FITPED

C/C++ fundamentals

Jiří Rybička
Viera Michaličková
Juan Carlos Rodríguez-del-Pino
José Daniel González-Domínguez
Zenón José Hernández-Figueroa
Małgorzata Przybyła-Kasperek

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

C/C++ Fundamentals

Published on

November 2021

Authors

Jiří Rybička | Mendel University in Brno, Czech Republic

Viera Michaličková | Constantine the Philosopher University in Nitra, Slovakia

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Reviewers

Anna Stolińska | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1779-8

Table of Contents

Main Function, Header Files, Standard Library, Simple Input/Output

Chapter 1

1.1 First program, main function

📖 1.1.1

When you want to write your program in C you have to write the so-called "main" function.

Functions in C/C++ are pieces of code that have their own name (indentificator) and body. The main function represents the whole program, so it must be always written in source code.

The general form of function is:

```
data_type function_name (parameters) { body }
```

So the main function may have the following form:

```
int main() {
    // here are statements of function body
   return 0;  // so called return value of function
}
```

The main function returns an integer value, thus the data type of function main is int.

📝 1.1.2

The main function represents _____ of C/C++-program. Its declaration consists of _____ type, identifier "main", parameters and body. The body of the main function contains a statement for _____.

- clear screen
- alternative part
- int
- whole program
- string
- integer returned value
- clear input buffer

📝 1.1.3

The body of the main function is enclosed by parenthesis.

- False
- True

1.1.4

Parameters of the main function are enclosed by:

- parenthesis.
- curly brackets.
- spaces.

1.1.5

A header file is a file with extension .h which contains declarations and definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

The use of a header file is necessary for almost every program. One of the most useful header files which come with the compiler (or operating system) is a file with the declaration of input/output operations.

Your request to use a header file name.h in your program by including it with the C preprocessing directive **#include**. The syntax of #include has the following two forms:

```
#include <name>
```

This variant is for the system (compiler) files, and

```
#include "name.h"
```

for files written by the programmer.

Inserting a custom file can also include the file access path, while the header files are located in system directories whose paths are not specified (they are set automatically when the compiler is installed). Therefore, files from these paths are always written without an extension and without a path.

Detailed information about preprocessing directives comes later in this course.

1.1.6

The header file is

- a file with function declaration and/or some definition in C/C++ language.
- a file with a part of function body or main function body.
- a file containing the first line of C/C++ source code.
- a file containing any part of the source code.

1.1.7

Header files have two types: system and programmer made.

- True
- False

1.1.8

The header file named aaa.h to come with the compiler or operating system is used to write:

- #include <|aaa>
- #using <|aaa>
- #include "aaa"
- #include <|./aaa.h>

1.1.9

The header file named aaa.h in the active directory written by the programmer is used to write

- #include "aaa.h"
- #define "aaa"
- #use "aaa.h"
- #include <|./aaa.h>

1.2 Standard library, simple input/output

🕮 1.2.1

Almost every program needs to input some values and output results. So we have to use appropriate commands for it.

The input and output operations depend always on the operating system environment. Every system and every situation is solved differently. But the programmer – the author of the program – has no information about it. The source code of the program must work on many systems and programmers have to write the same commands.

So the input and output operations are encapsulated into functions which bodies are implemented according to the operating system you actually use and their headers are always the same.

Every programming language has some commands for input and output operations but every programming language has a different approach for it.

We will discuss two approaches because of two different versions of C language: pure C and C++ versions.

🕮 1.2.2

In the C++ programming language, the C++ Standard Library is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself. The C++ Standard Library provides support for some language features and functions for everyday tasks such as finding the square root of a number or simple input/output.

There are differences between C and C++ standard libraries. We prefer the C++ approach, so we show C++ simple input/output and the appropriate part of the standard library for this.

We need some input values and write output in almost every program. In the very simple form, we have the statement for input: **cin** (it stands for "console input"), and a statement for output: **cout** (i.e. "console output").

When we need the mentioned commands we have to include a standard library for this in the following form:

```
#include <iostream>
```

1.2.3

For input/output operations in C++ we use the header file in the following form:

- #include <|iostream>
- #include <|stdio>
- #include "inputoutput"
- #input "iostream"

1.2.4

As mentioned earlier, cin and cout are available for input and output. The cin and cout are so-called streams. Their use is supplemented by a special operator "<<" and ">>". Multiple items can be read or written with one cin stream and one cout stream. Small hint: Operator ">>" points from cin to the variable, operator "<<" points from variable or expression to cout.

For example input to two variables:

```
cin >> Var1 >> Var2;
```

Output can be realized as a sequence of expressions in which values are first computed and then shifted to the output stream:

```
cout << "The value is " << Var1 + Var2*5 << "." << endl;
```

Note that cin, cout and endl (stands for "end of the line") are objects in namespace std. When we use them directly, we have to open this namespace via command:

```
using namespace std;
```

If we did not specify this command, it would be necessary to write a space name for each element contained in it:

```
std::cout << "The value is " << Var1 + Var2*5 << "." <<
std::endl;
```

Standard error output

Just as we can write to the standard output (stream cout), we can write to the standard error output, in the same way, using the stream cerr. Everything else remains the same as cout. We will always use the standard error output if we want to convey some non-standard situation or auxiliary information to the user. We never mix these auxiliary messages with data so we don't write them to standard output! For example:

```
double a, b;
cin >> a >> b; // input two non-zero numbers
if (a*b == 0) cerr << "invalid data!" << endl;  // error!
   else // valid data, output to stdout
      cout << "Contents of rectangle is " << a*b << endl;
```

1.2.5

We want to read two values of A and B. These values represent the sides of the rectangle. We should write to output the contents and perimeter of this rectangle. Fill in appropriate parts into following code:

```
#include <iostream>
using namespace std;
int main(){
   float A, B;
 _____
 _____
   return 0;
}
```

- cout >> "Contents ">> A*B >> ", perimeter " >> (A+B)*2 >>"."">>endl;
- cin <|<| A <|<| B
- cin >> A >> cin >> B
- cin >> A >> B;
- cout >> "Contents " cout >> A*B <|<| cout >> ", perimeter " cout >> (A+B)*2 cout >> "." <|<| endl;
- cout << "Contents "<< A*B << ", perimeter " << (A+B)*2 <<"."<<endl;

1.3 Introduction (programs)

1.3.1 Main function, header files

Write down source code with system header file iostream and with the main function which shows "Hello!" (use std::out) and returns exit code 0.

1.3.2 Simple input and output

Imagine an X is the outer diameter of the ball and a Y is its wall thickness. Both values are in millimetres. Read the X and Y values from the input and write the total ball volume and sum of the outer and inner surfaces. The output sentence will be:

Volume is .... mm3, surfaces are .... mm2.

Variables, Constants

Chapter 2

2.1 Variables

📖 2.1.1

A variable is a certain **place in the computer memory** which can hold some values. It's a different approach than in mathematics.

What place does the memory variable occupy? This depends on its **data type**. Data types will be discussed later in this course.

If a variable occupies some space, we can insert certain **values** into it.

Each memory space has its address. In order not to remember the numeric form of the address, these addresses are named, so we call it a **variable identifier**. It represents the numeric form of the address.

So each variable has its address (or name in form of an identifier) and its value.

📝 2.1.2

What is a variable in the programming language?

- A certain place in computer memory.
- A letter representing some real number.
- Number of program steps.

📝 2.1.3

The variable identifier in the programming language

- represents the address of the variable in memory.
- represents the value of the variable.
- represents number of possible values.

📖 2.1.4

Each variable can hold some values. Because the variable occupies a certain space in the computer's memory which is composed as a sequence of bits, each value is machine-displayed as a sequence of zeros and ones.

A sequence of ones and zeros can mean different values. The simplest interpretation is as a non-negative integer in the binary system.

Suppose a variable occupies a space of two bytes. What values can be stored in this space?

The minimum value is represented by all zeros and represents zero. The maximum value is formed by the ones themselves. Two bytes are 16 binary ones, which is a decimal number of 65,535. So a two-byte variable can hold a value between 0 and 65,535.

Generally, a variable on an *n*-bit space can hold a maximum value of $2^n - 1$.

2.1.5

What is the maximum integer decimal value of a variable occupying 8 bits in computer memory?

2.1.6

Each variable you planned to use has to be **declared**. This means we have to tell the computer how the variable will be named and what data type it will be. According to the data type, the compiler creates the appropriate space for this variable in computer memory. The declaration must always precede the use of the variable.

The declaration in C/C++ language is written as follows scheme:

```
data_type variable_identifier;
```

For example (int is an identifier for the integer data type):

```
int Count;
```

Now we can use the variable Count which can hold some integer value.

If we need more variables of the same type, we can use the notation where we specify a data type and a list of variables of this type divided by commas:

```
int suma, current_value, count;
```

After the declaration, the value of the variable is **undefined**. If we want the variable to have its initial value along with the declaration, we can write it directly into the declaration statement. In one declaration statement, we can combine variables without and with an initial value. For example:

```
int suma=0, current_value, count=1;
```

The variable current_value has an undefined value, but suma has value zero and count has value 1.

2.1.7

We want to declare two variables: Side and Radius. The data type is float. The variable Radius should have an initial value of 10. Fill in the appropriate pieces of code:

```
_____  Side,  _____ ;
```

- float
- Radius=10
- char
- Side=0
- 10=Radius
- radius=10
- int
- Radius

2.2 Constants

2.2.1

A constant represents memory space. It is similar to a variable. Compared to a variable, however, it has a value that cannot be changed. We assign this value when declaring a constant.

The constant declaration looks like a variable declaration but it is preceded by keyword const and we always have to define its initial value:

```
const int WeekDays=7;
```

2.2.2

We want to declare a constant that contains the number of minutes per day. Fill in the appropriate pieces of code:

```
_____  _____  _____  _____ ;
```

- =1440
- var
- char
- int
- =24
- float
- =60
- MinPerDay
- const

2.3 Variables (programs)

⌨ 2.3.1 Variables 1

Declare one variable of type char with identifier Pass and initial value asterisk and one variable of type double with identifier Num and initial value 2.7182.

⌨ 2.3.2 Variables 2

Declare a variable of type int with identifier Page and initial value 123 and a variable of type char with identifier Delim and initial value tilde.

⌨ 2.3.3 Constants 1

Define constant named Digit with a value of the character with ordinal value 48, and constant named Zone with integer value 48.

⌨ 2.3.4 Constants 2

Define constant named PI with value 3.14159, and constant named Sign with the value of minus character.

Data Types, Type Conversion

Chapter 3

3.1 Data types

📖 3.1.1

The data type specifies the **allowable values** and **allowed operations** with these values. The data type can be **predefined** or **user-defined**. Predefined data types are basic building blocks for user-defined types.

In this lesson, we will deal with numeric data types and types for expressing character information.

As mentioned above, a variable represents a place in computer memory. The data type tells us how to interpret this memory location and how we can process it. Thus, the variable is associated with the data type.

📖 3.1.2

There are numeric data types for number processing. We will start with integer data types. They are divided into two categories – **unsigned** (non-negative values only) and **signed** (both negative and positive values). They are listed in the following table:

Numeric data types – integer values

Type identifier	Short type identifier	Memory size	Values range
unsigned int	unsigned	4 B	0 to 4,294,967,295
unsigned long int	unsigned long	4 B	as above
unsigned long long int	unsigned long long	8 B	0 to 18,446,744,073,709,551,615
unsigned short int	unsigned short	2 B	0 to 65,535
unsigned char	(none)	1 B	0 to 255
signed int	int	4 B	–2,147,483,648 to 2,147,483,647
signed long int	long	4 B	as above
signed short int	short	2 B	–32,768 to 32,767
signed long long int	long long	8 B	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed char	char	1 B	–128 to 127

3.1.3

Match appropriate data type name and its value range.

signed short int _____

unsigned char _____

signed char _____

unsigned short int _____

- 0 to 65,535
- −65,536 to 65,535
- −32,768 to 32,767
- 0 to 255
- −256 to 255
- −128 to 127

📖 3.1.4

A pair of integers is used to store real numbers: mantissa (M) and exponent (E). These data types are generally referred to as floating-point numbers.

The number value is calculated as *M*·2^*E*. The number of bits of mantissa indicates the accuracy of the number, the number of bits of the exponent determines the size of the number. Accuracy is given as the number of significant digits. The number size is given in decimal order.

In C/C++ language there are three floating-point data types:

Floating-point data types

Data type name	Memory size [B]	Accuracy	Size
float	4	6 digits	$+/-10^{+/-38}$
double	8	15 digits	$+/-10^{+/-308}$
long double	10	18 digits	$+/-10^{+/-4932}$

Note also that the long double type is often stored at 16B to align memory to blocks of 4B.

📝 3.1.5

We have three data types for real numbers in the C/C++ language:

- float, double, long double.
- single, double, extended.
- real, comp, decimal.
- numeric, float, extended.

📝 3.1.6

Match the appropriate values:

double _____

long double _____

float _____

- accuracy to 12 significant digits
- accuracy to 10 significant digits
- accuracy to 8 significant digits
- accuracy to 15 significant digits
- accuracy to 6 significant digits
- accuracy to 18 significant digits

3.1.7

Assume the following variable declarations:

```
double A, B = 0;
float C, D=1.2E-2;
```

How much total memory do these variables take?

3.1.8

The char data type mentioned earlier is primary used to store **character values**. Characters are written into apostrophes, for example:

```
'a'   '7'   '('
```

Each character is stored in the computer memory as an integer. The base character set (called ASCII – American Standard Code for Information Interchange) contains 128 characters with codes 0 through 127. The character code table determines which number each character is stored in.

The character set is divided into two categories: **printable** (visible) characters (letters, digits, punctuation marks) and **control** characters. The control character is used to control the output devices, for example, go to a new line, ring bell or backspace. The control character is written with a special sequence starting with a backslash.

The control character can be written using its code in octal or hexadecimal, for some characters there are special sequences, see the following table.

Control characters

Name of the character	Acronym	Code (decimal)	Written as
null character (termination of strings)	NUL	0	'\0'
end of text file	EOT	4	'\4' or '\x4'
bell (ring bell when shipped to output)	BEL	7	'\7' or '\x7'
backspace	BS	8	'\10' or '\x8'
horizontal tab	HT	9	'\11' or '\x9' or '\t'
line feed (new line)	LF	10	'\12' or '\xa' or '\n'
form feed (new page)	FF	12	'\14' or '\xc' or '\f'
carriage return	CR	13	'\15' or '\xd' or '\r'
escape (start of printer control sequences)	ESC	27	'\33' or '\x1a'

3.1.9

Match the appropriate values.

LF _____

CR _____

form feed _____

BEL _____

- written as '\7'
- written as '\10'
- written as '\xc'
- written as '\n'
- written as '\r'
- written as '\xf'

3.1.10

Match the appropriate values:

A control character used for the start of new page _____

Control character for tab mark is _____

Control character for the new line _____

When you want to beep you send character _____

- HT

- EOT
- BS
- LF
- BEL
- FF

🕮 3.1.11

The user can define his own data types, which can then be treated similarly to predefined ones. This can be very advantageous if, e.g. we want to get rid of dependence on a particular type and we want to generalize the notation so that we can change the data type of the processed values by changing it in one place.

The **typedef** keyword is used to define a new type, followed by a type definition, and a new type identifier, such as:

```
typedef unsigned int Value_type;
```

Then we can use this new type to declare variables:

```
Value_type Current, Sum = 0, Max, Min;
```

Then, in the algorithm for calculating the sum and extreme values from the input sequence, we can process unsigned integers once, sometimes they can be decimal numbers, just replace the Value_type definition.

3.2 Data type conversion

🕮 3.2.1

The value stored in a particular memory location can be understood in different ways. So we sometimes need to tell the compiler that the value of a particular variable is to be processed differently than the data type of that variable.

Sometimes, the compiler automatically executes this change itself. For example, if you insert an int variable value into a long long variable, the compiler automatically extends the int value to long long. This is called **implicit data conversion**.

We can also specify **explicit type conversion**. It is written in two possible ways:

```
type (expression)
```

or if the data type is written in more words:

```
(type) expression
```

Examples:

```
char(48 + digit)
(long long) sum
```

If the resulting value does not fit in the converted type, this value is truncated. For example:

```
int A=1000000000;
long long B;
B=9*A; // we multiply value A by a 9
```

The result is 410065408, but the right result can be 9000000000. To solve this mistake we need to write:

```
int A=1000000000;
long long B;
B=9*(long long)A;
```

3.2.2

We have defined a variable X of the type short int with a value of 300. What do we get if we convert to an unsigned char?

- 44 (or character ',')
- 300 (or char 'ň')
- 255
- 344 (or two characters "ň,")

3.3 Data types (programs)

3.3.1 Data types

Declare one integer variable without sign which takes 2 bytes of memory, named MyVar. Then declare one real variable of 12 bytes, named MyBigVar.

c1_types1.cpp

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

int main(){
```

```
    // declare the variables here:
//---------------------------------

//=================================
// the test of proper declaration:
    cout <|<| sizeof(MyVar)<|<|" bytes."<|<|endl;
    cout <|<| sizeof(MyBigVar)<|<|" bytes."<|<|endl;
    return 0;
}
```

⌨ 3.3.2 Type conversion

Declare a variable named Letter of type char and 4-byte long integer variable named Number which will assign to an ordinal number of variable Letter.

c1_types2.cpp

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

int main(){
    // declare the variable Letter here:
//-------------------------------

//===============================
    cin >> Letter;

    // declare and assign the variable Number here:
//-------------------------------

//===============================
    cout <|<| "The character " <|<| Letter <|<| " has an
ordinal number "
         <|<| Number <|<| "." <|<| endl;
    return 0;
}
```

Assignment and Operators

Chapter 4

4.1 Expression, statement, assignment

📖 4.1.1

The **expression** prescribes some calculation. It is composed of operands and operators.

The **operand** is a constant value or variable. The **operator** may be a sign for arithmetic operation, logical operation or other manipulation with values of various types. The operator may be a function too.

Many expressions are written like in math, but there are differences.

Expression examples (a – circle content, b – triangle perimeter, c – root of the quadratic equation):

```
a)   Radius * Radius * 3.14
b)   sideA + sideB + sideC
c)   (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
```

Notes: Operators have their **priorities** (order of evaluation). Parentheses have the highest priority, they are used only round unlike in math. Operator for multiplication is written like "*" and unlike in math cannot be neglected. The sqrt function (square root) is in the cmath library, and this library must be appended with the #include <cmath> command. Each function has its parameters in parentheses, parentheses must be written even if the function has no parameters. For exact value of the number Pi, we can use the constant M_PI from cmath library. So the exact value of circle contents we obtain as

```
Radius * Radius * M_PI
```

📝 4.1.2

Suppose we have the variable Radius with the radius of the sphere. We want to calculate the volume of the sphere, $V=4/3\ \pi r^3$. Which expression will make the correct calculation?

- Radius*4*Radius/3*Radius*3.14
- 4/3M_PI Radius^3
- 4*3.14/Radius*3*Radius*Radius
- 4Radius*Radius*Radius/3M_PI

4.1.3

An assignment is a special operator that works from right to left. On the right side, it has an expression that is first calculated, and on the left side, it has a variable into which the calculated expression value is inserted. The assignment operator has various shapes. The simplest is "=", for example:

```
Volume = SideA * SideB * Height;
```

At the same time, the expression serves as a **command** that stores the calculated expression value in the specified variable. Therefore, it is more of an **assignment statement**, not an expression. The **assignment statement** is one of the simple commands and is one of the most widely used program elements.

The assignment has various shapes. In short, we can write the fact that the value on the right side is added (or multiplied, etc.) to the variable on the left side along with the assignment. For example, we need to add a SideC value to the TotalLength variable. We can write:

```
TotalLength = TotalLength + SideC;
```

or also abbreviated

```
TotalLength += SideC;
```

Similar to the + = operator, we can use *=, /=, -=, %= etc.

4.1.4

We have the following code:

```
int V, a = 10, b = 5, c = 9;
V = (a += 2) * (b -=3) * (c /= 3);
cout << "Volume = "<< V << endl;
```

What result will we see on the screen?

- Volume = 72
- Volume = 18
- Volume = 450
- Volume = 150

📖 4.1.5

Some operators are presented in the following table:

Operators in C++

Operator	Syntax	Example
Adding two numbers	+	a + b
Subtracting two numbers	-	a - b
Multiplying two numbers	*	a * b
Dividing two numbers	/	a / b
Modulo of the integer division of the two numbers	%	a % b
Greater than	>	a > b
Less than	<	a < b
Greater and equal than (without space between)	>=	a >= b
Less and equal than	<=	a <= b
Equal (attention: TWO equal signs in a row)	==	a == b
Not equal	!=	a != b
Logical and (conjunction)	**and** &&	a **and** b a && b
Logical or (disjunction)	or ||	a or b a || b
Negation	!	!a
Assignments	= += etc.	a += b

Operators can be divided into several groups. For example, additive operators are + and -, multiplicative are *, / and%, relational (used to compare values) are ==, ! =, >= etc. The operators differ in their priorities. Operators in the same group have the

same priority and are evaluated from left to right in the expression. The higher the operator's priority, the earlier it is calculated in the expression. You can change the priority of calculation by enclosing a part of the expression in parentheses.

Examples:

```
int A = 10, B = 5, C = 4, result;
result = A + B * C; // result is 30
result = (A + B) * C; // result is 60
result = A + C / B;  // result is 10
result = (A + C) / B; // result is 2
result = A * C / B; // result is 8
result = (A * C) / B; // result is 8 - the same as above
```

4.1.6

We need to calculate the following formula: $y=y \cdot x \,/\, [i \cdot (i-1)]$

Which of the following statements calculates the value of this formula?

- y *= x/(i*(i-1))
- y += y*x/i/(i-1)
- y = y*x*1/i*(i-1)
- y *= y*x /(i*i-1)

4.1.7

We need to recalculate the Celsius temperature to Fahrenheit. We know that $F = 9C / 5 + 32$, where C is Celsius temperature. Which of the following expressions correctly computes the Fahrenheit temperature?

- F = 9*C/5+32;
- F = 9C/5+32;
- F = 9/5/C+32;
- F = 9C/(5+32);

4.1.8

In the previous sections, it was stated that the assignment is one of the operators. Therefore, the assignment statement is also an expression. The value of this expression is the value inserted into the variable on the left side of the assignment.

We can use this fact if we want to insert this value into another variable. For example:

```
int CountA, CountB, CountC, StartingValue = 1;
CountA = CountB = CountC = StartingValue;
```

In this case, all three variables will have the same value that is equal to the contents of the StartingValue variable.

The assignment operator is evaluated from **right to left**. This is different from many other operators, for example, the addition is evaluated from left to right. So the first operation is assigning StartingValue to the CountC variable. This value is then inserted into the CountB variable (the same value as in CountC) and then is inserted into the CountA variable.

The assignment operator can be used in any part of another expression. We will achieve a double effect: assigning a value to a variable and at the same time using that value in another part of the expression. Example: We want to calculate the content and perimeter of a rectangle. The sides of the rectangle are in variables a and b. We can assign the initial values in the first expression and then use these values in the second expression:

```
int a, b, content, perimeter;
content = (a = 3) * (b = 4);
perimeter = (a + b) * 2;
```

This notation is not the best because assigning initial values is lost inside the first expression and is slightly confusing but illustrates the function of the assignment operator.

4.1.9

The engine of a certain car has 4 cylinders. What is the volume of ten such cars? The input values are the radius and height of one cylinder. Add the correct parts to the code.

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
  float radius, height, volume;
  cin >> _____ >> height;
  cout << "Volume of given cylinder is " <<
     ( _____ ) << endl;
```

```
  cout << "Volume of four such cylinders is " << ( _____ ) <<
endl;
  cout << "Volume of cylinders of ten cars with the same
engine is "<<
  _____ << endl;
  return 0;
}
```

- volume = radius * radius * M_PI * height
- radius
- volume * 10
- volume * M_PI * radius
- height
- volume
- radius * radius * 10 * M_PI
- 4 * radius * radius * height
- volume *= 4
- radius *= M_PI * height

🕮 4.1.10

A very frequently used operation is to add or subtract one. The so-called incremental operator serves to simplify the syntax of increasing or decreasing the value of the variable.

This operator is written as a two + or two – and it has two possibilities of use: as a **post-increment** (written after a variable) or as a **pre-increment** (written before a variable).

The post-increment works by first using the current value of the variable in the expression and then increasing it by one. Conversely, a pre-increment first increases the value of the variable by one and then the new value is only used in the expression. Examples:

```
int x = 10; y;
y = x++ * 5; // y is 50 and new value of x is 11
y = ++x * 5; // y is 60 and new value of x is 12
```

Thus, increasing the value of variable X by one can be done in four ways:

```
X = X + 1;  // this is long :-(,
            // but clear :-) and commonly used in other
languages
X += 1;     // the use of assignment operator
X++         // postincrement
```

```
++X          // preincrement
```

4.1.11

We have defined this part of the code:

```
int a = 6, b = 12, c = 4;
   cout << a++ + --b * c++ << endl;
```

What will appear on the screen?

- 50
- 80
- 72
- 95

4.2 Assignment and operators (programs)

4.2.1 Simple assignment 1

Try to complete code to obtain product and difference of two values X and Y.

c1_assign1.cpp

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

int main(){
    int X=3, Y=1;
    // complete the code here:


    cout <|<| "Product of numbers X and Y is:
"<|<|product<|<|endl;
    cout <|<| "Difference between X and Y is:
"<|<|differ<|<|endl;
}
```

4.2.2 Simple assignment 2

Try to complete code to obtain the sum of squares of values X and Y

```
input : none
output: 25
```

4.2.3 Simple assignment 3

Calculate the discriminant of a quadratic equation.

c1_assign3.cpp

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

int main(){
    int A=1, B=3, C=2; // coefficients of quadratic equation
    // complete the code here:


    cout <|<| "The discriminant is: "<|<| D <|<|endl;
}
```

4.2.4 Simple assignment 4

Calculate roots of quadratic equation with given coefficients A, B, C. Suppose two real roots only.

4.2.5 Simple assignment 5

Calculate three members of the Fibonacci sequence from the given two members.

c1_assign5.cpp

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

int main(){
    int M1=55, M2=89; // two members of Fibonacci sequence
    // complete the code here:


    cout <|<| "The members are: "<|<|M3<|<|", "<|<|M4<|<|",
"<|<|M5<|<|"."<|<|endl;
}
```

Formatted Input/Output

Chapter 5

5.1 Formatted input/output

📖 5.1.1

We suppose that the actual compiler is C++, so we have to link a library with standard C input/output features. This library is named "cstdio". The link to the header file is written as:

```
#include <cstdio>
```

For input, we may use the function scanf. This function has the first parameter which defines the data type of input values and the next parameters are memory addresses to store read values.

For output exists function printf. This function has the first parameter which defines the shape of the output string and the next parameters define output values (variables or expressions).

Both functions have some common types of format parameters. So we can define how may input value be read and the same way we define how may be output value written.

Some simple examples of format parameters:

```
%d -- decimal integer value
%f -- float value
%s -- string (array of chars)
%x -- hexadecimal integer value
%c -- one character
```

Some simple examples of read/write values:

```
scanf("%d", &count); /* one integer value is read to a
variable on address count. Note, that the address of the
variable is indicated by the & */
scanf("%f%s", &price, customer); /* one float value stored to
price and some characters stored to the variable named
customer. Customer is array of characters and this is direct
address, so we do not need the & */
printf("Total count is %d.", count); /* integer value from
variable count is written into given sentence */
printf("The customer %s have to pay $%f.", customer, price);
/* name of customer and price are written in given sentence */
```

5.1.2

The following program reads two integer values and stores them into two variables MeasureA and MeasureB. Fill the appropriate parts into code:

```
#include  _____
int main(){
  int MeasureA,  _____ ;
  scanf(" _____ ", &_____ , &_____ );
  _____  0;
}
```

- <cstdio>
- MeasureA
- halt
- %d%d
- <|iostream>
- MeasureB
- %2d%2f
- return
- MeasureB

5.1.3

We have the following piece of code:

```
int A=10, B=3, C=14;
printf("Values are %d.%d, more than %d times and %c is name of
first variable.", B, C, A, 65);
```

Which output exactly corresponds with the previous code?

- Values are 3.14, more than 10 times and A is the name of the first variable.
- Values are 10.3, more than 14 times and 65 is the name of the first variable.
- Values are 10 and 3, more than 14 times and A is the name of the first variable.
- Values are 14, more than 30 times and 65 is the name of the first variable.

5.1.4

The specifiers mentioned in the previous text (d, f, c, s, etc.) can be supplemented with four groups of sub-specifiers in the following order:

```
%[flags][width][.precision][length]specifier
```

Some of these sub-specifiers are listed in the following tables:

Flags

Sub-specifier Description

– Left-justify within the given field width; Right justification is the default (see width sub-specifier).

+ Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.

0 Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

Width

Sub-specifier Description

(number) The minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

Precision

Sub-specifier Description

.number For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0.
For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed.

The *length* sub-specifier modifies the length of the data type. For example output value of type int may be written as signed char, short int, long int, long long int:

Length

Sub-specifier Description

hh writes integer as signed char

h writes integer as short int

l writes integer as long int

ll writes integer as long long int

L writes double as long double

Examples:

```
printf("%010d", 1977);  // displays 000001977
printf("%+5.2f", 3.1);  // displays +3.10
printf("%-10d%d", 1, 1); // displays 1         1
```

5.1.5

We have three values stored in variables float A, int B, and unsigned char C. We need the following shape of the output:

```
The return count of negative numbers is 00014.
Minimal number with sign was: -796.85000 and its category was:
Q.
```

Fill the following statement with appropriate format specifiers and sub-specifiers:

```
printf(
_____
 , B, A, C);
```

- "The return count of negative numbers is %05d.\nMinimal number with sign was: %+10.5f and its category was: %c."
- "The return count of negative numbers is %50x. Minimal number with sign was: %-5.5d and its category was: %hh."
- "The return count of \nnegative \nnumbers is %d. Minimal \nnumber with sign was: %10f and its category was: %HH."
- "The return count of negative numbers is \n%5d. Minimal number with sign was: %3.5o and its category was: %C."

5.1.6

There is another approach to input and output values in the C++ language. We have to first include a library with input/output operations called iostream:

```
#include <iostream>
```

The operations for input and output are stored into namespace "std", so we have to write "std::" as a prefix of the operation name. More info about namespaces will be discussed later.

C++ input/output is solved as objects called streams. Input stream for standard input is called cin and output stream for standard output is called cout.

For the simple input can be used statement in the following form:

```
std::cin >> variable;
```

The name "cin" stands for "console input". The value from the standard input file is stored into variables according to the data type of this variable. When we use multiple operators ">>", we can input multiple values. This is the simplest form for common input.

Example:

```
std::cin >> first >> second;
```

For the simple output can be used following statement:

```
std::cout << expression;
```

where "expression" is a variable or combination of variables and operators. The name "cout" stands for "console output". Note that operator "<<" have certain priority and operators used in expression may have less priority. Then we have to enclose this expression into parentheses. To output multiple values can be written multiple operators "<<" and multiple expressions. Each expression is calculated first and then is written to the output stream.

Example:

```
std::cout << "The result value is " << first * second <<
".\n";
```

Note the "\n" is the control character for the new line.

The cout has some internal items accessible with dot notation, for example:

- precision(*n*) – sets the number of decimal positions of float numbers to *n*
- width(*n*) – sets the size of the next output expression to *n* characters.
- fill(c) – sets the character-filled to output expression instead of spaces.

Example:

```
float radius=2.752, volume;
volume = 4/3 * M_PI * radius * radius * radius;
std::cout.precision(4); std::cout.width(40);
std::cout.fill('/');
std::cout << volume<<endl;
```

The output value will be:

```
//////////////////////////////////65.48
```

Some of these formatting settings are usable via so-called manipulators (will be discussed later).

5.1.7

Choose appropriate options:

```
#include < _____ >
int main(){
  int lengthA, lengthB;
  cout _____ "Enter two side of the rectangle, please:\n";
  _____;
  _____ "The contents of this rectangle is " _____;
  return 0;
}
```

- cin >> lengthA >> lengthB
- iostream
- <<
- << lengthA * lengthB << ".\n"
- cout <<

5.1.8

So-called manipulators are used to adjusting the output shape of the values displayed. Manipulators are written in the same way as output variables or expressions in the cout command. The following table shows some useful manipulators:

Manipulators for output

Manipulator	Description
endl	end of line
dec	switches output of integers to decadic system (implicit)
oct	switches output of integer to octal system
hex	switches output of integer to hexadecimal system
showpos	shows plus sign for positive numbers (with zero)
noshowpos	switches off displaying plus sign for positive numbers
left	left aligning
right	right aligning
internal	fill character is displayed between sign and value
fixed	displays real numbers with a fixed position of the decimal point
scientific	switches displaying of real numbers to scientific shape (with exponent)
uppercase	all letters in numbers (hexadecimal or scientific) are in upper case
nouppercase	all letters in numbers (hexadecimal or scientific) are in lower case
setprecision(p)	sets the precision of real number to p
setw(x)	sets the width of the next output to x characters
setfill(c)	sets the fill character to c

Manipulators setw, setfill and setprecision are implemented in header file iomanip, so we have to include them (#include <iomanip>).

Some examples:

```
cout << showpos << 56;  // displays +56
cout << setw(5)<<left<<-56;  // displays two spaces and -56
cout << fixed << setprecision(3) << 3.141592;  // displays
3.142
cout << scientific << setprecision(2) << 142; // displays
1.42e+002
cout << setw(10) << left << setfill('/') << fixed
<<setprecision(3) << 3.115926; // displays 3.142/////
```

The settings you make in one output statement also apply to the following commands until you undo the settings. The exception is the setw manipulator which applies only to the following expression.

5.1.9

We have to calculate the volume of the sphere. We want to display the radius and volume below each other, aligning them to 10 positions on the right, as is shown on the following schema:

```
Radius of sphere:      2.752 cm,
Volume of sphere:     65.478 cm3
```

Which code displays exactly this shape of output?

- cout <|<| "Radius of sphere: " <|<| setw(10) <|<|setprecision(3)<|<|fixed<|<|right<|<|radius <|<| " cm," <|<| endl; cout <|<| "Volume of sphere: " <|<| setw(10)<|<|volume <|<| " cm3"<|<|endl;
- cout <|<| "Radius of sphere: " <|<| radius:setw(10):3 <|<|right <|<| " cm," <|<| endl; cout <|<| "Volume of sphere: " <|<| volume:setw(10):3 <|<| " cm3"<|<|endl;
- cout <|<| "Radius of sphere: " <|<| nouppercase <|<|setprecision(3)<|<|floated<|<| leftspaces <|<|radius <|<| " cm," <|<| endl; cout <|<| "Volume of sphere: " <|<| leftspaces <|<|volume <|<| " cm3"<|<|endl;
- cout <|<| "Radius of sphere: " <|<| width(10) <|<|precision:3<|<|fixed<|<|right <|<| radius <|<| " cm," <|<| endl; cout <|<| "Volume of sphere: " <|<| width(10) <|<| volume <|<| " cm3"<|<|endl;

🕮 5.1.10

The cin command with the ">>" operator assumes a simple input that omits the so-called white space characters (spaces, tabs, line breaks). This method is useful for entering numbers, individual characters, and some strings. Often, however, we need to work with input values somewhat differently, using other input stream tools.

To enter individual characters without skipping the white space we can use the *get* method of input stream *cin*. Example:

```
char ch;
cin.get(ch);
```

To modify the behaviour of input operations we can set the input stream with setf or unsetf. For example:

```
cin.unsetf(ios::skipws);
```

This flag sets/unsets skip of white space characters. Other flags for example are:

- boolalpha – reads (and writes) boolean values as a string true/false instead of 1/0.
- showbase – write integral values preceded by their corresponding numeric base prefix.
- showpoint – write floating-point values including always the decimal point.
- showpos – write non-negative numerical values preceded by a plus sign (+).

Notice that several manipulators have the same name as these flags (but as global functions instead).

For reading strings (a character array, not a string type), the getline method is used in addition to the basic form. The getline method has two or three parameters:

```
char S[20];
```

```
cin.getline(S, 10); // reads max. 9 characters plus \0 or to
the end of line
```

We can read to the given delimiter:

```
cin.getline(S, 15, ':');
  // reads max. 15 characters or to the colon or to the end of
line
```

The last method is useful for reading strings from CSV files for example.

5.1.11

Two characters are on the input. Find out what categories these characters include (white characters, letters, digits, punctuation). Add the corresponding parts to the following code.

```
char a, b;

 ______
if ( ______ ) cout << "a is white character"<< endl;
  else if (a>='0' and a<='9') cout << "a is digit"<<endl;
      else if ((a>='A' and a<='Z') or (a>='a' and a<='z'))
          cout << "a is letter" << endl;
          else cout << "a is punctuation" << endl;
if ( ______ ) cout << "b is white character"<< endl;
  else if (b>='0' and b<='9') cout << "b is digit"<<endl;
      else if ((b>='A' and b<='Z') or (b>='a' and b<='z'))
          cout << "b is letter" << endl;
          else cout << "b is punctuation" << endl;
```

- a==' ' or a=='tab'
- b<=' '
- a==' '
- cin >> (get)a; cin >> (get)b;
- a<=' '
- b==' ' or b=='tab'
- cin.get(a); cin.get(b);

5.2 Formated input/output (programs)

⌨ 5.2.1 Formatted output 1

We have two numbers – tax-free amount and VAT given on the input. Calculate amount with VAT and write: "Result is XXX.YY €." The number has to occupy the position of 10 characters have to be on 2 decimal digits and formatted to the right side.

⌨ 5.2.2 Formatted output 2

We have the name, surname and salary of the person given on the input. Write this information on output so that personal name will occupy 15 characters (padded by spaces), surname 25 characters and salary 7 characters without decimal places. The salary will be aligned right.

⌨ 5.2.3 Formatted output 3

We have one word and one integer value given on the input. Task: display one row on output where input word will be on the left side, occupy 20 positions, input integer value will be in the right side, occupy 7 positions and between the word and the number will be leaders dots, for example:

```
Glass.................35
```

⌨ 5.2.4 Formatted output 4

We have two integer numbers given on the input. Display these numbers and their sum in hexadecimal code, for example:

```
Input numbers 10 and 15 in hex: A + F = 19
```

⌨ 5.2.5 Formatted output 5

We have two input integer numbers. Display them in the octal system and display their product as well. The exact shape of output is:

```
Product of decimal numbers 10 and 8 in octal code: 12 * 10 =
120
```

5.2.6 Formatted output 6

We have given one real number, the requested number of positions and the requested number of decimal places on the input. Display the input real number with specified sizes and display underscores instead of spaces.

For example – input numbers are 7.78 10 4 and on output, we will see:

```
The number is: ____7.7800
```

5.2.7 Formatted output 7

In some spreadsheets, it is possible to format values so that a sign is placed on the left and a value on the right. There may be a fill between the sign and the number (most often spaces, but also other characters). Create a similar format: there is a real number at the input and the displayed width in characters. Display the number so that the sign is on the left, the number is on the right, and the fill is done with dots. For example, input number 2.45 on size 10 will be displayed as:

```
+.....2.45
```

5.2.8 Formatted output 8

The output of the typical SQL command "select": strings are aligned to the left, padded with spaces and wrapped with the character '|'. We have given two strings on the input. Display them to a size of 10 characters. For example input strings Alex and Barnie will be displayed following (remember to write an end of line):

```
|Alex      |Barnie    |
```

5.2.9 Formatted output 9

The hexadecimal system is used to easily express binary values always storing data on a computer in whole bytes or groups (for example, 4 bytes). Thus, the values are displayed so that the number of digits is always the same and corresponds to the desired byte width and is filled with zeros from the left. Assume that the input is an integer value and the required number of bytes to display. Display this value in hexadecimal with the appropriate width. For example, the input is a pair of 26 4, the output will be:

```
0000001A
```

5.2.10 Formatted output 10

There are five integer values at the input. Write a histogram of their relative shares in the total. Each 10% will be displayed as 5 stars. For example, for values 1 3 2 3 1 we obtain the following:

```
1: *****
3: ***************
2: **********
3: ***************
1: *****
```

form_output10.cpp

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

int main(){
  int num1, num2, num3, num4, num5;
  cin >> num1 >> num2 >> num3 >> num4 >> num5;
  // enter the appropriate code here:

  return 0;
}
```

Logical Operators

Chapter 6

6.1 Logical data type and logical operators

🕮 6.1.1

A **bit** is the minimum amount of information that we can imagine since it only stores either value 1 or 0, which represents either YES or NO, activated or deactivated, true or false, etc. This means two possible states each one opposite to the other without the possibility of any shades. We are going to consider that the two possible values of a bit are 0 and 1.

Several operations can be performed with bits either in conjunction with other bits or themselves alone. These operations receive the name of **boolean operations**, a word that comes from the name of one of the mathematicians who contributed the more to this field: George Boole (1815–1864). There is a special data type named **bool**. Values of the bool type are false and true. The value false is implemented as zero in computer memory, the value true is implemented as one (or any non-zero value). Boolean values are also called **logical values** and data type bool is also called **logical data type**.

Basic operators with logical values are **and**, **or**, **not**, **xor**. The following table shows the results of operations with these operators.

Logical operations

value A	value B	not A	A and B	A or B	A xor B
false	false	true	false	false	false
false	true	true	false	true	true
true	false	false	false	true	true
true	true	false	true	true	false

Input and output of logical values

For C language compatibility, both input and output logical values are represented by numeric values. A value of false is zero, and a value true is 1. We can change this behaviour by setting stream format flag boolalpha. Then we can read string "true" or "false" as a logical value and write the logical value in form of strings "true" or "false".

🕮 6.1.2

A logical expression is similar to an arithmetic expression. It is composed of operands and operators. However, operators calculate the results in the form of a logical value. For example the expression

```
(a or b) and not (c and d)
```

is a logical expression where a, b, c, d are operands (variables) and "or", "and", "not" are logical operators.

Similar to the arithmetic expression, logical operators have their priorities. The logical expression is evaluated according to these priorities. The change of priority can be prescribed by brackets.

All comparisons can be a part of a logical expression. For example, x == y is true when x and y are equal. Variables x and y may be any numbers, characters, strings and so on. Example:

```
(x != 5) and (x > 0)
```

The expression value is true when x is positive but not equal to five.

6.1.3

Suppose we have a declaration:

```
bool first=false, second=true;
```

What value will the following expression have:

```
not ((first xor second) and (first or second))
```

- False
- True

6.1.4

The value of variable Salary is between 1000 and 3000 € inclusive. Which of the following expressions will be true if Salary meets that condition?

- (Salary >= 1000) and (Salary <|= 3000)
- (Salary = 1000) or (Salary = 3000)
- (Salary <|= 1000) xor (Salary >= 3000)
- (Salary >= 1000 and <|= 3000)

6.1.5

We need to find out whether the variable Number has an odd non-negative value. Which of the following terms is true with the specified criterion?

- (Number >=0) and (Number % 2 == 1)
- Number positive and Number odd
- (Number !=0) or (Number % 2 != 0)
- Number between 0 and odd(100)

6.1.6

We have the following code:

```
char one, two;
cin >> one >> two;
cout << (((one == 'a') or (one == 'e')) and ((two == 'x') or
(two == 's'))) << endl;
```

For which inputs will the output be 1?

- axis
- especially
- average
- ethernet
- session

6.2 Logical operators (programs)

6.2.1 Logical expressions 1

Complete the program which reads a length of a square and writes value 1 if the content of this square is between 10 and 100.

logic.cpp

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

int main(){
  int SquareSide, Content;
  cin >> SquareSide;
  Content = SquareSide * SquareSide;
 // here enter code for output logical information about
content of square.

  return 0;
```

```
}
```

6.2.2 Logical expressions 2

Complete the program that reads two lengths and writes value 1 if these lengths are identical.

6.2.3 Logical expressions 3

Complete the program that reads personal height in centimetres and weight in kilograms and writes value 1 if BMI of this person is under 25, zero instead. The BMI is calculated as the weight divided by the square of the height in meters.

6.2.4 Logical expressions 4

Complete the program that reads personal height in centimetres and weight in kilograms and writes value 1 if BMI of this person is out of normal range (18.5, 25), zero instead. The BMI is calculated as the weight divided by the square of the height in meters.

logic4.cpp

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

int main(){
  int height, mass;
  cin >> height >> mass;

 // here enter code for output logical information about
personal BMI.

  return 0;
}
```

6.2.5 Logical expressions 5

A mortgage applicant wants to finance a € 100 000 property. He gets a loan when he asks for a maximum of € 80 000 and earns more than € 1 500 a month. When he

is younger than 35, he gets € 90 000 and can only earn € 1 000 a month. The entry is the required amount, monthly earnings and the applicant's age. Type 1 to get a loan, zero instead.

Conditional Statements

Chapter 7

7.1 Conditional statement "if"

📖 7.1.1

Flow control

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program.

But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C/C++ provides **flow control** statements that serve to specify what has to be done by our program, when, and under which circumstances.

Compound statement

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C/C++ statement, such as a single instruction, terminated with a semicolon, or a **compound** statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:

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

The entire block is considered a single statement (composed of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

The compound statement may have its own local declaration which is valid only in the range of this statement. This is useful for example to divide variables that are useless in other parts of the program.

📖 7.1.2

Condition

A condition is any boolean expression. We say that the condition is fulfilled if the value of this expression is true. Otherwise, the condition is not met. The condition is used as a part of some flow control statements.

Flow control – branching

The branching is the division of the program flow into two (or more) variant parts. Which part will be performed in a particular moment is most often driven by the value of the condition.

The most often situation is to divide flow control into two branches. The "if" statement can only execute a command if a condition is met or another statement if a condition is not met. The syntax of this statement is:

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

Note the condition is enclosed to parenthesis. Both statement1 and statement2 can be simple statements or compound statement.

Words "if" and "else" are **keywords**. Keywords cannot be used in another way than in their defined meaning. Therefore, they will be highlighted in bold to distinguish them from other parts.

We can use an incomplete **if** statement that omits the part that starts with **else**. In this case, nothing is done if the condition is not met.

7.1.3

Select the command that displays variable *Count* only if it is even.

- if (Count % 2 == 0) cout <|<| Count <|<| endl;
- if {Count % 2 == 0} (cout <|<| Count <|<| endl;)
- if (cout % 2 != 1) Count <|<| cout <|<| endl;
- if (Count == 2) {cout <|<| Count <|<| endl};

7.1.4

A student will receive a 1000 € yearly scholarship if his or her average is better than 1.2. Write a message about the scholarship of students. Fill in the corresponding parts to the following code.

```
_____ Average;
_____ Average;
if _____ cout << "This student receives €1000";
else cout << "This student receives no scholarship.";
```

- int
- cout >>;
- cin >>
- (Average < 1.2)

- (Average == 1.2)
- unsigned long
- float

7.1.5

We have a variable Measure with a value of 5. Will the following statement write a message "Insufficient value"?

```
if ((Measure > 0) and (Measure % 2 == 1) and (Measure <=10))
  cout << "Measure is OK";
else cout "Insufficient value";
```

- False
- True

7.1.6

We have the following code:

```
int Sum = 0, Current;
cin >> Current;
if (Current % 5 == 0)
   Sum += Current;
   Current++;
cout << "Sum is: " << Sum << ", Current is: " << Current <<
endl;
```

The input value was 11. What appears on the screen when we launch this code?

- Sum is: 0, Current is: 10
- Sum is: 11, Current is: 10
- Sum is: 0, Current is: 12
- Sum is: 11, Current is: 11

7.1.7

Sometimes it is necessary to branch a program to more than two branches. To do this we can use the **if** statement within a branch of another **if** statement.

For example, we need to determine whether a numeric variable X is positive, negative or zero. We can write:

```
if (X == 0) cout << "X is zero" << endl;
else // variable X is positive, or negative - we have to test
it:
   if (X < 0) cout << "X is negative" << endl;
   else cout << "X is positive" << endl;
```

In the same way, we can branch to even more branches. We can write additional if statements.

In order to see in the source text which branch belongs to which command, we use indentation as shown in the previous example.

7.1.8

Select a command that detects a larger value from the two specified integer variables Number1 and Number2.

- if (Number1 > Number2) cout <|<| "Number1 is greater than Number2" <|<| endl; else if (Number1 <| Number2) cout <|<| "Number2 is greater than Number1" <|<| endl;
- if (Number1 == Number2) else if (Number1 > Number2) cout <|<| "Number1 is greater than Number2" <|<| endl; else cout <|<| "Number2 is greater than Number1" <|<| endl;
- if (Number1 <| Number2) cout <|<| "Number2 is greater than Number1" <|<| endl; else if (Number1 != Number2) cout <|<| "Number1 is greater than Number2" <|<| endl;
- if (Number1 == Number2) cout <|<| "Number1 is greater than Number2" <|<| endl; else if (Number1 > Number2) cout <|<| "Number2 is greater than Number1" <|<| endl;

7.1.9

Assume a variable Distance which contains a distance of two stops on the railway. We need to calculate fares when we know that up to 10 km is paid 1 €, from 10 to 20 km 2 €, from 20 to 40 km 3 € and from 40 to 100 km 5 €. For distances of over 100 km, 3 € per 100 km is charged.

- if (Distance <| 10) cout <|<| "Fare is 1 €"; else if (Distance <| 20) cout <|<| "Fare is 2 €"; else if (Distance <| 40) cout <|<| "Fare is 3 €"; else if (Distance <| 100) cout <|<| "Fare is 5 €"; else cout <|<| "Fare is " <|<| (((Distance / 100) + 1) * 3) <|<| " €";

- if (Distance > 0) cout <|<| "Fare is 1 €"; else if (Distance >= 10) cout <|<| "Fare is 2 €"; else if (Distance >= 20) cout <|<| "Fare is 3 €"; else if (Distance >= 40) cout <|<| "Fare is 5 €"; else cout <|<| "Fare is " <|<| (((Distance / 100) + 1) * 3) <|<| " €";
- if (Distance <| 100) cout <|<| "Fare is 5 €"; else if (Distance <| 40) cout <|<| "Fare is 3 €"; else if (Distance <| 20) cout <|<| "Fare is 2 €"; else if (Distance <| 10) cout <|<| "Fare is 1 €"; else cout <|<| "Fare is " <|<| (((Distance / 100) + 1) * 3) <|<| " €";
- if (Distance <|= 10) cout <|<| "Fare is 1 €"; else if (Distance <|= 20) cout <|<| "Fare is 2 €"; else if (Distance <|= 40) cout <|<| "Fare is 3 €"; else if (Distance <|= 100) cout <|<| "Fare is 5 €"; else cout <|<| "Fare is " <|<| (((Distance / 100) + 1) * 3) <|<| " €";

7.1.10

The worker worked X hours. Norma's 40 hours. The worker receives a basic salary of €300 for meeting the standard. For a 20% overrun, he will receive a €50 bonus, but salary is reduced by a €100 for normas non-compliance. Calculate the salary of a worker who has worked for X hours. Fill in following code the appropriate pieces:

```
if ( _____ ) cout << "Salary is _____";
  else if ( _____ ) cout << "Salary is €300";
    else cout << "Salary is €350";
```

- X == 40
- €350
- X < 40*1.2
- €300
- €200
- X < 40

7.1.11

The teacher works especially with very good students on one hand and with very weak students on the other. The student is graded A to E. The teacher, therefore, needs to know if the student is very good (A or B) or weak (he has an E). Fill in the corresponding parts to the following code.

```
_____ Grade;
cin >> Grade;
if _____ cout << "Excellent student";
  else if _____ cout << "Weak student";
```

- (Grade >= 'A')
- char
- (Grade != 'C' and Grade != 'D')
- (Grade == 'E')
- (Grade == 'A' or Grade == 'B')
- int
- float

7.2 Command if (programs)

⌨ 7.2.1 Conditional statement 1

If you buy at least 100 €, you get a 5% discount. For a purchase of at least 200 €, you get a 10% discount and 20% off for purchases over 400 €. Calculate the discount amount for the amount you entered.

condstat.cpp

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

int main(){
    float Amount, Discount;
    cin >> Amount;
    // Enter appropriate code here:

    cout <|<| "Discount is " <|<| Discount <|<| endl;
    return 0;
}
```

⌨ 7.2.2 Conditional statement 2

The size of the mortgage loan is at most 80% of the property price and the applicant must earn at least 1500 € per month. If the applicant is at most 35 years old, he can get up to 90% of the property price and earn at least 1000 € a month. Otherwise, he won't get any credit. On input are the cost of the property, earnings and the applicant's age. Calculate the maximum amount of credit that this applicant will receive.

⌨ 7.2.3 Conditional statement 3

We have a math function: (y = sqrt{frac{x-1}{x^2-3}})

Task: Calculate the value of this function for given x on input. Display result value with 4 decimal places. In case the function doesn't have a solution in the range of real numbers, write the message "The function has no solution for given x."

⌨ 7.2.4 Conditional statement 4

We have three coefficients A, B, C of a quadratic equation (Ax^2+Bx+C=0). Calculate the roots of this equation in real and complex ranges. For any type of result write the following comments:

1. The solution has two real roots: xxxxx and yyyyy.
2. The solution has one double root: xxxxx.
3. The solution has complex roots xxxxx +/-i yyyyy.

All result numbers write with four decimal places.

⌨ 7.2.5 Conditional statement 5

To pass the credit, the student has three tests, 20 points for each. None of them can have zero points. If all of them meet at least 80%, they will not only have the credit, but also the exam for A. The input values are three numbers indicating the number of points obtained by the student in each test. The outputs are sentences corresponding to the evaluation in individual cases:

1. The student does not obtain credit.
2. The student obtains credit.
3. The student obtains credit and exam for A.

⌨ 7.2.6 Conditional statement 6

There are two names at the input. Write down which is the first in alphabetical order.

7.2.7 Conditional statement 7

There are three numbers at the input. Write these numbers in ascending order.

7.2.8 Conditional statement 8

If the driver is driving at a speed of 10 km / h higher than allowed, he gets a fine of 20 €. If he is driving at a speed of 20 km / h higher than allowed, he will receive a fine of 50 €. Input has two numbers – allowed speed and actual speed. Write out whether the driver will be punished and how. There will be a sentence at the output: The driver will not be fined. / The driver will be fined ... €.

7.3 Conditional expression

7.3.1

If the result of the branching is only the calculation of an expression, it is possible to use rather simple writing instead of the **if** statement.

For example, we need to calculate fares based on distance travelled. From 0 to 19 km is paid 1 €, over 20 km is paid 2 €. We can write the calculation with the **if** statement:

```
if (Distance < 20) Fare = 1;
else Fare = 2;
```

We can write the same with a **conditional expression**:

```
Distance < 20 ? Fare = 1 : Fare = 2;
```

or even more briefly

```
Fare = Distance < 20 ? 1 : 2;
```

Operator "?" and ":" is often called **ternary operator** – it has three operands: condition, first expression and second expression.

7.3.2

We have two pieces of code:

First:

```
if (SideA <= SideB) SideA *= 2;
else SideB *= 2;
```

Second:

```
SideA = SideA <= SideB ? SideA * 2 : SideB * 2;
```

Do both codes the same thing?

- False
- True

7.3.3

We need to insert a square content or a square perimeter into the *X* variable, according to whichever amount is greater. The square side is in the Side variable. Select the correct code that does this.

- X = (Side * Side > 4 * Side) ? Side * Side : 4 * Side;
- X = Side * (Side > 4 ? Side : 4);
- X = (Side * Side <| 4 * Side) ? Side * Side : 4 * Side;
- X = (Side * 2 <| 4 * Side) ? 2 * Side : 4 * Side;

7.3.4

We need an expression that returns the initial value of Count and adds one to the Count value if the Current variable is negative. Fill in the blanks with the corresponding code:

```
_____ ? _____ : _____ ;
```

- Count -=1
- -Current <| 0
- Count
- Count += 2
- (Current >= 0
- Count++
- Count = 1
- Current < 0

7.3.5

We have the following code:

```
if (Sum > 1000) { Count ++; Sum -= Current; }
else {Count --; Sum += Current;}
```

Which of the pieces of code does exactly the same action?

- Count = Sum > 1000 ? Count + 1 : Count - 1; Sum = Sum > 1000 ? Sum - Current : Sum + Current;
- Sum = Sum > 1000 ? Sum - Current : Sum + Current; Count = Sum > 1000 ? Count + 1 : Count - 1;
- Count = Sum <| 1000 ? Count ++ : Count --; Sum = Sum <| 1000 ? Sum - Current : Sum + Current;
- Sum += Sum > 1000 ? -Current : Current; Count += Sum > 1000 ? 1 : -2;

7.3.6

Rewrite the following code using the **if** statement:

```
Speed += Lap < 100 ? CurSpeed : ;
Lap += Lap < 100 ? 1 : 0;
```

- if (Lap <| 100) { Speed += CurSpeed; Lap++; }
- if (Lap > 100) Lap--; if (Lap <| 100) Speed = CurSpeed; else Speed = -CurSpeed;
- if (Lap <| 100) Speed = Speed + CurSpeed; Lap = Lap + 1;
- if (Lap <| 100) else {Speed += CurSpeed; Lap--;}

7.3.7

When calculating the arithmetic expression, we have to check whether the input values allow the calculation to be performed. Calculate the value of the expression log10 ((a2 −√b) / (a − b)). Input values are *a* and *b*.

Fill in the following code the appropriate pieces:

```
float a, b, result;
cin >> a >> b;
if _____ {
   result = _____ ;
   if _____ {
```

```
      result = log10(result);
      cout << "The result is: "<<result<<endl;
   }
   else cout << "The expression cannot be evaluated."<<endl;}
else cout << "The expression cannot be evaluated."<<endl;
```

- (log10(result) != 0)
- log10((a*a - sqrt(b)) / (a - b))
- (b >;= 0) or (a == b)
- (a*a - sqrt(b)) / (a - b)
- ((a - b != 0) and (b >=0))
- (result > 0)

7.3.8

Determine whether the entered numbers *x* and *y* satisfy the equation *y*=34 /(*x*+12).

```
float x, y, fraction;
string Message;
cin >> x >> y;
if (x + 12 != 0)
   Message = *** ? "Numbers are satisfying the equation. " :
         "Numbers aren't satisfying the equation.";
   else Message = "The fraction cannot be evaluated.";
cout << Message << endl;
```

Choose the right part of the code that needs to be replaced by three stars.

- (y == 34 / (x + 12))
- (y * x - 12) == 34
- (y == (34 / x + 12))
- (x + 12 != 34 / y)

7.4 Conditional expression (programs)

7.4.1 Ternary operator 1

If you buy at least 100 €, you get a 5% discount. For a purchase of at least 200 €, you get a 10% discount and 20% off for purchases over 400 €. Calculate the discount amount for the amount you entered.

ternop.cpp

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

int main(){
    float Amount, Discount;
    cin >> Amount;
    // Enter appropriate code here:

    cout <|<| "Discount is " <|<| Discount <|<| endl;
    return 0;
}
```

⌨ 7.4.2 Ternary operator 2

The size of the mortgage loan is at most 80% of the property price and the applicant must earn at least 1500 € per month. If the applicant is at most 35 years old, he can get up to 90% of the property price and earn at least 1000 € a month. Otherwise, he won't get any credit. On input are the cost of the property, earnings and the applicant's age. Calculate the maximum amount of credit that this applicant will receive. Use ternary operator instead of if statement.

⌨ 7.4.3 Ternary operator 3

We have a math function: (y = sqrt{frac{x-1}{x^2-3}})

Task: Calculate the value of this function for given x on input. Display result value with 4 decimal places. In case the function doesn't have a solution in the range of real numbers write value -100.0000. Use the ternary operator to calculate the conditions.

⌨ 7.4.4 Ternary operator 4

There are three numbers at the input. Write these numbers in ascending order. Use the ternary operator.

⌨ 7.4.5 Ternary operator 5

If the driver is driving at a speed of 10 km/h higher than allowed, he gets a fine of 20 €. If he is driving at a speed of 20 km/h higher than allowed, he will receive a fine of 50 €. Input has two numbers – allowed speed and actual speed. Write out whether the driver will be punished and how. There will be a sentence at the output: The driver will not be fined. / The driver will be fined ... €.

Use the ternary operator.

7.5 Switch to more branches

🕮 7.5.1

In some cases, branching into multiple branches can be written with the **switch** command. This command tests the value of an integer or enumerated expression and then decides which branch the program control moves to.

Syntax of **switch** command is:

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

The **break** statement always terminates the execution of the entire **switch** statement. If we do not specify a **break**, the **switch** statement continues to execute the next branch, even if it is a branch whose value does not match the evaluated expression. The **default** branch is executed if neither value matches the enumerated expression. The **default** branch may be omitted.

The **break** statement will also be discussed later.

📝 7.5.2

Suppose the following code:

```
char MyCharacter;
cin >> MyCharacter;
```

```
switch (MyCharacter) {
  case 'a': cout << MyCharacter;
  case 'e': cout << MyCharacter;
  case 'i': cout << MyCharacter;
  case 'o': cout << MyCharacter;
  case 'u': cout << MyCharacter;
  case 'y': cout << MyCharacter;
  default: cout << "+any consonant."
}
```

The "e" character was entered. What will be displayed on the screen after this code is executed?

- eeeee+any consonant.
- e+any consonant.
- eiouy
- eiouy+any consonant.

7.5.3

We need to quantify integer powers to the fifth degree. Variable Base contains a powered value, variable Power contains the required power of variable Base. We use the **switch** command. Which listing resolves the specified task?

- switch (Power){ case 0: result = 1; break; case 1: result = Base; break; case 2: result = Base*Base; break; case 3: result = Base*Base*Base; break; case 4: result = Base*Base*Base*Base; break; case 5: result = Base*Base*Base*Base*Base; break; default: result = 0; }
- result = Base; switch (Power){ default: result = 0; break; case 0: result = 1; break; case 5: result *= Base; case 4: result *= Base; case 3: result *= Base; case 2: result *= Base; case 1: ; }
- switch (Power){ case 0: result = 1; break; case 5: result *= Base; break; case 4: result *= Base; break; case 3: result *= Base; break; case 2: result *= Base; break; case 1: ; default: result = 0; }
- result = Base; switch (Power){ case 0: result += 1; case 1: result += Base; case 2: result += Base*Base; case 3: result += Base*Base*Base; case 4: result += Base*Base*Base*Base; case 5: result += Base*Base*Base*Base*Base; default: result += 0; }

7.5.4

We have two variables: Op1 and Op2. We want to process some arithmetical operations depending on variable Choice which contains one character specified

the operation: '+' for addition, '-' for subtraction, '*' for multiplication, or '/' for division. Fill in the following code the appropriate pieces:

```
int Op1=24, Op2=8;
char Choice;
cin >> Choice;
switch _____ {
  case '+': cout << "Result is: "<< _____
  case '-': cout << "Result is: "<< Op1 - Op2 << endl; break;
  case '*': cout << "Result is: "<< Op1 * Op2 << endl; break;
  case _____
  default: cout << "Unresolved operation." << endl;
}
```

- Op1 ++ Op2 <|<| break;
- '/': cout <|<| "Result is: "<|<| Op1 / Op2 <|<| endl; break;
- '/': cout << "Result is: "<< (Op2 !=0 ? Op1 / Op2:0) << endl; break;
- (Choice)
- Choice
- Op1 + Op2 << endl; break;

7.5.5

One character is entered from the input. You need to decide whether it's a vowel, a consonant, or a different character (not a letter). Fill in appropriate pieces of code:

```
char Character;
cout << "Enter any character: ";
cin >> Character;
Character = toupper(Character);  // converts letter to upper
case
if _____
  switch _____ {
   case 'A':
   case 'E':
   case 'I':
   case 'O':
   case 'U':
   case 'Y': cout << "The '"<<Character<<"' was a
vowel."<<endl; break;
   default: _____
   } else _____
```

- cout <|<| "The '"<|<|Character<|<|"' was a digit."<|<| endl;

- (Character is 'A' to 'Z')
- ('A' to 'Z')
- ((Character >='A') and (Character <= 'Z'))
- (Character)
- cout << "The '"<<Character<<"' wasn't a letter." << endl;
- ('A' <|= Character <|= 'Z')
- cout << "The '"<<Character<<"' was a consonant." << endl;

7.6 Switch command (programs)

⌨ 7.6.1 Switch application 1

Value-added tax is calculated according to the category to which the relevant goods belong. The categories are marked 1 (5% tax), 2 (10% tax), 3 (15% tax), and any other number represents a basic tax of 21%. On the standard input are two numbers: the tax category and the tax-free price of the goods. Calculate the price of goods with tax.

⌨ 7.6.2 Switch application 2

There is an integer value at the input and the required output after it: 0 is in decimal, 1 in octal, and 2 in hexadecimal. Write the entered number in the desired system.

Loops

Chapter 8

8.1 The while statement

🕮 8.1.1

Program loops represent the basis of almost all programs. Most algorithms are based on loop value processing. All loops can be divided into two groups: **conditional loops**, **counted loops**. Conditional loops are further divided into loops with a condition at the beginning, a condition at the end, and a condition in the middle. So we have four types of loops. First, we deal with conditional loops.

What is a loop exactly?

A loop is a few commands to be executed repeatedly but not forever. We must always determine when commands are to be again repeated and when the repetition is to end. For conditional loops, the point at which the repetition ends is determined by fulfilling a condition. For counted loops, the number of repetitions is determined in advance.

So conditional loop has a condition, i.e. logical expression that controls the repetition. A poorly determined condition may cause the loop to be performed forever (looping) or not at all. Looping is a serious bug in program construction.

🕮 8.1.2

A conditional loop has the following common structure:

- start of the loop,
- loop body (commands to be repeated),
- end of the loop.

The loop control condition can be placed at the beginning of the loop body or at the end. Loops with control conditions at the beginning or at the end are preferred. The loop with a control condition at the beginning has the following syntax:

```
while (condition) command;
```

The keyword **while** determines the start of the loop. Note the condition is always enclosed in parentheses. The loop body contains one statement only, but this statement may be a compound statement that is composed of many other statements enclosed by curly brackets. The end of this loop is composed of an end of body command.

Example:

```
while (Current > 0) {
   Count++;
   cin >> Current;
}
```

The most important thing is to build the condition correctly. There must be at least one statement in the body of the loop that affects the condition value. In this example, it is the cin command – inserts a value from the input into the Current variable and then controls the loop pass.

If the loop has a condition at the beginning, it is necessary to prepare everything before the loop so that the condition can be evaluated before the loop starts. So we need to input the first value of Current before the loop:

```
cin >> Current;
while (Current > 0) {
   Count++;
   cin >> Current;
}
```

8.1.3

Let's have the following loop:

```
int Count = 100;
while (Count > 50)
   cout << "Round No. " << Count-- << endl;
```

Is the loop condition built correctly to avoid an endless loop?

- True
- False

8.1.4

Often we need to process the numbers entered from the input. There are several ways that a job is given:

1. We know which value is the last; this last value may be part of the data or it may be a breakpoint (stop value) that we can't process with other values.
2. We know the count of input numbers.
3. We don't know their number nor do we know which value is the last.

Each of these cases has a somewhat different loop design that is able to read and process the entered numbers.

We start with the simplest situation, that is when we know the last number that is not a part of the data and shouldn't be processed together with the data.

8.1.5

There is a sequence of numbers on input, representing the prices of goods sold in one day. The last number in the input sequence is zero. We have to find out the total price of the goods sold. Fill in the correct parts to the following code:

```
int Price, Sum = 0;
cin >> Price;
while _____
  {
   _____
   _____
}
cout << "Total price is " << Sum << endl;
```

- Sum += Price;
- (Price != 0)
- cin >> Price;
- (Price == 0)
- Price++
- (Sum != 0)

8.1.6

The conditional **do**-**while** loop has a condition at the end. This means that the loop body will always run at least once. The syntax is as follows:

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

The loop starts with the keyword **do**. If we need more than one statement in the loop body, we use a compound statement. The condition is enclosed by parenthesis and its value true forces to repeat the loop body. Example:

The user has to enter an integer value between zero and 20. If he enters the wrong number, he will be repeatedly prompted to enter the correct value.

```
int Value;
do {
   cout << "Please, insert any number between 0 and 20: ";
   cin >> Value;
} while ((Value <0) or (Value > 20));
```

When this loop is over, we are sure that the Value variable has the correct value.

📖 8.1.7

Now we can return to the numeric input options. We will now deal with the case that we know the last value that is part of the data and should be processed with the data. For this case, a conditional loop of the do-while type is best suited.

Example: Suppose a sequence of real values is prepared at the input and the last value is 42. We have to determine how many values are on the input.

```
float Value;  // variable for input value
int TotalCount = 0; // variable for total count of input
values
do {
   cin >> Value;   // we read one value
   TotalCount++;  // counter of values is increased
} while (Value != 42);  // when input value is not 42, we
continue in the loop
```

Note that the end condition expresses the situation where the loop body is to be repeated again.

We can use this algorithm also if the end value does not belong to the data, but its processing does not affect the result. For example, the input contains integer values, last value is zero. We have to find out the sum of the numbers. In this case, the addition of the numbers with the ending zero does not affect the result.

📝 8.1.8

Let's assume that prepared input values represent the number of students in each exam during the current term. We know that there were exceptionally many students in the last exam – 48. We have to find out how many exam tests were done this term in total. Fill in the appropriate pieces of code:

```
int Exam, Total = 0;
// Exam - number of students in one exam;
```

```
// Total - Total exam tests
do {
   _____
   _____
} while _____
cout << "Total exam tests executed: " << Total << endl;
```

- (Exam != 48);
- cin <|<| Exam;
- (Exam == 48);
- Total ++;
- (Total != 48);
- cin >> Exam;
- Total += Exam;

8.1.9

Input values represent daily revenue from product sales in euros. The last number is zero and this is a sentinel only. Find out what was the biggest revenue in the input data. Fill in appropriate pieces of code:

```
float Revenue, MaxRevenue;
 _____
while _____ {
   _____
   cin >> Revenue;
}
cout << "The biggest revenue was " << MaxRevenue << " €." <<
endl;
```

- if (Revenue > MaxRevenue) MaxRevenue = Revenue;
- cin >> Revenue; MaxRevenue = Revenue;
- MaxRevenue = 99999; cin >> Revenue;
- if (Revenue > MaxRevenue) Revenue = MaxRevenue;
- (Revenue != 0)
- (MaxRevenue > 0)

8.1.10

Another case of input data processing is when we do not know the end value nor the number of input data. So we process the data as long as there is something in the input.

The moment when there is nothing on the input, we find out by testing the reading operation. When reading, the input stream "cin" is set to true if reading succeeds and values are inserted into the appropriate variables, and false otherwise. However, the unsuccessful reading may be another reason than the end of the data, for example, a misspelt numeric value containing illegal characters.

Example: Determine how many real numbers are on input.

```
int Count = 0;
float Current;
while (cin >> Current) Count++;
```

8.1.11

The input contains a sequence of real numbers representing the company's monthly profits. Find out the total profit of the company and write down the smallest positive profit for the given period. Add the appropriate parts to the following program.

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

int main(){
  float Profit, TotalProfit_____ _____;
  while (_____) {
    TotalProfit += Profit;
    if (Profit>0 and Profit>MinProfit) _____;
  }
  cout << "Total profit is " << TotalProfit << endl;
  cout << "The smallest positive profit is "<<
MinProfit<<endl;
  return 0;
}
```

- MinProfit = Profit
- MinProfit = 10000
- = 0,
- =-100
- Profit = MinProfit
- MinProfit = 1e50
- Profit != 0
- ,
- cin >> Profit

8.2 The while loop (programs)

8.2.1 While loop basics

If you buy at least 100 €, you get a 5% discount. For a purchase of at least 200 €, you get a 10% discount and 20% off for purchases over 400 €. Calculate the discount amount for the sequence of amounts you entered. The input sequence ends with a value of 999, which is not data.

8.2.2 While loop 2

A sequence of integer numbers is on input. Calculate the sum of non-negative numbers from the input sequence.

8.2.3 While loop 3

Calculate the average of the real numbers that are on input. You know that the last number is zero.

8.2.4 While loop 4

The average daily temperature is calculated from three measurements (at 7, at 14, at 21 o'clock). The temperature at 21 o'clock is counted twice. The entry is made up of a series of triplets of daytime temperatures, instead of the last triple, there is only one number -100. Calculate and display the average temperature over a given period, which is the average of daily averages. Determine the situation when only -100 is on input.

8.2.5 While loop 5

Every month, the funds deposited in the real estate fund account for a certain monthly percentage which depends on current conditions and may be different each month. We can also deposit a monthly additional amount to the fund. The task is to calculate the resulting amount on your account. The first input value is the initial state of the account, followed by a sequence of pairs of numbers. Each first number indicates the monthly additional deposit and the second number is the percentage of appreciation in the given month. After reading all input values, write down the resulting account status of the given fund. The result value display with two decimal places.

8.2.6 While loop 6

The input is a sequence of numbers representing potato yields per year from one hectare of land (100 kg/ha). Write the serial number of the year in which the highest yield was reached.

8.2.7 While loop 7

The input is a continuous text ending with a dot. Find out how many digits it contains and the percentage of spaces for this text.

while7.cpp

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

int main(){
    // Enter appropriate code here:


    cout.precision(2);
    cout <|<| "Number of digits: "<|<|digits<|<|", spaces: "

<|<|fixed<|<|float(spaces)/float(allchar)*100<|<|"%"<|<|endl;
    return 0;
}
```

8.2.8 While loop 8

The input is a sequence of numbers representing potato yields per year from one hectare of land (100 kg/ha). Write the serial number of the year in which the **highest increase** of yield was reached.

8.2.9 While loop 9

There is one integer number on input. Write this number on output in the binary system including possible negative signs.

⌨ 8.2.10 While loop 10

There is one integer number on the input. Determine if this number is a prime number. Write sentence: "Number ... is a prime number." or "Number ... is not a prime number.".

8.3 The do loop (programs)

⌨ 8.3.1 The do loop 1

Write the number of the positive real numbers that are on input. You know that the last number is zero.

⌨ 8.3.2 The do loop 2

The input is a continuous text ending with a dot. Find out how many letters and digits together it contains.

⌨ 8.3.3 The do loop 3

The sequence of ones and zeroes ended with "=" is on input. Convert this sequence to a decimal value and write it as a result.

⌨ 8.3.4 The do loop 4

If you buy at least 100 €, you get a 5% discount. For a purchase of at least 200 €, you get a 10% discount and 20% off for purchases over 400 €. Calculate the discount amount for the sequence of amounts entered on the input. The input sequence ends with a value of 0 that is not data.

8.4 Counted loop for

🕮 8.4.1

The **for** loop is intended for cases where we know the number of times a loop is repetitive. It is a complement to a conditional loop that is performed by meeting a condition.

The syntax of **for** loop is following:

```
for (initialization; condition; control) statement;
```

There are three expressions in brackets after keyword **for** that determine how the loop body is executed. The loop execution is usually controlled by the value of a numeric variable, which is called a **loop control variable**.

The first expression specifies the initial setting. Usually, the initial value of the control variable is set here. This expression is always calculated **only once before the loop starts**. We can also use there the declaration of a control variable that will only function inside the loop body and will be automatically deleted when the loop ends.

The second expression is **always evaluated before each execution** of the loop body. It is a condition whose fulfilment allows the execution of the loop body. If this condition isn't met, the loop is terminated.

The third expression is used to update the value of the control variable. It is automatically executed **at the end of the loop body**. We can add or subtract one or any other value (also float).

Simple example: We want to display numbers from 10 to 20:

```
for (int CV = 10; CV <= 20; CV++) cout << CV << endl;
```

We can omit the initialization expression if the initialization was performed earlier:

```
int CV = 10;
for ( ; CV <= 20; CV++) cout << CV << endl;
```

If we omit the second expression, the loop will be executed forever (infinite loop). In that case, there is no point in mentioning the third expression.

The expression can contain more commands divided by commas. For example, we can have two variables used in the loop body:

```
for (int a=1, b=3; a*b < 100; a+=3, b+=4)
  cout << a << ", " << b << endl;
```

The loop body can contain only one command. If we need more commands, we use a compound statement.

8.4.2

We have the following code:

```
int Value, Sum = 0;
for (int C = 1; C < 5; C++)
  {cin >> Value; Sum += Value;}
cout << Sum << endl;
```

Numbers 1 3 5 7 9 11 13 15 were entered when running this program. What was the output value?

- 16
- 21
- 7
- 5

8.4.3

Looking at the three expressions that need to be written in the **for**-loop bracket, it is basically the same thing we would have to do with the **while** loop. So what is the difference between using a **for** and while **loop**? If we want to write pure code, we will use the **for** loop just if we know in advance how many times it will be repeated. In all other cases, we use the **while** loop.

The **for** loop can be easily rewritten to a **while** loop. For example the following loop:

```
for (int i = 10; i <= 100; i+=3) {
   cin >> current;
   sum += current;
   cout << "We processed " << i << " values."<< endl;
}
```

can be rewritten to:

```
{int i = 10;
while (i <=100) {
  cin >> current;
  sum += current;
  cout << "We processed " << i << "values."<< endl;
```

```
  i+=3;
}}
```

Therefore, the **for** loop may seem unnecessary. Its benefit is that in the case of a known number of repetitions, we have all three essential elements of the loop (initialization, repetition condition, modification of the control variable) in one place in the parentheses.

Similarly, you can rewrite a **while** loop to a **for** loop. For example, to read a sequence of numbers ending with -1 and summing up the values, we can write:

```
cin >> cur;
while (cur != -1) {
   sum += cur;
   cin >> cur;
}
```

And in case of for loop we write:

```
for (cin >> cur; cur != -1; cin >> cur) sum += cur;
```

We strongly warn against various dirty tricks, such as modifying the control variable inside the **for** loop. For example:

```
for (char X = 'A'; X <= 'Z'; X ++) {
   cout << "We processed character: " << X << endl;
   if (X == 'P') X += 5;
}
```

Looking at the beginning of the loop, we see the repetition for all the characters of the uppercase alphabet, but 5 characters are skipped inside the loop.

8.4.4

We need to calculate the factorial of an input number N. The input number is non-negative. Which code solves this task?

- int N, F = 1; cin >> N; for (int I = N; I > 1; I--) F *= I;
- int N, F = 0; cin >> N; for (int I = N; I >= 1; I--) F += I;
- int N, F = 1; cin >> N; for (int N = I; I <| N; N--) F *= N;
- int I = 1, F = 0; for (cin >> N; I > 1; I--) F *= I;

8.4.5

We have a function y = (3 − x) / (x^2−2x + 1).

We need to list a table of functional values for x going from 1 to 2 with a step of 0.05. Fill in matching pieces to the following code.

```
float denom;
for ( _____ ){
   denom = _____ ;
   cout << x << " " << _____ << endl;
}
```

- (denom != 0 ? (3 - x)/denom : "--")
- int x=1; x <|= 2; x++
- x^2 - 2x +1
- (denom == 0 ? 3 - x / denom : 0)
- x * x - 2 * x + 1
- float x = 1; x <= 2; x += 0.05

8.4.6

Rewrite the following while loop to the equivalent for loop (choose the right equivalent):

```
int result = 1, coef = 1; cin >> inp;
while (coef <= inp) {
   result *= coef;
   coef ++;
}
```

- int result = 1; cin >> inp; for (int coef = 1; coef <|= inp; coef ++) result *= coef;
- int coef = 1; cin >> inp; for (int result = 1; result <|= inp; result ++) result *= coef;
- int result, coef; cin >> inp; for (coef = inp, result = 1; coef >= 1; coef --) result *= coef;
- int result = 1, coef; cin >> inp; for (coef = 1; coef >= 1; coef ++) result *= coef;

8.5 The for loop (programs)

⌨ 8.5.1 The for loop 1

If you buy at least 100 €, you get a 5% discount. For a purchase of at least 200 €, you get a 10% discount and 20% off for purchases over 400 €. Calculate the discount amount for the sequence of amounts you entered. The input sequence begins with an integer that represents the number of values entered.

⌨ 8.5.2 The for loop 2

A sequence of N integer numbers is on input. The N is the first value on input. Calculate the sum of non-negative numbers from the input sequence.

⌨ 8.5.3 The for loop 3

Calculate the average of the real numbers that are on input. You know that the first value on input is a number of input values.

⌨ 8.5.4 The for loop 4

There are two N and K values at the input. Calculate the value of the combination number (N over K). Note that the combined number is calculated as $N!/((N-K)!\,K!)$

⌨ 8.5.5 The for loop 5

There is a non-negative integer on input. Display this value in a binary system at 32 digits.

⌨ 8.5.6 The for loop 6

An arithmetic sequence is specified at the input: the first member (integer value), the difference (integer) and the required number of members. Write the members of the sequence so that there are comma and one space between the members, and a dot after the last member. Do not write a new line at the end.

⌨ 8.5.7 The for loop 7

A geometric sequence is specified at the input: the first member (real value), the quotient (real) and the required number of members. Write the members of the sequence so that there are comma and one space between the members, and a dot after the last member.

8.6 Affecting the passage through the loop

🕮 8.6.1

The **break** command allows you to end the loop. This means that anywhere within the loop body, it is possible to jump behind the loop and avoid further repetitions.

The **break** statement has already been mentioned by the switch statement. Its meaning was the same – to terminate the command and jump beyond its end.

The **break** command is one of the program jump commands. The principles of good programming practice do not allow the jump command because it always interferes with the clarity of the program.

Thus, the **break** statement should be always avoided. It is usually seen as a dirty trick. It presents potential bugs and makes looping unclear. If we write a conditional loop, it should be clear from its condition when the loop works and when exactly it will stop repeating. Similarly, for the **for** loop, we should determine how many times the loop body is repeated from the notation in parentheses. In both cases, the existence of a break statement in the loop body completely disrupts this information.

Example:

Suppose we are, to sum up, the sequence of the input values ending with zero. However, it may happen that the sum exceeds 10,000, in which case we should stop reading and addition. We can use "dirty trick" with the break command:

```
int Sum = 0, Value;
do {cin >> Value;
   if (Sum >= 10000) break;
   Sum += Value;
} while (Value != 0);
```

As you can see, the **break** command divides the body of the loop into two parts – the part before and after it. The body part before the **break** statement can be executed less than the part after it. This is very dangerous and annoying, for example, when looking for errors, because with a longer body of the loop, it may not

be obvious that some commands in the body of the loop were not executed due to the jump beyond the end of the loop.

However, we can also write the correct loop condition and avoid the annoying break statement:

```
int Sum = 0, Value;
do {Sum += Value;
    cin >> Value;
} while (Value != 0 and Sum <= 10000);
```

8.6.2

We want to calculate the factorial of a given number N. We have allocated an int variable for the result. However, it is possible that during the calculation it is found that the result exceeds the capabilities of this variable. The first variant of the program is based on the factorial calculation using the **for** loop. We add the condition to terminate the calculation if the result variable is going to overflow (it is over circa 400 million).

```
int N, result = 1;
cin >> N;
for (int i = 2; i<=N; i++) {
   if (result > 400000000) break;
   result *= i;
}
```

Rewrite the code without using the break command.

- int N, result = 1; cin >> N; for (int i = 2; i<|=N and result <| 400000000; i++) result *= i;
- int N, result = 1; cin >> N; for (int i = 2; (i>N or result <| 400000000; i++) result *= i;
- int N, i, result; cin >> N; for (i = result = 1; result <| 400000000; i++) result *= i;
- int N, i=2, result = 1; for (cin >> N; N<|=400000000; i++) result *= i;

8.6.3

The **continue** command causes the loop body to end and jump to a new loop. Similar to the undesirable use of the **break** statement, the **continue** statement is equally undesirable.

Example: We should sum the values of the input numbers ending with zero. However, if there are 10 between the numbers, we should not include it in the sum.

```
int Cur, Sum = 0;
cin >> Cur;
while (Cur != 0) {
   if (Cur == 10) {cin >> Cur; continue;} // jump to next loop
   Sum += Cur;
   cin >> Cur;
}
```

Similar to the **break** command, the loop body is divided into two parts by the **continue** statement. The part before the **continue** statement is always executed, while the part after the **continue** statement is executed only if the condition is not met. This can cause unwanted effects because the loop body commands are not executed the same number of times.

The **continue** statement can always be avoided by using a more understandable structure, such as:

```
int Cur, Sum = 0;
cin >> Cur;
while (Cur != 0) {
   if (Cur != 10) Sum += Cur;
   cin >> Cur;
}
```

8.6.4

We have the following code:

```
int Value, Func;
bool Test;
for (Value = 0; Value <= 100; Value ++){
   Test = Value % 10 == 0;
   if (Test) continue;
   Func = Value * Value - 2 * Value + 1;
   cout << Value << "   " << Func << endl;
}
```

How many lines are displayed on the output?

- 90
- 101
- 100
- 91

8.6.5

We have the following code:

```
int Value, Func;
bool Test;
for (Value = 0; Value <= 100; Value ++){
   Test = Value % 10 == 0;
   if (Test) continue;
   Func = Value * Value - 2 * Value + 1;
   cout << Value << "   " << Func << endl;
}
```

How can we rewrite this code to perform an identical action but not include the **continue**?

- int Value, Func; for (Value = 0; Value <|= 100; Value ++){ if (Value % 10 != 0) cout <|<| Value <|<| " " <|<| Value * Value - 2 * Value + 1 <|<| endl; }
- int Value, Func; bool Test; for (Value = 0; Value <|= 100; Value ++){ Test = Value % 10 == 0; if (Test) { Func = Value * Value - 2 * Value + 1; cout <|<| Value <|<| " " <|<| Func <|<| endl; } }
- int Value, Func; bool Test; for (Value = 0; Value <|= 100; Value ++){ Test = Value % 10 == 0; if (!Test) Func = Value * Value - 2 * Value + 1; cout <|<| Value <|<| " " <|<| Func <|<| endl; }
- int Value, Func; for (Value = 0; Value <|= 100; Value ++){ if (Value % 10 == 0) { Func = Value * Value - 2 * Value + 1; cout <|<| Value <|<| " " <|<| Func <|<| endl; } }

8.7 Loop damage (programs)

8.7.1 Loop damage 1

Employee age data are input. It is necessary to calculate the average age of the specified group. However, if an incorrect value appears in the data (a negative number, a number greater than 200), only a warning message "Incorrect input data." will be displayed to standard error output. Determine missing input data and display a warning message "Missing input data." to standard error output.

Avoid the command **break** in the program solution.

8.7.2 Loop damage 2

Employee age data are input. It is necessary to calculate the average age of the specified group. However, if an incorrect value appears in the data (a negative number, a number greater than 200), this value can't be processed.

Consider the variant using the **continue** command but write down the target solution without this command.

User-Defined Functions

Chapter 9

9.1 User-defined functions

🕮 9.1.1

Each program consists of smaller parts called subroutines. Subroutines are defined as logically integrated parts of a program that have a clearly defined activity and communicate with the environment in a known manner. For example, the subroutine is a log function or the main function.

Subroutines can be logically divided into functions and procedures. A **function** is a subroutine that returns a value and that is processed further. A **procedure** is a subroutine that does not result in a single value, but the result is a processed algorithm.

In C / C ++ language, procedures and functions are technically defined in the same way, differing only in expressing the output value. It is often said that there are only functions.

In addition to the procedures and functions that are already done and can only be used, the programmer can create his own. The definition of own subroutine is as follows:

```
<type> <identifier> (<parameters>) { <body> }
```

The type is any data type except array (see later). The identifier is the name of the function. Parameters are given in parentheses, these are the variables that the function communicates with the environment. The curly brackets list the commands that prescribe what a function/procedure does.

The function data type determines the type of value that the function calculates and returns. It's called a **return value**. If we want to define a procedure, we will use a special type of **void**.

There is a special **return** statement inside the function body. This command terminates the function and defines the return value. A function that returns a value must have that statement. A procedure that does not return a value does not contain a return statement. A return statement is similar to a **break** statement that forcibly terminates a cycle. The return statement forcibly terminates the execution of function statements. For the sake of clarity and clarity of the function body, the **return** statement should always be written as the last command in the function body.

A small example: We will declare a function that returns a less value from its two parameters:

```
int min(int a, b) {
   if (a < b) return a;
```

```
   else return b;
}
```

We see that the return value of the function is of the **int** type, the function is called min and has two parameters of type **int**: a and b. In the function body, it is determined which value of the two parameters is smaller, this value is then used as the output value using the **return** statement.

If we want to use this feature, we'll call it in any expression, such as:

```
int first, second;
cin >> first >> second;
cout << "The smaller value is: " << min(first, second) << "."
<< endl;
```

9.1.2

The function definition for calculating cylinder volume with radius R and height V. Fill in the corresponding part to the following code:

```
_____ Volume _____
 _____
```

- float
- (float R, float V)
- Result = M_PI*R*R*V;
- {float R; float V}
- int
- {return R*R*M_PI * V;}

9.1.3

The subroutine parameters can be imagined as variables to be used in the subprogram body for processing and calculation. These parameters are called **formal parameters**. When the subroutine is called up, the specific values with which the subroutine is to work are put in place. These parameters are called **actual parameters**.

At the time of the subroutine call, the actual parameter values are **copied** to the formal parameter locations. This process is called a **value call**. If we change the value of a parameter within the subroutine body, the actual value does not change.

If the function does not have any parameters, we must write empty parentheses in the definition and in the call too. However, the parameters express the subroutine

communication with the environment. So if we don't specify any parameters, it's suspicious because it's not clear how the subroutine will communicate.

9.1.4

We want to do a subroutine that lists a small multiplication table. Fill in the appropriate pieces into the following code:

```
_____ Multi _____ {
  for (int x=1; x<=10; x++) {
     for (int y=1; y<=10; y++) cout << setw(6) << x*y;
     _____
  }
}
```

- float
- (int x, y)
- x=1; y=1;
- ()
- void
- cout << endl;
- int

9.1.5

We have an integer variable X. Then we define a function in which we have the variable X again. Let's look at the following code of the whole program:

```
#include <iostream>
using namespace std;
int MyFunc(int a, int b) {
  int X;
  for (X=1; X<=b; X++) a *= b;
  return a;
}
int main() {
  int X=5;
  cout << MyFunc(2, 3) << endl;
  cout << "Variable X = " << X << endl;
  return 0;
}
```

What will appear on the output? The correct answer is:

```
54
Variable X = 5
```

The value of 54 is the multiplication of 2 * 3 * 3 * 3. The value of 5 is the original value of the variable X that was declared in the main program. The variable X, which is declared inside the MyFunc function, is called **local** because it applies only within this function. Because it is named the same as the X variable that is declared in the main program, it covers it and the X variable from the main program becomes unavailable in MyFunc.

The term local variable is called a variable that is defined in a given function. However, we can work with all the variables that are defined in the parent functions unless they have the same names as the local variables. Then we call such variables global.

9.1.6

We have the following code:

```
int a = 1;
float RealFunc(int a) {
  return a * 0.01;
}
int main() {
  int a = 10;
  cout << RealFunc(a) << endl;
  return 0;
}
```

What will be displayed after executing this code?

- 0.1
- 0.01
- 1
- 10

9.1.7

Among all subroutines, the **main function** plays a special role. It represents the main program, that is, what is run after compilation into machine code and what it

communicates with the operating system. Like all other subroutines, the main function can communicate through parameters and return a value.

What are the actual parameters of the main function and what does it return?

After the program is compiled, an executable code is created. This can be started by calling from the command line. Like other commands, a compiled program can accept parameters from the command line, ie, strings written on the command line after the executable file name.

For this reason, the main has **two parameters** in the following order: the first is an integer and tells how many parameters have been specified from the command line. The second is a pointer to the strings (to be discussed later) and allows you to work with the values of each parameter from the command line. Example:

```
int main (int NumPar, char** ArrayPar) {
   cout << "This program was executed with "
        << NumPar << " parameters." << endl;
   return 0;
}
```

After compilation and execution from the command line, the program writes out how many parameters were specified at the command line startup. We can work with the values of individual parameters using a string array (to be discussed later).

The first parameter is always the name of the program itself, including any access path. For example, we can ask from within the program what directory the program was started from. It also follows that the number of parameters is always at least 1.

The **return value** of the main function returns to the operating system and can process this value. The value is stored in an environment variable named "$?" (more detailed information about environment variables depends on the operating system used). It is normal that the output value 0 indicates to an operating system a successful execution of the program, the nonzero value indicates various error states (e.g., no file found, division by zero, etc.).

9.1.8

We have to create a program that decides what it will do according to the number of parameters from the command line. However, if no parameters are specified, it outputs output code 4 as the result, otherwise, it will have output code 0. Fill in the corresponding part to the following code:

```
_____ main _____ {
  if _____ {
   ProcessProgram(Params);
```

```
   return 0;
  } _____
}
```

- void
- (Number>1)
- ;
- (int Number, char** Params)
- (Params!=0)
- else Number=0;
- int
- ()
- (char Params, int Number)
- else return 4;

🕮 9.1.9

A subprogram can be defined in two places. This is called **the forward definition**. First, the header is specified (data type, identifier, parameters and semicolon) and then any time later, the subroutine body is added to the repeated header as-is on the following scheme:

```
float Forwarded(float B, int X);
```

and at any later position

```
float Forwarded(float B, int X){
for (int i=1; i<=X; i++) B *= i;
return i;}
```

For example, we need this property to call subroutines: subroutine A calls the subroutine B and the subroutine B, in turn, calls the subroutine A. This situation cannot be realized without a forwarded definition.

The second situation where we use the forward definition is to insert a subroutine into another subroutine. For example, we have a function to calculate the integral of a real function F. We define this function inside a function for calculating the integral, see example:

```
float Integral(float A, float B){  // function for calculate
the integral
   float result, X;
   float F(float X);  // integrand; forward definition
   result = (F(A) + F(B)) / 2;
   for (X=A; X<=B; X+=0.01) result += F(X);
```

```
   result *= 0.01;
   return result;
}
float F(float X){  // definition of body of forwarded function
   return 2 * sin(X - M_PI * 0.33) + 1.271;
}
```

🕮 9.1.10

Parameters passing

When a subroutine is used, the program activity is transferred to that subroutine and the parameters are passed. This parameter passing can take place in two ways with different effects.

It has already been mentioned above that the actual parameter is copied to the locations designated for formal parameters. This method of passing parameters is called a **value call**.

Thus, the process of passing parameters by value means that at the time the subroutine is called, the passed value of each parameter exists twice – once in the actual parameter and once in the place of the formal parameter. The copy in the formal parameter can then be changed in any way, but the actual parameter value will not be affected. Therefore, calling by value is used for parameters that represent input data for a given subroutine. The real parameter can also be not only a variable but any expression of a given type that is calculated first and then copied to a formal parameter.

The second way of passing parameters is by **calling by reference**. In this case, a copy of the value of the actual parameter is not made, but the **address** of the variable that is in the role of the actual parameter is passed. This means that a formal parameter refers to the same memory location as the actual parameter – that is, like only the variable is renamed. This implies that any manipulation of a formal parameter value within a subroutine is also a manipulation of the actual parameter value.

This mechanism is used if we want to insert the results into the actual parameters, ie for the output values, by the subroutine.

Distinguishing whether it is a parameter called by reference or by value is done by the & character, which is the operator to get an address (reference) in memory. So by writing **&vrbl**, we say that we do not mean the **value** of the variable vrbl but its **address** (reference).

To make matters worse, some data types are passed by reference automatically because they are represented as pointers. For example, the array data type. The pointers and arrays will be discussed later.

Examples:

The procedure reads the input numbers and passes the sum and the number of reading data as a result:

```
void Read(float &Sum, int &Count){
 float Number;
 Sum=0; Count=0;
 while (cin>>Number){Count++; Sum += Number;}
}
```

Functions for moving a king on a chessboard: the position is given by the X and Y coordinates (row and column of the chessboard, values in the range of 1 to 8), the desired direction of movement (one of eight options). If this move can be made (the new field is on the chessboard), the function returns true, otherwise false. The new position will be in parameters X and Y:

```
bool King(int &X, int &Y, int Where){
 switch (Where) {
  case 1: // move to the right
       if (X<8) {X++; return true;} else return false;
  case 2: // move up
       if (Y<8) {Y++; return true;} else return false;
  case 3: // move to the left
       if (X>1) {X--; return true;} else return false;
  // ... all other possible directions analogously
 }
}
```

9.1.11

We have the following program:

```
#include <iostream>
using namespace std;
void Multiply(int A, int &B){
 B = --A * B;
}
int main(){
 int X, Y;
 cin >> X >> Y;
 Multiply(X, Y);
 cout << "X = " << X << ", Y = ", Y << endl;
 return 0;
}
```

There are two numbers on input: 3 and 7. What will appear on output?

- X = 3, Y = 14
- X = 2, Y = 14
- X = 3, Y = 21
- X = 2, Y = 21

9.2 Functions (programs)

⌨ 9.2.1 Functions 1

Create a function that calculates the sin(*x*) value. Calculate the first 10 members of the Taylor series. Compare the result with the function available in the cmath library.

funct1.cpp

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

// enter function "mysinus" here:


int main(){

    double a, f1, f2;
    cout <|<| "  mysinus    library sin   difference"<|<|
endl;
    cout.precision(5);
    for (a=0; a<|=2*M_PI; a+=0.3){
       f1 = mysinus(a); f2 = sin(a);
       cout <|<| setw(9) <|<| right <|<| fixed
       <|<| f1 <|<| setw(11) <|<| f2 <|<| setw(14) <|<| f2-f1
<|<| endl;
    }
    return 0;
}
```

9.2.2 Functions 2

Define a function to determine the prime numbers. Use this function to write first N prime numbers value N is on input.

funct2.cpp

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

// enter function "isprime" here:


int main(){

    int N, count=0, Number=0;
    cin >> N;

    while (count<|N){
      Number++;
      if (isprime(Number)) {
         cout <|<| Number <|<|" ";
         count++;
      }
    }
    cout <|<| endl;
    return 0;
}
```

9.2.3 Functions 3

Write a subroutine that exchanges the values of two real variables in parameters. Use this subroutine in a program that reads three values and lists them in descending order.

funct3.cpp

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

// enter function "myswap" here:
```

```
int main(){

    double A, B, C;
    cin >> A >> B >> C;

    if (A<|B) myswap(A, B);
    if (B<|C) myswap(B, C);
    if (A<|B) myswap(A, B);

    cout <|<| A <|<|", "<|<|B<|<|", "<|<|C<|<| endl;
    return 0;
}
```

9.2.4 Functions 4

Write a function that calculates the factorial value of N. Use a non-recursive variant. If the result exceeds the allowed value range, the calculation result will be zero.

funct4.cpp

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

// enter function "factorial" here:


int main(){

    int Number;
    long long int NFact;

    cin >> Number;
    NFact = factorial(Number);
    cout <|<| "Factorial of "<|<|Number<|<|" is ";
    if (NFact==0) cout <|<| "too big to calculate" <|<| endl;
       else cout <|<| NFact <|<| endl;

    return 0;
}
```

9.2.5 Functions 5

Write a function to find out the number of digits of a given integer. Use this function to determine the total number of digits of the entered sequence of integer values.

funct5.cpp

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

// enter function "NumDigits" here:


int main(){

    long long int Number;
    int Digits=0;

    while (cin >> Number)
       Digits += NumDigits(Number);

    cout <|<| Digits <|<| endl;

    return 0;
}
```

9.2.6 Functions 6

Write a function that reads and verify that the entered personal identification number is correct. Note that personal identification numbers have 10 digits and have to be divisible by 11.

funct6.cpp

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

// enter function "CheckPIN" here:


int main(){

    if (CheckPIN()) cout <|<| "This personal id is correct."
<|<| endl;
```

```
       else cout <|<| "This personal id is incorrect." <|<|
endl;

    return 0;
}
```

9.3 Recursion

🕮 9.3.1

In some tasks, the resulting value is defined by a specific previous value. For example, the value of the 10th member of the geometric series is defined as the value of the 9th member multiplied by the quotient. If we have calculated a previous member with some function, then we can use this function for the next member. So we can write:

$f(x_i)=f(x_{i-1})$

If we programmed the function f, we need to call the same function inside her body, and this case is called **recursion**.

Recursion is a kind of loop. But each function call creates new variables that this function contains, so there is a significant difference between recursion and loop. It is possible to replace the loop with recursion, but sometimes we cannot replace recursion with just a loop.

Evaluation of member of geometric sequence:

```
int StartMember = 2, Quotient = 3;
int Geom(int Member){
   if (Member>1) return Geom(Member - 1) * Quotient;
      else return StartMember;
}
```

Note that, just like a loop, you need to think here about ending a recursive call. So in this example, we have to ask how many members we count. If it is the first member, then we do not recursively call the function, but we assign an initial value only.

9.3.2

We need to sum the numbers entered from the input. The numbers end with zero. We define the task recursively:

$Si=S_{i-1}+c_i$; $S_0=0$

According to the recursive assignment, we compose the program. Fill in the appropriate pieces into code:

```
_____ {
   int Current;
   cin >> Current;
   if _____ return 0;
   else return _____;
}
```

- int Sum()
- (Sum != 0)
- X + Current
- int Sum(int X)
- Sum() + Current
- (Current == 0)

🕮 9.3.3

As mentioned, recursion is a kind of loop. However, the main difference to a loop command is that each time a subroutine is called, its parameters and local variables are stored. If we do not use this effect, the use of recursion instead of an ordinary loop is a really stupid idea.

In every Beginner's Guide, we read that factorial calculation can be done by the recursive call. Considering the common mathematical definition of factorial:

$n!=n\cdot(n-1)!$ *and* $1!=1$

then it is natural to program the calculation using a simple recursive function:

```
int Fakt(int N){
   if (N>1) return N * Fakt(N-1);
   else return 1;
}
```

However, such a function is an expression of a totally erroneous approach with a waste of memory resources. In such cases, the use of an ordinary loop is incomparably more efficient.

However, if we use recursive calls to store local variables, we get very efficient writing and execution of the given task. A good example is reversing the sequence of input values. We have for example a sequence of integer values ended by –100 and we need to write this sequence in reverse order. The whole program with the recursive procedure is:

```
#include <iostream>
using namespace std;
void Reverse(){
  int Inp; // local variable
  cin >> Inp;
  if (Inp != -100) {Reverse(); cout << Inp << endl;}
}
int main(){
  cout << "Enter numbers ended with -100: ";
  Reverse();
  return 0;
}
```

Appropriate recursive subroutines will be discussed later.

9.3.4

We want to calculate the nth member of the Fibonacci sequence. The calculation formula is:

$F_i=F_{i-1}+F_{i-2}$; $F_0=0$; $F_1=1$.

Will it be efficient to use recursion for this task?

- False
- True

9.4 Recursion (programs)

9.4.1 Recursive subroutines 1

The input is a sequence of real values. Write this sequence in reverse order. Use a recursive subroutine.

9.4.2 Recursive subroutines 2

Write a function that calculates the factorial value of N. Use a recursive variant. Determine if the result exceeds the allowed value range, otherwise, the calculation result will be zero.

recfunct2.cpp

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

// enter function "factorial" here:


int main(){

    int Number;
    long long int NFact;

    cin >> Number;
    NFact = factorial(Number);
    cout <|<| "Factorial of "<|<|Number<|<|" is ";
    if (NFact==0) cout <|<| "too big to calculate" <|<| endl;
       else cout <|<| NFact <|<| endl;

    return 0;
}
```

9.4.3 Recursive subroutines 3

Write a function that calculates the first N members of Fibonacci sequence. Use a recursive variant. Value N is on input.

WARNING! This implementation is very inefficient and should not be used at all. The example is included only as a recursion exercise. The calculation time for small values of N increases to large values and for N exceeding 50, you do not have to wait for the result.

recfunct3.cpp

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

// enter function "Fibon" here:


int main(){

    int Number;

    cin >> Number;
```

```
    for (int i=1; i<|=Number; i++)
        cout <|<| Fibon(i) <|<| " ";
    cout <|<| endl;

    return 0;
}
```

9.4.4 Recursive subroutines 4

The input is a sequence of integer values. Write the sum of this sequence. Use a recursive subroutine.

Arrays

Chapter 10

10.1 Array

🕮 10.1.1

Structured data types

The term **structured data type** refers to a data type that is composed of multiple components, so it can store multiple values at a time. Structured type items can be both simple and structured types, so structures of almost any complexity can be created. The simplest and the best-known structured type is an array.

Why array?

We often need to process several (many) values of the same data type. To do this, we need the appropriate number of variables. However, it would be very inconvenient if we had to declare and process the entire set of variables separately. Therefore, we can make one variable with multiple folders. Such a variable with multiple folders of the same type is called an **array**. Individual components are distinguished by serial numbers, which we call **indexes**.

The indexes always start with zero in the C/C++ language, so for example a ten-component array has indexes of 0 to 9.

In previous algorithms, we were processing a series of values, yet we didn't need an array. So when is the array necessary? This is in the case when we need to access a series of data **repeatedly**. In all previous algorithms, we were gradually processing the data and we no longer needed the processed values, so we have rewritten them with new ones (for example, read from the input or calculated from various sources). However, there are many algorithms that need to process the input data multiple times, to reorder (sort) it or otherwise modify it. There everywhere is necessary to use the array.

How array?

If we want to declare an array, we use the same method as when declaring a simple variable, just add the required number of items to the square brackets. For example, a five-element array of Payments composed of integer items will be declared:

```
int Payments[5];
```

We get an array whose components will have indexes 0 through 4.

Similar to simple variables, you can insert an initial value directly into an array when declaring it. However, it is necessary to use a slightly different syntax here. Values inserted into individual items must be written in a list enclosed in curly braces separated by commas, for example:

```
int Payments[5] = {100, 80, 300, 250, 140};
```

10.1.2

Choose from the following options to declare an array to store float numbers as your business's monthly turnovers.

- float Flow[12];
- signed char Flow[12];
- int Flow[11];
- double Flow{1,2,3,4,5,6,7,8,9,10,11,12};

10.1.3

As we already know, if we want to work with a variable, we write its identifier. If we want to work with individual array components, we write the index of the component to the array identifier in square brackets. An index can also be entered in the form of an expression that is first calculated and then used to access the appropriate item. The index can only be **integer values** or values that can act as integers.

Suppose declaration of Payments array:

```
int Payments[5];
```

Examples:

```
Payments[3] = 100;
int m = 0;
Payments[m+3] = Payments[m+2] - 5;
```

Very often, we need to process all the components of the array in the same way, for which the for loop is great. For example, a list of all items together with their indexes can be written:

```
for (int i=0; i<5; i++) cout << i << ": "<< Payments[i] <<
endl;
```

Note that the variable *i* passes through the indexes from zero to 4.

When declaring the array, we specify the number of folders, which is always 1 greater than the value of the last index. It is therefore very advantageous for array manipulation to have a constant specifying the number of array items. We can then

refer to it everywhere, giving us the opportunity to change the size of the array in one place:

```
const int NumOfPayments = 10;
int Payments[NumOfPayments];
for (int i=0; i<NumOfPayments; i++) cout << i << ": " <<
Payments[i]<<endl;
```

10.1.4

We have the following code:

```
char Vowels[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
cout << Vowels[3] << endl;
```

Does the character "i" appear on the screen (without quotes)?

- False
- True

10.1.5

We have a daily close stock price array for the last working week. In the following code, complete the parts so that the prices will be read from standard input and then will be correctly displayed for each day.

```
#include <iostream>
using namespace std;
int main(){
 _____ ;
   int Day;
   _____ cin >> Stock[Day-1];
   _____
      cout << "Day No. " << Day+1 << ": The price is " <<
Stock[Day] << endl;
   return 0;
}
```

- for (Day=1; Day=5; Day++)
- float Stock[5];
- for (Day=1; Day<=5; Day++)
- int Stock[4];
- for (Day=0; Day<5; Day++)

- for (Day=0; Day<|=5; Day++)
- char Stock[Week];

10.1.6

It is very advantageous to use an array if we create its data type in advance. We can then simplify some manipulations, such as declaring totally identical arrays at different locations or passing such arrays as subroutine parameters.

As mentioned earlier, it is possible to define custom data types using the **typedef** keyword. In the case of an array data type definition, it is necessary to add a number of array items into the type definition, for example:

```
typedef float TEnterprises [10];
```

We get a data type that we can use to declare multiple identical arrays:

```
TEnterprises Salaries, Profits = {0, 0, 0, 0, 0, 0, 0, 0, 0,
0};
```

10.1.7

Task: We have the input values that consist of pairs of data: enterprise number and profit for goods sold. The company has ten enterprises, the enterprise numbers are from 1 to 10. Write down the total profits of every enterprise from all input data.

The program is:

```
#include <iostream>
using namespace std;
int main(){
   //    ... HERE IS MISSING CODE ...
   int Num; int Profit;
   while (cin>>Num>>Profit) Profits[Num-1]+=Profit;
   for (Num=0; Num<10; Num++)
      cout << Num+1 << ": " << Profits[Num] << endl;
   return 0;
}
```

Select the appropriate missing part of the code.

- typedef int TProfits[10]; TProfits Profits = {0,0,0,0,0,0,0,0,0,0};
- int Profits[11];
- typedef int Profits[10]; Profits TProfits = {0,0,0,0,0,0,0,0,0,0};

- int TProfits[10] = {0,0,0,0,0,0,0,0,0,0};

📖 10.1.8

Indexes are used to access array items. Arrays can always be indexed from zero in the C/C++ language. When declaring an array, the number of entries M is specified. If you use an index outside of 0 to M – 1 to access the array entry, the compiler does not check that the index does not belong to that array. Therefore, it is always necessary to check whether the index is correct or not.

If we access an array item outside the specified range, we can read the contents of the memory that is behind the allocated array. But we can also modify this memory, which can lead to very tricky and unexpected errors, which only occur under certain circumstances and are very difficult to find.

📝 10.1.9

We want to insert values read from standard input into the array. We don't know how many values are on the input. Is the following code correct?

```
typedef float TValues[100];
TValues Values;
int Num=0;
while (cin>>Values[Num]) Num++;
```

- False
- True

📖 10.1.10

The *sizeof* function is used to determine the size of a variable in the computer's memory. If we want to determine the size of the array, we can use this function. If we want to find out how many components an array has, we have to divide the result by the size of one component. In this case, it is advantageous if user data types are defined for both the array and its components.

For example:

```
typedef int Value;
typedef Value TValues[15];
```

Then we can write:

```
  cout << "Number of items is "<< sizeof(TValues) /
sizeof(Value) << endl;
```

10.1.11

How many items does the Values array have when we know that sizeof (Values) = 64 and the array component is double data type? (We assume that there is no alignment of values in the memory to larger units.)

- 8
- 16
- 10
- 64

10.1.12

The **enumeration** type (**enum**) is a subset of an integer type. Its values are named identifiers, so each value can be very clearly understood in the program. For example, while an integer 5 used in a condition or an expression does not give an idea if we write Friday instead of 5, it is clear that it is the day of the week.

Definition of enum data type:

```
typedef enum {
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday
} TWeek;
```

The values are automatically numbered from zero. However, we can assign an arbitrary number to each value, so values don't have to go in the order in which they are written. For example:

```
typedef enum {
   red = 0xFF0000,
   green = 0x00FF00,
   blue = 0x0000FF,
   white = 0xFFFFFF,
   black = 0x000000
} TRGBColors;
```

Enum type values cannot be read from standard input and identifiers are not displayed on output. The integer representation of identifiers is displayed on output only.

Because the enum type is a subset of an integer type, it can also be used for array indexing. We have for example an array indexed by day of the week – the type TWeek mentioned above:

```
int SaledPieces[6];
```

Suppose that this array is filled with numbers of saled pieces of cars and we need to display the content of this array for workdays only:

```
for (TWeek W = Monday; W <= Friday; W = TWeek(int(W)+1))
   cout << SaledPieces[W] << ", ";
cout << endl;
```

10.1.13

Suppose that the input is a sequence of pairs of values – the price of the car sold and its colour. Output the sum of the car prices for each colour. Select the correct part of the code in the following program.

```
   typedef enum {black, red, green, blue,
       yellow, magenta, cyan, white, silver} TColor;

   // select the appropriate part of code HERE

   TCars Sales;
   int Price, NumColor;
   for (int i=black; i<=silver; i++) Sales[i]=0;
   while (cin >> NumColor >> Price) {
     if (NumColor >= black and NumColor <=silver)
        Sales[NumColor]+=Price;
   }
   cout << "Overview of sold cars by color:" << endl;
   for (int i=black; i<=silver; i++)
      cout << setw(10) << left << ColorNames[i]
      << setw(10)<<right<<Sales[i] << endl;
```

- typedef int TCars[silver+1]; const string ColorNames[silver+1] = {"black", "red", "green", "blue", "yellow", "magenta", "cyan", "white", "silver"};
- typedef int TCars[silver]; const string ColorNames[silver] = {"black", "red", "green", "blue", "yellow", "magenta", "cyan", "white", "silver"};

- typedef int TCars[7]; const string ColorNames[7] = {"black", "red", "green", "blue", "yellow", "magenta", "cyan", "white", "silver"};
- typedef int TCars[silver]; const string ColorNames[silver] = {'black', 'red', 'green', 'blue', 'yellow', 'magenta', 'cyan', 'white', 'silver'};

10.2 Arrays (programs)

⌨ 10.2.1 Arrays 1

The input is a sequence of integer values (maximum 100 values). Write this sequence in reverse order. Use an array.

⌨ 10.2.2 Arrays 2

The input is a sequence of integer values, maximum 100 values. Write all values which are higher than the average input values.

⌨ 10.2.3 Arrays 3

There are pairs of values at the input. The first number indicates the day of the week (1 is Monday, etc.), the second number indicates the volume of production in euro. Sum the production volume for each day of the week and output these totals.

⌨ 10.2.4 Arrays 4

There is a series of integer values at the input, the last value is zero. Determine the frequencies of input numbers at intervals of 1..10, 11..20, etc., up to 91..100.

⌨ 10.2.5 Arrays 5

There is a series of integer values at the input. Determine the number of individual decimal digits from which the input values are composed.

10.2.6 Arrays 6

There is a sequence of integers on the computer input. The task is to encrypt this input so that individual digits, spaces, positive and negative signs are replaced with other characters that the user enters before a series of numbers. So the input starts with 13 characters – the first ten are used to encrypt the digits 0..9, the other two encrypt the plus and minus signs respectively, and the last character encrypts the space. The output is composed of an encrypted sequence of input numbers.

Multidimensional Arrays

Chapter 11

11.1 Array of arrays

📖 11.1.1

C/C++ knows only one-dimensional arrays. However, the array component can be an array too, then you can get multidimensional arrays. The declaration is:

```
double Matrix[12][10];
```

This gives you a 12-item array where each item has ten float-type items. Note that adding the next index will create an array of items that consist of the previous array.

We can do it in another way that is more flexible and clearer. We will use type definitions and constant definitions (the constants will be mentioned later):

```
const int NumCols = 10;  // number of columns
const int NumRows = 12; // number of rows
typedef double TRow [NumCols];
    // type definition of the row of matrix as an array of
double
typedef TRow TMatrix [NumRows];
    // type definition of whole matrix as an array of rows
TMatrix Matrix;  // declaration of matrix variable
```

In this shape, we can change the size of a matrix at any time by simply changing constants, which is very flexible. The constants can be further used whenever we refer to the dimensions of a matrix (reading elements, listing elements).

📝 11.1.2

Assume that the input contains 57 real values. Create a matrix of 3 rows of 19 values and enter the input numbers. Then write the matrix to standard output so that the numbers are arranged in rows. Add the corresponding parts to the following code:

```
const int Items = 19;
const int Rows = 3;

 ______
TMatrix Mat;
for (int I=0; I<Rows; I++)

______
for (int I=0; I<Rows; I++){

______
```

```
    for (int J=0; J<Items; J++) cout << Mat[I][J] << "  ";
    cout << endl;
}
```

- for (int J=0; J<Items; J++) cin >> Mat[I][J];
- cout <|<| Mat[I+1][J+1] <|<| ": ";
- typedef double TRow[Rows]; typedef TRow TMatrix[Items];
- typedef double TRow[Items]; typedef TRow TMatrix[Rows];
- cout << "Row No.: " << I+1 << ": ";
- for (int J=0; J<|Rows; J++) cin >> Mat[I][J];

11.1.3

We have two matrices of equal dimensions and they are filled with integers. The *TMyMatrix* data type has been defined, and the matrix dimensions are given by the *NumRows* and *NumCols* constants. Choose from the following code options that sum these two matrices.

- TMyMatrix MA, MB; int B=0, A; while (B<|NumRows){A = 0; while (A<|NumCols) {MA[B][A] += MB[B][A]; A++;} B++;}
- TMyMatrix MA, MB; int X, Y; for (X=0; X<|NumCols; X++) for (Y=0; Y<|NumRows; Y++) MA[X][Y] += MB[X][Y];
- TMyMatrix MA, MB; int A=0, B=0; while (B<|NumRows) while (A<|NumCols) MA[B][A] += MB[B][A];
- TMyMatrix MA, MB; for (int A=0; A<|NumRows; A++) for (int B=0; B<|NumCols; B++) MA[B][A] += MB[B][A];

11.1.4

In the following code, complete the correct parts to transpose the square matrix.

```
const int Order = 10;  // order of the matrix
 ______
QMatrix Mat; double aux;
for (int x=0; x<Order; x++)
for (int y=0; y<Order; y++) cin >> Mat[x][y];
for (int i=0; i<Order; i++){
_____
_____
for (int I=0; I<Order; I++){ cout << "Row: " << I+1 << ": ";
for (int J=0; J<Order; J++) cout << Mat[I][J] << "  ";
cout << endl;}
```

- for (int j=0; j<|=Order; j++)
- Mat[i][j]=Mat[j][i]; j++;}
- typedef double QMatrix[Order][Order];
- {aux = Mat[i][j]; Mat[i][j] = Mat[j][i]; Mat[j][i] = aux;}
- typedef double QMatrix[Order];
- for (int j=i+1; j<Order; j++)

11.1.5

Task: There is a sequence of pairs of numbers on the input: the department number (1 to 7) and the salary of a worker of that department. List average salaries for individual departments.

From the following options, select the part of the code that belongs to the following program:

```
#include <iostream>
using namespace std;
int main(){
  typedef int TDept[2];
  typedef TDept TFactory[7];
  TFactory OurFact;
  int Num, Salary;
  for (Num=0; Num<7; Num++){
      OurFact[Num][0]=OurFact[Num][1]=0;
  }
  while (cin>>Num>>Salary){
      OurFact[Num-1][0]++;
      OurFact[Num-1][1]+=Salary;
  }
  // SELECTED PART OF CODE BELONGS HERE
  return 0;
}
```

- for (Num=0; Num<|7; Num++) if (OurFact[Num][0]!=0) cout <|<| "Dept. No. " <|<| Num+1 <|<| ", avg. salary: " <|<| OurFact[Num][1]/OurFact[Num][0] <|<| endl; else cout <|<| "No workers. " <|<| endl;
- Num=0; while (Num<|7){ if (OurFact[Num][1]==0) cout <|<| "Dept. No. " <|<| Num+1 <|<| ", avg. salary: " <|<| OurFact[Num][1]/OurFact[Num][0] <|<| endl; else cout <|<| "No workers. " <|<| endl; }
- Num=0; while (Num<|7){Num++; if (OurFact[Num][1]==0) cout <|<| "Dept. No. " <|<| Num <|<| ", avg. salary: " <|<| OurFact[Num-1][0]/OurFact[Num-1][1] <|<| endl; else cout <|<| "No workers. " <|<| endl; }

- for (Num=1; Num<|=7; Num++) if (OurFact[Num][0]!=0) cout <|<| "Dept. No. " <|<| Num <|<| ", avg. salary: " <|<| OurFact[Num][1]/OurFact[Num][0] <|<| endl; else cout <|<| "No workers. " <|<| endl;

11.2 Multidimensional arrays (programs)

⌨ 11.2.1 Matrix 1

There is a number N (max 20) on the input, indicating the order of the square matrix, followed by real numbers representing the row-entered matrix. Write the entered matrix and follow the transposed matrix.

matrix1.cpp

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

// define data type TMatrix here:


void DisplayMat(TMatrix M, int Count){  // display matrix
   cout <|<| "-----------------------------------"<|<| endl;
   cout.precision(3);
   for (int i=0; i<|Count; i++){
      for (int j=0; j<|Count; j++)
         cout<|<| setw(9)<|<|fixed<|<|right<|<|M[i][j];
      cout<|<|endl;
   }
   cout <|<| endl;
}


int main(){
 // enter your code here:

    return 0;
}
```

⌨ 11.2.2 Matrix 2

There is a number N on the input, indicating the order of the square matrix, followed by real numbers representing the row-entered matrix. Recalculate the matrix so that it has a value of 1 on the main diagonal. (Hint: you have to divide each row by the value on the main diagonal. In the case of the zero elements of the main diagonal, the entire row will be zero.)

Strings

Chapter 12

12.1 Strings

📖 12.1.1

Typical programs contain two categories of data – numbers and character strings. These elements then build other structures. Numeric data types have already been discussed, now we look at character strings.

The character string is implemented in most languages as a kind of array whose components are individual characters. The data type for characters has already been specified – this is char (unsigned char).

Like numeric value arrays, we can construct an array of character values:

```
char Name[35];
```

This will give you an array of up to 35 characters, with each character you can manipulate using the appropriate index.

However, a string is not exactly the same as a character array but is different in several respects. Each character string must have its current length specified. If we have a Name variable that can store up to 35 characters, we would work with unoccupied folders when saving a specific name less than 35 positions. Therefore, the end of the string is indicated by a special zero character '\0'. Strings of this type are called null-terminated '\0'. Therefore, when determining the length of a string, we always have to assume that there is one extra character.

In order not to determine exactly how many items the character array has, we can make a declaration when the number of items is determined by the value entered:

```
char Address[] = "Rodeo Drive, Holywood";
```

The length of this string will then be given by the number of characters plus the terminating character null.

📝 12.1.2

Assume a Description variable with the following definition:

```
char Description[] = "This is sum of values.";
```

The number of components of this variable will be 23. True or false?

- True
- False

12.1.3

We have a string variable to store the ISBN. What value will this variable have after performing the following code?

```
char ISBN[14] = "9887074283225";
ISBN[1] = '7'; ISBN[3] = '8'; ISBN[11] = '1';
```

- 9788074283215
- 9877874283221
- 7887074283125
- 1887074288275

12.1.4

We have a string variable to store the ISBN. What will be displayed after processing the following code?

```
char ISBN[18] = "978-80-7428-321-5";
ISBN[3] = '\0'; ISBN[6] = '/'; ISBN[11] = '/'; ISBN[15]='/';
```

- 978
- 978\80/7428/321/5
- 978\80\7428\321\5
- 978\080\07428\0321\05

12.1.5

To work with a character array, there are operations stored in the standard string.h library. If we want to use this library in the C++ language, it is available under the name cstring, so we can write

```
#include <cstring>
```

Functions:

- strlen(*s*) – integer function for determination of the length of string *s* (i.e. the number of characters without end null character);
- strcpy(*dest*, *source*) – function copies string *source* into string *dest*;
- strcat(*dest*, *source*) – function appends string *source* to the end of string *dest*;

- strchr(*str*, *ch*) – this function searches character *ch* in string *str* and returns a pointer to the first position of character *ch*. If the character is not found, this function returns NULL;
- strstr(*str*, *substr*) – this function searches substring *substr* in string *str*. The result is similar to strchr function;
- strcmp(*s1*, *s2*) – integer function compares strings *s1* and *s2*. The result of this function is a negative number if *s1*<*s2*, result 0 is returned if *s1*==*s2* and a positive number otherwise.

12.1.6

Assume that in the standard input is a name and surname (on each line separately). Put them in one string variable, write down the total number of characters and write only the last name from the variable. Add the appropriate parts to the following code.

```
char Name[35], Surname[40], FullName[75]="";
cin >> Name >> Surname;
strcpy(FullName, Name);

 _____
cout << "The length is "<< _____ << endl;
cout << "The surname is: "<< _____ << endl;
```

- strlen(FullName)
- length(FullName)
- strcat(FullName, " "); strcat(FullName, Surname);
- strcat(FullName, Surname); strcat(FullName, " ");
- strcpy(FullName, strstr(Name, ' ')+1)
- Surname
- sizeof(FullName)

12.1.7

An array of characters still has an array behaviour, so we can't simply do a number of useful operations. For example, we have the following code:

```
char Name[20];
Name = "Smith"; // an error will be reported here
```

We get an error because the Name variable is of type char[20] and we try to assign a value of type const char[6] on the right side of the assignment.

Therefore, the C++ language has an std library that implements the string data type. So code:

```
std::string Name;
Name = "Smith";
```

implements variable Name which can be processed both as an array of characters and as a whole string. If we use the *using namespace std* construction, which we normally do for standard input and output, we don't have to write *std :: string*, but just a *string*.

📖 12.1.8

The **string** data type available in C ++ is created somewhat differently, it is implemented as an object. The objects will be discussed later, at this point we will show the operations that can be performed with the string type.

Because the string data type is an object, the operations are stored "inside" it and are available through the variable name and period. Thus, the syntax is completely different from the operations that were shown at the character array.

Suppose the following declaration:

```
string SomeChars;
```

We can then use these operations:

- SomeChars = "any string" – assigning of any constant value into string variable;
- SomeChars.length() – the length of string
- Concatenation with a "+" operator, for example: SomeChars = "Paul" + " " + "Smith";
- SomeChars.clear() – deleting all characters from string variable
- SomeChars.find("Sm") – find a position of string "Sm" in variable SomeChars. If the substring is not found, the result of this function is a number greater than the maximum number of characters (i.e. value greater than length()).
- SomeChars.substr(3, 5) – a substring from index 3 and of length 5. In our case, this will return substring "l Smi".

12.2 String (programs)

12.2.1 Strings 1

There is a sequence of strings at the input (one string per line). Write the longest string from the input sequence.

12.2.2 Strings 2

There is a sequence of strings at the input (one string per line). Find a string with the largest relative proportion of punctuation and white characters.

12.2.3 Strings 3

There is a sequence of strings at the input (one string per line). Remove from all strings all non-alphabet characters.

12.2.4 Strings 4

There is a sequence of strings at the input (one string per line). Write a string that is first alphabetically and a string that is last alphabetically.

12.2.5 Strings 5

There is a sequence of strings at the input (one string per line). Find out which strings are palindromes (i.e. both left and right read are the same).

Struct Data Type

Chapter 13

13.1 Structured data type struct

🕮 13.1.1

Data type **struct** is a collection of items. Each item may be of a different type. This structure wrapped its items and may be as one variable.

The common shape of structure definition is:

```
struct {
   type item1;
   type item2;
   ...
} identifier;
```

For example:

```
struct {
   string FullName;
   int Salary;
   float Weight;
} MyPerson;
```

To manipulate with items we use the dot convention:

```
MyPerson.FullName = "James Bond";
MyPerson.Salary = 1000000;
MyPerson.Weight = 75;
```

It is very useful that two structures of the same type can be assigned to each other, unlike arrays. Therefore, it is useful to create a data type of the appropriate structure first and then declare variables of this type:

```
typedef struct {
   string FullName;
   int Salary;
   float Weight;
} TPerson;
TPerson MyPerson, Brother, MyTeacher, MyWife; // four
variables
MyPerson.FullName = "Jean Gabin"; MyPerson.Salary = 60000;
MyPerson.Weight = 90; // insert some values to items
Brother = MyPerson; // Brother got the same values as MyPerson
Brother.FullName = "Paul Gabin";
```

13.1.2

We have the following code:

```
struct {int IdMat; float Length;
   string Description;} StoreItem;
StoreItem GreenCanvas;
```

Is declaration of variable GreenCanvas correct?

- False
- True

13.1.3

The basic motivation for using a structure is to concentrate several different items into one variable. If we build an array from such records, we can move entire records instead of individual items as needed, for example, when we need to swap items in sorting or searching.

To create an array with records we use a known process. We need for example an array of engines:

```
typedef struct {string Series; int Weight; int Distance;}
TEngine;
const int MaxNumOfEngines = 75;
typedef TEngine WholeDepot[MaxNumOfEngines];
```

Example of engine array usage: Suppose the array is filled with values and the total number of engines is in the *ActualNumOfEngines* variable. Put the engine with the smallest weight into the *Shunting* variable:

```
TEngine Shunting; int ActualNumOfEngines = 58;
int MinW=0; // index of the lightest locomotive
for (int i=1; i<ActualNumOfEngines; i++)
   if (WholeDepot[i].Weight<WholeDepot[MinW].Weight) MinW = i;
Shunting = WholeDepot[MinW];
```

13.1.4

Define a person's record array (name, height in centimetres, weight in kilogrammes). Fill this array with data read from the input and display the name of the person with the smallest BMI (Body Mass Index = weight in

kilogrammes/square of height in metres). Add the corresponding section to the following code:

```
_____
float BMI(TPerson P){
   float h = float (P.Height);
   return P.Weight / ((h/100)*(h/100));
}
const int MaxNumOfPerson = 100;
typedef TPerson TDepartment [MaxNumOfPerson];
TDepartment OurDept;
int ActNum=0, MinBMI = 0;
while (cin _____ ) {
    if ( _____ ) MinBMI = ActNum;
   ActNum++;
}
cout << "Min. BMI has " << OurDept[MinBMI].Name << endl;
```

- BMI(OurDept[ActNum]) < BMI(OurDept[MinBMI])
- typedef struct {int Name; string Height; float Weight;} TPerson;
- >> OurDept[ActNum].Name >> OurDept[ActNum].Height >> OurDept[ActNum].Weight
- typedef struct {string Name; int Height; int Weight;} TPerson;
- BMI(Height, Weight) <| BMI(MinBMI.Height, MinBMI.Weight)
- >> OurDept[ActNum]

📖 13.1.5

Structure items can have one more meaning – they can be used to determine what space to store. It is advisable that the sequence of items forms integral bytes. The number of bits on which the item is to be stored is written after the colon for each item. For example, have the following definitions:

```
  typedef enum {woman, man} TSex;
  typedef enum {nonsmoker, smoker} TSmoker;
  typedef enum {student, teacher} TRole;
  typedef enum {driver, nodriver} TDriving;
```

Now we will create a structure that will contain some data, but will store it in minimal space:

```
  typedef struct {
    TSex Sex:1;
    TSmoker Smokes: 1;
```

```
    TRole Role: 1;
    TDriving Driver: 1;
    int NumChild: 4;
  } TPerson;
```

This structure takes up only 8 bits in memory, ie one byte, and 5 information is stored on it. So we can easily work with the individual bits of the respective byte, for example:

```
  TPerson MyWife; // one byte of memory
  MyWife.Sex=woman;
  MyWife.Smokes=smoker;
  MyWife.Role=teacher;
  MyWife.Driver=driver;
  MyWife.NumChild=2;
```

13.1.6

We have the following structure:

```
typedef struct {
   bool readonly: 1;
   bool system: 1;
   bool hidden: 1;
   bool archive: 1;
   bool directory: 1;
   bool shared: 1;
} TFileAttrib;
TFileAttrib MyFile;
```

Will the size of the MyFile variable be greater than one byte?

- False
- True

13.2 Struct (programs)

13.2.1 Structs 1

There are data of three persons on input: name, age and salary. List these people in ascending order by age.

⌨ 13.2.2 Structs 2

There are a number of data pairs at the input: the title of the goods and the number of pieces in the warehouse. List the names of the goods that have the amount of pieces above average in the warehouse.

⌨ 13.2.3 Structs 3

It is necessary to store product information: category (1..13), new/used (bool), number of pieces (0..50), how much is reserved (0..25). Create a record that stores this information in minimal space. Display amount of memory for this record. There are data on several items at the input. List these goods in reverse order. Check the correct values entered. An incorrect category will be inserted as 0, an incorrect logical value as false, an incorrect number of pieces as 0, and an incorrect reserved value as the number of pieces; if it exceeds 25, then as half the number of pieces.

⌨ 13.2.4 Structs 4

There are a number of Cartesian coordinates of points in the plane at the input (max. 100). Write these points sorted by distance from the origin of the coordinates.

⌨ 13.2.5 Structs 5

There is a series of integer values at the input. Determine which value occurred most times at the input.

Union Data Type

Chapter 14

14.1 Union data type

🕮 14.1.1

The **union** data type is very similar to the structure (**struct** data type), the difference is in the way the items are stored in memory. For the **struct** type, items are placed one after the other in the order they were written in the type definition. For **union**, all items are located starting with the *same memory address*, that is, they share the same memory space. What good is this way for? We assume that we always need *only one option* for real data. The rest of the items, therefore, does not take up unnecessary memory. A type definition is like the **struct** definition, but the keyword **struct** is replaced by keyword **union**:

```
typedef union {
   int intnumber;
   double realnumber;
   char txt[7];
} TNumbers;
```

For example, we need to store locomotive parameters. For each type of locomotive, however, we need something different – for the steam locomotive the heating surface of the boiler, for the diesel locomotive the number of engine cylinders and for the electric locomotive the voltage in the power system. For example, if we have an array of such records, each array entry may contain a record of another locomotive type, but unused items will not occupy any additional space. Let us define such structure:

```
typedef enum {steam, diesel, electric} TTypeLok;

typedef union {
    float Boiler;
    int Cylinders;
    int Voltage;
} TLokParam;

typedef struct {
    string Label;
    float Weight;
    TTypeLok Traction;
    TLokParam LocoPar;
} TLocomotive;

const int MaxNumLoco = 150;
typedef TLocomotive TDepot[MaxNumLoco];
```

Note that we have placed locomotive-bound traction parameters in a structure where there are other components common to all types, as well as a component that lists the locomotive type. This item is called the *distinguishing item* because it can be used to tell which of the union items is valid. Typically, the distinguishing component is defined as an enumeration or an integer.

14.1.2

We have the following **union** structure:

```
typedef union {
   int anynumber;
   char bytes[4];
} TTwo;
TTwo MyVariable;
```

Next, we have an integer stored in memory so that the most significant byte is the first. We use a defined structure to display the values of individual bytes that make up an integer in memory:

```
cin >> MyVariable.anynumber;
for (int i=0; i<4; i++) cout << (int)MyVariable.bytes[i]<<" ";
```

After entering a certain number was displayed:

```
0 0 1 0
```

What number has been entered?

- 256
- 100
- 1024
- 1

14.1.3

We have a structure TDepot that we defined above:

```
typedef enum {steam, diesel, electric} TTypeLok;

typedef union {
    float Boiler;
    int Cylinders;
```

```
    string PowerType;
} TLokParam;

typedef struct {
    string Label;
    float Weight;
    TTypeLok Traction;
    TLokParam LocoPar;
} TLocomotive;

const int MaxNumLoco = 150;
typedef TLocomotive TDepot[MaxNumLoco];
```

and a variable Manchester of this type:

```
TDepot Manchester;
```

Next, assume that the variable Manchester is filled with data about 88 locomotives. The task is to display all steam locomotives with a weight above 80 tons and all diesel locomotives with a number of cylinders less than 16. Select the appropriate parts in the following code:

```
const int NumLoco = 88;
cout << "Steam locomotives with weight above 80 tons:"<<endl;
for (int L=0; L<NumLoco; L++)
   if ( _____ )
     cout << _____ <<endl;
cout << "Selected diesel locomotives: "<<endl;
for (int L=0; L<NumLoco; L++)
   if ( _____ )
     cout << _____ << endl;
```

- Manchester[L].TTypeLok==steam and Manchester[L].Weight>80
- Manchester[L].Traction==diesel and Manchester[L].Cylinders<|16
- Manchester[L].Label <|<|", num of cylinders is " <|<| Manchester[L].Cylinders
- Manchester[L].TLocomotive.Label <|<|", weight is " <|<| Manchester[L].TLocomotive.Weight
- Manchester[L].Traction==diesel and Manchester[L].LocoPar.Cylinders<16
- Manchester[L].Traction==steam and Manchester[L].Weight > 80
- Manchester[L].Label << ", num of cylinders is " << Manchester[L].LocoPar.Cylinders
- Manchester[L].Label << ", weight is " << Manchester[L].Weight

14.2 Union (programs)

⌨ 14.2.1 Union 1

There is a number of types double on the input. Display them in the binary system as stored in the computer-s memory. Show the most significant bits on the left.

⌨ 14.2.2 Union 2

Create an array whose components can be integers or real numbers or strings of 7 characters. The input is a sequence of pairs - type value (0 = integer, 1 = real number, 2 = string) and followed by the value of the type. The last pair has only the first value of –1. Fill these values into the array and list first all integer values, then all real values and then all string values.

Pointers I

Chapter 15

15.1 Pointers

🕮 15.1.1

The pointer represents the address in the computer's memory. It is used for many purposes and to manage pointers is one of the basic skills in writing programs.

The pointer definition is very simple – the asterisk character is given before the variable name.

Pointer declaration example:

```
int *ptrint;
```

We can also define a new pointer type. Note that the asterisk is written after the data type, for example

```
typedef float* tdataptr;
```

Pointers allow you to work with addresses in memory, which in some cases is very important for optimizing memory usage and optimizing data access.

🕮 15.1.2

The pointer (address) of the variable itself is not useful but is important to use this address to access the variable located at that address. If P is a variable of type pointer, then *P allows you to work with a variable whose address is in P.

How to get a variable address?

The address of the variable in memory is automatically assigned by the compiler when declaring the variable. We do not normally need this address because we work directly with the variable through its identifier. The variable is created at the time of declaration and expires with the end of the block in which it is declared. The address of such a variable can be obtained by the unary operator & written before the variable identifier. The operator & will be discussed later, see the Reference. This is the first way to get an address. For example:

```
float MyData;  // the address of MyData was given by the
compiler
float *AddrData; // pointer to float data type
AddrData = &MyData;  // address of MyData is stored to pointer
AddrData
*AddrData = 4.5;  // store data to variable of address of
MyData;
```

```
cout << MyData << endl; // will be displayed 4.5
```

The second way to get an address is quite different. In the previous case, we worked with the address assigned by the compiler at the usual variable declaration. We can use this variable throughout the block in which it is declared. However, if we no longer need it, we cannot release it from our memory. The second option is a system where memory is allocated only when the variable is needed and can be released at any time. These are dynamic memory allocation and dynamic variables. For dynamic variables, completely different memory space is used than for non-dynamic variables. Typically, this block has a much larger size than the memory block for the declared variables. So it is natural that we try to place the data in dynamic variables. The dynamic variable address assignment is made by the **new** operator:

```
float *AddrData;  // declaration of dynamic variable
AddrData = new float; // memory allocation and assignment of
new address
*AddrData = 18.78; // the use of dynamic variable
```

When allocating memory, we can also assign an initial value:

```
AddrData = new float(18.78);
```

We can allocate memory for the array. At the time of allocation, we can specify the required number of field items:

```
typedef int TMyData;  // data type of array items
TMyData *ActValues;  // pointer to array item
ActValues = new TMyData[52];  // allocation of array with 52
items
```

We can assign an address constant. We have only one constant for pointers of all types. This constant is NULL and represents an empty pointer (pointer which points nowhere). We'll use this pointer wherever we want to provide an empty address that we can test.

```
AddrData = NULL;
```

15.1.3

Task: Input is a sequence of decimal numbers. The first input value is the count of numbers entered. Write the numbers to the output in reverse order.

Choose from the parts offered here that belong to the following code:

```
  int Num;
```

```
  cin >> Num;
// choose one answer HERE:

  for (int i=0; i<Num; i++) cin >> Values[i];
  for (int i=Num-1; i>=0; i--) cout << Values[i] << endl;
```

- float *Values; Values = new float[Num];
- float Values; Values = new float[Num];
- float *Values[Num];
- float Values = float [Num];

📖 15.1.4

The **delete** operator is used to free memory and remove the dynamic variable.

```
delete AddrData;
delete [] ActValues;
```

To remove the array from memory, you have to write brackets after keyword **delete**.

The value of the pointer is undefined after deallocating of memory. This it is strongly recommended to assign freed pointer with constant NULL. We can then simply detect that this pointer is no longer valid.

Deallocating memory is a very important operation. When the memory is released, we can allocate the same space in the next step of the program.

Because the process of allocating and freeing memory is random, it is possible that the free blocks are still shrinking in memory. If we do not release a variable but lose a pointer to it, the block will become inaccessible but will remain in memory and cannot be released. This situation is called a memory leak.

Typical error:

```
int *p;
p = new int;
// ... using p ...
p = new int; // old address is replaced by new one!
```

The old address is no longer accessible and the first allocated block remains in memory.

15.1.5

Task: Input is a sequence of decimal numbers. The first input value is the number of numbers entered. Write the numbers to the output in reverse order.

Choose from the parts offered here that belong to the following code:

```
  int Num;
  cin >> Num;
// choose one answer HERE:

  for (int i=0; i<Num; i++) {AnyData[i] = new float; cin >>
*AnyData[i];}
  for (int i=Num-1; i>=0; i--) cout << *AnyData[i] << endl;
  delete [] AnyData;
```

- float *AnyData[Num];
- float AnyData[Num];
- float *AnyData = new float;
- float AnyData = new [Num]

15.1.6

Later we will discuss the relationship between an array and a dynamic variable, now we will look at structures where we don't need to know in advance how many elements we will need, but we can make them as large as the data we are currently processing. The basic structure is a linear unidirectional dynamic list.

A list element is a dynamic variable containing two items - the data and a pointer to another such variable:

```
struct Node {
   float Data;
   Node *Next;
};
```

To store data in such a structure, we create an element in memory and assign the appropriate values to the individual components:

```
typedef Node* Nodeptr;  // data type for the pointer to Node
Nodeptr Node;
Node = new Nodeptr;
cin >> (*Node).Data;  // data from input is put into structure
(*Node).Next = NULL;  // the next element does not exist
```

To access the structure components, we need both an asterisk for dereferencing the pointer and a dot for selecting the component. However, the asterisk does not have the necessary precedence to properly serve as a dereference, so we need to connect it to the pointer name with a round bracket (*First).Data. This syntax can be simplified with the special operator "->" and we can then write Prvni->Data.

Suppose we have one element with data stored after the previous commands. Now we want to save a second element and append it to the first. There are two ways to do this - join the new element before the existing one, or after it.

We will create a new element. For this operation, we need a second pointer because we must not lose the address to the first element.

```
Nodeptr Pom;
Pom = new Nodeptr;
cin >> (*Pom).Data;  // we put the new data into the created
structure
(*Pom).Next = First;  // existing element is connected to the
new one
Frist = Pom;  // the list has two elements: the new one is the
first and the latter is the second
```

If we repeat these steps, we get successively new elements at the beginning of the list and the stored data is placed in reverse order.

For the second method, that is, appending a new element after an existing one, we can write:

```
(*First).Next = Pom;
```

Suppose we already have several elements in the list and we want to add a new element to the end:

```
Pom = First;
while ((*Pom).Next!=NULL) Pom=(*Pom).Next;  // finding the end
(*Pom).Next= new Nodeptr;  // creating a new element and
connecting to the last one
Pom = (*Pom).Next;  // Pom points to the new elementPom nyní
ukazuje na nový prvek
cin >> (*Pom).Data;  // new data from input to new element
(*Pom).Next = NULL;  // this is the end of the list
```

To avoid having to scroll through the entire list looking for the last element when inserting each new element, it is very useful to store the address of the last element in a special Last. Then the lookup and the while loop are eliminated and the creation of the new element, including the binding to the current end, is written:

```
(*Last).Next = new Nodeptr;
Last = (*Last).Next; // a new last element of the list
```

If we add new elements to the top, the list acts as a stack. If we add new elements to the end, the list works as a queue.

15.1.7

The input is a sequence of decimal numbers. Write them in reverse order. Use a dynamic list in the form of a stack. Select one of the options to insert it into the following code:

```
  typedef struct Node {
     float Data;
     Node *Next;
  } Node;
  typedef Node* Nodeptr;
  Nodeptr Top=NULL, Help;
  float curdata;
  while (cin>>curdata){
     // ------> select appropriate code HERE:
  }
  Help=Top;
  while (Help!=NULL){
    cout << (*Help).Data << endl;
    Help=(*Help).Next;
  }
```

- Help = new Node; Help->Data=curdata; Help->Next=Top; Top=Help;
- Top = Help; Help = new Node; (*Help)->Data=curdata; (*Help)->Next=Top;
- while (Help->Next!=NULL) Help = Help->Next; Help = new Node; Help->Data = curdata; Help->Next = NULL;
- Top->Next = new Node; Help = Top->Next; Help->Data = curdata; Help->Next = NULL;

15.1.8

Input data consists of a sequence of real numbers. List those that are above average. Choose the appropriate part of the following code:

```
typedef struct Node {
     float Data;
```

```
    Node *Next;
  } Node;
  typedef Node* Nodeptr;
  Nodeptr Top=NULL, Tail=NULL, Aux;
  float curdata, Sum=0;
  int Count=0;
  while (cin>>curdata){
    Sum+=curdata; Count++;
// -------> select appropriate code HERE:

    Tail->Data=curdata;
    Tail->Next=NULL;
  }
  curdata = Sum / Count;
  Aux=Top;
  while (Aux!=NULL){
    if (Aux->Data > curdata) cout << Aux->Data  << endl;
    Aux=Aux->Next;
  }
```

- if (Tail==NULL) { Top = new Node; Tail = Top; } else { Tail->Next = new Node; Tail = Tail->Next; }
- if (Tail==NULL) { Tail = new Node; Tail = Top; } else { Tail = new Node; Tail = Tail->Next; }
- if (Tail==NULL) { Top = new Node; Top = Tail; } else { Top->Next = new Node; Tail = Top->Next; }
- if (Tail!=NULL) { Tail = new Node; Tail = Tail->Next; } else { Top = new Node; Tail = Top; }

🕮 15.1.9

The pointers we used were associated with the dynamic variable data type. So, if you needed to allocate memory, the compiler knew exactly how much memory to allocate and how it would work with dynamic variables.

A special data type is **void**, it is an empty type. When we make a pointer to void, the compiler does not know what variable it will be but knows that it is a pointer. We call this pointer a general pointer.

For example, we can create a linear list whose data component can be anything – we make it as a general pointer. We can then use such a list for any data or we can store something else on each node.

If we are working with a dynamic variable that is referenced by a general pointer, we must typecast the data with the appropriate data type, giving the compiler the necessary information to perform the operation.

We use for example general linear list:

```
typedef struct Node {
     void *Data;
     Node *Next;
} Node;
typedef Node* NodePtr;
NodePtr Top;
```

If we want to add a value to such a list, we will create a dynamic variable of any type, and then insert that pointer as data:

```
float *CurData = new float;
cin >> *CurData; // dynamic variable with inserted number as a
data
Top = new Node;
Top->Data = CurData; // store pointer to node
```

Note that each pointer can be converted to void* but not vice versa. The reverse conversion requires typecasting. So if we get the original value from the list later, we have to write:

```
CurData = (float*)Top->Data;
```

15.1.10

Input data consists of a sequence of real numbers. List those that are above average. Use the linear list with general pointers to data. Choose the appropriate part of the following code:

```
 typedef struct Node {
     void *Data;
     Node *Next;
  } Node;
  typedef Node* Nodeptr;
  Nodeptr Top=NULL, Tail=NULL, Aux;
  float *curdata, Sum=0, Average;
  int Count=0; curdata = new float;
  while (cin>>*curdata){
     Sum+=*curdata; Count++;
     if (Top==NULL) {
```

```
         Top = new Node;
         Tail = Top;
      } else {
         Tail->Next = new Node;
         Tail = Tail->Next;
      }
      Tail->Data=curdata;
      Tail->Next=NULL;
      curdata = new float;
   }
   Average = Sum / Count;
   Aux=Top;
   while (Aux!=NULL){
// enter appropriate code HERE:

     if (*curdata > Average) cout << *curdata  << endl;
     Aux=Aux->Next;
   }
```

- curdata = (float*)Aux->Data;
- curdata = Aux->Data;
- curdata = (float)Aux->Data;
- curdata = float*(Aux->Data);

📖 15.1.11

In the previous section, we discussed a linear list as an example of a dynamic structure. The second commonly used structure is a **tree**. A simple case of a tree is a binary tree, where each node has no more than two successors.

The basic element of the binary tree is a node containing data and two pointers to successors:

```
struct Node {
   int data;
   Node *toleft;
   Node *toright;
};
```

A binary tree is typically used to store data in an ordered form. It is often called a **binary search tree**. In such a tree, the data on the left has values smaller and the data on the right has values greater than the value of the node.

The basic operations are inserting a new node into the tree, searching for a node with the entered data and systematically listing the tree contents.

A binary tree is a typical recursive structure. Order rules apply in each of the three nodes father – left son – right son. Therefore, when implementing operations, we encounter recursive notation.

Suppose we want to store a sequence of integer values from standard input in a tree structure. The previously defined Node data type can serve as the node of this tree. We declare:

```
Node *Root;
int InpNum;
```

Firstly we create an empty tree:

```
Root = NULL;
```

To read the input data and save it to the tree, do the following:

```
while (cin>>InpNum)
   TInsert(Root, InpNum);
```

The TInsert operation is a recursive procedure defined as follows:

```
void TInsert(Node* &father, int v){
  if (father==NULL) {
     father=new Uzel;
     father->data=v;
     father->toleft=NULL;
     father->toright=NULL;
  } else {if (father->data>v) TInsert(father->toleft, v);
          else TInsert(father->toright, v);}
}
```

Now we can do a search in the tree of values. Suppose we want to search the value C:

```
if (IsInTree(Root, C)) cout << "The value was found." << endl;
    else cout << "The value was not found." << endl;
```

The function IsInTree is defined as:

```
bool IsInTree(Node *father, int what) {
   Node *Aux;
   Aux = father;
   while ((Aux != NULL) and (Aux->data != what)){
```

```
      if (what < Aux->data) Aux = Aux->toleft;
          else Aux = Aux->toright;
  }
  return Aux!=NULL;
}
```

You can print the entire tree to standard output using the following recursive procedure:

```
void InOrder(Node* father){
  if (father != NULL) {
     InOrder(father->toleft);
     cout << father->data << " ";
     InOrder(father->toright);
  }
}
```

If the data in the tree is ordered according to the given rules, this statement will get a sequence that is sorted in ascending order. Thus, the binary search tree can be used to sort values.

15.1.12

We have an ordered binary tree with real numbers as data, "Left" and "Right" are pointers to successors. The tree is full of data, the root node is pointed by pointer Top. Select the appropriate piece of code for reading and searching of value F in this tree.

- cin >> F; Cur = Top; while ((Cur != NULL) and (Cur->data != F)) { if (F <| Cur->data) Cur = Cur->Left; else Cur = Cur->Right; } cout <|<| (Cur == NULL ? "No." : "Yes") <|<| endl;
- cin >> F; Cur = Top; while ((Cur == NULL) and (Cur->data == F)) { if (F <| Cur->data) Cur = Cur->Right; else Cur = Cur->Left; } cout <|<| (Cur == NULL ? "No." : "Yes") <|<| endl;
- cin >> F; Cur = Top; while ((Cur->data != NULL) and (Cur != F)) if (F <| Cur->data) Cur = Cur->Right; else Cur = Cur->Left; cout <|<| (Cur == NULL ? "No." : "Yes") <|<| endl;
- cin >> F; while ((Top != NULL) and (Top->data != F)) if (F >= Top) Top = Top->Right; else Top = Top->Left; cout <|<| (Top == NULL ? "No." : "Yes") <|<| endl;

🕮 15.1.13

As previously mentioned, a *reference* is used to get the address of an object that is already in memory. It does not matter whether it is a static variable declared by a normal declaration or is a dynamic variable that was created by an allocation during a program. The reference is written by the & operator before the variable identifier. For example:

```
int Count;
int* IntPtr; // pointer to int
IntPtr = &Count;  // the address of Count is stored to IntPtr
```

So we have two ways to access the value of variable Count:

```
Count = 10;
*IntPtr = 20;
```

Since the variable address Count is stored in the variable IntPtr, dereference gives the same memory space as writing the Count identifier, which also represents the memory address. After executing these commands, we can see the effect of working with the address. By output,

```
cout << Count << endl;
```

we will see the value 20.

Further broad usage of references can be found when passing subroutine parameters. The parameter passed by the reference can change the value within the subroutine, but because the *address* of the actual parameter is passed (no value only), the changed value is reflected in the actual parameter. Example:

We have two variables, x and y, and we need to swap their values. This is a typical task as a part of sorting algorithms. So we construct the procedure Swap:

```
void Swap(float &x, float &y){
  float Aux = x;
  x = y;
  y = Aux;
}
```

If we realize that we are working with the addresses of the actual parameters, this is the possibility of working in two ways with the variable above: by specifying its identifier, or through the address obtained by the & operator.

15.1.14

We have to construct a procedure that sums values of two parameters into a third parameter. The proposed shape of this procedure is:

```
void Sum(double Op1, double Op2, double Result){
  Result = Op1 + Op2;
}
```

We have the following code:

```
double A = 0, B = 7, C = 10;
Sum(A, B, C);
cout << C << endl;
```

The output contains: 10

True or false?

- True
- False

15.2 Pointers (programs)

15.2.1 Pointers 1

The input is a series of 32 binary digits, which can be preceded by any number of spaces. Display the float value that this binary sequence represents.

15.2.2 Pointers 2

The input is a sequence of real numbers (max. 100). Write this sequence in reverse order. Use dynamically allocated array.

15.2.3 Pointers 3

The input is a sequence of integer numbers. Write this sequence in reverse order. Use array with dynamically allocated items. Free all allocated items at the end.

15.2.4 Pointers 4

The input is a sequence of integer numbers. Write this sequence in reverse order. Use dynamically allocated linear list.

15.2.5 Pointers 5

The input is a sequence of integer numbers. Calculate the standard deviation of this sequence; the formula is $s = \mathrm{sqrt}\{1/N \ \mathrm{sum}_{i=1}^{N} (x_i\text{-}avg)^2\}$. Use dynamically allocated linear list as a queue.

15.2.6 Pointers 6

Inputs include worker performance records in the form of number pairs – worker number (integer, interesting are only numbers 1..20) and working hours (real). Create an overview of each worker's performance in the form of a list of worker numbers and a list of their working hours. List only workers with non-zero working hours. Use an array of dynamically allocated linear lists.

15.2.7 Pointers 7

Input can contain values (max. 100) of three different types: integers, real numbers and strings. The first value is a character that specifies the type of the following values (i = integer, r = real, s = string). It is followed by appropriate values. Write a program that lists the values you entered in reverse order. Use an array with void* items.

15.2.8 Pointers 8

Inputs are the dimensions of circles and rectangles. The circle information starts with c and continues with the radius (real number), the rectangle information begins with r and continues with the two sides. List the circles and the specified number of rectangles. Use the list with data of void* type.

⌨ 15.2.9 Pointers 9

The input is a sequence of integer numbers. Write this sequence in descending order. Use a binary tree to sort.

15.3 Pointers and arrays – array implementation

🕮 15.3.1

The array is actually implemented as a dynamic variable. The difference is that it does not allocate with new, but when declaring the array, needed space is allocated in memory. Therefore, a variable whose value at the time of the declaration determines the required number of items can be used to delimit the array size.

The array identifier can be understood as a pointer to the beginning (first component) of the array in memory.

In some situations, an array behaves like a pointer – for example, if you use an array identifier without an index, or if you pass an array as a subroutine parameter:

```
int MyNumbers[20];  // an array declared
cout <<"My magic number: "<<MyNumbers<< endl;
```

On output, we will see for example 0x7fffc2c4c550, i.e. address of an array in memory.

The same effect as the mentioned array declaration has the following construction:

```
int *MyNumbers;
MyNumbers = new int [20];
```

A small difference is the possibility to allocate and deallocate with **new** and **delete** in a random time.

Remember the Swap procedure with slightly changed parameters:

```
void Swap(int A[6], int x, int y){
// procedure works with array A which is passed as a pointer
  int aux = A[x];
  A[x] = A[y];
  A[y] = aux;
}
int VV[6] = {1, 2, 3, 4, 5, 6}; Swap(VV, 1, 3);
for (int i=0; i<6; i++) cout << VV[i] << " ";
```

On output, we will see 1 4 3 2 5 6

Conversely, if you want to prevent array component values from being changed within a subroutine, you specify this parameter with **const** modifier, for example:

```
double Sum(const double X[30]){ // array X will not be changed
  double Result = 0;
  for (int i=0; i<30; i++) Result+=X[i];
  return Result;
}
```

15.3.2

We have the following function:

```
bool Mirror(double Values[10]){
   if (Values[3] > 7.5 and Values[5] <= 17){
      Values[4]+=Values[3];
      return true;
    } else return false;
}
double MyArray[10]; for (int i=0; i<10; i++) cin >>
MyArray[i];
if (Mirror(MyArray)) cout << "Done.\n";
```

On output we see "Done." Was the MyArray changed?

- True
- False

15.4 Pointers and arrays (programs)

15.4.1 Array allocation 1

The first value on the input is a quantity N and then integer numbers of this quantity. Calculate the standard deviation of this sequence; the formula is

$s = sqrt{1/N\ sum_{i=1}^{N} (x_i\text{-}avg)^2}$.

Use dynamically allocated array and after the calculation free used memory.

⌨ 15.4.2 Array allocation 2

There are integer numbers on the input. The quantity of numbers doesn't exceed 50. Use a dynamic linear list as a stack and a static array for input values. Calculate products of opposite items (ie first * last, second * penultimate etc.) and write these products to the standard output. Compare the amount of memory needed for the stack and the static array. If the stack takes up more space, write the string "stack!" at the end of the data.

15.5 Pointer arithmetic

🕮 15.5.1

As we have already mentioned, the array identifier is essentially a pointer to where the array lies. Thus, array indexing is the same as finding the corresponding component address. However, this address can also be determined by direct manipulation of the array address, ie the value of the pointer.

Arithmetic operations with pointers

Suppose we have pointers p1 and p2 of any type (for example int *p1, *p2;) and integer x.

Pointer arithmetic – list of operations

Operation	Result type	Description
`p1 + x`	as p1	new address shifted up by x items (by type of pointer)
`p1 - x`	as p1	new address shifted down by x items (by type of pointer)
`p1++`	as p1	new address shifted up by 1 item
`p1--`	as p1	new address shifted down by 1 item
`p1 - p2`	int	number of items between p1 and p2
`p1 == p2`	bool	is the address p1 the same as address p2?
`p1 != p2`	bool	is the address p1 different from address p2?
`p1 > p2`	bool	is the address p1 greater than address p2?
`p1 >= p2`	bool	is the address p1 greater than or equal to address p2?
`p1 < p2`	bool	is the address p1 lower than address p2?
`p1 <= p2`	bool	is the address p1 lower than or equal to address p2?

15.5.2

We have an array and a pointer:

```
single Temperatures[5] = {0.5, 1.6, 5.5, 10.2, 21.2};
single *temp, result;
```

Now we assign:

```
temp = &Temperatures[1];
result = *temp - *(temp+1) + *(temp+3);
cout << result << endl;
```

What we will see on screen?

- 17.3
- 17.3
- 9.1

- 28.3

15.5.3

We have an array:

```
int Open[200];
```

This array is filled with some values and we need to write it to output:

```
int *i, k; i = &Open[0];
for (k=0; k<200; k++) {
   if (k % 10 == 0) cout << endl;
   //  ***************
}
```

Select the appropriate option in a starred location.

- cout <|<| *(i + k) <|<| " ";
- cout <|<| *i + k <|<| " ";
- cout <|<| *i[k] <|<| " ";
- cout <|<| *[i + k] <|<| " ";

15.5.4

We have the following declarations:

```
struct Node{
   int Id;
   int Num;
   double Descr;
};
Node *Arr, *Current;
```

and we process the following statements:

```
Arr = new Node[10];
Current = Arr + 1;
cout << Arr << " " << Current << endl;
```

What will be the difference between the values of written addresses?

- 16

- 1
- 4
- 8

15.6 Pointers arithmetic (programs)

⌨ 15.6.1 Pointer arithmetic 1

The input contains a sequence of integer values. List these values in ascending order. Use adaptive bubble sorting. Use pointer arithmetic to access neighbouring values in the array.

⌨ 15.6.2 Pointer arithmetic 2

The input is a sequence of integer values. Write this sequence in reverse order. Use an array with access via pointer arithmetic.

⌨ 15.6.3 Pointer arithmetic 3

The input is a sequence of integer values. Write all values which are higher than the average input values. Use an array and access into elements via pointer arithmetic.

⌨ 15.6.4 Pointer arithmetic 4

There are pairs of values at the input. The first number indicates the day of the week (1 is Monday, etc.), the second number indicates the volume of production in euro. Sum the production volume for each day of the week and output these totals. Use an array with pointer arithmetic to access it.

Pointers II

Chapter 16

16.1 Pointers to subroutines

🕮 16.1.1

Pointers represent addresses in memory. At a certain address in memory, there may be data, but also instructions – a sequence of instructions, or subroutines.

Consider the following function:

```
int First(){
   return 2;
}
```

When we want to call this function, we have to write for example

```
cout << First() << endl;
```

and obtain result 2.

And what is happen when we write

```
cout << First << endl;
```

This is the use of function identifier which represents the address of function First, so we can obtain somewhat as 0x4007e4 (or 1 only – it depends on compiler interpretation of address).

Address of function can be stored into pointer variable which can be declared similar to subroutine header as:

```
int (*funPtr)();
```

Note that syntax construction

```
int *funPtr();
```

has a quite different meaning – this is a forwarded declaration for a function named funPtr that takes no parameters and returns a pointer to an integer.

So back to the previous construction: we have now a variable funPtr which can be assigned:

```
funPtr = First;
```

The return data type and all parameters of the declared pointer and assigned function must match.

We can call stored function in two ways – explicit dereference, or implicit dereference:

```
(*funPtr)();  // explicit dereference
funPtr();  // implicit dereference
```

As you can see, the implicit dereference method looks just like a normal function call -- which is what you'd expect, since normal function names are pointers to functions anyway. However, some older compilers do not support the implicit dereference method, but all modern compilers should.

16.1.2

We need to declare a pointer to a boolean function that takes two integer parameters.

```
_____
```

Now we declare two functions of the appropriate type:

```
bool Descending(int a, int b){
    return a > b;}
bool Ascending(int a, int b){
   return a < b;}
```

And finally, we decide which function will be in process and will be assigned to a pointer variable:

```
char C; cin >> C;
if (C=='A') _____
```

- funCompare = (*Ascending); else funCompare = (*Descending);
- funCompare = Ascending; else funCompare = Descending;
- int (*funCompare)(bool a, bool b);
- bool (*funCompare)();
- bool (*funCompare)(int a, int b);

16.1.3

The complex syntax of the pointer definition on a subroutine can be somewhat simplified by the type definition. For example, let's define a type of relation function for two real numbers that produce a logical value:

```
typedef bool (*TRelation)(double, double);
```

After that, we can simply write:

```
TRelation AscDouble;
```

or we can simply write this type as a parameter of another subroutine, for example:

```
void InsertSort(TMyValues Values, int Size, TRelation HowSort)
...
```

Function for comparison of array items is passed as a parameter, so we can determine how the sorting will be processed (ascending, descending, for decimal part only etc.).

16.1.4

We want to write a simple calculator. The user will enter one character (operator +, -, *, /) and two numbers, the calculator will process the appropriate operation and display a result.

```
float Add(float a, float b){return a + b;}
float Subtr(float a, float b){return a - b;}
float Multi(float a, float b){return a * b;}
float Divid(float a, float b){if (b!=0) return a / b; else
return 0;}
//*****************
char op;
float num1, num2;
cin >> op >> num1 >> num2;
switch (op){
 case '+': Calc = Add; break;
 case '-': Calc = Subtr; break;
 case '*': Calc = Multi; break;
 case '/': Calc = Divid; break;
}
cout << num1 << op << num2 << " = " << Calc(num1, num2)<<
endl;
```

Select the correct variant to the starred place.

- typedef float (*TAritOp)(float, float); TAritOp Calc;
- typedef float *TAritOp (float, float); TAritOp Calc;
- typedef (*TAritOp) float (float, float); TAritOp Calc;

- typedef float (float, float) TAritOp; TAritOp Calc;

16.1.5

For example, if we use a linear list, its structure is exactly the same for any type of data. Without dependence on the data item, a new element is inserted, the list goes through, etc. Some operations, of course, need some knowledge of the data, for example, deleting a list element needs to first find which data is to be deleted. If the data is in the form of general pointers, we cannot do this until we know the actual data type to which the general pointer points.

The subroutine, which is then used to process the data item, will always change according to the actual data in the list. It is therefore advantageous to make such subroutines in the form of a pointer to a subroutine in which the corresponding specific function is always assigned.

Typically, we need a function that is able to compare two data items (greater than, lower than, equal to), or a procedure that is capable of displaying data into an output:

```
typedef bool (*TRelation)(void*, void*);  // to compare data
typedef void (*TDisplay)(void*);  // to display data
```

16.1.6

We have a linear list with data items in the form of general indicators. We need to list the contents of the list on standard output. We know that at this point, the data is an integer type.

Select the correct variant of code to starred place:

```
void DispInt(void* X){
  cout << (*(int*)X) << endl;
}
struct Node {
   void * data;
   Node * Next;
};
typedef *Node ItemPtr;
//***************************
ItemPtr Start, Aux;  // list is already filled with data
HowToDisplay = DispInt;
Aux = Start;
while (Aux != Null){
```

```
    HowToDisplay(Aux->data);
    Aux = Aux->Next;
}
```

- typedef void (*THow)(void*) THow HowToDisplay;
- typedef void (*HowToDisplay)(void*) HowToDisplay THow;
- typedef int (*THow)(void*) HowToDisplay THow;
- typedef void (THow)(void) THow DispInt;

16.2 Pointers to subroutines (programs)

⌨ 16.2.1 Subroutine pointers 1

The input contains a sequence of integer values. Write these values in ascending or descending order. The sorting order is determined by the letter A or D entered as the first input value.

⌨ 16.2.2 Subroutine pointers 2

The input contains a sequence of integer values. Depending on whether the last value entered is even or odd, write every second or every third value of the input series.

⌨ 16.2.3 Subroutine pointers 3

The input contains a sequence of integer values. Use binary tree with void* data type to ascending sort and display these values.

⌨ 16.2.4 Subroutine pointers 4

The input contains a sequence of integer values. Use the dynamic linear list as a stack with void* data type to display these values in reverse order. Write one space between the output values, but if the total number of values is less than 10, type a semicolon and space between the values. Create output shape variants as two different procedures.

⌨ 16.2.5 Subroutine pointers 5

The input contains a sequence of characters ended by a dot. Display cypher of this sequence so that uppercase letters will be replaced by corresponding downcase letters, downcase letters will be replaced by letters shifted by 4 in alphabet and spaces will be replaced by a plus sign. Other characters will be displayed without changes. For various modes of displaying create separate functions.

16.3 Struct as a member of dynamic structure

🕮 16.3.1

Struct is a user-defined data type that is essentially a collection of variables of different types under a single name. For example, we have a structure with data about a person:

```
struct TPerson {
   string Name;
   int Age;
}
```

We can declare a variable for personal data and this data type can serve as a basis for the pointer definition:

```
TPerson Worker;
typedef TPerson *TPersPtr;
```

We will typically work with multiple people. These can be inserted into a linear dynamic list or into an array. So we can create the following data types:

```
struct Node {  // one member of linear dynamic list
   TPersPtr Person;  // pointer to personal data
   Node *Next;  // pointer to next member of linear list
}
```

Or:

```
typedef TPersPtr TPersons [35];  // array of pointers to
personal data
```

16.3.2

Input data are names and ages of persons. List people whose age is below the group average. In the following program, select the piece of code that belongs to the starred location.

We use the common list and put the definition of personal data as a data item.

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

bool IsRead(TPersPtr &P){
   P = new TPerson;
   return cin >> P->Name >> P->Age;
}
int main(){
  typedef struct Node { // common queue
     void *Data;
     Node *Next;
  } Node;
  typedef Node* Nodeptr;
  Nodeptr Top=NULL, Tail=NULL, Aux;
  int Sum=0, Count=0;
  float Average;
  TPersPtr Curdata;
  while (IsRead(Curdata)){
     Sum+=Curdata->Age; Count++;
     if (Top==NULL) {
        Top = new Node;
        Tail = Top;
     } else {
        Tail->Next = new Node;
        Tail = Tail->Next;
     }
     Tail->Data=Curdata;
     Tail->Next=NULL;
  }
  Average = Sum / Count;
  Aux=Top;
  while (Aux!=NULL){
    Curdata = (TPersPtr)Aux->Data;
    if (Curdata->Age < Average) cout << Curdata->Name  <<
endl;
    Aux=Aux->Next;
```

```
  }
  return 0;
}
```

- typedef struct { string Name; int Age; } TPerson; typedef TPerson *TPersPtr;
- TPerson struct { string Name; int Age; }; typedef TPersPtr *TPerson;
- typedef struct { string Name; int Age; } TPerson; typedef *TPerson TPersPtr;
- typedef struct { string Name; int Age; } TPersPtr; typedef TPerson *TPersPtr;

16.3.3

The same task as in the previous example – it is necessary to determine from the list of persons who have below-average age. This solution is done using an array of people.

In the following program, select the piece of code that belongs to the starred location.

```
#include <iostream>
using namespace std;
typedef struct {
     string Name;
     int Age;
} TPerson;
typedef TPerson *TPersPtr;
const int MaxPerson = 65;
typedef TPersPtr TPersArray[MaxPerson];
bool IsRead(TPersPtr &P){
   P = new TPerson;
   return cin >> P->Name >> P->Age;
}
int main(){
  int Sum=0, Count=0;
  float Average;
  TPersArray Persons;
  TPersPtr Curdata;
  for (int I=0; I<MaxPerson; I++) Persons[I]=NULL;
//**************************

  Average = Sum / Count;
  for (int I=0; I<Count; I++)
     if (Persons[I]->Age < Average)
        cout << Persons[I]->Name << endl;
  return 0;
```

```
}
```

- while (IsRead(Curdata)){ Persons[Count]=Curdata; Sum+=Curdata->Age; Count++; }
- while (IsRead(Curdata)){ Persons[I]=Curdata; Sum+=Curdata.Age; Count++; }
- while (IsRead(Curdata)){ Persons[Count]=*Curdata; Sum+=(*Curdata).Age; Count++; }
- while (IsRead(&Curdata)){ Persons[Count]=&Curdata; Sum+=Curdata->Age; Count++; }

🕮 16.3.4

As mentioned early the structures can be not only array items but also members of dynamic structures such as linear lists or trees. A very frequent dynamic structure is a linear list. Its element is a record containing some data and a pointer to the next same record:

```
  typedef struct Node {
     float Data;
     Node *Next;
  } Node;
```

We can define the datatype of struct pointer:

```
  typedef Node* Nodeptr;
```

The basic list operations include adding an element (at the beginning, at the end, at another location) and removing the element (from the beginning, from the end, from another location).

Suppose we have a pointer to the existing start of the list and any auxiliary pointer:

```
Nodeptr Start, Aux;
```

Adding an element at the beginning can be processed by the following code:

```
Aux = new Node;   // pointer Aux has address to a new empty
element
Aux->Data = 6.88; // value to Data item
Aux->Next = Start; // pointer to the next member points recent
starter element
Start = Aux;
// pointer to the beginning of the list points now to the new
element
```

Removing an element from the beginning can be processed by the following code:

```
  Aux = Start; // pointer Aux points to the start of list
  Start = Start->Next; // new beginning of list is now on
second member
  delete Aux;  // recent starter element was deleted
```

16.3.5

The input is a sequence of integer values. Write this sequence in reverse order. Use a dynamic linear list as a stack. Add the corresponding parts to the following code:

```
#include <iostream>
using namespace std;
int main(){
  typedef struct Node {
     long int Data;
     Node *Next;
  } Node;

_____
  Nodeptr Top=NULL, Aux;
  long int curdata;
  while (cin>>curdata){
     Aux = new Node;
     Aux->Data=curdata;
     _____
  }
  Aux=Top;
  while (Aux!=NULL){
    cout << Aux->Data << endl;
    _____
  }
  return 0;
}
```

- Top->Next=Aux; Top=Aux;
- Top=Top->Next; delete Aux; Aux=Top;
- typedef Nodeptr * Node;
- Node* Nodeptr;
- Top=Aux->Next; delete Top; Aux=Top;
- Aux->Next=Top; Top=Aux;
- typedef Node* Nodeptr;

16.4 Pointers and structs (programs)

⌨ 16.4.1 Pointers to struct 1

Inputs are the dimensions of circles and rectangles. The circle information starts with c and continues with the radius (real number), the rectangle information begins with r and continues with the two sides. List shapes in descending order of area. Display input values and calculated areas. Use the binary tree with data of pointer to a struct type.

16.5 Pointers to functions as a parameter

🕮 16.5.1

Some standard operations, such as search and sort, are treated as library procedures. These subroutines are able to work with any array. However, it is necessary to define a comparison function for the search or sorting itself, which can handle the data.

The *qsort()* function is defined in C++ header *cstdlib* and sorts a given array in ascending order using the Quicksort algorithm. The *qsort()* function uses a comparison function to decide which element is smaller/greater than the other.

The header of the *qsort()* function looks like this:

```
void qsort (void* base, size_t num, size_t size, int
(*compare)(const void*,const void*));
```

The *qsort()* function sorts the given array pointed by the base in ascending order. The array contains *num* elements, each of *size* bytes. The function pointed by *compare* is used to compare two elements of the array. This function modifies the content of the array itself in ascending order. However, if two or more elements are equal, their order is undefined.

16.5.2

Task: We have a sequence of the real numbers on input. Write this sequence to standard output ascending sorted by absolute values. Use the standard qsort function.

Fill in the following program the correct piece of code into starred place:

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

int main() {
  const int MaxValues = 1000;
  typedef float TValArray[MaxValues];
  TValArray VA;
  int Count=0;
  while (cin>>VA[Count]) Count++;
  qsort(VA, Count, sizeof(float), MyComp);
  for (int I=0; I<Count; I++) cout << VA[I]<<", ";
  cout << endl;
  return 0;
}
```

- int MyComp (const void* a, const void* b){ if (*(float*)a<|*(float*)b) return -1; if (*(float*)a>*(float*)b) return 1; return 0; }
- int MyComp (void* a, void* b){ if (*(float*)a<|*(float*)b) return -1; if (*(float*)a>*(float*)b) return 1; return 0; }
- int MyComp (const void* a, const void* b){ if (*a<|*b) return -1; if (*a>*b) return 1; return 0; }
- int MyComp (float a, float b){ if (a<|b) return -1; if (a>b) return 1; return 0; }

🕮 16.5.3

Effective retrieval of an element's presence in the data can only be realized with prepared data. If the data is jumbled in a linear structure (array, list), the only sequential search can be used. A binary search algorithm (BSearch) can be used in ordered array data.

The bsearch() function in library cstdlib of C++ language performs a binary search of an element in an array of elements and returns a pointer to the element if found. The bsearch() function requires ascending ordered array.

The header of bsearch function looks like this:

```
void* bsearch (const void* key, const void* base, size_t num,
size_t size, int (*compare)(const void*,const void*));
```

The principle is similar to the qsort function. The first parameter is a key (searching value), the base is a pointer to an array start (or simple array identifier), num is a number of array elements and size is the size of one array element. The last parameter is the function for the comparison of values (key and array element). In

order to perform the search, the bsearch() function makes a series of calls to the function pointed by comparing with key as the first argument and an element from the array as the second argument. Therefore, the key data type may not be identical to the data type of the array item. For example, we can only search for goods records by product name or stock number.

16.5.4

Task: We have an ordered array with float values. Find the index of values entered from standard input. For the searching use the standard function bsearch.

Fill in the following code with the correct option into starred place.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int MyComp (const void* a, const void* b){
  return *(float*)a - *(float*)b;
}
int main() {
  const int MaxValues = 1000;
  typedef float TValArray[MaxValues];
  TValArray VA;
  int Count;
  float Key, *Result;
  cin>>Count;
  for (int I=0; I<Count; I++) cin>>VA[I];
  qsort(VA, Count, sizeof(float), MyComp); // preparation of
data
  while (cin>>Key){
// ***************************
  }
  return 0;
}
```

- Result = (float*)bsearch(&Key, VA, Count, sizeof(float), MyComp); if (Result!=NULL) cout <|<| "A number "<|<|*Result<|<|" was found on index " <|<| Result - VA <|<| endl; else cout <|<|"Value "<|<| Key<|<|" was not found."<|<|endl;
- Result = bsearch(&Key, VA, Count, sizeof(float), MyComp); if (Result!=NULL) cout <|<| "A number "<|<|*Result<|<|" was found on index " <|<| VA[Result] <|<| endl; else cout <|<|"Value "<|<| Key<|<|" was not found."<|<|endl;

- Result = (float*)bsearch(&Key, VA, Count, sizeof(float), MyComp); if (Result!=0) cout <|<| "A number "<|<|VA[Result]<|<|" was found on index " <|<| Result <|<| endl; else cout <|<|"Value "<|<| Key<|<|" was not found."<|<|endl;

16.6 The use of qsort, bsearch (programs)

⌨ 16.6.1 qsort 1

The input contains a sequence of integer values (max. 100). Use the qsort function to sort it and display ascending sequence.

⌨ 16.6.2 qsort 2

The input contains a sequence of characters ended by a dot (max. 300). Use the qsort function to descending sort of these characters.

⌨ 16.6.3 bsearch 1

The input contains a sequence of integer values. Use the qsort function to sort it and use the bsearch function to find numbers from 1 to 20 in the input sequence. Display all found numbers.

⌨ 16.6.4 bsearch 2

The input contains a sequence of characters ended by a dot. Use the qsort function to ascending sort of these characters and use the bsearch function to display characters 'r' to 'z' found in the input sequence.

Memory Management

Chapter 17

17.1 Memory management

🕮 17.1.1

The memory for program variables is divided into two parts: for dynamic variables and for static variables. We have already discussed this fact in the data type pointer.

Now let's look at the efficiency of working with these two parts of memory.

Working with **static variables** is done by the compiler. The variable is created in memory at the time of declaration – the compiler assigns the corresponding address and this address is accessible to the programmer via the variable identifier. By simply writing this identifier, we immediately get to the value that is stored in the variable.

The variable is automatically released when the block in which it was declared ends. It never happens that a memory variable does not become loose, remains partially in memory, or has a bad location.

This means that the variables used are as effective and secure as possible. So what's the disadvantage? We can see two disadvantages: we cannot influence (allocate or release) memory during the program, and this area of memory is small for large data. Nowadays, however, this smaller size is usually quite sufficient for most applications.

Why this part of memory is called **stack**: as mentioned, memory allocation is done when declaring a variable. This declaration can be specified in a block (subroutine, command). It follows from the principle of program operation that a subroutine (or command) with a local variable starts working, thus occupying the appropriate memory, but the previous block from which it was called cannot end earlier. Therefore, the memory that was last occupied is always released first. This is the principle of the stack. Thus, there is never a situation where incoherent sections are created in this part of the memory.

🕮 17.1.2

A so-called heap is a memory area that is used to allocate dynamic variables via pointers. The basic technique to allocate this memory is the **new** operator, and deallocate with **the delete** operator.

In connection with what has been said about stack, modern C++ code tends to use **new** quite rarely, and **delete** very rarely. From a memory standpoint, the disadvantage is that **new** allocates memory of the heap while local objects allocate memory off the stack. Heap allocation times are much slower than allocations off

the stack. However, there are still times when it's appropriate to do so, and a solid understanding of how these low-level facilities work can help with the understanding of what normally happens "below the hood". There are even times when **new** and **delete** are too high-level, and we need to drop back to **malloc** and **free** from C language – but those situations are rare exceptions indeed.

The basic idea of **new** and **delete** is simple: new creates an object of a given type and gives a pointer to it, and delete destroys an object created by new, given a pointer to it. The reason that **new** and **delete** existing in the language is that code often does not know when it is compiled exactly which objects it will need to create at runtime, or how many of them. Thus **new** and **delete** expressions allow for "dynamic" allocation of objects.

For those of you familiar with the C programming language, **new** is a kind of "type-aware" version of malloc: the type of the expression "new int" is "int*". Because new is type-aware, it can also initialize the newly created objects, calling constructors if appropriate. Another enhancement of **new** and **delete** compared to **malloc** and **free** is that the C++ standard provides a standard way to change how **new** and **delete** allocate memory; in C this is normally achieved using a non-standard technique known as "interpositioning".

All dynamically allocated memory must be released before the pointer (except smart pointers) pointing to it goes out of scope. So, if the memory is dynamically allocated for a variable within a function, the memory should be released within the function unless a pointer to it is returned or stored by that function.

The basic **new** and **delete** operators are intended to allocate only a single object at a time; they are supplemented by **new[]** and **delete[]** for dynamically allocating entire arrays. Uses of new[] and delete[] are even rarer than uses of basic new and delete; usually, a std::vector is a more convenient way to manage a dynamically allocated array.

Note that when you dynamically allocate an array of objects, you *must* write **delete[]** when freeing it, not plain **delete**. Compilers cannot usually give an error if you get this wrong; most likely your code will crash when you run it.

When a call to **delete[]** runs, it first retrieves the information stored by **new[]** describing how many elements are present in the dynamically allocated array and then calls the destructor for each element before deallocating the memory. The actual address of the memory block that was allocated may differ from the value returned by **new[]** to allow room to store the number of elements; this is one reason why accidentally mixing the array form of **new[]** with the single-element form of **delete** may lead to crashes.

The particularly astute reader might be wondering if it would be possible to eliminate the need to remember which of **new**/**new[]** and **delete**/**delete[]** to use, and make the compiler figure it out instead. The answer is that it would be perfectly

possible, but doing so would add overhead to each single-object allocation (as delete would need to be able to work out whether the allocation was for a single object or an array), and a design principle behind C++ has been that you "don't pay for what you don't use", so the trade-off made is that single object allocations remain efficient, but users have to take care when using these low-level facilities.

17.1.3

Task: Input data include the name of the goods and the number of pieces in stock. List the goods that are below the average number of items in stock and a list of goods that are no longer in stock (ie, have zero pieces).

The first variant is solved via a simple array, the second variant via a dynamic linear list.

The record of one good:

```
struct TGoods {
   char Name[50];
   int Pieces;
};
```

Suppose we have 150 records on input and an array is declared with 160 elements.

Which of these variants take more memory and how many bytes it will be?

- An array: 9280 bytes; linear dynamic list 9900 bytes. The array is more economical.
- An array: 9280, a linear dynamic list 8700. The list is more economical.
- Both of variants takes the same size of memory.

17.1.4

Assume processing of the order of millions of real numbers (double data type). In the first variant, we decide to use the array and dimension it to 2 million items. In the second variant, we use a linear one-way dynamic list. How many items does the memory consumption of the linear list exceed the memory consumption when using the array?

(Suppose that one pointer takes 8 bytes; the double data type takes 8 bytes.)

- 1 million
- 1.5 million
- 2 millions

- 0.5 million

📖 17.1.5

A dynamic array is similar to an array, but with the difference that its size can be dynamically modified at runtime.

The elements of an array occupy a contiguous block of memory, and once created, its size cannot be changed. A dynamic array can, once the array is filled, allocate a bigger chunk of memory, copy the contents from the original array to this new space, and continue to fill the available slots.

In the backend, dynamic arrays allocate a predetermined amount of memory on creation, which then grows by a certain factor when needed. These parameters, initial size and growth factor are central in terms of performance. If the initial size and growth factor are small, you'll end up reallocating memory often, which isn't good; on the other hand, if they have a higher value, you'll probably have a lot of unused memory, and the resize operation could take longer. The tradeoff is obvious here.

So we have to realize that the possibility of extending field size is usually associated with considerable overhead, so it is necessary to consider when a dynamic array is really needed and when it is quite effective to allocate a large enough array without the possibility of its expansion.

📝 17.1.6

We need to process integer values. We create a dynamic array whose size increments will have ten items. Because we don't know in advance how many items we need, the basic size will also be 10 items. How many times will it be necessary to reallocate memory and copied values if we read 4582 numbers from the input?

- 458
- 460
- 46
- 0

17.2 Storage classes

📖 17.2.1

A storage class defines the scope (visibility) and a lifetime of variables and/or functions within a C++ program. These specifiers precede the type that they modify. There are the following storage classes, which can be used in a C++ program:

- **auto** – this is the default storage class for local variables. The definition without the auto keyword has the same effect.
- **register** – storage class define local variables that should be stored in a CPU register instead of memory. This is useful for quick access to this variable. Typically is used for example in a loop as a control variable or index. This variable must have a maximum size correspond to the register size. This variable cannot be referenced via & operator.
- **static** – the **static** storage class instructs the compiler to keep a local variable in existence during the lifetime of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared. When static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.
- **extern** – is used to giving a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
- When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference to the defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.
- The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions
- **mutable** – applies only to class objects, which are discussed later. It allows a member of an object to override the const member function. That is, a mutable member can be modified by a const member function.

📝 17.2.2

We have the following code:

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

```
XXXXX int Index;
void ToSet(){
  Index = 20;
}
void ToModify(){
  for (YYYYY int I=Index; I>=Index /2; I--) cout << I << ", ";
  Index /=2;
}
int main() {
  ToSet();
  while (Index>0) ToModify();
  return 0;
}
```

Which storage classes fit to XXXXX and YYYYY places?

- XXXXX = static, YYYYY = register
- XXXXX = register, YYYYY = static
- XXXXX = register, YYYYY = extern
- XXXXX = extern, YYYYY = static

17.3 Const and volatile qualifiers

🕮 17.3.1

Qualifiers **const** and **volatile** are written before the datatype in variable declaration.

A qualifier const is used with a datatype declaration or definition to specify an unchanging value. Examples:

```
const int five = 5;
const double pi = 3.141593;
```

The following operations are illegal and end with error:

```
five = 7;
pi += 1;
```

A qualifier volatile specifies a variable whose value may be changed by processes outside the current program. One example of a volatile object might be a buffer used to exchange data with an external device:

```
int CheckBuf() {
   volatile int io_buf;
```

```
   int val;
   while (io_buf == 0) {}; // wait until io_buf is changed by
external process
   val = io_buf; io_buf = 0; // value processed, buffer
cleared
   return val;
}
```

If io_buf had not been declared volatile, the compiler would notice that nothing happens inside the loop and thus eliminate the loop.

Qualifiers const and volatile can be used together to make sure the compiler knows that the variable should not be changed (because it is input-only for example) and that its value may be altered by processes other than the current program.

17.3.2

We have the following code:

```
const float X = 1.5;
void SetFn(float &X){
   X = 4.5;
}
int main() {
   float X = 0;
   SetFn(X);
   cout << X << endl;
}
```

What we will see on output?

- 4.5
- 1.5
- An compilation error.
- 0

17.3.3

We have the following code:

```
const float X = 1.5;
void SetFn(float &X){
```

```
   X = 4.5;
}
int main() {
   float X = 0;
   SetFn(X);
   cout << X << endl;
}
```

What we have to change to obtain value 1.5 on output? (Hint: two changes.)

- To remove & in function parameter, to remove declaration float X = 0;
- To remove const, to remove declaration float X = 0;
- To remove & in function parameter, to remove const
- To remove const, to remove declaration float X = 0;

PRISCILLA

priscilla.fitped.eu