pre> 2 8 3 27 4 64 </pre>	

2.13 Data structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not only sequences of elements all of the same data type, but sets of different elements with different data types.

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```

struct structure_name
{
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    ...
}    object_names;

```

where `structure_name` is a name for the structure type, `object_names` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{ }` there is a list with the data members, each one is specified with a type and a valid identifier as its name. [45]

The first thing we have to know, is that a data structure creates a new type. Once a data structure is declared, a new type with the identifier specified as `structure_name` is created and can be used in the rest of the program, as if it was any other type. For example:

```

struct product
{
    int weight;
    float price;
};
product apple;
product banana, melon;

```

We have first declared a structure type called `product` with two members: `weight` and `price`, each of a different fundamental type. We have then used this name of the structure type (`product`) to declare three objects of that type: `apple`, `banana` and `melon`, as we would have done with any fundamental data type.

Once declared, `product` has become a new valid type name like the fundamental ones `int`, `char` or `short` and from that point on we are able to declare objects (variables) of this compound new type, like we have done with `apple`, `banana` and `melon`.

Right at the end of the `struct` declaration and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `banana` and `melon` at the moment we define the data structure type this way:

```

struct product
{
    int weight;
    float price;
}
    apple, banana, melon;

```

It is important to differentiate clearly between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like `apple`, `banana` and `melon`) from a single structure type `product`).

Once we have declared our three objects of a determined structure type (`apple`, `banana` and `melon`) we can operate directly with their members. To do that, we use a dot (`.`) inserted between the object name and the member name. For example, we could operate with any of these elements, as if they were standard variables of their respective types:

Table 2.41.	Structure sample program.
<pre>#include <cstdlib> #include <iostream> #include <string.h> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { struct product { char name[100]; float price; int amount; } p1, p2; strcpy(p1.name,"CPU"); p1.price = 350.20; p1.amount = 70; cout << "Enter the name of the product: "; gets (p2.name); cout << "Enter the price of the product: "; cin >> p2.price; cout << "Enter the amount of the product: "; cin >> p2.amount; cout << "Available products:\n"; printf("%s\t%7.2f\t%d\n", p1.name, p1.price, p1.amount); printf("%s\t%7.2f\t%d\n", p2.name, p2.price, p2.amount); system("PAUSE"); return EXIT_SUCCESS; }</pre>	

Table 2.42.	Output of program.
<pre>Enter the name of the product: HDD Enter the price of the product: 215.81 Enter the amount of the product: 150 Available products: CPU 350.20 70 HDD 215.80 150 Press any key to continue ...</pre>	

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Each one of these has the data type, corresponding to the member they refer to variables: `apple.weight`, `banana.weight` and `melon.weight` are of type `int`, while `apple.price`, `banana.price` and `melon.price` are of type `float`.

Let's see a real example where you can see, how a structure type can be used in the same way as fundamental types (see Table 2.41). In this program we use `gets()` to input a string and `printf()` to perform the output. Table 2.42 shows output of program.

Data structures can be used to represent databases, especially if we consider the possibility of building arrays of them (see Table 2.43). Like any other type, structures can be pointed by its own type of pointers:

Table 2.43.	Structure array.
<pre> #include <cstdlib> #include <iostream> #include <string.h> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { struct product { char name[100]; float price; int amount; } p[100]; int n; cout << "Enter the amount of different products" << " you would like to specify:\n"; cin >> n; for(int i=0; i<n; i++) { cout << "Enter the name of the product: "; cin >> p[i].name; cout << "Enter the price of the product: "; cin >> p[i].price; cout << "Enter the amount of the product: "; cin >> p[i].amount; }; cout << "Available products:\n"; for(int i=0; i<n; i++) { printf("%s\t%7.2f\t%d\n", p[i].name, p[i].price, p[i].amount); }; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.44. Combinations of pointers and structure members.		
Expression	What is evaluated	Equivalent
<code>a.b</code>	Member <code>b</code> of object <code>a</code>	
<code>a->b</code>	Member <code>b</code> of object pointed by <code>a</code>	<code>(*a).b</code>
<code>*a.b</code>	Value pointed by member <code>b</code> of object <code>a</code>	<code>*(a.b)</code>

```

struct product
{
    char name[100];
    float price;
    int amount;
}

p1, *p;
p = &p1;
p -> price = 150;

```

Here `p1` is an object of structure type `product`, and `p` is a pointer to point to objects of structure type `product`. The previous code includes an important introduction: the arrow operator (`->`). This is a dereference operator, that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference.

In the example we used: `p->price` which is equivalent to: `(*p).price`. Both expressions are valid and both mean, that we are evaluating the member `price` of the data structure pointed by a pointer, called `p`. Table 2.44 summarizes possible combinations of pointers and structure members.

Structures can also be nested so that a valid element of a structure can also be in its turn another structure. It is possible to create connected list of structures using pointers and dynamic memory allocation as presented in Tables 2.45a, 2.45b.

Table 2.45a.	List of structures using pointers.
--------------	------------------------------------

```

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

struct person
{
    char    name[50];
    int     age;
    person *next;
    // We will use this pointer to access the next person
};

```

Table 2.45b.

List of structures using pointers.

```

int main(int argc, char *argv[])
{
    person  *p1 = NULL,
            *p2 = NULL;
    int      n = 0;
    char t;
    cout << "Would you like to enter a person? (y/n)\n";    cin >> t;
    while((t=='y')||(t=='Y'))
    {
        if (p1==NULL)
        {
            // dynamic memory allocation for pointer p1
            p1 = (person *)malloc(sizeof(person));
            p2 = p1;
        }
        else
        {
            p2 -> next = (person *)malloc(sizeof(person));
            // dynamic memory allocation for pointer p2
            p2 = p2 -> next;
        };
        // function malloc allocates the requested size of memory
        // we use type casting to convert general pointer
        // returned by malloc to specified type of pointer
        cout << "Enter the person's name: ";    cin >> p2 -> name;
        cout << "Enter the person's age: ";    cin >> p2 -> age;
        p2 -> next = NULL;
        n++;
        cout << "Would you like to enter a person? (y/n)\n"; cin >> t;
    }
    cout << n << " persons are entered.\n";
    cout << "The list of persons:\n";
    p2 = p1; // we will start from the beginning of the list
    while(p2 != NULL)
    {
        cout << p2 -> name << '\t' << p2 -> age << endl;
        p2 = p2 -> next; //we move to the next person
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

2.14 Classes

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name
{
    access_specifier_1:  member1;
    access_specifier_2:  member2;
    ...
}
    object_names;
```

where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All of this is very similar to the declaration on data structures. But in this case we can include also functions and members. A new thing is called access specifier. An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights, that the members following them acquire:

- i) `private` members of a class are accessible only from within other members of the same class or from their friends;
- ii) `protected` members are accessible from members of the same class and from their friends, but also from members of their derived classes;
- iii) finally, `public` members are accessible from anywhere, where the object is visible.

By default, all members of a class declared with the `class` keyword have `private` access for all its members. Therefore, any member, that is declared before one other class specifier automatically has `private` access. For example:

```
class CRectangle
{
    int x, y;
public:
    void set_values (int,int);
    int area (void);
}
    rect;
```

Declares a class (i.e., a type) called `CRectangle` and an object (i.e., a variable) of this class called `rect`. This class contains four members: two data members of type `int` (member `x` and member `y`) with `private` access (because `private` is the default access level) and two member functions with `public` access: `set_values()` and `area()`, of which for now we have only included their declaration, not their definition.

After the previous declarations of `CRectangle` and `rect`, we can refer within the body of the program to any of the `public` members of the object `rect` as if they were normal functions

or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we were doing with plain data structures before. For example:

```
rect.set_values (2,5);
S = rect.area();
```

The only members of `rect` that we cannot access from the body of our program outside the class are `x` and `y`, since they have private access and they can only be referred from within other members of that same class.

The complete example of class `CRectangle` is presented in Table 2.46.

The most important new thing in this code is the operator of scope (`::`, two colons) included in the definition of `set_values()`. It is used to define a member of a class from outside the class definition itself.

You may notice, that the definition of the member function `area()` has been included directly within the definition of the `CRectangle` class because of its extreme simplicity, whereas `set_values()` has only its prototype declared within the class, but its definition is outside it. In this outside definition, we must use the operator of scope (`::`) to specify, that we are defining a function, that is a member of the class `CRectangle` and not a regular global function.

The scope operator (`::`) specifies the class which member being declared belongs to, granting exactly the same scope properties, as if this function definition was directly included within the class definition. For example, in the function `set_values()` of the previous code, we have been able to use the variables `x` and `y`, which are private members of class `CRectangle`, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second case it will be a normal (not-inline) class member function, which, in fact, supposes no difference in behaviour.

Table 2.46.

Class `CRectangle`.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: void set_values (int,int); int area () { return (x*y); } };</pre>	<pre>void CRectangle::set_values (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect; rect.set_values (3,4); cout << "area: " << rect.area(); system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	--

Members `x` and `y` have private access (remember, that if nothing else is said, all members of a class defined with keyword `class` have private access). By declaring them private, we deny access to them from anywhere outside the class.

This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. It is difficult to see any utility in protecting these two variables in such simple example as given, but in greater projects it may be very important, that values cannot be modified in unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. Each object has its own variables and, in some way, has also its own function members, that use its object's own variables to operate.

That is the basic concept of object-oriented programming, OOP. Data and functions are both members of the object. We no longer use sets of global variables, that we pass from one function to another as parameters, but, instead, we handle objects, that have their own data and functions embedded as members. Notice, that we have not had to give any parameters to `rect.area()`. The member function directly used the data members of the object `rect`.

Objects often need to initialize variables or assign dynamic memory during the process of their creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function `area()` before having called function `set_values()`? Probably, we would have gotten an undetermined result since the members `x` and `y` would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and can not have any return type, not even `void`.

We are going to implement `CRectangle` including a constructor (see Table 2.47). As you can see, the result of this example is identical to the previous one. But now we have removed the member function `set_values()`, and have included instead a constructor, that performs a similar action: it initializes the values of `x` and `y` with the parameters, that are passed to it.

Table 2.47.	Class CRectangle.
<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: CRectangle (int,int); int area () { return (x*y);} };</pre>	<pre>CRectangle::CRectangle (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect (3,4); cout << "rect area: " << rect.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>

Notice, that these arguments are passed to the constructor at the moment, at which the objects of this class are created:

```
CRectangle rect (3,4);
```

Constructors can not be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how, neither the constructor prototype declaration (within the class) nor the latter constructor definition, includes a return value; not even void.

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator `delete`.

The destructor must have the same name as the class, but preceded with a tilde sign (`~`) and it must also return no value. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

Like any other function, a constructor can also be overloaded with more than one function that has the same name, but different types or number of parameters.

Remember, that for overloaded functions the compiler will call the one, whose parameters match the arguments, used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one, that matches the arguments passed on the object declaration (see Table 2.48).

Table 2.48.

Class CRectangle.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: CRectangle (); CRectangle (int,int); int area () {return (x*y);} }; CRectangle::CRectangle () { x = 1; y = 1; }</pre>	<pre>CRectangle::CRectangle (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect1 (3,4); CRectangle rect2; cout << "rect1 area: " << rect1.area() << endl; cout << "rect2 area: " << rect2.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	---

In this case, `rect2` was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both `x` and `y` with a value of 1.

Important: Notice, if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses (`()`):

```
CRectangle rect2;    // right
CRectangle rect2();  // wrong!
```

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So, you have to declare all objects of that class according to the constructor prototypes, you have defined for the class.

But the compiler not only creates a default constructor if you do not specify your own. It provides three special member functions in total, that are implicitly declared if you do not declare your own. These are the copy constructor, the copy assignment operator, and the default destructor. The copy constructor and the copy assignment operator copy all the data, contained in another object to the data members of the current object.

It is perfectly valid to create pointers, that point to classes. We simply have to consider, that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer.

For example:

```
CRectangle *prect;
```

is a pointer to an object of class `CRectangle`. As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (`->`) of indirection. Next, you have a summary of how can you read some pointer and class operators (`*`, `&`, `.`, `->`, `[ ]`) that can be used with classes as presented in Table 2.49. Expression can be read as combinations of pointer and class operators.

Table 2.49. Combinations of pointer and class operators.	
Expression	Can be read as
<code>*x</code>	pointed by <code>x</code>
<code>&x</code>	address of <code>x</code>
<code>x.y</code>	member <code>y</code> of object <code>x</code>
<code>x->y</code>	member <code>y</code> of object pointed by <code>x</code>
<code>(*x).y</code>	member <code>y</code> of object pointed by <code>x</code>
<code>x[0]</code>	first object pointed by <code>x</code>
<code>x[1]</code>	second object pointed by <code>x</code>
<code>x[n]</code>	(n+1)th object pointed by <code>x</code>

2.15 Friendship and inheritance

Private and protected members of a class, usually, cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends. Friends are functions or classes declared with the **friend** keyword.

Declaring an external function as friend of a class, allows this function to have an access to the private and protected members of this class. That is done by declaring a prototype of this external function within the class, and preceding it with the keyword **friend** (see Table 2.50).

The **sum()** and **dif()** functions are **friends** of **complex** class. From within these functions we have been able to access the members **a** and **b** of different objects of type **complex**, which are private members. Notice that neither in the declaration of **sum()** nor in its later use in function **main()** it is considered **sum** a member of class **complex**. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can be used, for example, to conduct operations between two different classes. In general, the use of friend functions is out of an object-oriented programming methodology, so it is better to use members of the same class to perform operations with them whenever it is possible.

Just as there is the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

Consider that friendship is not corresponded if we do not explicitly specify it. For example, if first class is considered as a friend class by second one, but the first class does not consider the second one to be a friend, so the first class can access the protected and private members of the second one but not the reverse way.

Another property of friendship is that it is not transitive: The friend of a friend is not considered to be a friend unless is explicitly specified.

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own ones. For example, we suppose that we want to declare a series of classes that describe different geometrical objects (circle, cylinder). They have certain common properties, such as radius (see Table 2.51).

This could be represented in the world of classes with a class **circle** from which we would derive the **cylinder** class.

Classes that are derived from the others inherit all the accessible members of the base class. That means that if the base class includes the member **A** and we derive it to another class with another member called **B**, the derived class will contain both members **A** and **B**.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

where identifier **derived_class_name** is the name of the derived class and identifier **base_class_name** is the name of the class on which it is based.

Table 2.50. Friend functions.

```

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

class complex
{
    int a, b;
public:
    complex() {a=1; b=1;}
    complex (float x, float y) {a=x; b=y;}
    void set (float x, float y) {a=x; b=y;}
    void print() {cout << a << " + " << b << "*i";}
    friend complex sum (complex, complex);
    friend complex dif (complex, complex);
};

complex sum (complex k, complex l)
{
    complex m;
    m.a = k.a + l.a;
    m.b = k.b + l.b;
    return m;
}

complex dif (complex k, complex l)
{
    complex m;
    m.a = k.a - l.a;
    m.b = k.b - l.b;
    return m;
}

int main(int argc, char *argv[])
{
    complex z1, z2, z3;
    z1.set(7,9);
    z2.set(3,6);
    cout << "z1= ";    z1.print();  cout << endl;
    cout << "z2= ";    z2.print();  cout << endl;
    z3=sum(z1,z2);
    cout << "z1+z2= ";  z3.print();  cout << endl;
    z3=dif(z1,z2);
    cout << "z1-z2= ";  z3.print();  cout << endl;
    system("PAUSE");  return EXIT_SUCCESS;
}

```

The `public` access specifier may be replaced by any one of the other access specifiers - `protected` and `private`. This access specifier limits the most accessible level for the members inherited from the base class.

The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

The objects of the class `cylinder` contain members inherited from `circle`. The `protected` access specifier is similar to `private`. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members. Since we wanted `radius` to be accessible from members of the derived class `cylinder` and not only by members of `circle`, we have used `protected` access instead of `private`.

Table 2.51.

Derived classes.

```

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

class circle
{
protected:
    float r;
public:
    void set(float x) {r=x;}
    float area() { return 3.14*r*r;}
};

class cylinder:protected circle
{
    float h;
public:
    void set(float x, float y) {r=x; h=y;}
    float volume() {return 3.14*r*r*h;}
};

int main(int argc, char *argv[])
{
    circle c;
    c.set(5);
    cout << "area= " << c.area() << endl;
    cylinder cc;
    cc.set(5,3);
    cout << "volume= " << cc.volume() << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.52. Different access types.

Access	Public	Protected	Private
Members of the same class	yes	yes	yes
Members of derived classes	yes	yes	no
Not members	yes	no	no

We can summarize the different access types according to who can access them in the following way as presented in Table 2.52.

”Not members” represent any access from outside the class, such as from `main()`, from another class or from a function. For example see Table 2.53.

A derived class inherits every member of a base class except:

- i) its constructor and its destructor;
- ii) its operator `=()` members;
- iii) its friends.

Although the constructors and destructors of the base class are not inherited by themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

Table 2.53. Derived classes and constructors.

```

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

class circle
{
protected:
    float r;
public:
    circle() {r=1;}
    circle(float x) {r=x;}
    void set(float x) {r=x;}
    float area() {return 3.14*r*r;}
};

class cylinder:protected circle
{
    float h;
public:
    cylinder():circle() {h=1;}
    cylinder (float x, float y) : circle(x) {h=y;}
    void set (float x, float y) {r=x; h=y;}
    float volume() {return 3.1415926 * r * r * h;}
};

int main(int argc, char *argv[])
{
    circle c(5);
    // c.set(5);
    cout << "area= "
         << c.area() << endl;
    cylinder cc(5,3);
    // cc.set(5,3);
    cout << "volume= "
         << cc.volume() << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    derived_constructor_name (parameters) :
        base_constructor_name (parameters) {...}
}
```

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by separating the different base classes with commas in the derived class declaration.

2.16 Polymorphism

One of the key features of derived classes is that a pointer to a derived class is a type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings *Object Oriented Methodologies* to its full potential.

We are going to start by writing a program about the rectangle and the triangle (see Table 2.54).

Table 2.54.

Pointers to base class.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CPolygon { protected: int width, height; public: void set_values (int a, int b) { width=a; height=b; } }; class CRectangle: public CPolygon { public: int area () { return (width * height); } };</pre>	<pre>class CTriangle: public CPolygon { public: int area () { return (width * height / 2); } }; int main(int argc, char *argv[]) { CRectangle rect; CTriangle trgl; CPolygon *ppoly1 = &rect; CPolygon *ppoly2 = &trgl; ppoly1 -> set_values (4,5); ppoly2 -> set_values (4,5); cout << rect.area() << endl; cout << trgl.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	---

Table 2.55.	Virtual members.
-------------	------------------

```

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

class circle
{
protected:
    float r;
public:
    circle() {r=1;}
    circle(float x) {r=x;}
    void set(float x) {r=x;}
    virtual float area() {return 3.14*r*r;}
};

class cylinder:public circle
{
    float h;
public:
    cylinder ():circle() {h=1;}
    cylinder (float x, float y):circle(x) {h=y;}
    void set (float x, float y) {r=x; h=y;}
    float area()    {return 2 * 3.14 * r * h;}
    float volume() {return      3.14 * r * r * h;}
};

int main(int argc, char *argv[])
{
    circle c(5);
    circle *pp;
    pp=&c;
    cout << "area= " << pp->area() << endl;
    cylinder cc(5,3);
    pp=&cc;
    cout << "cylinder side area= " << pp->area() << endl;
    system("PAUSE");    return EXIT_SUCCESS;
}

```

Table 2.56.	Output of program.
-------------	--------------------

```

area= 78.5
cylinder side area= 94.2
Press any key to continue ...

```

In function `main()`, we create two pointers that point to objects of class `CPolygon` (`ppoly1` and `ppoly2`). Then we assign references to `rect` and `trgl` to these pointers, and as both are objects of the classes derived from `CPolygon`, both are valid assignment operations.

The only limitation in using `*ppoly1` and `*ppoly2` instead of `rect` and `trgl` is that both `*ppoly1` and `*ppoly2` are of type `CPolygon*` and therefore we can only use these pointers to refer to the members that `CRectangle` and `CTriangle` inherit from `CPolygon`. For that reason when we call the `area()` members at the end of the program we have had to use directly the objects `rect` and `trgl` instead of the pointers `*ppoly1` and `*ppoly2`. [45]

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword `virtual`. For example see Table 2.55. Table 2.56 represents output of program.

Two classes (`circle` and `cylinder`) have the same members: `r`, `set()` and `area()`.

The member function `area()` has been declared as virtual in the base class because it is later redefined in derived class. You can verify if you want that if you remove this virtual keyword from the declaration of `area()` within `circle()`, and then you run the program and the result will be 78.5 instead of 94.2. That happens because instead of calling the corresponding `area()` function for each object (`circle::area()`, `cylinder::area()`). Function `circle::area()` will be called in all cases since the calls are via a pointer whose type is `circle *`.

Therefore, the `virtual` keyword allows a member of a derived class with the same name as one in the base class to be appropriately called from a pointer and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the example above.

A class, that declares or inherits a virtual function, is called a polymorphic class.

2.17 Input/Output with files

C++ provides the following classes to perform input and output (i/o) of characters to/from files:

- i) `ofstream`: stream class to write to files;
- ii) `ifstream`: stream class to read from files;
- iii) `fstream`: stream class to read from and write to files both.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects of these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes, that are related to our file streams.

And, in fact, we can use our file streams in the same way we already used to use `cin` and `cout`, with the only difference, that we have to associate these streams with physical files. Let's see the example (Table 2.57).

This code creates a file called `example.txt` and inserts a sentence into it in the same way we used to do with `cout`, but using the file stream `myfile` instead.

We can also use input/output functions inherited from C. Let us observe the work with text files.

The first operation generally performed on an object of one of these classes is to associate it to a real file.

This procedure is known as to open a file.

Table 2.57. Basic file operations.

```

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

int main(int argc, char *argv[])
{
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    system("PAUSE"); return EXIT_SUCCESS;
}

```

An open file is represented within the program by a stream object (an instantiation of one of these classes, in the previous example it was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

<code>ios::in</code>	open for input operations.
<code>ios::out</code>	open for output operations.
<code>ios::binary</code>	open in binary mode.
<code>ios::ate</code>	set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
<code>ios::app</code>	all output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
<code>ios::trunc</code>	if the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator `OR (|)`. For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```

ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);

```

Each one of the `open()` member functions of the classes devoted to file operation `ofstream`, `ifstream` and `fstream` has a default mode, that is used if the file is opened without a second argument as presented in Table 2.58.

Table 2.58. Default mode parameters.

Class	Default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function. The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter, the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

As the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms used to open a file are valid and equivalent. If you want to check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a bool value of `TRUE` in the case that indeed the stream object is associated with an open file, or `FALSE` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

When we finished input and output operations on the file we must close it, so that its resources become available again. In order to do that, we have to call the stream's member function `close()`. This member function takes no parameters and its function is to flush the associated buffers and to close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes. In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`. Text file streams are those where we do not include the `ios::binary` flag in their opening mode.

These files are designed to store text and thus all values that we input from or output to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value. Data output operations on text files can be performed in the same way we operated with `cout` (see Table 2.59).

Data input from a file can also be performed in the same way that we did with `cin` (see Table 2.60).

Table 2.59. Writing to a text file.

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

int main(int argc, char *argv[])
{
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n" << "This is another line.\n";
        myfile.close();
    }
    else
        cout << "Unable to open file";
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 2.60. Reading from a text file.

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

int main(int argc, char *argv[])
{
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( myfile.good() )
        {
            getline (myfile,line);
            cout << line << endl;
        };
        myfile.close();
    }
    else
        cout << "Unable to open file";
    system("PAUSE"); return EXIT_SUCCESS;
}
```

This last example reads a text file and prints out its content on the screen. Notice, how we have used a new member function, called `good()`, that returns `TRUE` in the case that the stream is ready for input/output operations. We have created a while loop that finishes, when indeed `myfile.good()` is no longer `TRUE`, which will happen either if the end of the file has been reached or if some other error occurred.

In addition to `good()`, which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a `bool` value):

- `bad()` - returns `TRUE` if a reading or writing operation fails. For example, in the case when we are trying to write to a file that is not open for writing or if the device, where we are trying to write, has no space left.
- `fail()` - returns `TRUE` in the same cases as `bad()`, but also in the case when a format error happens, like when an alphabetical character is extracted, when we are trying to read an integer number.
- `eof()` - returns `TRUE` if a file open for reading has reached the end.
- `good()` - this is the most generic state flag: it returns `FALSE` in the same cases in which calling any of the previous functions would return `TRUE`.

In order to reset the state flags checked by any of these member functions, we have just seen, we can use the member function `clear()`, which takes no parameters.

All i/o streams objects have, at least, one internal stream pointer: `istream`, like `istream`, has a pointer, known as the `get()` pointer, that points to the element to be read in the next input operation.

Objects of `ofstream`, like `ostream`, has a pointer, known as the `put()` pointer, that points to the location, where the next element has to be written.

Finally, objects of `fstream` inherits both, the `get()` and the `put()` pointers, from `iostream` (which is itself derived from both `istream` and `ostream`).

These internal stream pointers, that point to the reading or writing locations within a stream, can be manipulated using the following member functions:

- `tellg()` and `tellp()` - these two member functions have no parameters and return a value of the member type `pos_type`, which is an integer data type representing the current position of the `get` stream pointer (in the case of `tellg`) or the `put` stream pointer (in the case of `tellp`);
- `seekg()` and `seekp()` - these functions allow to change the position of the `get` and `put` stream pointers.

Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions `tellg` and `tellp`: the member type `pos_type`, which is an integer value. The other prototype for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the position of the `get` or `put` pointer is set to an offset value relative to some specific point determined by the parameter `direction`. Offset is of the member type

`off_type`, which is also an integer type; and `direction` is of type `seekdir`, which is an enumerated type (`enum`) that determines the point from where offset is counted from and that can take any of the following values:

- `ios::beg` - offset counted from the beginning of the stream,
- `ios::cur` - offset counted from the current position of the stream pointer,
- `ios::end` - offset counted from the end of the stream.

The following example uses the member functions we have just seen to obtain the size of a file (see Table 2.61).

It is also possible to operate with different portions of data while working with files. Let us create a sample program, which rewrites a matrix from one file to another. We will perform this task in three different ways: operating by characters (see Table 2.62), by strings (see Table 2.63) and by numbers (see Table 2.64).

It is not so easy to obtain the end of line working with numbers. That is why we use for loops to read the data. As it was mentioned earlier, we can also use file input/output function inherited from C. We must include header files `stdio.h` and `conio.h` to use these functions. The process is rather similar. First of all we define a pointer to a file and then open it.

```
FILE * pointertofile;
pointertofile = fopen (<file name>, <file use TYPE>);
```

After the assignment of the file name, we need to define file usage type one of the following:

- "r" - open for reading, the file has to be created;
- "w" - open for writing, and, if the file already exists, its contents will be cleaned, rewritten;
- "a" - opening file for appending, data will be added at the end of the file, if the file does not exists, it will be created;
- "r+" - open file for reading or writing, the file has to be created;

Table 2.61	Obtaining file size.
------------	----------------------

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

int main(int argc, char *argv[])
{
    long begin,end;
    ifstream myfile ("example.txt");
    begin = myfile.tellg();
    myfile.seekg (0, ios::end);
    end = myfile.tellg();
    myfile.close();
    cout << "size is: " << (end-begin) << " bytes.\n";
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 2.62.

Rewriting the matrix
while working with characters.

```

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

int main(int argc, char *argv[])
{
    char s;
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    while(f.get(s))              // we read till end of file
        g.put(s);                // we write characters to the second file
    f.close();                   // first file is closed
    g.close();                   // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.63.

Rewriting the matrix
while working with strings.

```

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

int main(int argc, char *argv[])
{
    char s[100];
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    while(f.getline(s,100)!=NULL) // we read till end of file
        g << s << endl;        // we write strings to the second file
    f.close();                   // first file is closed
    g.close();                   // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.64.

 Rewriting the matrix
while working with numbers.

```

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

int main(int argc, char *argv[])
{
    int s;
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            f >> s;              // we read numbers
            g << s << " ";        // we write numbers to the second file
        };
        g << endl;
    };
    f.close();                  // first file is closed
    g.close();                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

"w+" - open file for reading or writing, and if the file already exists, its contents will be rewritten;

"a+" - open for reading and appending, if the file does not exist, it will be created.

We should not forget to close file using `fclose()` function. Let us observe the same three programs with matrix copying while working with these functions. We will perform this task in three different ways: operating by characters (see Table 2.65), by strings (see Table 2.66) and by numbers (see Table 2.67).

Table 2.65.

Rewriting the matrix
while working with characters using C functions.

```

#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

int main(int argc, char *argv[])
{
    char s;
    FILE *f, *g;                // the pointers are declared
    f=fopen("data.txt","r");    // we open the first file for reading
    g=fopen("result.txt","w");  // we open the second file for writing
    while((s=getc(f))!=EOF)     // we read characters
        putc(s,g);             // we write characters to the second file
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.66.

Rewriting the matrix
while working with strings using C functions.

```

#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

int main(int argc, char *argv[])
{
    char s[100];
    FILE *f, *g;                // the pointers are declared
    f=fopen("data.txt","r");    // we open the first file for reading
    g=fopen("result.txt","w");  // we open the second file for writing
    while(fgets(s,100,f)!=NULL) // we read strings
        fputs(s,g);            // we write strings to the second file
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.67.

Rewriting the matrix
while working with numbers using C functions.

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

int main(int argc, char *argv[])
{
    int s;
    FILE *f, *g;                // the pointers are declared
    f = fopen("data.txt","r");   // we open the first file for reading
    g = fopen("result.txt","w"); // we open the second file for writing
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            fscanf (f,"%d",&s); // we read numbers
            fprintf(g,"%d ",s); // we write numbers to the second file
        };
        fprintf(g,"\n");
    };
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}
```
