МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Нижегородский государственный университет им. Н.И. Лобачевского

К.А. Баркалов, С.И. Бастраков

Основы программирования

Учебно-методическое пособие

Нижний Новгород

2013

Баркалов К.А., Бастраков С.И. Основы программирования: Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2013. – 48 с.

Ministry of Education and Science of the Russian Federation

State educational institution of higher education
«Lobachevsky State University of Nizhni Novgorod»

K.A. Barkalov, S.I. Bastrakov

Programming Basics

Tutorial

Nizhni Novgorod

2013

Introduction

This course introduces very basic concepts of programming using the C++ programming language. C++ is a descendant of C programming language and we will mostly study a subset of C++ that is also present in C, sometimes this subset is referred as the C/C++ programming language. This tutorial covers the following topics: introduction to programming, basics of the C++ programming language, conditional and loop statements, arrays. It does not cover more advanced programming techniques: pointers and references, functions, strings, object-oriented programming (OOP), and standard template library (STL).

Most of currently popular programming languages has been significantly influenced by C/C++, therefore, good understanding of C++ makes studying many other programming languages, such as C# or Java, much easier. Furthermore, the basic concepts and general principles of programming, which are the focus of this course, are essentially the same for all programming languages.

Course program

1. Introduction to programming

It is well known, that modern computers have amazing capabilities: in numerous areas and applications they are far ahead of what humans could possibly do. Nevertheless, computers do not share human creativity and can only execute instructions given by humans. These instructions are defined by **software** – a set of programs – that computer executes. People who create software are called **software engineers** or **programmers**.

An **algorithm** is an exact order of actions resulting in solution of a given problem. Algorithm is a mathematical term, and all problems solvable by a computer could be formulated as mathematical problems.

A **program** is a text representation of an algorithm using a special language that computer can interpret, called **programming language**. There are a plenty of programming languages: some are general-purpose, some are specialized to certain domains or hardware. Among the most popular general-purpose languages are: C, C++, Java, C#, PHP, Python, Visual Basic.

Throughout this course we will study basics of programming using the C++ programming language. C++ is a descendant of C programming language and we will mostly study a subset of C++ that is also present in C, sometimes this subset is referred as the C/C++ programming language. We cover the following topics: introduction to programming, basics of the C++ programming language, conditional and loop statements, arrays. We do not cover pointers and references, functions, strings, object-oriented programming (OOP), and standard template library (STL).

Most of currently popular programming languages were significantly influenced by C/C++, therefore, good understanding of C++ makes studying other programming languages, like C# or Java, much easier. Moreover, the basic concepts and general principles of programming, which are the focus of this course, are essentially the same for all programming languages.

On a very low level, computers operate with sets of zeroes and ones, and machine instructions are encoded by sequences of 0 and 1. It would be almost impossible to directly write programs as sequences of 0 and 1. Moreover, the particular encoding depends of lots of factors, such as hardware and operating system, which would make software bounded to particular machine. Instead, modern programming languages offer much higher level of abstraction that is closer to natural mathematical thinking and, to a certain extent, to natural language.

The original text in programming language, called **source code** or, simply, **code**, is then transformed into an **executable** (**binary**) **file** – application, that computer could launch. This process of transforming source code into binary file is called **build**, or **compilation**. It is done by special software – **compilers** and **linkers** – that handle all the low-level details. Thus, a software engineer writes a program on relatively high-level programming language, and it is then automatically translated into an executable file. A programmer basically does two things: figure out an algorithm for a given task, create a program that implements the algorithm following rules of a chosen programming language. We will study both basics of algorithms and C++ programming language.

1.1 "Hello world" program

Traditionally the first program one writes while studying a new programming language is a program that prints words "Hello world" to the screen. Let us dive into this program in the C++ programming language:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```

Figure 1.1. "Hello world" program

If we build and run this code, we will get a black screen, called console, with words "Hello world" on it, and that is all this program does.

Let us now go through the code of this program and explain some basic constructions of C++ on this example. The first two lines of this program are a standard way to enable input/output – we need it to print the "Hello world" text later. This is not the only way to input/output in a C++ program, but is one of the most widely used. The line `int main()` defines an **entry point** of the program – a point from which the program starts the execution. A pair of curly braces `{ }` is used to group several lines of code, so-called **block**, together. In this particular case, there is only one block. `cout` stands for console output and is a standard way to print a text or numerical data on the screen. The quotation marks tell that the exact text inside should be put to the screen. The `endl` in the end stands for the end of line and affects the output screen: the next output or a utility 'press any key to continue' will appear

on the next line and not clamp with the “Hello world” text. The `return 0;` line marks the end of the program. Notice that some words in the source code are given in blue: these are special constructs of C++ with a predefined meaning or behavior, called **keywords**. It is important to note, that C++, as most of other programming languages, is case-sensitive: capital and small letters are different. For this reason, one cannot use `Cout` instead of `cout` or `Return` instead of `return` – that would be a violation of C++ rules, and the corresponding source code would not build into an application.

Let us now modify the given program to output a slightly different text:

“Hello world.

This is our first C++ program!”.

Apparently, we need to add another `cout` command after the first one:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world" << endl;
    cout << "This is our first C++ program!" << endl;
    return 0;
}
```

Figure 1.2. Modified “Hello world” program

1.2. More examples

As you could notice, the programs given in the previous subsection do not interact with a user: on launch they just print a message on the screen. In reality software usually interacts with a user: its behavior depends on data the user provides and actions the user performs.

As an example of a very simple interaction with a user let us create a simple calculator program that inputs two integer number and computes a sum of the given numbers. Even though this program is very simple and is not practically useful (there are far more powerful calculator programs), it raises a fundamental problem. We need the program to operate on the data that user inputs: the same program should work for any numbers user could input. When we create a program by writing a source code, numbers are not prescribed, they are only input when the program is

running. Therefore, the only way to make such a program is to make is general, not dependent on specific values of input numbers. Luckily, it is not that hard for computing a sum: if we denote input numbers as `a`, `b`, then `sum = a + b` for any `a`, `b`. So what we need to do is to save the data that user inputs in pieces of computer memory (RAM) with names `a`, `b`, and then execute a command that takes values in those pieces of memory, computes their sum, and prints it on the screen. This concept of a named piece of memory, called **variable**, is a fundamental concept of programming. In this case, we need two variables for input numbers, and one variable for the result.

In C++ and many other programming languages a variable is defined not only by name, but also by **type** that determines a set of values the variable could take. In our example we need integer numbers that correspond to C++ keyword `int`. Any variable should be created before it is used, to create a variable `a` of type `int` use a line `int a;`. We also need to input data from a user, C++ provides a standard `cin` (console input) command for this purpose, that acts as an inverse of `cout`. The complete program is:

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    int b;
    cout << "Input two integer numbers: ";
    cin >> a;
    cin >> b;
    int sum;
    sum = a + b;
    cout << "Sum = ";
    cout << sum << endl;
    return 0;
}
```

Figure 1.3. Computing sum of two integer numbers

As in the previous program, the first two lines enable input/output to be able to use `cin` and `cout` later, `int main()` defines an entry point. Then we create two variables of integer type (`int` is C++ keyword for integer) `a`, `b`. We invite a user to input two numbers by printing the corresponding message using `cout`. Then we save the values that a user inputs into the pieces of memory that correspond to variables `a`, `b`, we will refer to that just as values of variables `a`, `b`. We create another variable for

the sum and compute it. C++ supports basic arithmetic operations +, –, *, / and some basic mathematical functions (square root, trigonometric, exponential and others). Note that the value of `sum` depends of the values of `a`, `b`, which in turn are depend on the numbers user inputs. That is a way to make programs general, and variables are the only way to do so. After the sum is computed we output the value to the screen. Note that we put the text `Sum =` in quotation marks which means the program will print the exact text (without quotation marks), and then we put the name of the variable without quotation marks which means the program will print the value of this variable. For example, for the input "5 3" the program will print "Sum = 8", and for input "10 –37" the program will print "Sum = –27".

C++ allows combining several similar commands to one command. In the previous example we can combine creation of variables `a`, `b`, input of `a`, `b`, creation and computing of `sum`, output of `sum`. This will not affect how the program works but will make it smaller in terms of number of lines of code. Generally it does not matter too much and do not affect speed or any other characteristics of a program, but it is recommended to combine commands that are naturally similar to each other. After these modifications our program will look as follows:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    cout << "Input two integer numbers: ";
    cin >> a >> b;
    int sum = a + b;
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Figure 1.4. A modified program computing sum of two integer numbers

Consider another problem: create a program that inputs salary size and tax rate and computes the tax size. Denote salary size as `s`, tax rate as `r` and use a decimal fraction for rate (so that, for instance, 15% rate correspond to `r = 0.15`), tax size as `t`. Clearly, due to `r` being measured as a fraction, `t = s * r`. Note that here salary size, tax rate and tax size are not necessarily integer, therefore we cannot use `int` type and have to use a type for real numbers. C++ has two data types for real numbers, `float` and `double`, for this purposes we could use either of them. Here is the source code:

```
#include <iostream>
using namespace std;
int main()
{
    double s, r;
    cout << "Input salary size and tax rate: "
    cin >> s >> r;
    double t = s * r;
    cout << "Tax size = " << t << endl;
    return 0;
}
```

Figure 1.5. Computing tax size

For example, for salary size 200 and tax rate 0.2 (20%), the tax size is 200 * 0.2 = 40 and for input “200 0.2” a user will see “Tax size = 40”.

In this section we introduced some very basic concepts of programming, such as program, source code, variable, and wrote some simple programs using the C++ programming language. We only explained the constructs of C++ on examples and tried to give a general impression of how programs look like. The following sections provide a much more methodological and detailed description of basic features and constructs of C++ accompanied by more interesting examples.

2. Basics of the C++ programming language

Source code of a program is a text written in a special language that computer can interpret – a programming language. All programming languages have strict rules that software developer must follow. Otherwise a code cannot be properly interpreted by a computer and, therefore, is incorrect. This section describes very basic rules of the C++ programming language.

2.1. Elementary constructions

C++ language alphabet is a set of symbols C++ supports, that is, can recognize. It consists of: characters (a–z, A–Z, _), numerals (0–9), special characters ({}. [], +, –, *, /, |, &, #, ~, etc.), whitespace and tab symbols, line feed character (usually denoted \n). Other symbols are forbidden. The only exception is comments: a special lines or pieces of the source code written for humans and ignored by a compiler. There are two ways to make a comment: everything between `/*` and `*/` is a comment, a line after `//` is a comment. In the examples of programs we use comments to give explanations and clarifications directly in the source code; we highlight comments with green color.

Elementary constructions of C++ are basic blocks that form a program. These include identifiers, keywords, operations, constants, and separators.

Identifier is a name of an object in a program (e.g. name of a variable). A name can consist of characters a–z, A–Z, numerals 0–9, underscore symbol _. C++ is a **case-sensitive** language: capital and small letters are different; thus “abc”, “Abc” and “ABC” are three different identifiers.

C++ has a set of **keywords** – special words reserved to express programming language functionality. It is forbidden to use them as identifiers. Among the C++ keywords are `main`, `int`, `double`, `if` ,`else`, `switch`, `case`, `for`, `while`, `do`, `new`, `delete`. In code examples we give keywords are given in blue color.

C++ supports some mathematical **operations** using common operation signs. Unary operations apply to one argument: unary minus (e.g. –x), increment ++ (increase by 1), decrement –– (decrease by 1). Binary operations +, –, *, / apply to two arguments. Semantics of operations depends on types of its arguments. For example, division operation / performs division of real numbers using fractions when applied to real numbers, but does integer division when applied to integer arguments.

C++ is capable to operate with several kinds of data: integer, real and complex numbers, characters, logical values (true / false). A **data type** is described by three characteristics:

- Internal representation in memory
- Set of possible values
- Operations and functions applicable to values of this type

Let us now introduce several widely used data types of C++. Names of data types are keywords.

Data type **`int`** is used to represent integer numbers. The range of values of this type depends on the computer and operating system. Usually a value of type `int` takes 4 bytes of memory and is bounded by range between roughly –2 billion to 2 billion. It supports usual arithmetic operations +, –, *, / (integer division), remainder operation %, bitwise operations that work with internal representation.

Data type **`char`** is used to represent symbols. A value of char type takes 1 byte of memory and stores a number between –128 and 127. There is a table that maps symbols and their numerical codes (so-called ASCII table). Thus, char represents symbols by storing their numerical codes.

There are two types to represent real numbers: **`float`** and **`double`**. `float` takes 4 bytes, `double` takes 8 bytes and has higher precision. The names are somewhat misleading and have a historical foundation.

There is a type **`bool`** to represent logical values that can only have two values: **`true`** and **`false`** (which are also keywords of C++). True and false correspond to integer values of 1 and 0, respectively. Logical values are also called Boolean which clarifies the name `bool`.

2.2. Program structure

In this tutorial we only consider basics of C++ and the simplest program structure. In this case a program has an **entry point**, from where the execution starts. An entry point is denoted by `int main()` which is a necessary element of any C++ program. The line `int main()` is preceded by enabling of additional functionality, such as input/output or mathematical functions. Throughout all the examples in this tutorial we use `cin/cout` for input/output, so we always start with lines

`#include <iostream>` and `using namespace std;` to enable those. This, all programs in this tutorial have the following structure:

```
#include <iostream>
using namespace std;
int main()
{
    // Main body. We write the code here
    // …

    return 0; /* This line and the following
closing curly brace end the program */
}
```

Figure 2.1. Structure of simple C++ programs

The main body of a program consists of **statements** which are elementary commands of C++. Examples of statements are input/output, variable definition and assignment. Statements are separated by semicolon ; (except of complex statements that we will learn in the next sections). We usually write each statement on a separate line to improve readability of the code, it is not required by C++ rules.

2.3. Variables

Variable is a fundamental concept of programming and the main way to handle data in program. A variable is the named piece of memory with a certain type. Variable is characterized by **value** and **name**. Variable name is an identifier used to access to the piece of memory that contains value of the variable. Variable type defines set and semantics of operations applicable to the variable.

Each variable must be **defined** before its first usage. Definition allocates a piece of memory for a variable, so that we can use it to store some data of a certain type. Variable definition has the following syntax:

```
<type> <name>[=<value>];
```

where

`<type>` is variable type, e.g. int, double, etc.,

`<name>` is variable name,

`<value>` is initial value (optional).

Initial value is optional, in case it is not present, a variable may be initialized with some default value, which depends on a particular computer. Thus, a programmer cannot rely on default initial values and should always explicitly initialize the variables either in definition or later, but before any other usage of the variable. Figure 2.2 presents some examples of variable definitions:

```
int a = 3; // Integer number a, initial value 3
float b; // Real number b, default initial value
int c = a + 2; /* Integer number c, initial value
is value of a + 2, in this example is 5 */
int d, e, f; /* It is possible to define several
variables of the same type in one statement */
```

Figure 2.2. Examples of variable definitions

Initialization is not the only way to set a value of a variable. The most common way to modify a value of a variable is to use assignment operation =. The general syntax is `<variable> = <expression>;` Expression may include numbers and variables that are already defined. Consider some examples:

```
int a = 1; // Initialization
int b; // No initialization
b = a + 5; // Assignment, now b is 6
a = a - 3; /* Assignment may contain the same
varialbe in right-hand side. The value of right-
hand side will be computed using old value of a,
and then will become a new value of a, in this
case -2. */
```

Figure 2.3. Examples of assignment operations

Note that is it possible to use the same variable in both left and right hands of assignment operation. In this case, the old value will be used to compute the right-hand part, and only then assigned to the variable.

Another way to modify a variable is to use increment or decrement operations ++, --, as shown on Figure 2.4. These operations are only applicable for integer types.

```
int a = 5; // Initialization
a++; // Now a is 6
a--; // Now a is 5
a--; // Now a is 4
```

Figure 2.4. Examples of assignment operations

Finally, we can save a value user inputs in a variable using `cin`. `cin` is a standard C++ mechanism to input values of any standard types. The program will wait until a user inputs a value (and presses “Enter” key) and then save it to a given variable.

```
int a;
double b;
cin >> a; // Input a
cin >> b; // Input b
```

Figure 2.5. Examples of cin for different types

Sometimes we want a variable to not be changeable. For example, such variables may represent physical or mathematical constants (e.g. speed of light or π) or some parameters supposed to be constant. In this case we may define a variable using `const` keyword before data type, thus allowing only initialization and no subsequent modification. Such variables are called **constant variables** or simply **constants**. Trying to assign a value to a constant (except of initialization) is a violation of C++ rules. Figure 2.6. illustrates usage of variables and constants.

```
int a;
const int b = 3;
a = b + 3; // Change a, don't change b - OK
b = a - 1; // Try to change a constant – forbidden
```

Figure 2.6. Examples of variables and constants

2.4. Sample programs

Let us now apply new knowledge to solve some sample tasks and demonstrate various aspects of C++.

Consider a problem: find the average of 3 real numbers input by the user. Let us denote the numbers as `a`, `b`, `c`. Then the average is `(a + b + c) / 3`. We need to create three variables for input numbers using type `float` or `double` (as they are real numbers), input the values using `cin`, then create another variable for average and compute it using the given formula, then output the average value. This is a set of actions we need to perform to solve the given task – the algorithm. For this task in may seem obvious, nevertheless it is strongly recommended to specify an algorithm as clearly as possible before starting to write to code. When the algorithm is ready we need to express the same actions in C++ source code. For this task it is done in a straightforward manner, the source code is given in Figure 2.7 and comments explain the actions we perform.

```
// The next two lines enable cin/cout
#include <iostream>
using namespace std;
// An entry point
```

```
int main()
{
    // Create three variables for input numbers
    float a, b, c;
    cout << "Input three real numbers: ";
    // Input numbers
    cin >> a >> b >> c;
    // Compute average
    float average = (a + b + c) / 3;
    // Output average
    cout << "Average = " << average << endl;
    // Finalize the program
    return 0;
}
```

Figure 2.7. Computing average of three real numbers

For example, imagine that a user inputs 1 1.5 2. Then value of variable `a` will be 1, the value of variable `b` will be 1.5 and the value of `c` will be 2. The expression `(a + b + c) / 3` will result in 1.5 which will be the value of `average` variable. A user will see output "Average = 1.5".

Let us now consider another problem. A bank provides money transfer service for a fee that is a percentage of the transfer amount. We need to create a program that inputs transfer sum and bank percentage and outputs the amount of transfer including bank fee. For example, if transfer amount is 200 and percentage is 7%, the amount of transfer including bank fee is $200 + 200 * 7\% = 200 + 14 = 214$. The general formula to compute the resulting amount is $result = amount + amount * percentage = amount + amount * percentage\,rate\,/\,100$. Now let us create a program that creates variables for amount and percentage, inputs values of these variables, computes resulting amount according to the given formula, and prints it to the screen. Amount, percentage and result are real numbers, this time let us use type `double`. The source code of this program is given in Figure 2.8.

```
#include <iostream>
using namespace std;
int main()
{
    double amount, percentage;
    cout << "Input transfer amount and percentage
rate: ";
    cin >> amount >> percentage;
    double result = amount + amount * percentage /
100;
```

```
        cout << "Result = " << result << endl;
        return 0;
    }
```

Figure 2.8. Computing amount of transfer including percentage-based fee

If we run this program and input values from the example 200 and 7 we will get the message “Result = 214”.

3. Conditional statements

As you probably noticed, all programs that we wrote till this moment followed the same execution path. That is, each time they executed exactly the same command for different data. There was no decision about which of two execution paths to choose, that is, which of multiple groups of statements to execute. In reality, this kind of decisions are needed even for simplest operations, such as finding the maximum of two or several numbers, or checking if the data user inputs is valid.

All programming languages support constructs for making decisions, called **conditional statements** (or, somewhat jargon, branch statements). Usually there are two types of conditionals: simple choice between two alternatives (**binary choice**) and choice between multiple alternatives (**multiple choice**). C++ supports three kinds of conditional statements: **if statement** for binary choice, **conditional operation "?"** as a simplified form of if statement convenient in specific cases, **switch statement** for multiple choice.

3.1. If statement

If statement is the basic and most widely used conditional statement that allows to choose between two alternatives. It evaluates the specified condition (to get either true or false) and depending on its value executes one of the two specified pieces of code called **branches**.

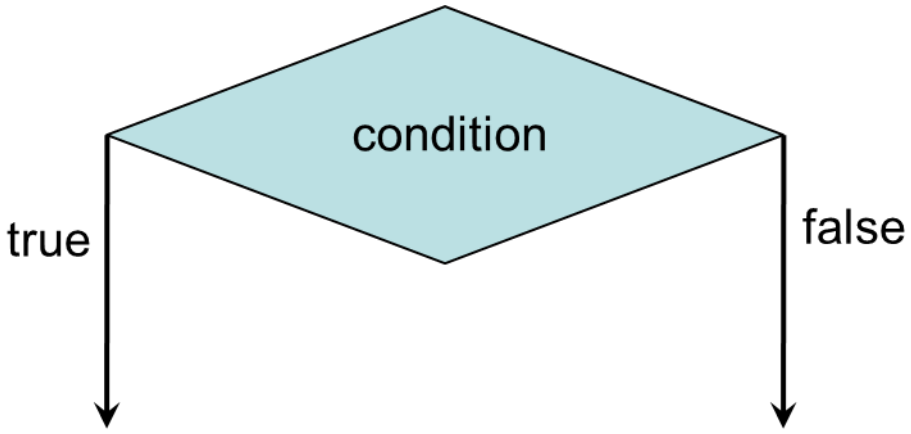

Figure 3.1. If statement scheme

The syntax of if statement is as follows.

```
if (condition)
{
    /* code executed if condition is true */
}
else
```

```
{
    /* code executed if condition is false */
}
```

Figure 3.2. If statement syntax

If a branch consists of only one statement, the curly braces could be omitted.

Let us demonstrate usage of if statement for the task of finding the maximum of two integer numbers. The algorithm is very simple: if the first number is greater than the second number, the maximum is the first number; otherwise the maximum is the second number. Here is the implementation of this algorithm:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, maximum;
    cin >> a >> b;
    if (a > b)
        maximum = a;
    else
        maximum = b;
    cout << "Max = " << maximum << endl;
    return 0;
}
```

Figure 3.3. Computing maximum of two numbers

A condition in if statement could be any logical expression, that is any expression that return a Boolean value of true or false. In particular, this could be a variable of type bool.

Let us give several examples on if statements. Consider the following problem: input an integer number, print whether it is even or odd. Number being even is equivalent to its remainder by modulus 2 being 0. So we need to check if the remainder is 0 or 1 and depending on that print the corresponding message. The implementation is as follows:

```
#include <iostream>
using namespace std;
int main()
{
    int a;
    cin >> a;
    if (a % 2 == 0)
        cout << "The number is even." << endl;
    else
        cout << "The number is odd." << endl;
```

```
    return 0;
}
```

Figure 3.4. Checking whether a given number is even or odd

Note that we use operation == to compare, and = used to assign. It is a common mistake to misuse assignment instead of comparison that results in wrong logic of a program. Most compilers give warnings if you try to assign a value in a condition of an if statement.

Let us now solve a classical problem of computing the roots of a quadratic equation

$$ax^2 + bx + c = 0,$$

where $a \neq 0$. It is well known that real solutions exist if the discriminant is non-negative:

$$D = b^2 - 4ac \geq 0$$

In this case the roots can be computed as:

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$$

Therefore, we need to input coefficients a, b, c, compute the discriminant D, check if D is non-negative and depending on that either compute the roots or print the message that there are no real roots. We will also check is a is zero and print the corresponding message. This program needs to compute a square root. C++ has a standard way to compute it: `sqrt(<expression>)` returns $\sqrt{< expression >}$. To be able to use sqrt it is required to add `#include "math.h"` in front of the program.

```
#include <iostream>
using namespace std;
#include "math.h"
int main()
{
    double a, b, c;
    cin >> a >> b >> c;
    if (a == 0)
    {
        cout << "a = 0, the equation is not quadratic" << endl;
        return 0;
    }

    double D = b * b - 4 * a * c;
    if (D >= 0)
    {
```

```
            double x1 = (-b + sqrt(D)) / (2 * a);
            double x2 = (-b - sqrt(D)) / (2 * a);
            cout << "x1 = " << x1 << endl;
            cout << "x2 = " << x2 << endl;
        }
        else
            cout << "There are no real solutions" <<
endl;
        return 0;
}
```

Figure 3.5. Solving a quadratic equation

Note that the first if statement does not have else branch. If the condition is true, the statements inside if-branch are executed, otherwise, no statements are executed and the program executes the next statement after if-branch. Thus, omitting else-branch is equivalent to an empty else-branch `else { }`.

Consider a problem of finding the maximum of three numbers. There are two principally different ways to solve it, we will illustrate both. First, one can note, that $\max\{a, b, c\} = \max\{\max\{a, b\}, c\}$. Thus, the problem of computing the maximum of three numbers reduces to solving the problem for two numbers twice. This gives an algorithm of two steps: compute $d = \max\{a, b\}$, the result is $\max\{d, c\}$. We can use the if statement similar to given in Figure 3.3 to compute maximum of two numbers. This results in the following program:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, c, d, maximum;
    cin >> a >> b >> c;
    // Compute d = max{a, b}
    if (a > b)
        d = a;
    else
        d = b;
    // Compute maximum = max{d, c}
    if (d > c)
        maximum = d;
    else
        maximum = c;
    cout << "Max = " << maximum << endl;
    return 0;
}
```

Figure 3.6. Finding maximum of three numbers via reduction to maximum of two numbers

The second way is to modify conditions in if statement to make comparisons of three numbers. Note that we cannot make any decision just based on comparison of two numbers, say `a` and `b`. Even if `a` is greater than `b`, from only this fact it is wrong to say that `a` is the maximum of all three numbers, because `c` can be greater than `a` and we do not check that. Instead, we need to use the following condition: if `a` is greater than `b` and `a` is greater than `c`, then `a` is the maximum. This is a so-called **complex condition**: it consists of two (in general, several) parts and we need both of them to be true. C++ provides logical operations that allow to write such conditions: **&&** stands for **logical AND**, || stands for **logical OR**, **!** stands for **logical NOT**. Logical AND is a binary operation and is true if and only if both of its parts are true. Logical OR is a binary operation and is true if and only if one or both of its parts is true. Logical NOT is a unary operation and is true if and only if its operand is false. Thus, the condition "`a` is greater than b and `a` is greater than `c`" is expressed via condition `if (a > b) && (a > c))`. Here is the full program:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, c, maximum;
    cin >> a >> b >> c;
    // Check if a is the maximum
    if ((a > b) && (a > c))
        maximum = a;
    else
        // Check is b is the maximum
        if ((b > a) && (b > c))
            maximum = b;
        // Otherwise c is the maximum
        else
            maximum = c;
    cout << "Max = " << maximum << endl;
    return 0;
}
```

Figure 3.7. Finding maximum of three numbers via complex conditions

3.2. Conditional operation

C++ supports a unique conditional operation "?" that simplifies conditional statements in some cases. In essence, it is a simple if statement written in one line. The syntax of "?" operation is: `<condition> ? <expression1> : <expression2>`. The condition is evaluated, if it is true the value of "?" operation

is the <expression1>, else the value of "?" operation is the <expression2>. Unlike if-else statement, <expression2> ('else-section') cannot be omitted.

We will illustrate the "?" operation on the example of finding maximum of two numbers.

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, maximum;
    cin >> a >> b;
    maximum = (a > b) ? a : b;
    cout << "Max = " << maximum << endl;
    return 0;
}
```

Figure 3.8. Finding maximum of two numbers with "?" operation

We compare a and b, and if a is greater the "?" operation returns a, otherwise it returns b. The return value is then assigned to the maximum variable. Thus, this program is completely equivalent to one in Figure 3.3.

3.3. Switch statement

If-else statement and "?" operations support only binary choice. In principle this is enough: every condition can be expressed as a sequence of (probably complex) binary conditions. So, theoretically, if-else statement suffices to implement any condition. But sometimes it is more natural to use multiple-choice condition. C++ provides switch statement as an instrument.

The syntax of switch statement is:

```
switch (expression)
{
    case constExpr1:
    {
        /* The first sequence of operations */
        [break;] // explicit break
    }
    …
    case constExprN:
    {
        /* The N-th sequence of operations */
        [break;]
    [default:
    {
```

```
        /* Statements to execute if none of cases
occur */
    }]
}
```

Figure 3.9. The syntax of switch statement

The expression is evaluated. It is required to have a discrete set of values, that is, be of integer or character type. The value of the expression is matched against all values given in `case` blocks. If it matches, the sequence of operations inside that case block is executed. Then the statements of all following case blocks are executed (independent of value comparison), until the break statement is encountered. In most cases we need only one of case blocks, so break is used in every case block. If none of values of case blocks are matched, the statements in `default` section are executed. The default section is not required, its absence is equivalent to an empty default section `default: { }`.

Let us use switch statement to create a simple calculator program that operates on two real numbers and supports four arithmetic operations: +, –, *, /. We will input an arithmetic expression in typical form: <first operand> <operation> <second operand>. The operands are real numbers, and operation is a symbol, which can be stored in C++ type `char`. We use ‘’ to represent constants of type char: ‘+’ for symbol +, etc.

```
#include <iostream>
using namespace std;
int main()
{
    double a, b, result;
    char op;
    cin >> a >> op >> b;
    switch (op)
    {
        case '+':
        {
            result = a + b;
            break;
        }
        case '-':
        {
            result = a - b;
            break;
        }
        case '*':
        {
            result = a * b;
            break;
```

```
            }
            case '/':
            {
                result = a / b;
                break;
            }
            default:
            {
                cout << "The operation is not
supported";
            }
        }
        cout << "Result = " << result << endl;
        return 0;
}
```

Figure 3.10. A simple calculator program using switch statement

Note that for this kind of comparison all four options are uniform, which is why it is natural to use switch statement. Nevertheless, we can implement the same logic using multiple if statements:

```
#include <iostream>
using namespace std;
int main()
{
    double a, b, result;
    char op;
    cin >> a >> op >> b;
    if (op == '+')
        result = a + b;
    else
    if (op == '-')
        result = a - b;
    else
    if (op == '*')
        result = a * b;
    else
    if (op == '/')
        result = a / b;
    else
        cout << "The operation is not supported";
    cout << "Result = " << result << endl;
    return 0;
}
```

Figure 3.11. A simple calculator program using if statements

4. Loop statements

Computers are exceptionally good at handling lots of data. It is usually done by applying the same operations to several pieces of data. In terms of a programming language it means that there exist a way to repeat same actions. Loops are a construct of programming language to allow that. In combination with sequential execution of commands and conditional statements, loop statements form the third basic construct of a programming language. Theoretically any algorithm can be expressed as a combination of those three fundamental constructs.

Generally any loop looks like:

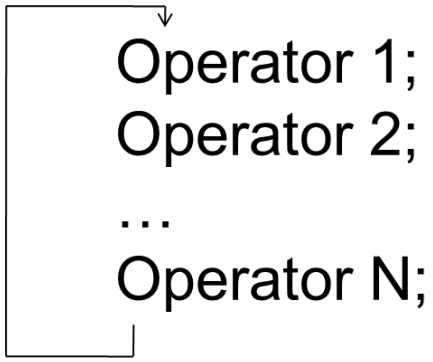

Figure 4.1. The general structure of a loop

The operations inside the loop are called **loop body**. One repetition (execution) of a loop body is called **loop iteration**, sometimes it is also called step. Of course, no loop should repeat infinitely, so there must be a condition on where to stop repetitions. This condition is usually checked either before an iteration starts or after an iteration ends. Based on this subdivision, loops are classified into **pre-condition** and **post-condition loops** that are subject of the following subsections.

4.1. Pre-condition loops

C++ supports two kinds of pre-condition loops: **while** and **for** loops. We will start with the while loop. The syntax of the while loop is:

```
while (condition)
{
    <loop body>
}
```

Figure 4.2. The syntax of while loop

The condition here is of the same kind as in if statement. Each time before executing the loop body, condition is checked. If it is true, the loop body is executed.

Otherwise the loop is over and the next statement after the loop is executed. It is important that the condition is only checked when iteration begins, so if it becomes false in the middle of the iteration, the iteration will not stop immediately, and will only stop in the beginning of the next iteration.

To illustrate usage of while loop consider the following task: print a table of values of function $y(x) = x^2 + 1$ on a given interval $[a, b]$ with a given $step$. That is, for all points in $[a, b]$ with a given $step$ print the value of x and the corresponding value of y. For example, for $a = 1$, $b = 1.9$ and $step = 0.2$ the corresponding table is:

x	y
1	2
1.2	2.44
1.4	2.96
1.6	3.56
1.8	4.24

We will start from point $x = a$ and then increase x in a loop by step using variable `point` as our current x value. Inside a loop body we compute the value of the function in the current point and print a pair of x and y to the screen. The program is given in Figure 4.3.

```
#include <iostream>
using namespace std;
int main()
{
    float a, b, step;
    cout << "Input a ";
    cin >> a;
    cout << "Input b ";
    cin >> b;
    cout << "Input step ";
    cin >> step;
    cout << "x y" << endl;
    float point = a;
    while (point <= b)
    {
        cout << point << " " <<
            point * point + 1 << endl;
        point = point + step;
    }
    return 0;
}
```

Figure 4.3. Building function table using while loop

Consider another problem: input an integer n, compute the sum of all natural numbers from 1 to n: $s = \sum_{i=1}^{n} i$. Note that this problem is already formulated in terms of loops: the mathematical sum and product operations are natural loops themselves: they are formulated in terms of operations with current value and specify the initial value and stop condition. The program is given on Figure 4.4.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int s = 0;
    int i = 1;
    while (i <= n)
    {
        s += i;
        i++;
    }
    cout << "s = " << s << endl;
    return 0;
}
```

Figure 4.4. The summation program using while loop

Note that (almost) every `while` loop has three parts: initialization of a loop variable that stands before the loop, condition involving the loop variable and modification of the loop variable in the loop body.

Another pre-condition loop in C++, **`for` loop**, combines the three parts together. The syntax of the `for` loop is:

```
for (<initialization>; <condition>;
 <modification>)
{
    // loop body
}
```

Figure 4.5. The syntax of for loop

The initialization section is used to set initial values of loop variables (that can also be done before the loop, but usually it is more convenient to do in this section). The condition plays the same role as for while loop and is checked in the beginning of iteration. The modification part is executed in the end of iteration. To illustrate for loop usage let us rewrite the program in Figure 4.3 using for loop instead of while

```
#include <iostream>
using namespace std;
int main()
{
    float a, b, step;
    cout << "Input a ";
    cin >> a;
    cout << "Input b ";
    cin >> b;
    cout << "Input step ";
    cin >> step;
    cout << "x y" << endl;
    for (float point = a; point <= b; point += step)
    {
        cout << point << " " <<
            point * point + 1 << endl;
    }
    return 0;
}
```

Figure 4.6. The function table program using for loop

Note that we mostly just rearranged the code of the while loop so that all loop-related parts are now inside the first line of the for loop.

For loops are particularly convenient when the values of a loop variable a discrete, as it appears in the program in Figure 4.4. Now let us show an equivalent program with for loop instead of while:

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

int main()
{
    int n;
    cin >> n;
    int s = 0;
    for (int i = 1; i <= n; i++)
    {
        s += i;
    }
    cout << "s = " << s << endl;
    return 0;
}
```

Figure 4.7. The summation program using for loop

Consider a little bit more complicated problem: check if a given integer number is prime or not. An integer number is prime if it is not a multiple of any integer number except of one and itself. Therefore, to check is a given number n is prime we should try all integer numbers in $[2, n-1]$ and check if n is a multiple of any of those numbers. For loop is the most convenient tool to use in this case. We will also use a Boolean variable which signifies whether we have found that the number is not prime or not.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    bool isPrime = true;
    for (int i = 2; i < n; i++)
        /* Check if n is a multiple of i, in this
case n is not prime */
        if (n % i == 0)
            isPrime = false;
    if (isPrime)
        cout << "The number is prime" << endl;
    else
        cout << "The number is not prime" << endl;
    return 0;
}
```

Figure 4.8. Checking whether a given number is prime using for loop

Note that it would be wrong to write else branch directly in the if statement that is inside the loop: if n is not multiple of a particular number (say, 2) it does not necessarily mean it is not multiple of other numbers. That is why we cannot make an immediate decision whether the number is prime and have to go through all iterations of the loop and use a Boolean variable. If all the checks in the for loop will fail, the number is indeed prime and the value of `isPrime` variable will remain true. If the number is not prime, on some iteration (-s) of the loop the if statement condition will be true, the value of `isPrime` variable will become false and after the loop is over a user will see the correct message.

4.2. Post-condition loops

Post-condition loops check stop condition in the end of the iteration. The major difference is that that a post-condition loop always performs at least one iteration,

while a precondition loop may do no iteration if the check before the first iteration fails.

C++ has one post-condition loop: **do-while** with the following syntax:

```
do
{
    // loop body
}
while (condition);
```

Figure 4.8. The syntax of do-while loop

Except condition check time it is essentially the same as while loop.

All three loop constructs of C++ are equivalent in the sense that any program with loop statement can be written with either while, for, or do-while loops. Let us illustrate usage of do-while loop for programs in Figures 4.3 and 4.4.

```
#include <iostream>
using namespace std;
int main()
{
    float a, b, step;
    cout << "Input a ";
    cin >> a;
    cout << "Input b ";
    cin >> b;
    cout << "Input step ";
    cin >> step;
    cout << "x y" << endl;
    float point = a;
    do
    {
        cout << point << " " <<
            point * point + 1 << endl;
        point += step;
    } while (point <= b)
    return 0;
}
```

Figure 4.9. Building function table using do-while loop

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int s = 0;
```

```
    int i = 1;
    do
    {
        s += i;
        i++;
    } while (i <= n)
    cout << "s = " << s << endl;
    return 0;
}
```

Figure 4.10. The summation program using do-while loop

5. Arrays

With introduction of loop statements we are now able to create programs that operate on large sets of data. But we also need a way to store large sets of data in memory. **Arrays** are a way of storing big amounts of uniform data; they serve as the simplest data structure of that kind and a fundamental concept in programming.

5.1. Static arrays

Imagine we have a group of quantities that have the same data type and need to perform uniform operations on each piece of data we have. In this case we can give the name for this group and have access to each element using index. Array is the named sequence of elements that have the same data type.

Static array definition has the following syntax:

```
<mem> <type> <name>[<number>];
```

where `<mem>` is memory class (auto, etc.), `<type>` – data type of array elements, `<name>` is name of array, number is number of elements in array, which must be positive integer constant. The array is called static because memory is allocated during the stage of compilation (static allocation). That is why the array size should be known at compile time and required to be constant.

Array can be optionally initialized in the definition:

```
<mem> <type> <name>[<number>]={<values>};
```

If initialization is present, array size can be omitted, in this case it will be computed from initialization. Here are some examples of static array definitions:

```
int a[5] = {2, -1, 3, 4, 9};
float b[4];
int c[] = {-8, 2}; // initializer is essential
const int n = 10; double d[n];
```

Figure 5.1. Examples of array definitions

`a[i]` is element of the array with index `i`. Indexes always start from 0. Thus, a set of valid indexes for array of size n is $\{0, 1, 2, \ldots, n-2, n-1\}$. Obviously, as arrays are collections of many elements, they are mostly processed in loops with loop variable being an index in the array. `For` loops are particularly convenient for most operations on arrays. The typical input and output of arrays of size n looks like:

```
const int n = 20;
int a[n];
for (int i = 0; i < n; i++)
  cin >> a[i];
for (int i = 0; i < n; i++)
  cout << a[i];
```

Figure 5.2. Typical array input and output

Let us now consider several typical operations with arrays. First, let us input an integer array of size 10 and find the sum of the elements. The corresponding program is given in Figure 5.3. To compute the sum we create a variable `sum` that accumulates a sum. After iteration is over we have processed part of the array and accumulated the corresponding partial sum into the `sum` variable. Thus, after the first iteration is over `sum` holds the first element, after the second iteration it holds value of a[0] + a[1], then a[0] + a[1] + a[2], etc. When the loop is over, all elements of the array are processed, and the `sum` variable holds the value of the total sum of elements that we need.

```
#include <iostream>
using namespace std;
int main()
{
    const int n = 10;
    int a[n];
    // Input array elements
    for (int i = 0; i < n; i++)
        cin >> a[i];
    // Compute the sum
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    cout << "Sum = " << sum << endl;
    return 0;
}
```

Figure 5.3. Computing sum of array elements

Note that although we know array size is 10 we still create a constant `n = 10` and use `n` everywhere later. It is a very common approach that aims to make the code more general: in case we had to change the program to work for array of size 20 we would have to change all loop conditions from 10 to 20, while due to our approach it suffices to charge `n` to 20. This approach is even more valuable for larger programs as is helps to localize the changes. It is an example of a good coding style. We will consistently use this approach throughout this tutorial.

The next typical problem we will consider is finding the minimum value of an array of size 7. The code to create and input an array is standard and exactly the same (except different value of `n`) as in the previous program. Let us describe the algorithm to find the minimum value of an array. It is very similar to the algorithm used to find sum. Again, we create a variable to store result of processing part of the array – in this case the value of `minimum` variable is the minimum element of the processed part of the array. Initially it is just the first element `a[0]`, after the first iteration of the loop is over it is $\min\{a[0], a[1]\}$, after the second iteration of the loop it is min{a[0], a[1], a[2]}, etc. When the loop is over, the value of `minimum` variable is the minimum element of the whole array. The code of this program is given in Figure 5.4. Another important note is that we cannot initialize `minimum` variable with, say, 0 or 1000, or any other particular value (except of maximum possible value of type double). In that case the program will not work for arrays that have all elements greater than the initial value of `minimum` variable: all checks in the loop would fail and the program would output the initial minimum value that is not present in the array. On the contrary, with initial value being one of the array elements, even if all checks in the loop fail, it is fine as in this case the actual minimum is a[0] and after the loop the `minimum` variable has the correct value.

```
#include <iostream>
using namespace std;
int main()
{
    const int n = 7;
    double a[n];
    // Input array elements
    for (int i = 0; i < n; i++)
        cin >> a[i];
    // Compute the minimum
    double minimum = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] < minimum)
            minimum = a[i];
    cout << "Minimum = " << minimum << endl;
    return 0;
}
```

Figure 5.4. Computing minimum of array elements

Both of sum and minimum problems are **reduction problems**. And they are solved in similar way: we create a variable that is the partial result – result of processing a part of the array. During an iteration of a loop we add a new element to the processed part and update the reduction variable correspondingly.

Another typical class of operations of arrays is **filter operations**: we are given an array and need to create another array that has all elements of the original array that satisfy some condition. There is a typical pattern of code to solve problems of this kind. Let us consider two problems of this kind and illustrate the pattern.

First, we will solve the following problem: input integer array of size 6, make another array that has only negative elements of the original array. The typical approach is to create the second array of the same size (at most it will have the same number of elements as the original array), and then control how many elements are actually in this array using a variable. The code is given in Figure 5.5. The two statements inside if branch could be replaced with one statement `result[resultSize++] = a[i];`.

```
#include <iostream>
using namespace std;
int main()
{
    const int n = 6;
    int a[n], result[n];
    // Input array elements
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int resultSize = 0; /* actual size of the
resulting array */
    for (int i = 0; i < n; i++)
        if (a[i] < 0)
        {
            result[resultSize] = a[i];
            resultSize++;
        }
    // Print the resulting array
    for (int i = 0; i < resultSize; i++)
        cout << result[i] << endl;
    return 0;
}
```

Figure 5.5. Filtering negative elements of array

Now we will solve an important problem that arises as part of many more complicated algorithms. It is to create an array without repetitions: the resulting array will have exactly the same elements as the original one, but each element is only present once. For input array {1, 3, 2, 3, 1, 2, –5} it will return array {1, 3, 2, –5}. The algorithm to solve this task is a bit more complicated compared to other tasks in this section. As in previous task, we will create the resulting array and keep track of

its current size using `resultSize` variable. We will go through the original array in a loop and check, whether the current element `a[i]` is present in the resulting array. If it is present, we don't need to include this element as it is already present. Otherwise, we need to add this element to the resulting array. The check, whether an element is present in the resulting array, we need to check every element of the resulting array in a loop. Thus, this program will have two nested loops: one through the original array and internal loop through the resulting array. This program is given in Figure 5.6.

```
#include <iostream>
using namespace std;
int main()
{
    const int n = 20;
    int a[n], result[n];
    // Input array elements
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int resultSize = 0;
    // Go through the original array
    for (int i = 0; i < n; i++)
    {
        /* Check if a[i] is present in the result
array */
        bool isPresent = false;
        for (int j = 0; j < resultSize; j++)
            if (result[j] == a[i])
                isPresent = true;
        /* If a[i] is not present, add it to
result */
        if (!isPresent)
        {
            result[resultSize] = a[i];
            resultSize++;
        }
    }
    // Print the result array
    for (int j = 0; j < resultSize; j++)
        cout << result[j] << endl;
    return 0;
}
```

Figure 5.6. Building array without repetitions

5.2. Dynamic arrays

Static arrays require that size of an array is known at compile time: thus, it should be a variable with the `const` qualifier, or just a number. The size must be known because a compiler needs to reserve memory for an array. And this is the only way of working with static memory: all sizes should be predetermined in compile time.

However, there is another segment of memory, called **dynamic memory**, or **heap**, that allows allocation at run-time. Arrays that reside in dynamic memory are called **dynamic arrays**. Unlike static memory, dynamic memory is manually controlled: we need to ask for pieces of dynamic memory using `new` keyword of C++. Here is a simple code to allocate memory for dynamic array at run-time, note that `n` is now a variable, not constant:

```
int n;
cin >> n;
int * a = new int[n];
```

Figure 5.7. Allocation of dynamic array

In this piece of code user inputs the size of an array and we manually allocate a piece of dynamic memory to fit that many elements. Because we allocate memory manually, the compiler is unable to keep track of this memory, and we have to deallocate (free) this memory after array is no longer used via `delete` keyword. The typical pattern of dynamic array usage looks like that:

```
int n;
// Determine the size of an array
int * a = new int[n]; // Allocate memory
// Use array for computations
delete [] a; // Free memory
```

Figure 5.8. A typical pattern of dynamic array usage

After execution of delete statement the array cannot be used any more – it could result in crashing a program. If we allocate dynamic memory and do not free it, it does not result in crash. But the computer will treat this piece of memory as reserved and will not allow allocating it again. This error is called **memory leak** and is one of the most common errors of C++ developers. If we allocate a large amount of dynamic memory and do not free it, the computer can run out of memory and memory allocations will fail.

The only difference between static and dynamic arrays is memory allocation and deallocation. Other operations such as element access `a[i]` does not depend on array

type. Therefore, all algorithms that we learned for static arrays can be easily implemented for dynamic arrays as well.

To demonstrate that let us rewrite a program that computes sum of array elements from Figure 5.3. for a dynamic array. Now we have additional advantage that we don't have to fix array size and can input it from a user, thus supporting arbitrary size arrays. The program is given in Figure 5.9.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cin >> n;
    int * a = new int[n];
    // The core of the program do not change
    for (int i = 0; i < n; i++)
        cin >> a[i];
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    cout << "Sum = " << sum << endl;
    delete [] a;
    return 0;
}
```

Figure 5.9. Computing sum of array elements for a dynamic array

5.3. Two-dimensional arrays

Arrays are a natural way to represent a collection of uniform data, or mathematical vectors. However, some data is naturally two-, three-, or higher dimensional. An obvious example is table, or matrix in algebra which is a two-dimensional collection of data. We will only consider the most common two-dimensional case.

Unlike one-dimensional arrays, one index is not enough to specify an element of table or matrix – we need row and column indexes. For two-dimensional array `a` the element on row `i` and column `j` is given by `a[i][j]`. Thus, we can also treat a two-dimensional array as a one-dimensional array, each element of which is again a one-dimensional array – in matrix case, array of rows, where each row is an array of element. And we can interpret the `a[i][j]` as if we first apply `a[i]` getting a one-dimensional array (row) and then taking `j`-th element of that

array. To create a static two-dimensional array we need to specify two sizes: size for the first dimension and for the second dimension. The total number of elements if their product (like the total number of elements of a matrix is a product of number of rows and columns). The code in Figure 5.10 creates a two-dimensional array of size `n = 10` by `m = 20` and inputs its values from the user. Note that while accessing elements of this array the first index must be between 0 and `n - 1`, and the second index between 0 and `m - 1`. That justifies stop conditions in the `for` loops.

```
const int n = 10, m = 20;
int a[n][m];
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        cin >> a[i][j];
```

Figure 5.10. A 10x20 static array

Let us now write a program that compute the sum of two matrices of the same size.

```
#include <iostream>
using namespace std;
int main()
{
    const int n = 5, m = 6;
    int a[n][m], b[n][m], result[n][m];
    // Input matrices
    cout << "Input the first matrix: " << endl;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    cout << "Input the second matrix: " << endl;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> b[i][j];
    // Compute sum
    cout << "Input the first matrix: " << endl;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            result[i][j] = a[i][j] + b[i][j];
    // Output resulting matrix
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
            cout << result[i][j] << " ";
```

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

Figure 5.11. Computing sum of two matrices.

Allocation and deallocation of dynamic two-dimensional array are a little trickier compared to one-dimensional dynamic arrays, as shown on Figure 5.12. It illustrates on of general rules of programming: deallocation of memory should happen in exactly reverse order of allocation.

```
int n, m;
cin >> n >> m;
int ** a = new int*[n];
for (int i = 0; i < n; i++)
    a[i] = new int[m];
// Use a the same way as static matrix: a[i][j]
for (int i = 0; i < n; i++)
    delete [] a[i];
delete [] a;
```

Figure 5.12. Allocation and deallocation of a two-dimensional dynamic array

6. Home exercises

Conditional statements

1. During the Christmas sale a shop reduces all prices. For all products with original price of 300$ or higher the new price is 0.7 of the original price. For all other products the new price is 0.8 of the original price. Write a program that inputs the original price of a product (use type float or double) and outputs the new price.
Examples:

1) Input: 500
   Output: 350
2) Input: 100
   Output: 80.

2. Create a program to convert lengths given in several units to meters (e.g. from inches to meters). On input the original unit of length is given in 1-letter code:

- i is inch, 1 inch = 0.0254 meters;
- f is foot, 1 foot = 0.3048 meters;
- v is versta, 1 versta = 1066.8 meters.

Input the length (use type float or double) and unit of length (use type char) and output the corresponding length in meters. Use switch statement.
Examples:

1) Input: 10 f
   Output: 3.048
2) Input: 2 v
   Output: 2133.6

3. Students take four exams, for each exam there are three possible marks: 3, 4, 5. University pays scholarship to every student. The size of the scholarship is determined by the following rules:

a) if at least one of the marks is 3, scholarship size is 100;
b) if all marks are greater than 3 and average mark is less or equal than 4.25 scholarship size is 300.
c) if all marks are greater than 3 and average mark is greater than 4.25 scholarship size is 500.

Write a program that inputs student's marks for four exams and outputs the scholarship size computed according to the rules.
Examples:

1) Input: 4 4 5 5
   Output: 500
   Explanation: all marks are greater than 3, average mark is 4.5 > 4.25 so according to c) scholarship size is 500.
2) Input: 5 3 5 5
   Output: 100
   Explanation: one of the marks is 3 so according to a) scholarship size is 100.

Loop statements

1. Input an integer number *n*. Print the count of integers in closed interval [1, *n*] that are multiples of 4.

Examples:

1) Input: 15
   Output: 3
   Explanation: there are 3 numbers that are multiples of 4: 4, 8, 12.
2) Input: 24
   Output: 6
   Explanation: there are 6 numbers that are multiples of 4: 4 8 12 16 20 24.

2. Input real numbers *a*, *b*. Print values of function *f(x) = x - 2* on closed interval [*a*, *b*] with step = 0.1.

Example:

Input: 1.4 2.1
Output: -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0 0.1

3. Input an integer *n* and then *n* integer numbers. Print the maximum of those *n* integers. Hint: you don't need to store all numbers in memory, just input one number per loop iteration and immediately process the number.

Examples:

1) Input: 5
   1 5 -3 6 0
   Output: 6
2) Input: 4
   -7 8 1 4
   Output: 8

4. Input an integer number n. Print all prime divisors of n (i.e. all prime numbers in [2, n] which n is a multiple of).

Examples:

1) Input: 12
   Output: 2 3
   Explanation: 2, 3 are divisors of 12 and are prime; other divisors of 12 are 4, 6, 12, all of them are not prime.
2) Input: 84
   Output: 2 3 7
   Explanation: 2, 3, 7 are divisors of 84 and are prime; other divisors of 84 are not prime.

Arrays

1. Input an array of 5 integers. Find and print the maximum of array elements and index of this element.

2. Input an array of 7 elements of type double. Compute count of positive numbers in this array.

3. Input an array of 4 integers and another integer key. Find the key in the array: output the corresponding index in the array or print the message that key is not present in the array. If the key is present several times, find its first encounter.

4. Initialize an array of 10 integers with values 1, 2, 3, …, 10 (use initialization at definition). Then replace all the elements that are < 4 with 0 and find the sum of elements of the array.

5. Input an array of 5 integers. Copy all positive elements to another array and output the resulting array.

7. Examination questions

Question № 1

1. Memory organization. Cell structure. Variables.

2. Annie goes on vacation. She got some money, and now decides whether the hotel with 10$ per day payment is affordable for her. Ask a user to input amount of money and number of days of vacation. Help Annie to decide. (Ex. If Annie has 50 $ and vacation is 3 days long, the hotel is affordable for her).

Question № 2

1. Data types.

2. Implement a program to compute roots of a square equation in form $\frac{a}{b}x^2 + cx + (f - d) = g$. Suppose that coefficients a, b, c, d, f, g are real numbers and are input by a user. Control situation of division by zero in case $b = 0$.

Question № 3

1. Logical operations and logical expressions.

2. Implement a program to compute roots of a square equation in form $ax^2 + \left(\frac{c}{d} + b\right)x + f = g$. Suppose that coefficients a, b, c, d, f, g are real numbers and are input by a user. Control situation of division by zero in case $d = 0$.

Question № 4

1. Selective structure if-else. Nested selective structures.

2. Each student reports the time (hour and minute) he/she came to school. Number of students is input by a user. Find the count of students who was late, classes start at 8:00 (8 A.M.).

Question № 5

1. Multiple selection: statements switch and break.

2. Pavel has written several books. The publishing company pays 1.5$ per page. Write a program that inputs number of books, number of pages for each book, and computes overall payment for all Pavel's books.

Question № 6

1. Loop structure. Post-condition loop.

2. Write a program to analyze the information about research assistants. Each assistant has a personal ID (integer). For each of 5 assistants, input an ID and a number of publications. Print an ID of the assistant who has the maximum number of publications.

Question № 7

1. Loop structure. Precondition loop.

2. Antonio sorts apples. Apples greater than 4 cm. in diameter are large, others are small. Antonio sorted 20 apples. For each apple input diameter. Print number of small apples.

Question № 8

1. Loops with known number of iterations.

2. Antonio sorts apples. Apples greater than 6 cm. in diameter are large, others are small. Antonio sorted 10 apples. For each apple input diameter. Print number of large apples.

Question № 9

1. Loops with unknown number of iterations.

2. Input N – number of students in the group. Then input marks that students got at the exam (possible marks are 2, 3, 4, 5). Find the best mark at the exam.

Question № 10

1. Selective structure if-then. Nested selective structures.

2. Input N – number of students in the group. Then input marks that students got at the exam (possible marks are 2, 3, 4, 5). Find the worst mark at the exam.

8. References

1. Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. Prentice Hall, 1988.
2. Steve Oualline. Practical C Programming. O'Reilly Media, 1997.
3. Stephen G. Kochan. Programming in C. Sams, 2004.
4. Herbert Schildt. C++: A Beginner's Guide. McGraw-Hill Osborne Media, 2003.
5. Herbert Schildt. C++: The Complete Reference. McGraw-Hill Osborne Media, 2002.

9. Table of contents