ПЕРМСКИЙ
ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ

И. В. Ильин

FUNDAMENTALS AND METHODOLOGY OF PROGRAMMING

PROCEDURALLY ORIENTED PROGRAMMING IN C++

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

И. В. Ильин

FUNDAMENTALS AND METHODOLOGY OF PROGRAMMING

PROCEDURALLY ORIENTED PROGRAMMING IN C++

*Допущено методическим советом
Пермского государственного национального
исследовательского университета в качестве
учебного пособия для студентов, обучающихся
по направлению подготовки бакалавров
«Бизнес-информатика»*

Пермь 2024

УДК 004.43(075.8)
ББК 32.973.22я73
И46

Ильин И. В.

И46 Fundamentals and methodology of programming. Procedurally oriented programming in C++ [Электронный ресурс] : учебное пособие / И. В. Ильин ; Пермский государственный национальный исследовательский университет. – Электронные данные. – Пермь, 2024. – 1,97 Мб ; 201 с. – Режим доступа: http://www.psu.ru/files/docs/science/books/uchebnie-posobiya/Ilyin-Fundamentals-and-methodology-of-programming.pdf. – Заглавие с экрана.

ISBN 978-5-7944-4087-4

В учебном пособии рассматривается теоретический материал о технологиях и методах программирования, базовых структурах алгоритмов, функциях, статических и динамических структурах данных. Представлены типовые алгоритмы их обработки. Приемущественно представлен процедурный подход к программированию на языке C++. Отдельным разделом обозначены индивидуальные задачи для самостоятельного решения. Приведенные фрагменты программного кода были реализованы с использованием последних версий доступных компиляторов C++.

Учебный материал излагается последовательно и сопровождается наглядными примерами и практическими задачами с детальным разбором их решений. К каждому параграфу прилагается список контрольных вопросов.

Учебное пособие на английском языке предназначено для студентов, обучающихся по направлению подготовки бакалавров «Business Informatics», program «Information systems and Big data».

УДК 004.43(075.8)
ББК 32.973.22я73

Издается по решению ученого совета экономического факультета Пермского государственного национального исследовательского университета

Рецензенты: кафедра экономического анализа и статистики Пермского института (филиала) Российского экономического университета имени Г. В. Плеханова (зав. кафедрой – канд. эконом. наук, доцент ***О. И. Агеева***);

доцент кафедры прикладной информатики, информационных систем и технологий Пермского государственного гуманитарно-педагогического университета, канд. физ.-мат. наук ***А. Ф. Кузаев***

ISBN 978-5-7944-4087-4

CONTENTS

INTRODUCTION

Trends in the development of modern languages and programming technologies require the future IT specialist to master a wide range of theoretical knowledge and practical skills in this area. The study of programming, as a rule, begins in the course of computer science and ICT in high school. Students are introduced to basic algorithms and data structures. Further, the study of algorithmization and programming fundamentals continues in the computer science course at the university. Along with this, the curriculum includes many individual disciplines in the "programming" category. A novice developer has difficulty studying complex technical literature, so this textbook is intended primarily for those who are just starting to dive into programming (junior-year IT students at universities). The manual provides theoretical information and examples of solving typical problems, which will later be used in practice.

Despite the widespread use of RAD rapid application development frameworks (e.g., Visual Studio, Delphi, Xamarin, etc.) and higher-level programming languages (e.g., C #, Python, 1Cetc.), their choice as tools for teaching programming seems to be in the author's opinion, inappropriate. Firstly, in modern RAD environments and frameworks, the application interface is formed from a set of ready-made components (modifying their structure requires serious knowledge and programming skills). Secondly, many standard algorithms (for example, search, sorting, etc.) are already implemented in the form of ready-made functions (for novice programmers, it is important to master these algorithms, and not the ability to call a ready-made function by its name).

The classic C/C++ language combination was chosen as the programming language, which is actively used in the field of system and application programming, development of algorithms for hardware devices, software for high-load systems and industrial facilities.

The textbook consists of four sections in which the educational material is presented according to the principle from simple to complex. The manual mainly discusses the procedural approach to programming, which is typical for learning the basics of programming. Immersion in object-oriented and other approaches, as a rule, occurs in senior years of universities.

The first section provides information about programming technologies and basic algorithm structures. The main content lines of the section include: linear algorithms, branching, loops, subroutines, recursion.

The second section describes static data structures and typical tasks for processing them. Content lines of the section: processing various data structures, working with external files.

The third section provides information about dynamic data structures. Content lines of the section: program models of lists, stack, queue, binary trees and container classes of the standard library.

The fourth section is intended for independent work of students and contains practical work.

A number of paragraphs within the sections are provided with tasks for students' independent work. Program codes and standard solutions to problems will help students master practical programming techniques.

The study guide also contains a bibliography.

The material of the textbook complies with the Federal State Educational Standard for Higher Education of the third generation for IT areas, such as «Business Informatics», program «Information systems and Big data».

SECTION 1. PROGRAMMING LANGUAGES AND TECHNOLOGIES. BASIC STRUCTURES OF ALGORITHMS

TOPIC 1. BASIC CONCEPTS OF ALGORITHMIZATION AND PROGRAMMING

Any person in his everyday and professional activities is faced with the need to solve some tasks, the implementation of which usually requires several sequential actions (for example, getting dressed for the street, cooking porridge, etc.). Such a sequence of steps is called an algorithm, and each individual action is its step.

An algorithm is a description of the order of actions (instructions) that a performer (some abstract or real system that knows a system of commands) must perform to achieve the result of solving a problem in a finite time.

The algorithm must have a set of the following *properties* :

- discreteness – the algorithm consists of individual commands executed in a finite time;
- certainty (determinism) – an algorithm with the same input data produces the same result;
- understandability – the algorithm must contain commands that are part of the executor's command system;
- finiteness (effectiveness) – the algorithm must complete its work in a finite time;
- universality (massiveness) – the algorithm must be applicable to various sets of input data.

The algorithm can be represented as a "black box" (Fig. 1.1), where X is the set of input data, Y is the set of output data. Accordingly, the algorithm itself performs the transformation from X to Y.

Rice. 1.1. Algorithm in the form of a "black box"

The following *methods of writing algorithms are distinguished:*

- natural language;

- block diagram (graphical way of presenting algorithms);
- program (in a programming language);
- pseudocode (a mixture of programming language and natural language commands).

Let's consider *the basic structures of algorithms (algorithmic constructions):*

1. *Linear*– a group of algorithm steps performed sequentially one after another. In Fig. 1.2 shows a linear sequence consisting of two steps.

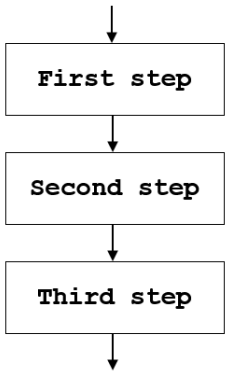


Fig. 1.2. Flowchart "Linear"

2. *A fork (conditional operator)* is a form of organizing an algorithm in which, depending on the fulfillment or non-fulfillment of a condition, one or another sequence of commands is executed. The “branching” construction (Fig. 1.3) is written in full form: if the condition is true, then action 1 is performed, if the condition is false, then action 2 is performed.

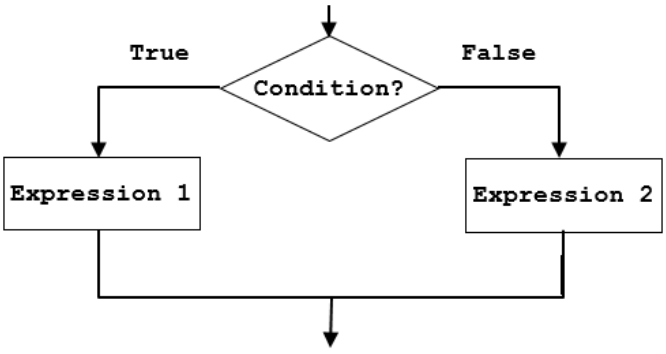


Fig. 1.3. Block diagram "Fork (conditional operator)"

If a branch contains actions for only one branch, then it is said to be written in abbreviated form.

3. *A cycle* is a form of algorithm organization in which the same sequence of commands is executed several times until a certain goal is achieved. Each execution of the

loop once is called an iteration. If the body of the loop is executed N times, it is said that N iterations have been completed.

Theoretically, two types of cycles are distinguished: cycles with a previously known number of repetitions (cycle with a parameter, Fig. 1.4 c) and cycles with a previously unknown number of repetitions. In loops with an unknown number of repetitions, a loop condition is used to determine when to stop executing the loop body. There are: cycles with a precondition (Fig. 1.4 a) and cycles with checking the condition after completing the next iteration (cycles with a postcondition) (Fig. 1.4 b).

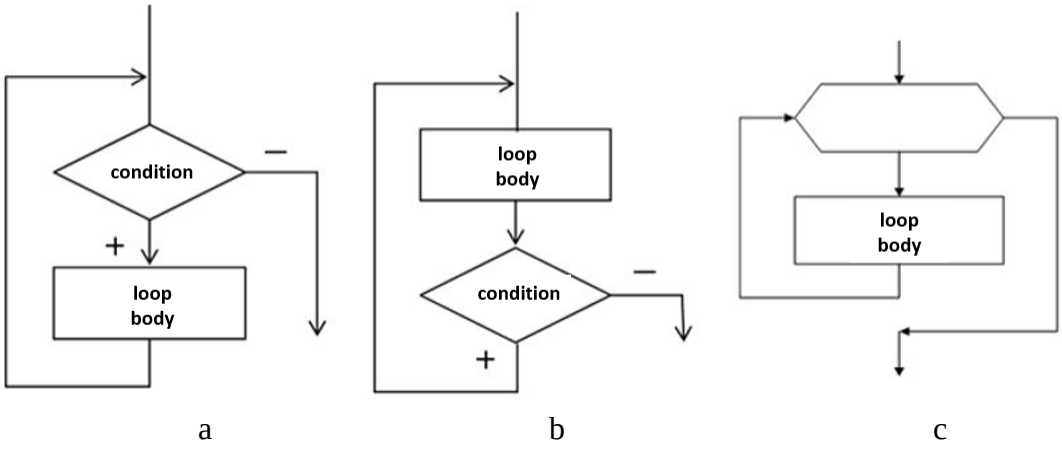


Fig. 1.4. Block diagram with types of cycles:
a) with a precondition; b) with postcondition (for C / C ++); c) with parameter

TOPIC 2. PROGRAMMING LANGUAGES AND TECHNOLOGIES

In a narrow sense, programming refers to the process of developing software in a specific programming language. Programming language (PL) – a formal sign system designed for recording computer programs. It defines a set of lexical, syntactic and semantic rules that determine the appearance of the program and the actions that the performer (i.e., the computer) will perform under its control.

The first programs were written in *machine code* for a specific computer architecture, which was very labor-intensive and technically difficult. And now critical parts of the program code are implemented in low-level language (this makes it possible to realize all the capabilities of the processor, programs become efficient both in speed and in the amount of memory used).

Example: program "Hello, world!" for x86 architecture processor (MS DOS OS, output using BIOS interrupt int 10h)

```
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 656C 6C 6F
2C20 57 6F72 6C64 21
```

Historically, languages with special symbolic notations (mnemonic commands) began to appear. This is *an assembly language* , which still remained machine-oriented (each processor has its own command system, its own assembly language and, accordingly, the program cannot be run on a different type of processor). The very system for translating these mnemonic commands into machine code is called *an assembler*.

Example :

```
mov ax, 3
mov bx, 2
sub ax, bx
```

Of course, a programmer wants to "talk" to a computer in natural language, without thinking about the type of processor. In the 1960s High-level programming languages began to appear in which commands consist of natural language words. To translate a program from a high-level language into machine codes (low-level language), a special program is needed – a *translator* .

There are two types of translators:

1. *The interpreter* analyzes the program in logical parts. The program code is decrypted and immediately executed. The program itself remains in the original language and cannot be run without an interpreter. Advantages: programs are portable (only an interpreter is needed) and easy to debug. The disadvantage is that an interpreter is required

for execution and programs execute more slowly (due to the presence of an additional software layer).

2. *The compiler* translates the entire program into machine code and builds an executable file. In fact, the program code is converted by the compiler into object modules, which are assembled by the linker into an executable program. Advantages: no translator is needed for execution, programs run quickly (since they are already in machine code). The disadvantage is that the executable program only runs on one OS.

As for the latter, ideally you would like to develop one program for different operating systems. Therefore, the modern stage of development of the program development process offers an intermediate option (combining the advantages of compilation and interpretation) – *translation into pseudocode*. The algorithm is as follows:

a) the source program code is processed by a compiler, which builds "intermediate code (pseudocode)". For example, bytecode is used as a type of pseudocode, in which each command occupies 1 byte.

b) the intermediate code is executed by an interpreter (on a virtual machine), which ensures the portability of programs (pseudocode) to any computer that has a virtual machine (they are developed for any architecture and OS).

Thus, ready-made programs for ActionScript , Java, etc. are distributed in the form of bytecode, and virtual machines (Flash player, .NET, Java machine, etc.) are used for their execution. When executed, the bytecode is converted into instructions for a specific processor. Due to the additional software layer, the program runs slower than "native" programs for the selected computer architecture.

There are many *programming paradigms* , and teaching each of them has its own specifics. *A programming paradigm* is a set of ideas that determine the style of writing computer programs.

1. *Imperative (procedural-oriented) programming* describes the computation process in the form of a step-by-step sequence of instructions that change the state of the input data. This allows the programmer to define each step in the process of solving a problem.

2. *Declarative programming* is a fundamentally different approach to programming, in which programs do not prescribe a certain sequence of actions, but describe the rules for generating a result (thus giving "permission" to the performer to independently determine the way to achieve the goal). The program specifies what needs to be done, i.e. the result, not how it should be done. There are two directions:

- *Functional programming* assumes that the computation process is treated as the calculation of function values;

- *Logical programming* assumes that the program is formulas of mathematical logic, and the performer, to solve the problem, tries to derive logical consequences from them. Thus, the Prolog programming language is used for problems of artificial intelligence and symbolic information processing. Prolog is based on formal predicate logic, which provides a convenient means for representing knowledge in the form of facts and rules of inference. A Prolog program describes not a procedure for solving a problem, but a logical model of the problem domain.

So, a declarative program states (declares) what is to be achieved as a goal, and an imperative program prescribes how to achieve it. For example, you need to get from point A to point B. The procedural approach will give a sequential list of commands: from point A to the street. Permskaya north to the Ural Volunteers Square, from there along the street. Lenin two blocks, then turn right to house No. 2, this is point B. In the declarative approach, a city plan will be given, which indicates both points, and "permitted" traffic rules. The courier himself looks for the way from point A to point B.

3. *Object-oriented programming* consists of modeling domain objects through hierarchically related classes.

Modern programming languages allow the use of different paradigms – multi-paradigm programming.

The classic and most common is the imperative (procedural) paradigm. Since most programming languages support this paradigm, therefore, most often in schools, colleges and universities, the beginning of programming is associated with the procedural approach.

Let's consider history of development of programming technologies. *Programming technology* is a set of methods and tools used in the process of program development. Highlight:

a) technologies used at specific stages of development (for specific tasks);

b) technologies covering the entire development process (the basis is an approach (paradigm) that defines a set of methods used at different stages of development (i.e. methodology)).

Let us consider the process of historical development programming technologies.

I: "spontaneous" programming (from the advent of the first computers to the mid-1960s)

During this time period, the development of programming languages can be represented in the form of a diagram: replacement of machine languages → assemblers → algorithmic languages (Fortran, Algol), as well as reuse of subroutines, which increased

the productivity of the programmer. At this time, programming was considered an art, but problems still began to arise:

- the complexity of developed programs has increased;
- low program reliability;
- large time and cost of program development;
- Difficulty in testing and debugging programs.

As a result of this, in the late 1960s. A programming crisis has arrived.

II: " structural approach" (1960–1970)

Key ideas:

- decomposition of complex systems for the purpose of subsequent implementation in the form of separate small subroutines (hierarchy of subtasks);
- using only basic algorithmic structures when compiling programs and prohibiting the use of the unconditional jump operator;
- top-down program design.

Later this approach was called *procedural decomposition* . Support for the principles of structured programming is the basis of procedural programming languages (PL/1, ALGOL-68, Pascal, C).

From these positions, *the concept of modular programming is being developed* , which consists of separating subroutines into separately compiled modules (libraries).

Added to the ideas discussed above:

- principle of modularity: the program is divided into separate modules that can be developed and debugged independently of each other;
- restriction of access to global program data and the principle of locality (the module uses its local variables, and global ones only in extreme cases);
- development of custom data structures;
- principle of subordination: connection between modules “top-down”.

III: object-oriented programming (mid-1980 – late 1990) – software creation technology based on representing the subject area and program as a collection of objects, each of which is an instance of a certain type (class), and the classes form a hierarchy with inheritance properties. The interaction of software objects in such a system is carried out by passing messages.

Advantages of OOP:

- “natural” decomposition (compared to the modular approach), which facilitates its development, which leads to more complete localization of data (class properties) and

their integration with processing routines (class methods), which allows independent development of program objects;

- reuse of program codes and creation of class libraries.

Within the framework of this technology, *visual programming is being developed* (Delphi, C++ Builder, Visual C++, etc.).

IV : component approach and CASE technologies (since the mid-1990s) Component approach enables the creation and reuse of components for any programming language supported by some software platform.

Key ideas:

- building a program from components (separate parts of software that interact with each other through standardized interfaces). For example, technologies:
  - OLE (Object Linking and Embedding);
  - COM technologies (Component Object Model);
  - CORBA (Common Object Request Bracer Architecture);
  - and etc.;
- components are assembled into dynamically called libraries or executable files distributed in binary form (without source code);
- use of computer technologies (CASE technologies) for creating and maintaining software at all stages of the life cycle;
- software platforms and common language execution environments. For example, the platform. NET and its programming languages.

The development of modern programming technologies has led to the emergence of concepts of unified development and execution platforms (for example, .NET Framework (Core)). Within its framework, it is possible to develop in different programming languages. New languages (C #) appear and old ones are modified (extended). Thus, C++/ CLI is an extension of the C++ language, J # is the Java language in relation to the . NET (it is possible to import both .NET and Java libraries).

The question often arises of which programming language to choose when implementing a particular application. It is important to understand that a specific programming language will be convenient for solving a specific problem. Thus, assembler and C languages are used for system programming , and C ++, C #, Java are used for application programming . For programming Internet sites: PHP, JavaScript, Perl, ASP, Python, etc. Therefore, for a successful career as a programmer, you must have a set of competencies in a number of programming languages.

Let's look at the concepts of translation and the operation of popular programming languages.

The C/C++ translation concept involves assembling object code for a specific hardware platform (for example, Windows x86, Windows x64, etc.), i.e. for a physical machine. Direct memory management makes the language unsafe, and all responsibility is placed on the programmer (for example, irrational memory management creates memory leaks and the creation of "garbage"). In this case, the machine code is called unmanaged code. But C/C++ programs are efficient in both speed and memory.

The C# translation concept involves creating programs for the .NET platform (read "dot no", from the English dot – dot) from Microsoft. Programs are compiled not into processor instructions, but into code in the intermediate language CIL (from the English "common intermediate language"). This code is executed by a special CLR program – a virtual machine of the .NET environment, which is positioned as a general execution environment for its programming languages. In fact, the program code is generated for a virtual machine, not a physical machine. Thus, program modules in C ++, C#, F #, Pascal can be compiled into CIL code . NET , VB . NET , etc. Accordingly, where there is a virtual machine (the .NET platform installed), the C# object code will work there, which ensures its cross-platform functionality. C# is positioned as a safe language due to the automation of the problem of garbage collection, additional checks that occur during execution, etc. Due to the additional software layer, programs run slower and are inferior to C ++ in efficiency, but are more secure. In this case, the term "managed code" is used to refer to the program code executed under the "control" of the .NET virtual machine. The term "managed" here refers to the method of exchanging information between the program and the runtime environment (at any point in execution, the control environment can pause execution and obtain state information). The information needed to do this is provided in managed CIL code and in the metadata associated with that code. On UNIX -like operating systems (such as Linux , macOS), such programs can be executed in the Mono environment .

The modern C++/ CLI for the .NET platform has tools for creating and executing both unmanaged and managed code. Therefore, it is of interest for the development of programs, some components of which are implemented in efficient unmanaged code.

Classification of programming languages:

1. By broadcast method:

- compiled languages (Pascal, C/C++, etc.);
- interpreted languages (Python, PHP , JS , AS , etc.).

2. By levels:

- low-level languages (machine language, assembler);
- high-level languages (Pascal, C/C++, C#, Java, etc.).

3. By paradigms:

- imperative (procedural) languages (Basic, Pascal, C, etc.);
- declarative languages:
  - functional (Haskell, Lisp, ML, F#, etc.);
  - logical (Prolog, etc.);
- object-oriented languages (C++, C#, Java, Python, etc.).

4. By area of destination:

- general purpose languages (C, C++, Java, C#, Visual Basic, Delphi, etc.);
- languages for teaching programming (BASIC, Pascal, Logo, Python, etc.);
- website programming languages (PHP, JavaScript, Perl, ASP, Python, etc.);
- programming languages for artificial intelligence tasks (Lisp, Prolog, etc.).

Today, to conveniently develop programs, programmers use specialized programs – *programming systems* or *integrated development environments (Integrated Development Environment, abbr. IDE)* . It is a system of software tools used by programmers to develop and debug new programs. For example , Microsoft Visual Studio, DEV-C++, Turbo Pascal, Delphi, Lazarus, C-Free and etc.

Typically, an IDE includes:

1. Text editor – a tool for editing program code.

2. Translator – a tool designed to convert source program code (files *.c, *.cpp) into object modules (*.o).

3. Linker (assembler, linker) – a tool that assembles program modules and functions of standard libraries into a single executable file.

4. Debugger (from English debugger) – a tool designed to find errors, execute a program step-by-step, implement breakpoints, and visually view the values of variables.

5. Profiler (from English profiler) – a tool that estimates the operating time of program code components to optimize it.

6. Separate CASE tools – tools for designing program elements, etc.

So, after writing the program text, it is necessary to translate it into machine code.

Stages of converting a program into machine code:

1. The source program code is transferred to the preprocessor, which pre-processes it before the translator (adds the files described in the #include directive) . The result is the complete program code.

2. The complete program code is input to the compiler, which builds an object module (.o).

3. The linker (link editor) assembles object modules (including those containing standard library functions, for example, input-output) and generates an executable file (.exe)

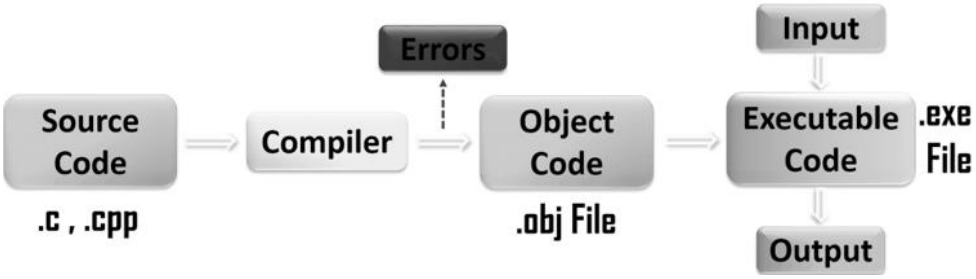


Fig. 1.5. Stages of translating a program into machine code

History of creation and scope of C/C++ languages

In 1972, Bell Labs employee D. Ritchie introduced the C language (as a development of the B language) as a language for developing the UNIX OS. The C language is also called "general-purpose assembler", "high-level assembler" or "middle-level language". Programs written in C are more efficient than those written in many other languages (only manually optimized assembly code will work even faster).

The corresponding compilers have been developed for all major computer architectures and have been optimized for several decades.

In 1983, Bjarne Stroustrup created the first version of the C++ language, adding object-oriented features from the Simula (OOP) language and fixing bugs. Early versions of the C++ language, known as “C with classes,” began to appear in 1980. In 1983, the language “C with classes” was renamed C++. The language combines the properties of both high-level and low-level programming languages. The author left traditional C types, operators, pointers, etc. The C++ language is positioned as a compiled, statically typed general-purpose programming language.

Scope of application C/C++:

1. OS and system programming

- UNIX OS kernel 1969 г. (developed in assembler), in 1972 г. rewritten into C language;
- Windows OS 1.0 kernel 1985 г. – "C" language with assembly inserts;
- Linux OS – C language, 97% of all supercomputers in the world run on Linux;
- Mac OS kernel – C language;
- mobile iOS, Android and Windows Phone – C language.

2. Drivers for devices: "C" language with assembly inserts.

3. Device microcontrollers (ATM, cash register, traffic light, video surveillance, automatic transmission control).

4. DBMS and DB: combination of C/C++: Oracle Database, MySQL, MS SQL Server, PostgreSQL.

5. High-load systems, server software, graphics, video, special effects, games.

Integrated development environments are used to develop programs: Microsoft Visual Studio, Microsoft Visual C++ Express (2008, 2010, 2013), C-Free, Dev-C++. Algorithm for creating a Microsoft Visual C++ 2010 IDE console application:

- create a new project (solution name) → Win32 Console application (empty project) → add a new .cpp file (file name) to the "source files" folder;
- create a new project (solution name) → CLR → Empty CLR project.

Listing. The simplest program in C /C++ (option 1):

```
int main ()
{
// this is the main program
/* operators */
return 0; // mechanism for terminating the main function
}
```

Every program must have a main function, `main ()` , which receives control when the program starts. All other functions needed to solve the problem are called from `main ()`.

The word `int` before `main ()` indicates that the program at the end of its work transfers the result of its work to the OS – the integer 0 `(return 0;)`. Using this number, the OS determines whether the program completed successfully or with an error.

Listing. The simplest program in C /C++ (option 2):

```
void main()
{
// this is the main program
/* operators */
}
```

The word `void` (from the English “empty”) before `main ()` indicates that the function will not return a result.

Standard Library in C

The standard library is a collection of classes and functions written in the base language. Supplied with the compiler <cstdlib> (stdlib.h) “standard library” (standard library) – header file of the standard library of the C language, containing functions that allocate memory, control the process of program execution, convert. types, etc.

Module `<cstdio>` (stdio.h (from the English standard input/output header – standard input-output header file)) is a header file of the standard C language library, containing definitions of macros, constants and declarations of functions and types used for various operations of the standard input and output.

Module `<conio.h>` (from the English console input-output – console input-output): description of functions for working with the keyboard and monitor.

Module `<cctype> (ctype.h)` is a header file of the standard library of the C programming language, containing declarations of functions for classifying and converting individual characters.

module `(float.h)` is a header file of the C programming language standard library that contains a macro that defines various restrictions and parameters of floating-point types.

module `(math.h)` is a header file of the C programming language standard library designed to perform simple mathematical operations.

Module `<cstring> (string.h)` is a header file of the standard C library containing functions for working with null-terminated strings.

Standard Library in C ++

The C++ language is compatible with the C language (in C++ you can use C libraries). The C library header files are included in the C++ library (by stripping the .h extension and adding a 'c' at the beginning (for example, 'time.h' is replaced by 'ctime').

Stream I/O Modules

Module `<iostream>` – implements the basics of input and output of the C++ language.

`<istream>` module – implements the std::istream class template and other necessary classes for input.

`<ostream>` module – implements the std::ostream class template and other necessary classes for output.

`<iomanip>` module – implements tools for working with output formatting, for example, a base used when formatting integer and exact floating point values.

Module `<fstream>` – implements tools for file input and output.

`<sstream>` module – implements the std::sstream class template and other necessary classes for working with strings.

Containers

Module `<deque>` – implements the container class template std::deque – a doubly connected queue.

Module `<list>` – implements the container class template std::list – a doubly linked list.

`<map>` module – implements container class templates std::map and std::multimap – associative array and multimap.

Module `<queue>` – implements the adapter-container class std::queue – a one-way queue.

`<set>` module – implements container class templates std::set and std::multiset – sorted associative containers or sets.

Module `<stack>` – implements the container class std::stack – stack.

Module `<vector>` – implements the container class template std::vector – a dynamic variable-length array.

Are common

`<algorithm>` module – implements definitions of many algorithms for working with containers.

`<iterator>` module – implements classes and templates for working with iterators.

String

`<string>` module – implements standard string classes and templates.

The Standard Template Library (STL) is a subset of the C++ Standard Library. STL objects are declared within the namespace STD (Standard Template Definition).

Connecting library functions

In any language there is a way to connect to the program various structures used in another module.

For example,

```
#include <iostream>
```

`#include` directive is used to include other files in the code. The `<iostream>` module is in the "standard library" and is responsible for standard input streams and console output.

Header file , or included file, is a file “inserted” by the compiler into the source text in the place where the #include <file.h> directive is located (Extension.h ; sometimes .hpp.).

TOPIC 3. BASIC STRUCTURES OF ALGORITHMS

Variables and data types

A variable is a value that has a name, type, and value. The value of a variable can be changed while the program is running. A variable (C/C++) stores data of a certain type. The variable characterizes:

- size of the allocated memory area and data storage format;
- the range of values it can store;
- a set of operations with this variable.

Data types:

- `int` // integer
- `long int` // long integer
- `float` // real
- `double` // double precision real
- `bool` // logical values
- `char` // character
- `and etc.`

Declaring a variable is defining a name, type, and allocating space in memory. The initial value can be specified directly in the variable definition statement (initialization).

Declaring a variable in C / C ++: Specifies the data type for that variable, then the name of that variable. If we declare, for example, a variable of type `int ,` then at the machine level it is described by two parameters – its address and the size of the stored data.

```
int a; // declaration of a variable a of integer type
float b; // declaration of variable b of floating point data type
double c = 9999.99; // initialization of a double type variable
char d = 's'; // initialization of a variable of type char
bool k = true; // initialization of the logical variable k
```

Note that when memory is allocated, global variables are filled with zeros, and local variables contain “garbage” (remaining data in memory from other programs).

The most important operator is the assignment operator.

An assignment operator is a command to write a new value to a variable.

```
int i = 1024; or int i ( 1024 );
string p = "Hello"; // or string p("Hello");
```

Built-in data types have a special syntax for specifying a null value:

```
int i = int();
double d = double();
```

Variable *i* gets the value 0, and variable *d* gets the value 0.0

There are also special variables for storing memory addresses – these are *pointers* . Pointer declaration syntax:

```
<type> *<name>;
```

An example of a pointer declaration is shown below:

```
int a = 5; // variable of integer type
int * p ; // declaration of a pointer (from English Pointer )
p = & a ; // the address of an existing static variable is written to the pointer
```

Data recording formats

The octal format is described by the prefix "`0`" . For example:

```
int b = 010; // 8th SS
cout << b +1 << endl ; // 9
```

The hexadecimal format is described by the prefix "`0x`" . For example:

```
int a = 0xFFFFFF; // hexadecimal
cout << a + 1 << endl; // 16777216
```

Description of constants

The const keyword can be added to an object's declaration to make that object a constant rather than a variable.

```
const int m = 1;
```

Input and output (C library functions)

Format is a character string that shows what types of data the input/output is organized for. A distinctive feature of the format is the presence of the "%" symbol in front of it:

```
printf ("%d", var); // output on display
```

Formats:

- `%d` – signed integer in decimal notation;
- `%x` – signed integer in hexadecimal notation;
- `%o` – signed integer in the octal number system;
- `%f` – real number;
- `%c` – symbol;
- `%s` – character string;
- and etc.

Printing data in a specified format:

```
int a = 1234;
int b = 1234.567;
printf ("%8d", a); // total 8 positions
printf ("%9.4f", b); // 9 positions in total, 4 fractional digits
```

`scanf ()` function implements formatted input by passing the input format string and memory cell addresses.

```
scanf("%d%d", &a, &b);
```

Listing. Finding the sum and difference of two numbers

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a, b, c;
```

```
printf ("Enter two integers\ n ");
  scanf("%d%d", &a, &b);
c = a + b;
printf("Amount: %d \n ", c);
printf("Difference: %d-%d=%d", a, b, a-b);
getch ();
}
```

Input and output streams (C ++ language library functions)

Data stream in programming is an abstraction used for read and write operations (i.e. moving data from source to destination).

The module `<iostream>` supports file input/output of built-in data types. I/O operations are performed using: istream class (streaming input), ostream class (streaming output), iostream class supports bidirectional I/O.

The library defines standard stream objects:

1. Data entry. `cin` object belongs to the `istream` class and corresponds to the standard input stream, which in general allows you to read (extract) data from the terminal (keyboard). Input is done using the overloaded bit shift operator >>. In C ++ language, operations can be overloaded, i.e. can perform different actions depending on the context in which they occur.

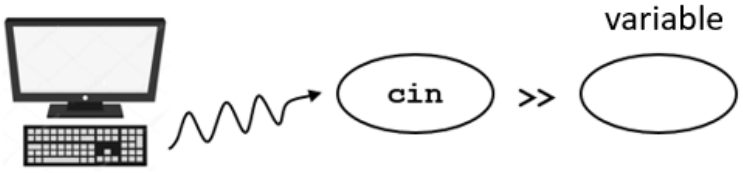


Fig. 1.6. Reading (extracting) data (>>) from standard input using `c in object`

Example: data entry

```
cin >> a;
cin >> a >> b;
```

2. Data output. An object `cout belongs to the ostream` class and corresponds to the standard output stream, which in general allows you to output (put) data to the

terminal. The output is done using the overloaded left shift operator << . In this case, this is a write operation to a stream (put data into a stream).

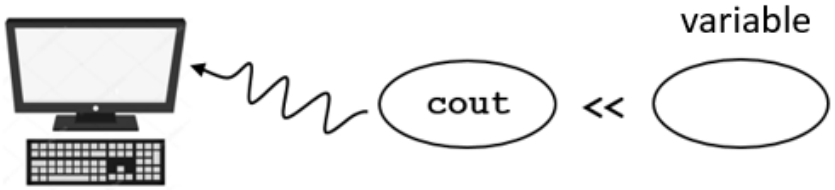


Fig. 1.7. Writing data (<<) to standard output using `cout`

Example: Data Output

```
cout << a;
cout << " Answer : " << c;
```

There are also additional objects, for example, `cerr` – an object of the `ostream class` , corresponding to the output of program error messages.

In practice, the names of user variables may coincide with the names of standard programming language library objects. Therefore, *a namespace is introduced* – an additional identifier to any names declared in it to prevent name collisions. Used when using libraries developed by various independent vendors. If, for example, you do not include the namespace of the standard template library `"using namespace std"` , then you will have to separately register `std::"object"`.

For example, `std::cout << "Programming";`

Listing. Finding the sum and difference of two numbers (version 2)

// without `using namespace std`

```
#include <iostream>
#include <iomanip>
void main()
{
 int a, b, c;
 std::cin >> a >> b;
 c = a + b;
 std::cout << std::setw(10) << c; // set width – set width
 std::cout << a << "-" << b << "=" << a – b;
 system (" pause ");
}
```

Namespaces can be at the level of a specific programming language library, for example:

```
using namespace System;
...
Console::WriteLine("C++/CLI language (Visual C++)");
```

There may also be custom ones, for example:

```
...
namespace X{int a = 2, b = 8;};
namespace Y{int b = 3;};
using namespace X;
using namespace Y;
int main()
{
Y::b = X::b / X::a;

printf("%d", Y::b); // Print : 4
}
```

Cyrillic in the console

Text written in Latin in the Windows command line will be displayed correctly. And if the text message is written in Cyrillic, then instead of the transmitted message, an "incomprehensible" sequence of letters and symbols will be displayed. Reason: use of different character encoding tables (cp866, cp1251, utf-8, etc.). `SetConsoleCP()` function, setting the locale (`setlocale()` function performs character conversion in accordance with the required language).

Option 1:

```
# include < Windows.h >
...
SetConsoleCP (1251); // set the win code page – cp 1251 to the input stream
SetConsoleOutputCP (1251); // set the win code page – cp 1251 to the output stream
```

Option 2:

```
#include <iostream>
...
system("chcp 1251");
...
```

Option 3:

```
#include <locale.h>
...
setlocale(LC_ALL,"RUS");
// setting the locale
...
```

Command Line Options

When running a program from the command line, it is possible to pass parameters to it in text form. Parameters are described as arguments to the `main () function`. The first argument `argc` is the number of parameters passed to the program. The second argument `* argv []` is a pointer to an array storing the remaining parameters (see section 3 for more information on pointers). In fact, `argc` is the array counter `* argv []`.

Listing. Passing parameters on the command line

```
#include <iostream>
using namespace std;
void main(int argc, char *argv[])
{
for (int i = 0; i < argc; i++) {
// Output arguments in a loop
cout << "Argument " << i << " : " << argv[i] << endl;
}
}
```

Next, I launch the compiled program file from the command line, passing parameters, for example:

```
C :\> test \ test 1. exe 5 0 1 2 3 4
```

The output will be as follows (with `argv [0]` containing the name of the program file):

```
Argument 0 : test\test1.exe
Argument 1:5
Argument 2: 0
Argument 3: 1
Argument 4:2
Argument 5:3
Argument 6:4
```

Evaluating Expressions

C/C++ uses linear notation of expressions with the following priority (precedence) of operations :

- brackets;
- multiplication and division;
- addition and subtraction.

The division operation (C / C ++) has a special feature: the result of dividing an "integer by an integer" is an integer (the data type is determined by the operands).

Example :

```
int x = 3, y = 4;
float z;
z = 3 / 4; // = 0
z = 3.0 / 4; // = 0.75
z = float( x ) / 4; // = 0.75
```

If you need to get the remainder of a division, use the % operation. Example :

```
17 = 8 * 2 + 1
17 / 2 = 8 (whole part)
17 % 2 = 1 (remainder)
```

Example :

```
int a, b, d;
```

```
d = 75;
b = d / 10; // output 7
a = d % 10; // output 5
```

In C/C ++, abbreviated notations of operations are used (primarily this is justified by the convenience of the computational process, and not by the brevity of the notation):

```
int x, y ;
...
x ++; // full form x = x + 1;
x --; // full form x = x – 1;
x += y ; // full form x = x + y ;
x -= y ; // full form x = x – y ;
x *= y ; // full form x = x * y ;
```

and etc.

The abbreviated notation of the operation can be located to the left or to the right of the object, according to which *a prefix-increment and a post-increment are distinguished.* The post-increment operation (i++) returns the value of variable i before the increment was performed, and the prefix-increment operation (++i) returns the value of the variable that has already been modified.

Example :

```
int a = 10;
int b = 10;
cout << a++ << endl; // output 10
cout << ++b << endl; // output 11
```

In the above example, the result of the program will be the output of the numbers 10 and 11, since the post-increment operation returns the value of the incremented variable before the increment is performed.

Example :

```
int x = 5;
int y;
y = ++x; cout << y << x << endl; // output y = 6, x = 6, t . to .
first x increases by 1, then y = x .
```

```
    y = -- x ; cout << y << x << endl ; // output y = 5, x = 5, because
x decreases by 1, then y = x .
    y = x ++; cout << y << x << endl ; // output y = 5, x = 6, because
the value y = x is assigned , then x is increased by 1.
```

Real numbers

In most languages, the integer and fractional parts of a real number are separated by a dot.

Example:

```
float f = 987.654;
cout . width (10); // width 10 characters
cout . precision (5); // significant 5 characters
cout << f << endl ; // output: _ _ _ _ 987.65
```

In scientific calculations, real numbers are often represented in exponential format. Example:

```
float x = 987.654;
cout.width(10);
cout.precision(2); // in the fractional part
cout << scientific << x; // _9.87e+002
```

Standard Math Functions

To connect the mathematical library of the C language, you need to write the following line:

```
#include <math.h> // or < with math>
...
abs ( x ) — modulus of an integer;
fabs ( x ) — module of a real number;
sqrt ( x ) — square root;
sin ( x ) — sine of the angle specified in radians;
cos ( x ) — cosine of the angle specified in radians;
exp ( x ) — exponent ex;
ln ( x ) — natural logarithm;
```

```
pow ( x , y ) – raising the number x to the power y ;
floor ( x ) – rounding down;
ceil ( x ) – rounding up
```

Pseudorandom numbers

In a number of tasks (simulation modeling, modeling of complex processes, etc.) it is necessary to obtain a random value in a certain range (random number sensor (RNS)). For these purposes , the library function `rand ()` is provided , returning a “pseudorandom” number in the range from 0 to RAND_MAX (for the MS compiler Visual Studio constant value is 32767).

Listing. Random Number Sensor (RNS)

```
#include <iostream>
#include <ctime>
using namespace std;
void main()
{
system("cls"); // clear screen
srand(time(0));
cout<<RAND_MAX<< endl;
int a = rand()%1000;
cout << a;
system("pause");
}
```

Test questions and general tasks:

1. Define an algorithm. List the properties of the algorithm.
2. Name ways to write algorithms.
3. Name the basic algorithmic structures and give them a brief description.
4. List the main programming paradigms.
5. Provide basic C ++ data types.
6. How do you stream data input and output in C++?
7. Describe the input and output format in C language.
8. Describe the rules for evaluating an algebraic expression.
9. Explain basic logical operations and give examples of their use in programs.

10. General task. Organize the calculation of the area of a rectangle (data input, calculation, data output).

Conditional operator

In some cases, it is necessary to implement the choice of one of two possible paths of program execution depending on some condition. To check the condition, a logical statement (expression) is formulated, which can be true (True) or false (False) (Fig. 1.8).

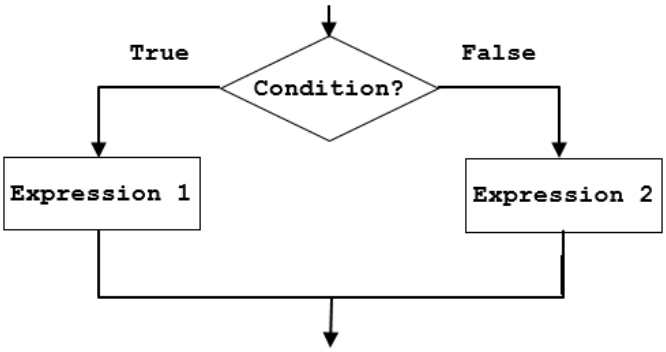


Fig. 1.8. Full form of branching organization

The syntax for the full form of branching is:

```
if <logical expression>
   { exptession 1; }
 else
   { exptession 2; }
```

The shorthand syntax for branching is:

```
if <logical expression> { statement block; }
```

To organize a logical expression, relation signs are used:

- Equality ==
- Inequality !=
- Greater than or equal to >=
- Less than or equal to <=
- More >
- Less <

Listing. Maximum of two numbers

```
void main ()
{
 int a , b , max ;
 printf ("Entering numbers\ n ");
 scanf("%d%d", &a, &b );
 if (a > b)
      max = a;
 else
      max = b;
 printf ("Maximum number = % d ", max );
}
```

It is possible to use nested conditions:

```
if (< boolean expression 1>)
{
operator block 1;
}
else if (< Boolean expression 2>)
 {
      operator block 2;
 }
 else
 {
      operator block 3;
 }
```

The logical expression in a conditional statement can be as complex as desired. *A complex condition* is a condition consisting of several simple conditions (relations) connected using basic logical operations:

`!` – NOT (not, negation, inversion);

`&&` – AND (and, logical multiplication, conjunction, simultaneous fulfillment of conditions);

`||` – OR (or, logical addition, disjunction, fulfillment of at least one of the conditions).

The programming language defines the order in which complex conditions are executed:

- `expressions in brackets`
- `! (NOT, negation)`
- `<, <=, >, >=`
- `==, !=`
- `&& (AND)`
- `|| (OR)`

Example: condition for hiring employees aged 16–65 years (inclusive).

```
if (v >= 16 && v <= 65) cout << "fits";
                 else cout << "not suitable";
```

A shortened way of writing the conditional operator (`"? :"`) is also possible . Syntax operator :

```
 "condition" ? "expression 1" : "expression 2" .
```

The conditional operator (?:) takes three operands. The first operand is evaluated and implicitly converted to type `bool` . If the result of the first operand is true (1), then `"expression 1" is executed` . If the result of the first operand is false (0), then `"expression 2" is executed .`

Listing. Maximum of two numbers

```
int i = 1, j = 2;
int max = 0;
i > j ? max = i : max = j;
   cout << max << " is max";
```

The above code can be replaced with the following:

```
cout << ( i > j ? i : j ) << " is max";
```

Example: Finding a Positive Number

```
int i=3;
int j=(i>0) ? eleven;
```

The above code can be replaced with the following:

```
if (i>0)
   j=1;
else
   j=-1;
```

Multiple selection operator (switch)

`switch` selection statement is a replacement for the multiple use of `if statements` . The operator is designed to organize a choice from many different options. The key variable in parentheses is compared with the values described after the `case statement` . If none of the keys is equal to the expression, then control is transferred to the `default operator` . The `break` operator ensures that the execution of the innermost `of the switch, do, for, and while statements` that unite it is terminated .

Listing. Select one of 4 options

```
#include <iostream>
using namespace std;
void main()
{
int key;
int a,b;
cout << "First number: ";
  cin >> a ;
  cout << "Second number: ";
  cin >> b ;
  cout << "1-addition; 2-subtraction; 3-multiplication; 4-divi-
sion: ";
  cin >> key;
switch (key)
```

```
{
case 1:
{
cout << a << " + " << b << " = " << a + b << endl; break;
}
case 2:
{
cout << a << " – " << b << " = " << a – b << endl; break;
}
case 3:
{
cout << a << " * " << b << " = " << a * b << endl; break;
}
case 4:
{
cout << a << " / " << b << " = " << float (a) / b << endl; break;
}
default:
cout << "Invalid input" << endl;
     }
 system (" pause ");
}
```

Test questions and general tasks:

1. Draw a block diagram of a fork (complete).
2. Draw a block diagram of the fork (incomplete).
3. What type of expression can act as a condition when organizing branching?
4. What is a complex condition?
5. When is it appropriate to use the switch multiple choice operator ?
6. General task. Find the sum of the maximum and minimum values of the three entered numbers.

Cycles (Loops)

Cyclic algorithms are designed for repeated execution of program fragments. From a theoretical point of view, there are two types of cycles:

- cycles with an unknown number of steps (with precondition and postcondition);

- cycles with a known number of steps.

1. Loop with precondition

The block diagram of a cycle with a precondition is shown in Fig. 1.9. In the diagram, the “loop body” is one or more operators, in the general case a compound operator. A condition (logical expression) is an expression of the condition for completing a loop. The body of the loop is executed as long as the boolean expression evaluates to true. When the boolean expression is false (equal to 0), control is transferred to the statement behind the loop.

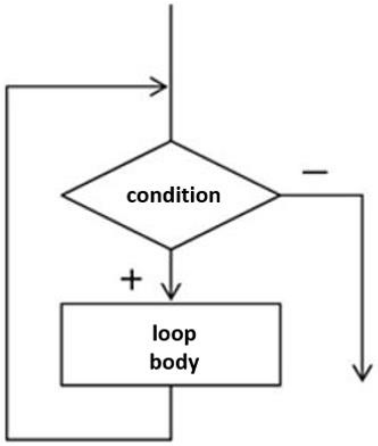


Fig. 1.9. Block diagram of a loop with a precondition

Syntax:

```
while (<Condition: while the condition is true, execute>)
{
Loop Body
}
```

Listing. Converting units of measurement (loop with precondition)

```
#include <iostream>
using namespace std;
int main()
{
 int sm(0);
```

```
 double d;
 while (sm<100)
 {
 sm++;
 d=sm/2.54;
 cout << sm <<" centimeters equals "<< d <<" inches\ n ";
 }
    return 0;
}
```

2. Loop with postcondition

The block diagram of a cycle with a postcondition is shown in Fig. 1.10. The diagram shows that when entering the loop, the condition is not checked and the body of the loop is always executed at least once. As soon as the condition (logical expression) becomes false (i.e. equal to 0), the loop ends and control is transferred to the next operator in order.

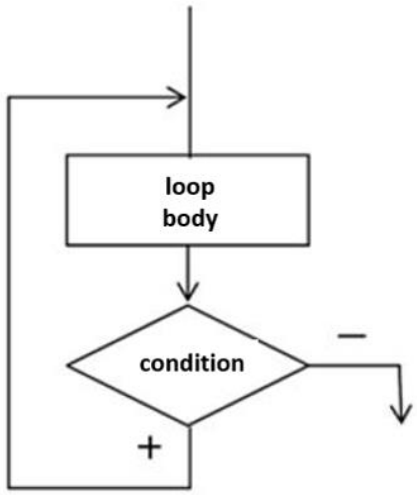


Fig. 1.10. Block diagram of loop with postcondition (C/C ++)

Syntax

```
do
 {
  Loop body
 }
while (<Condition: while the condition is true, execute>)
```

Listing. Converting units of measurement (loop with postcondition)

```
#include <iostream>
using namespace std;
int main()
{
 int sm(0); double d;
 do
 {
 sm++;
 d=sm/2.54;
cout << sm <<" centimeters equals "<< d <<" inches \n";
 }
 while (sm<100);
return 0;
}
```

3. Cycle with parameter

If the number of repetitions (iterations) of a loop is known in advance, then it is more convenient to use a loop with a parameter. Its block diagram is shown in Fig. 1.11.

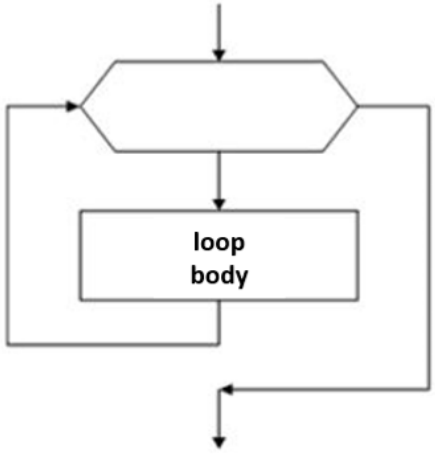


Fig. 1.11. Block diagram of a loop with a parameter

Syntax:

```
for (expression_1; expression_2; expression_3)
{
Loop body;
}
```

Where

`expression_1` : action before the start of the loop, i.e. the initial value is set to the loop parameter (counter);

`expression_2` : logical condition for continuing the loop;

`expression_3` : action at the end of each iteration of the loop, i.e. changing the cycle parameter.

Listing. Converting units of measurement (cycle with parameter)

```
#include <iostream>
using namespace std;
void main()
{
 int sm;
 double d;
 for (sm=1; sm<=10; sm++)
 {
 d=sm/2.54;
 cout << sm <<" santimetrov ravno "<< d <<" duimov \n";
 }
}
```

The above loop can be replaced with the following construction:

```
for ( ; ; )
{
d=sm/2.54;
cout << sm <<" see "<< d <<" inches \n";
 if(sm>=100)
 break;
sm++;
}
```

Listing. Calculate with a given accuracy the sum of the series S=x+x3/3!+x5/5!+…+x2n+1/(2n+1)!+…

```
#include <iostream>
void main()
{
int b,c; double x,y,s,e;
```

```
std::cin >> x >> e;
c=1; b=0;
s = x ; // sum of series
y = x ; // separate term
do {
     b=b+2; c=c+2;
     y=y*x*x/(c*b);
     s=s+y;
}
while (y>e);
std :: cout <<"The sum of the series is: " << s ;
system (" pause ");
}
```

We get the following result on the screen:

```
at x=0.5 e=0.1           result: s =0.520833
at x=0.5 e=0.001         result: s =0.521094
at x=0.5 e=0.00000001    result: s =0.521095
```

Test questions and general tasks:

1. What is a cycle?
2. Draw a block diagram of a loop with a precondition.
3. Draw a block diagram of a loop with a postcondition.
4. What are the differences between loops with a precondition and a postcondition?
5. In what case can a program with a loop (with a precondition) go into a loop?
6. What is the structure of a loop statement with a parameter?
7. How many times will the program fragment be executed: a = 2; b = 6; while (a < b) a ++;
8. For the previous program fragment, write the program equivalent using a loop with a postcondition.
9. General task. Display the values “RUBLE” – “DOLLAR” – “EURO” from 1 to 100.

TOPIC 4. FUNCTIONS AND RECURSIVE ALGORITHMS

Typically, C/C++ programs contain a set of functions, including the main function `main()` , to which control is transferred when the program's executable file is launched. The language also includes libraries of standard functions (`math.h`, `stdio.h`, `iomanip.h`, etc.).

Functions are combined into modules. A module is a separate file in which functions and associated data are grouped. The module solves some kind of data processing problem. Modules are coded and debugged separately from each other (the more independent they are, the easier the debugging process). To use a module, it is enough to know only its interface (the set of available functions and objects), without going into details of the implementation.

The main task can be broken down into a number of simpler subtasks. *Functional decomposition* is a method of program development in which a problem is divided into a number of subtasks, the solutions of which, taken together, provide a solution to the original problem as a whole.

In the process of multi-level decomposition, a hierarchical problem solution tree is created, where each level of the tree is a solution to a more detailed problem than the previous level.

The more general term for functions is subroutine. *A subroutine* is a named part of a program that contains a description of a specific set of actions. Application area:

- performing the same calculations in different places in the program;
- creation of publicly available function libraries.

Subroutines are divided into 2 types: procedures and functions.

A procedure is a named part of a program that, after being described once, can be called repeatedly by name from subsequent parts of the program to perform certain actions (returns a formal void result, indicating the absence of a result).

The parameters listed in the header of the function description are called *formal parameters* , or simply parameters, and those written in the function call operator are called *actual parameters* , or arguments.

Listing. Exchange values of two variables

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

```
    void swaping ( int a, int b ) // formal parameters, passing pa-
rameters “by value”
    {
    int temp;
    temp = a; a = b; b = temp;
    }
    void main()
    {
    int x = 1, y = 2;
      swaping ( x, y ); // actual parameters
    cout << x << " " << y;
    system("pause");
    }
```

Oddly enough, “1 2” will be printed on the screen instead of “2 1”. In this case, the procedure works with copies of the passed parameter values. The procedure creates temporary local variables in memory named *a* and *b* (formal parameters) and copies the passed values of the main program variables *x* and *y into them* . Therefore, operations are performed on their copies, but the values of *x* and *y* have not changed. The subroutine does not have access to the original parameter values, and therefore does not have the ability to change them. This type of parameter passing is called *call by value*.

In order for the replacement to occur, it is necessary to tell the subroutine to work with the same memory cells as the main program – it is necessary to add the address removal operator “&”. In fact, references (aliases) to variables are passed. This type of parameter passing is called *“call by reference”* .

```
    void swap (int &a, int &b)
```

A function is a named part of a program that returns a result value (a number, character, or other type of object). A function can return a value and therefore has a type. The return statement is used to return the value calculated by the function. The return type of a function can be anything except an array and a function (but can be a pointer to an array or a function). If the function does not return any value, then specify the void type, i.e. this is a procedure.

Syntax:

```
<Return type argument>Function_name (formal parameters) {function body}
```

Listing. Finding the smallest value

```
int min ( int a, int b ) // formal parameters
{
if ( a < b ) return a;
else return b;
}

void main()
{
int x, y, z;
cout << " Enter 2 numbers \n";
cin >> x >> y;
z = min ( x, y ); // actual parameters
cout << "Smallest number" << z;
}
```

It is worth noting that when a function is called, memory is allocated on the stack for formal parameters in accordance with their type, and each of them is assigned the value of the corresponding actual parameter (argument).

Another option for passing parameters is called " *call by pointer" (call by value)* . In fact, pointers are passed (for more information about pointers, see Section 3) to variables that store values. In the calling function, the values also change.

Listing. Parameter passing methods

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

int function_val(int x) // value parameter x is passed to the function
{
return ++x;
}
```

```
    int function_ref(int &x) // a reference is passed to the function
that refers to an object from outside
    {
    return ++x;
    }

    int function_ptr(int *x) // a pointer is passed to the function,
which contains the address of an external variable
    {
     return +*x; // in order to work with a pointer as a variable, you
need to dereference it
    }

    // inside the main function the functions are called one by one
    void main()
    {
    int val=10; // source variable

    // Passing a parameter to the function (by value)
      cout<<function_val(val)<<endl; // result : 11
    cout<<val<<endl<<endl; // result : 10

      // Passing a reference parameter to the function
    cout<<function_ref(val)<<endl; // result: 11
      cout<<val<<endl<<endl; // result : 11

      // Passing a pointer parameter to the function
    cout<<function_ptr(&val)<<endl; // result: 12
      cout<<val<<endl<<endl; // result : 12

      system("pause");
    }
```

Local and global variables

Local variables are declared within a subroutine and are visible only within that subroutine. Global variables are declared outside the body of a function (procedure), and therefore their scope extends to the entire program. Below is an example of initializing a global variable:

```
    int g = 0; // initializing a global variable
    int main()
```

```
{
...
}
```

Built-in functions (inline functions)

When a function is called, its arguments are pushed onto the stack, which takes time. You can implement function code substitution at the place of the call. *Built-in functions (inline functions)* are a function that the compiler inserts into the code at each call location. Such function substitutions speed up program execution, but lead to an increase in the size of the source code. Therefore, the scope of application of this technique is optimization of program code for speed.

It is advisable to use *inline functions* if

a) a small function (i.e., insert the code of only small functions);

b) the program code of the function significantly affects performance (as follows from the results of the profiler).

Listing . Using inline functions

```
inline int max(int a, int b)
{
return a > b ? a : b;
}

void main()
{
int x=1, y=2;
printf("The largest of the numbers %d and %d is: %d\n", x, y,
max(x, y));
system("pause");
}
```

The above code is equivalent to the following:

```
// ...
printf("The largest of the numbers %d and %d is: %d\n", x, y,
(x>y ? x : y));
return 0;
}
```

Function prototype

A function prototype is a function declaration (description of its interface) that does not contain the function body, but specifies the name of the function, the number (arity) and types of arguments, and the return data type.

Syntax prototype functions :

```
void function(int arg1, double arg2);
```

Function prototypes are used for the following purposes:

1. Function prototypes are used by the compiler to eliminate errors when calling it. To do this, the compiler analyzes the arguments (number and types) passed to the function in the call, and the subsequent use of the return value.

2. Development of library interfaces (by placing function prototypes in a header file, you can describe the interface for libraries).

3. Class declarations (function prototypes are used to describe class methods, in particular when separating the interface and implementation).

Below is an example of the behavior of an implicitly declared function `f()` .

```
#include <stdio.h>

int f(int n); /* Function prototype */

int main() /* Function call */
{
printf("%d\n", f());

return 0;
}
```

When implementing the prototype, the compiler will generate an error message: “there are not enough arguments when calling the function.”

```
int f ( int n ) /* Function to call */
{
if (n == 0)
return 1;
else
return n * f(n – 1);
}
```

Recursion

Recursion is a technique that uses a subroutine (procedure or function) that calls itself directly or through other subroutines. This technique allows you to replace cyclic algorithms.

For example, consider the traditional problem of calculating the factorial of a number (N! = 1*2*3*…* N). The function with the iterative solution algorithm is presented below:

```
int Factorial (int N)
{
int F;
F = 1;
for(i = 2; i<=N; i++)
F = F * i;
return F;
}
```

To solve a problem using the “recursion” technique, it is necessary to construct an appropriate recurrent formula. In the case of the factorial problem, it is given below. The first part of the formula is the condition for stopping the recursion (base case), the second part of the formula is the transition to the next step.

$$N! = \begin{cases} 1, & N = 1 \\ N \cdot (N-1)!, & N > 1 \end{cases}$$

Let's consider the corresponding recursive algorithm in the programming language and the protocol for calling `Factorial (3)`:

```
int Factorial ( int N )
{
  int F ;
  printf("input N= %d \n ", N);
    if ( N <= 1 )
   F = 1;
    else F = N * Factorial (N – 1);
  printf("output N= %d \n ", N);
  return F;
}
```

Print:

```
input N= 3
  input N= 2
    input N= 1
    output N= 1
  output N= 2
output N= 3
```

As can be seen from the operation protocol, the call to Factorial (2) occurs before the call to Factorial (3) ends. That is, the state of local variables and the return address after Factorial (2) completes must be remembered somewhere. For this purpose, a special area of RAM is used – the stack (English stack – "stack"), the work of which is organized according to the LIFO principle (English Last In – First Out , "last in, first out"). So, the stack is a memory area in which local variables and their return addresses are stored (for this, the stack pointer (SP register) is used, which stores the address of the last occupied stack cell). When a subroutine is called, parameters, a return address are placed on the stack, and space is allocated for local variables, and when returning from a subroutine, the state of the stack changes in the opposite direction.

Disadvantages of the "recursion" technique: a) possible stack overflow due to recursive calls and many local variables; b) slowdown due to the performance of service operations on the stack.

Consider the problem: given a recursive algorithm, find the sum of numbers that will be output when calling F(1).

```
void  F ( int n )
{
  printf ("% d \ n ", n );
  if ( n < 3){
    F ( n +1); // 1st RV
    F ( n +2); // 2nd RV
}
}
```

Since for n < 3 two recursive calls (RC) are performed , it is more visual to solve such a problem graphically in the form of a binary tree (the nodes contain the values of the input parameters `F ( n )` when calling a subroutine).

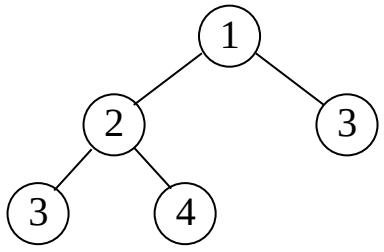


Fig. 1.12. Graphical solution in the form of a binary tree

Let's try to understand the algorithm:

a) during the initial call F(1), “1” is printed, the condition is checked (true!) and entry into the 1st PB (i.e. in F(n + 1)), where the sent parameter n = 1 + 1 = 2;

b) when implementing the 1st RT call (i.e. F(2)), “2” is printed, the condition is checked (true!), and the entry is again into the 1st RT, where the sent parameter is already n = 2 + 1 = 3;

c) when F(3) is called, “3” is printed, the condition is checked (false!), and then there is a return (since after the condition there are no more operators) from the current RT to the place of the call with parameter n = 2. And immediately enters 2nd PB (i.e. F(n + 2)): 2 + 2 = 4). Similarly, the return from the 1st RT and the entrance to the 2nd RT are again implemented: (1 + 2) = 3.

On the screen we will see the following numbers: “1”, “2”, “3”, “4”, “3”, respectively, their sum is 13.

To make the algorithm work more clearly, let’s add another output operator after the condition. Detailed version:

```
#include <iostream>
using namespace std ;
void F(int n)
{
cout << " input n=" << n << "\n";
if (n < 3){
F(n + 1);
F(n + 2);
 }
cout << " out put n=" << n << "\n";
}

int main(void){
 F(1);
 system("pause");
}
```

The data output will be as follows (data related to the previous task is highlighted in gray):

```
input n=1
 input n=2
  input n=3
```

```
 output n=3
  input n=4
  output n=4
 output n=2
  input n=3
  output n=3
output n=1
```

Test questions and general tasks:

1. Define the concepts procedure and function. What is the difference between procedures and functions?
2. Which parameters are called formal and which actual?
3. Name the methods for passing parameters to subroutines.
4. What variables are called local?
5. What are recursive routines?
6. What is a stack and what role does it play in program execution?
7. General task. Develop an application that implements a procedure for finding the maximum value from two numbers and a function for finding the minimum value from two numbers.
8. General task. A positive integer is entered (no more than 255_{10}, for example, 89_{10}) and converted into binary code (format the program code into a procedure), the sum of its digits is calculated (format the program code into a function).
9. General task. Write a function that "reverses" a number, i.e. returns a number with the digits in reverse order. Example: enter a natural number, 1234, after the revolution: 4321.
10. General task. Write a function that calculates the greatest common divisor (GCD) and least common multiple (LCD) of two natural numbers. Example: input numbers 10 and 15, output GCD(10,15)=5, LCM(10,15)=30.

SECTION 2. STATIC DATA STRUCTURES

TOPIC 5. ARRAYS AND ALGORITHMS FOR THEIR PROCESSING

In some cases, it is inconvenient to store a lot of the same type of data related to a specific subject area in ordinary variables. Therefore, they are combined into a group with a common name.

An array (for C / C ++) is a group of variables of the same type, located nearby in memory (in adjacent cells) and having a common name.

An array element has three characteristics (Fig. 2.1):

1) name (for example, `M`);

2) index – an integer that uniquely determines the location of the element in the array. A variable or an arithmetic expression of an integer type (for example, 0, 1, i , i +1, etc.) can also be used as an index ;

3) value – the actual value of the element (for example, `M[0]`).

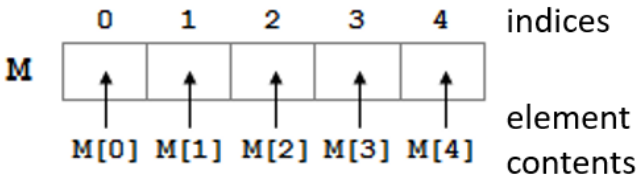


Fig. 2.1. Numbering of array elements

Array declaration syntax: `Type <name>[size]` , for example, `int M[5]`. Array elements are numbered starting from zero: `M[0], M[1], M[2], M[3], M[4]`

It is more rational to declare static arrays using a constant:

```
const int N = 5;
int M[N];
```

The above array declaration has the following purposes:

- appoint boolean name;
- determine the data type of elements;
- set the number of elements;
- allocate space in memory.

In C / C ++ it is important to define the initial values (initialization) of array elements. If they are not specified, then the cells will contain “garbage” (foreign data in memory from previous programs).

Array initialization syntax:

```
Type <name>[size] = {list of values}
```

Example:

```
int M[5] = { 7, 3, -14, 9, 4 };
float F[3] = { 1.0, 2.0, 7.1 };
```

Processing array elements:

```
int i = 0;
while(i<N)
{
// processing M[i]
i++;
}
```

```
for(int i = 0; i < N; i++ )
{
// processing M[i]
}
```

Let's look at typical array processing algorithms that are often found in program development practice.

Listing. Entering data into array elements from the keyboard

```
for (int i = 0; i < N; i++ )
{
cout << "M[" << i << "]=";
cin >> M[i];
}
```

Listing. Printing the contents of array elements to the console

```
cout << "Array M:\n";
for (int i = 0; i < N; i++ )
cout << M[i] << " ";
```

Listing. Analysis of array elements (calculate the number and sum of array elements M, the values of which are from 25 to 40)

```
int count = 0, sum ma = 0;
for (int i = 0; i < N; i++ )
if (M[i] > 25 && M[i] < 40 ) {
count++;
sum ma += M[i];
}
```

Listing . Search maximum element array And his index

```
MAX = M[0]; nMAX = 0;
for (int i = 1; i < N; i++ )
if ( M[i] > MAX ) {
MAX = M[i];
nMAX = i;
}
cout << "M[" << nM AX << "]=" << M AX ;
```

Listing. Selection of elements by condition into a new array

```
 int new_M[N];
 for(int i = 0; i < N; i++) new_M[i]=0;
 int count = 0; // internal counter of the number of selected el-
ements
 for(int i = 0; i < N; i++)
 if (M[i]>0) {
           new_M[count]=M[i];
           count++;
 }
      cout << " New array : ";
 for (int i = 0; i < count; i++) cout<< new_M[i] <<" ";
```

Passing an array to subroutines

When passing an array to a subroutine, the address of the first element of the array is actually sent (that is, passing by reference, by a pointer to the zero element).

```
int SUM(int M[], ... ) // or int *M
{
...
}
```

Example: subroutine (function) to find the sum of array elements.

```
int SUM ( int M[], int N )
{
int sum = 0;
for (int i = 0; i < N; i++ )
 sum += M[i];
return sum;
}
```

This is what a function call looks like:

```
void main()
{
 int M[5], summa;
 ...
 summa = SUM(M, 5);
 ...
}
```

Array sorting algorithms

Sorting is the process of rearranging the elements of a given set in a certain order, usually in order to facilitate the subsequent search for elements. In fact, this is a transformation of the original array into an array containing the same elements, but in descending (or ascending) order of values.

Sorting algorithms can be divided into two categories:

- simple, but ineffective in terms of speed for large amounts of data (“bubble method”, “selection method”, “insertion method”, etc.);
- complex, but speed-effective for large data arrays (“quick sort”, heap sort, etc.).

Bubble sorting (exchanges).

Let's consider the classic algorithm “ *bubble method (sorting by exchanges)* ”. The name of the method comes from the physical phenomenon of an air bubble floating in a vessel with water from the bottom to the top: the smallest (“lightest”) element moves upward (“floats”).

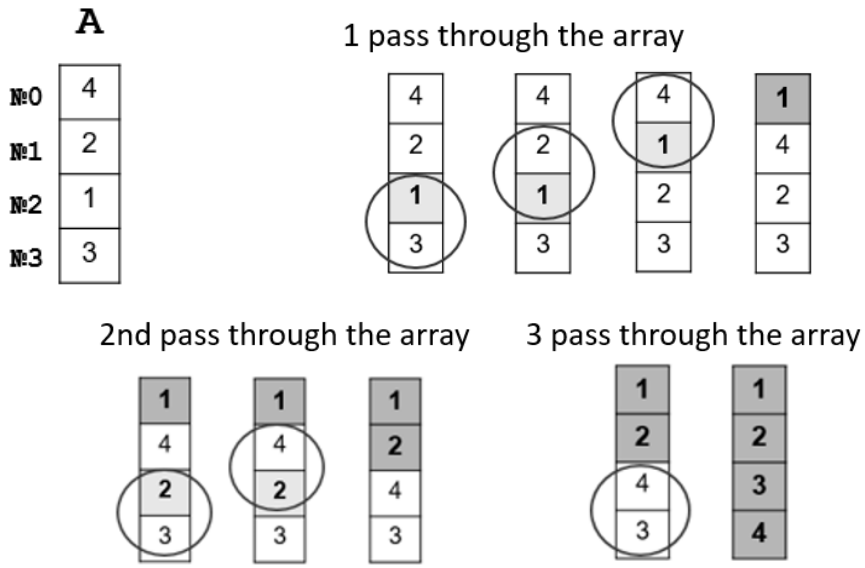


Fig. 2.2. Bubble sort (exchanges)

Iterating over elements starts from the end (bottom). In the inner loop, two adjacent elements are compared, and if they are placed incorrectly (the current one is greater than the lower one `A[j] > A[j+1]`), then we swap them. In one pass of the inner loop through the array, one element (the smallest) is placed in the "right" place. At each subsequent pass of the inner loop, the number of processed elements decreases by one (the boundary `i` in the outer loop for already sorted elements). In the last iteration, the subarray of one element does not need to be sorted, so to sort N elements, no more than N-1 iterations of the outer loop are required.

In fact, to sort an array of N elements, you need N-1 passes (the outer loop specifies the number of passes through the array). In our example, for N = 4, 3 passes are needed.

```
     for (int i=0; i<N-1; i++ ) // number of passes
     for (int j=N-2; j>=i; j-- ) // inner loop from the end of the
array
     if ( A[j] > A[j+1] ) {
             temp = A[j];
             A[j] = A[j+1];
             A[j+1] = temp;
     }
```

The complexity of the algorithm in this case is $O(n^2)$, since there are two nested loops.

Sorting by selection (highlighting)

Selection sort involves selecting the smallest element in the array and moving it to the top, i.e. selecting the minimal element and placing it at the top (switching places with A[0]), from the remaining ones we again look for the minimal element and place it at a position lower relative to the top (switching places with A [1]), etc. In the figure, the underlined element is the smallest of the remaining ones.

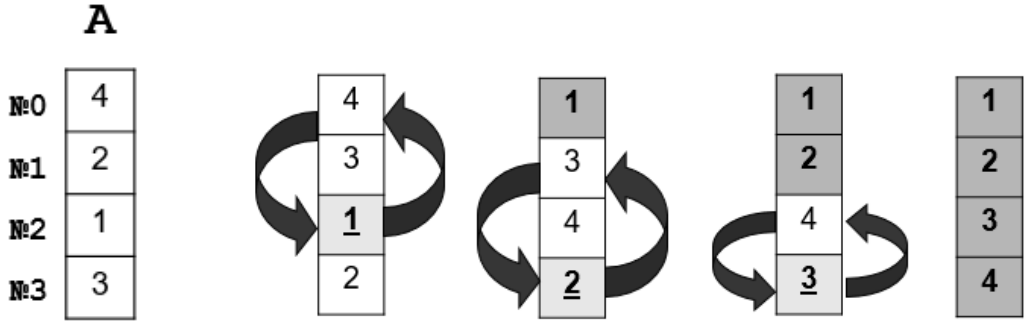


Fig. 2.3. Sorting by selection

```
     for( i = 0; i < N-1; i++ ) {
     nMin = i ;
     for ( j = i+1; j < N; j++)
     if( A[j] < A[nMin] ) nMin = j; // if the current element is less
than the minimum, then remember its number
     if( nMin != i ) { // if a new minimum is found
     c = A[i]; // swap the i -th and the found new minimum with num-
ber nMin
     A[i] = A[nMin];
     A[nMin] = c;
}
     }
```

In each iteration of the inner loop, we look for the number of the minimum element from the remaining ones and put it in the *i* th place. For example, in the first iteration we set the topmost element as the minimum (`nMin = i`) and in the loop from the next element we look for the one that will be smaller than it and remember its number (`nMin = j`). And then, if the new minimum has changed its index (`nMin != i`), i.e. does not remain equal to *i* , then we swap *the i* -th and the found new minimum with number `nMin` .

So, the outer loop shifts the border of the already found minimal elements, in which the search goes to the penultimate element (up to N-1), since the last one will already be in its place. In the inner loop, the search for the minimum of the remaining elements is

carried out and it starts with `j=i+1` ,since the i-th place already has the minimum one and we should not touch it.

There is an easier to understand option without checking the index change at the minimum value:

```
int min, n_min, j;
for (int i = 0; i < N – 1; i++)
{
min = A[i]; n_min = i;
for (j = i + 1; j < N; j++)
if (A[j] < min)
{
min = A[j];
n_min = j;
}
A[n_min] = A[i]; // exchange via min
A[i] = min;
}
```

The complexity of the algorithm in this case is $O(n^2)$, since there are two nested loops.

Insertion (inclusion) sort.

Insertion sort involves conditionally dividing the elements of an array into two groups:

- “ready sequence” (where the leftmost element is initially placed and then sorted elements will be stored here);
- "original sequence" (remaining unsorted elements on the right).

At each step, the i-th element (circled) is inserted into the “appropriate place” in the finished sequence (on the left).

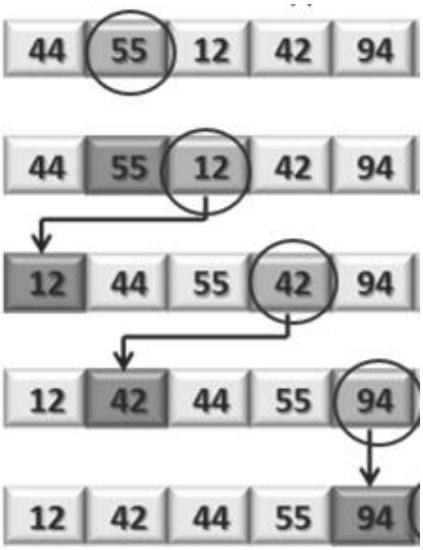


Fig. 2.4. Insertion sort

Let's take a closer look at this algorithm.

Let's look at the iterations of the outer loop. In the first iteration, we remember the initial element of the original sequence (`element = A[ 1 ]; )` and begin to compare it with all the elements of the finished part (in this case 44 < 55, so we do not make any rearrangements).

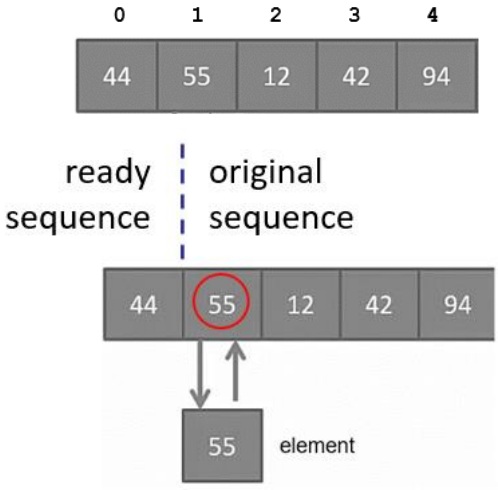


Fig. 2.5. Insertion sort. First iteration of outer loop

We go to the second iteration of the outer loop, where the finished part will already have two elements. And we do the same thing: we remember the next element of the original sequence (`element = A[ 2 ]; )` and begin to compare it with all the elements of the finished sequence (we compare `element =12` with the values 44 and 55) and shift

large elements of the finished part to the right . After this, the element in question `is element` place it in the vacant space.

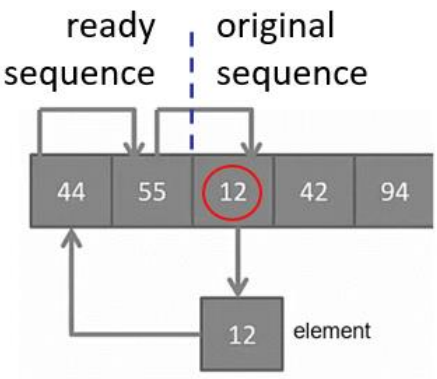


Fig. 2.6. Insertion sort. Second iteration of the outer loop

We go to the third iteration of the outer loop, where the finished part will already have three elements. Similarly, we compare `element = 42` with the values 12, 44 and 55.

- If 55 is greater than 42 (true), then shift 55 to the right.
- If 44 is greater than 42 (true), then shift 44 to the right.
- If 12 is greater than 42 (false), ..., the element in question is placed in the free space

`element` are sent one position to the right, i.e. the selected element is compared with the next element of the sorted part, starting from `j=i-1` (from the adjacent element to the left). If the selected element is greater than `A[i]` , then it is included in the sorted part, otherwise `A[j]` is shifted by one position, and the selected element is compared with the next element of the sorted sequence. The process of finding a suitable location ends under two different conditions:

- if element `A[j] > A[i] is found` ;
- and the left end of the finished sequence is reached `j >= 0` (this condition is necessary so as not to go beyond the left boundary of the array).

Moreover, the condition `j > 0` should be in 1st place so as not to check the second condition when `j` becomes `< 0`

```
int j, element;
// outer loop dividing the array into finished and initial parts
for (int i = 1; i < N ; i++)
```

```
 {
     element = A [i]; //remember the element that we will insert
     j = i – 1; // start searching from the adjacent element on the
left
while ( j >= 0 && element < A [j] ) // search for a suitable location
{
            A [ j +1] = A [ j ]; // shift values to the right
j--; // shift the view to the beginning of the array ( left )
}
     A [j + 1] = element ; // insert the element in question into the
right place
}
```

After the inner loop ended, we stopped at the position after which we need to insert the element (i.e., we checked the element, it is larger and, accordingly, the element in question needs to be placed in front of it).

Consider the library function `qsort()` , which implements the "*quicksort method*" `qsort()` function implements a "quick sort" algorithm for an array of N elements, each of W bytes in size. The `base` argument is a pointer to the beginning of the array to be sorted. `qsort()` function replaces the array with sorted elements.

Syntax:

```
void qsort(
void *base
size_t N,
size_tW,
int (__cdecl *compare)(const void *, const void *)
);
```

Last parameter `compare()` – this is a pointer to a custom routine for comparing two array elements. The parameters `compare ( ( void *) & elem 1, ( void *) & elem 2 )` are pointers to the two array elements being compared. The routine returns a value indicating their relationship:

- result < 0, i.e. elem1 is less than elem2;
- result = 0, i.e. elem1 is equivalent to elem2;
- result > 0, i.e. elem1 is greater than elem2.

Listing. Sorting an array using the qsort() function

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

int comp(const void* i, const void* j)
{return (*(int*)i - *(int*)j);}

void main()
{     int M[20];
 for(int i = 0; i < 20; i++) {
      M[i] = rand()%100;
      cout << M[i] << ' ';
 }
 cout << endl << endl;
 qsort(M, 20, sizeof(int), comp); // call a library function
 for(int i = 0; i < 20; i++) cout << M[i] << ' ';
 cout << endl;
 system("pause");
}
```

The array is sorted in ascending order as determined by the comparison function. To sort an array in descending order, you need to swap the parameters in the comparison function:

```
return (*(int*)j - *(int*)i);
```

Algorithms search

In practice, linear and binary search algorithms are used.

Linear search implements iterating over all elements of an array. It is possible to exit early (`break` in the body of the loop) if the element is found and it is the only one (or the first).

Listing. Linear search

```
int X = 5; // searched element = 5
nX = -1; // haven't found it yet...
for ( i = 0; i < N; i++) // loop through all elements
 if ( A [ i ] == X ) { // if found, then ...
         nX = i; // ... remember the number
break; // exit from cycle
}
```

```
if (nX < 0) // output "Not found..."
else  // output "A[...]=...", nX, X;
```

Binary search is applicable when there is preliminary sorting of data.

Listing. Binary search (option 1)

```
int X, L, R, c;
L = 0; R = N; // initial segment
while ( L < R-1 )
{
c = (L+R) / 2; // found the middle
if ( X < A[c] ) // segment compression
       R = c;
else L = c;
}
if ( A[L] == X )
 // output "A[...]=...", L, X );
else // output "Not found!" ;
```

Listing. Binary search (option 2)

```
nX = -1;
L = 0; R = N-1; // boundaries: search from A[0] to A[N-1]
 while ( R >= L ){
c = (R + L) / 2;
if (X = = A[c]) {
nX = c;
break;
}
if (X < A[c]) R = c – 1;
if (X > A[c]) L = c + 1;
   }
if (nX < 0) // output "Not found...";
 else // output "A[...]=...", nX, X;
```

C ++ Standard Library has a `bsearch() function` that implements *"binary search on a sorted array"*.

```
void *bsearch(
```

```
    const void *key, // pointer to the desired object
    const void *base, // pointer to the beginning of the array
    size_t N, // number of elements
    size_t W , // size of each element in bytes
    int *compare (const void *key, const void *datum)
);
```

Pointer `int *compare (const void *key, const void *datum)` means the address of a routine that compares two elements of an array and returns a value that indicates their relationship. The first parameter is a pointer to the key to search, the second is a pointer to the array element that will be compared with the key. Note that `the bsearch() function` returns a pointer to the `key number` in the array, which is indicated by base. If key is not found, the function returns NULL.

Test questions and general tasks:

1. What is an array? Scope of their application in programs?
2. How are array elements accessed?
3. Why might it be dangerous to cross an array boundary?
4. Describe the bubble sort algorithm.
5. General task. Develop an application in which to describe a one-dimensional static array and implement (in one program):

a) data entry (manually || random || programmatically specified values of array elements);

b) displaying the values of array elements;

c) search for maximum and minimum elements and their numbers;

d) searching for the sum and average value in an array;

e) array sorting;

f) rewriting the elements of the sorted array into a new array whose values are greater than the average value, as well as displaying it on the screen;

g) reverse the elements of the original array;

h) cyclically shift the original array by 1 element to the left.

TOPIC 6. MULTIDIMENSIONAL ARRAYS

C /C++ standard only defines one-dimensional arrays. Arrays of larger dimensions are treated as “arrays of arrays.” So, for example, a multidimensional array is an array of arrays, where each element has two indices (row number and column number), and the numbering of rows and columns starts from zero (Fig. 2.7).

Initialization of a two-dimensional array `M [3][4]` (3 rows, 4 columns):

```
int M [3][4] = {
{0, 3, 4, 6}, // values of zero row elements
{-1, -5, 0, 1}, // values of the elements of the first row
{8, 3, 5, 10} // values of the second row elements
};
```

M

	0	1	2	3
0	0	3	4	6
1	-1	-5	0	1
2	8	3	5	10

Fig. 2.7 . Two-dimensional array 3*4

When placed in memory, space will be allocated to record 3 × 4 = 12 elements, and they will be placed linearly in order. Therefore, it is possible to sequentially enumerate the matrix elements:

```
int M [3][4] = {0, 3, 4, 6, -1, -5, 0, 1, 8, 3, 5, 10};
```

When passing a static matrix to a function, the entire structure, as well as its dimensions, is used as a parameter. But the compiler needs to be told the mechanism for splitting this static structure into lines. Therefore, the number of rows of the matrix need not be specified, but the number of data in each row must be specified as a constant expression in square brackets. Prototype of a function that receives a matrix as an input parameter (`COLUMNS` is a constant):

```
void function(int M[][COLUMNS],int n,int m);
```

Listing. Processing a two-dimensional DNG array

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

// function for searching for the largest matrix element and its
indices
int MaxElem(int a[][4], int n, int m, int &imax, int &jmax)
{
 int i, j, max = a[0][0];
 imax = jmax = 0;
 for(i = 0; i < n; i++)
      for(j = 0; j < m; j++)
           if(a[i][j] > max)
           {
                max = a[i][j];
                imax = i;
                jmax = j;
           }
           return max;
}

void main()
{
 const int ROWS = 3, COLUMNS = 4; // constants
 int matrix[ROWS][COLUMNS]; // announcement
 int imax =0, jmax =0; // numbers of maximum indexes

 for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
           matrix[i][j] = rand()%10;
           cout << matrix[i][j] << " ";
      }
      cout << endl;
 }
 cout << "max = " << MaxElem(matrix, ROWS, COLUMNS, imax, jmax) <<
endl;
 cout << "imax = " << imax << endl;
 cout << "jmax = " << jmax << endl;
 system("pause");
}
```

Listing. Rearranging matrix rows (rows numbered 1 and 2)

```
for (int j = 0; j < COLUMNS; j++)
  {
c = d[1][j];
d[1][j]= d[2][j];
d[2][j]= c;
}
```

Test questions and general tasks:

1. Define the concept of "two-dimensional array".
2. Where can two-dimensional data structures be used?
3. General task. Develop an application in which to describe a two-dimensional array and implement (in one program):

a) data entry (manually || random || programmatically specified values of array elements);

b) displaying array values on the screen (in the form of a rectangular matrix);

c) search for the maximum element indicating its location;

d) outputting sums of array rows (rewriting into a new one-dimensional array);

e) sorting all rows of the array;

f) Finding the line number with the most zero elements.

TOPIC 7. PROCESSING CHARACTERS AND STRINGS

A character type (*char*) is a type that contains ASCII table values (codes) in its internal representation. An integer value is visually represented as a character surrounded by single quotes (for example, `'A'`). For example,

```
char c ='A'; // character with code No. 65 of the ASCII table
```

There is no base type for working with a set of characters (strings), instead an array of characters is used.

A character string (in array form) is an array of characters (one-byte data of an integer type) that are located side by side in memory. A string, like any objects and variables, is represented in memory as a set of bytes. Therefore, there are two implementation options for determining the end of a line:

1) storing the length of the string in a separate cell (Pascal language);

2) highlighting the special character “end of line” (C / C ++ language).

Thus, a character string (in C / C ++) is a sequence of characters that ends with the special character `'\0'` ("backslash", "null character", "terminator character", etc.). Since a string differs from a static array in that it ends with a line terminator, it is necessary to allocate 1 byte more memory (i.e. the total number of characters in the string is determined taking into account '\0' , and is equal to "string length + 1 ") .

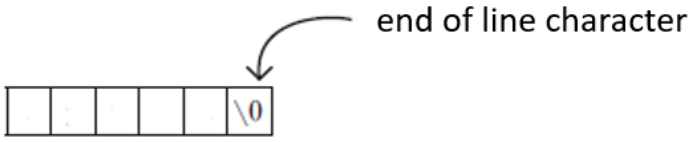


Fig. 2.8. The string ends with a null character

In this case, a character one-dimensional array receives the properties of a string, which can be used as a parameter for library functions, including when printing the string to the screen using the `printf() function` or the `cout object`.

In fact, a string is a static character array, so it is declared like this:

```
char s[10];
```

In the above example, the length of the string should be no more than 9 characters, since using a static array of characters as a string allocates 1 byte more memory (to store `'\0'`). The initial value of a string can be specified when declared in double quotes:

```
char s[10] = "Hello!";
```

The characters in double quotes will be written to the beginning of the character array *s* , followed by the line ending `'\0'` . When using library functions for calculating string length, the `'\0'` character is not taken into account. The remaining characters do not change, and there will be “garbage” in local strings.

As an example, let's manually replace the contents of each element of the character array (with the word " Hello "):

```
s[0] = 'H';
s[1] = 'e';
s[2] = 'l';
s[3] = 'l';
s[4] = 'o';
s[5] = '\0';
```

The last line is a manual offset of the null character. Otherwise, the `'t' character` from the previous contents of the line (`"Prive t !"`) will be printed.

also possible to set element values in operator brackets:

```
char s[10] = {'P','r','i','v', 'e','t','!','\0'};
```

Another way to initialize a character array:

```
char s[] = "Hello!";
```

In this case, the compiler counts the number of characters in quotes, allocates 1 byte more memory and writes the string and the line ending `'\0'` into this memory.

There is also a way to declare a character array by allocating memory to a pointer. In this case the string is dynamic. During initialization, memory is allocated for writing the string values. In fact, the result will be the same as in the previous options.

```
    char *s;
    s = new char[10];
    s = "0123456789101112"; // in this case, going oversize will not
be an error
```

Another initialization option:

```
    char * s = "0123456789101112";
```

If the string will not change, then it is declared as a constant:

```
    const char s [] = "Hello!";
```

Library functions in C

For input/output lines (up to the first space), standard library ones are used (module `<stdio.h>`) `scanf(), printf()` functions with the `"%c" format` (for one character) and the `"%s" format` (for strings). For example , `scanf ("%s", s )` and `printf ("%s", s )` . Please note that in `scanf()` you do not need to put & in front of the name of the input string (records `S` and `&s[0]` are identical in this case).

Input/output of individual characters is implemented by special functions `getchar()` and `putchar()` . The `getchar()` function reads one character per access from the input stream. The `putchar()` function prints a character to the standard stream. So, the example below implements the input and output of one character:

```
    char ch = getchar();
    putchar(ch);
```

To input/output arbitrary strings (texts with spaces), use `the gets()` and `puts() functions. The gets()` function receives a sequence of characters from the input stream before the Enter key is pressed. `The puts()` function puts a string on the standard output. So, the example below implements input and output of strings:

```
    char ch [10];
    gets ( ch ); // get from stream
    puts ( ch ); // put into stream
```

Typical functions for working with strings are connected using the `string.h module` of the standard C library.

- `strlen(str1)` – the function returns the length of the string, the '\0' character is not taken into account;
- `strcmp(str1, str2)` – the function performs a string comparison (the difference between the codes of the first two different characters is calculated);
- `strcpy(str1, str2)` – the function copies str2 to str1 (the old value is completely erased);
- `strncpy(str1, str2, n)` – function copies n characters from str2 to str1;
- `strcat(str1, str2)` – the function connects str1 and str2, and the result is placed in str1;
- `strchr(str, ch)` – the function searches for the first specified character from the beginning of the string;
- `strrchr(str, ch)` – the function searches for the last specified character in the string;
- and etc.

Library functions in C ++

The C ++ string data type is also not a base type and is a container class. To use string types, you must first include the corresponding header file:

```
#include <string>
```

Next, in the `main() function` we initialize the object:

```
string s = "Hello, World!";
```

Just as in a character array, access to the elements of a line is implemented by indicating its number (characters in a line are numbered starting from zero):

```
s[4] = 'e';
```

The peculiarity of entering strings using the `cin object` (for example, `cin >> s)` is that the input is implemented before a space. To enter lines with spaces, use the `getline() function` , for example:

```
getline(cin, s);
```

Screen output is implemented in the standard way (regardless of spaces):

```
cout << s;
```

Let's look at operations with strings, including methods of the `string class (C++)`:

1. Combining a string (concatenation):

```
string s, s1, s2;
s1 = "Hello";
s2 = "World";
s = s1 + ", " + s2 + "!";
```

2. Determining the length of a string – `size() method`:

```
int n = s.size(); // returns an integer
```

3. Selecting a substring – the `substr() method` . The first parameter is the position number, the second parameter is the number of characters. If we send only one parameter, then the substring from the specified number to the end of the line is selected.

```
s = "Hello World!";
s 1 = s.substr (8, 3); // "World"
s2 = s.substr (8); // "World!"
```

4. Removing a substring – `erase () method.` The first parameter is the position number, the second parameter is the number of characters. If we send only one parameter, then part of the line from the specified number to the end of the line is deleted.

```
string s = "Hello World!";
s.erase (6, 5); // the line remains "Hello!"
```

```
s.erase (6); // the line remains "Hello"
```

5. Inserting a substring – `insert () method.` The first parameter is the position number, the second parameter is the line to be inserted.

```
string s = "Hello World!";
s.insert ( 6,"I love you" ); // in the line: "Greetings, World"
```

6. Search for characters (substrings) in a string – `find () method.` Parameter – search string.

```
string s = "Hello World!";
int p = s.find(' e '); // 4,
```

You can also search for a sequence of characters. If the characters are not found, the method returns the value “-1”.

7. Convert from string to number – function `atoi()` (short for ASCII to integer):

```
string s = "1234";
int n = atoi ( s . c _ str () ); // conversion to C language string
```

Starting from version C++11, the `stoi() function is used` (short for “ string to integer ”):

```
int n = stoi ( s );
```

8. Conversion from number to string – `to_string() function` (since C++11):

```
int n = 1234;
string s = to_string(n);
```

It is also possible to perform number to string conversion using string streams (`# include < sstream >`).

```
int n = 1234;
string s;
```

```
ostringstream ss;
ss << n;
s = ss.str(); // s = "1234"
);
```

There are a variety of examples of string processing algorithms. For example, in the form of a visual application, the user enters his first name, middle name, last name and date of birth. It is necessary to split the lines and convert them to the form "last name, initials" and an abbreviated form of recording the date of birth, for example, for storage in a database. The general idea of implementation: using the `find (), substr (),` etc. methods, search for spaces, highlight first name, patronymic, last name, date. Then cut off the extra characters and concatenate the corresponding elements.

Comparing and sorting strings

Strings are compared character by character, based on their location in the encoding table. To determine the character code, use the `getchar ()` function of the `<conio.h>`. This function reads a character from the keyboard and gets its code (integer data type).

Listing. Getting the code of the entered character

```
#include <stdio.h>
#include <conio.h>
#include <cstdlib>
void main()
{
char char1 = getchar();
printf("%d", char1);
system (" pause ");
}
```

The string comparison mechanism is used when sorting strings.

Listing. Entering strings and displaying them in alphabetical order

```
#include <Windows.h>
```

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
 SetConsoleCP(1251);
 SetConsoleOutputCP(1251);
 const int N = 10;
 string temp, s[N];
 int i, j;
 cout << "Enter data:" << endl;
 for ( i = 0; i < N; i++ ) getline ( cin, s[i] );

 for ( i = 0; i < N-1; i++ ) // classic "bubble" method
      for ( j = N-2; j >= i; j -- )
           if ( s[j] > s[j+1] ) // in the first pass we compare
lines 8 and 9, etc.
           {
                temp = s[j];
                s[j] = s[j+1];
                s[j+1] = temp;
           }

           cout << "Output data after sorting:" << endl;
           for ( i = 0; i < N; i++ ) cout << s[i] << endl;
           system (" pause ");
}
```

Test questions and general tasks:

1. How are symbols stored and processed in a computer? What is an ASCII table?
2. What is a character string?
3. In what ways can you declare and initialize a string? Name the end of line marker.
4. List the main operations (standard functions and procedures) for working with strings.
5. General task. A text string in the format first name, patronymic, last name, date of birth is given (for example, “Ivan Petrovich Sidorov April 12, 1961”). Convert the string to a new format with last name and initials (for example, “Sidorov I.P.”). The date must be presented separately in string format (for example, “1961-04-12”) for future storage in the database. The algorithm must be invariant to the length of the source string.

TOPIC 8. CUSTOM DATA TYPES (STRUCTURES, ENUMERATIONS, UNIONS)

Arrays store data of the same type. But in some cases it is necessary to combine different types of data into one logical block. Of course, you can create many arrays, each of which will describe its own data type, but in this case it will be inconvenient to process (sort, search, etc.) the data. Therefore, for the task of combining heterogeneous types into one block, many programming languages have a specialized construction that allows you to combine arbitrary data of various types into a single whole.

A structure is an aggregated user data type that can include several fields – elements of different types (including other structures). The data included in the structure have their own names, and it is possible to perform any operations with them that are permitted by their type.

Structure declaration:

Option 1:

```
typedef struct
    {
    type <field name 1>;
    type <field name 2>;
    ...
} <structure name>;
```

Option 2:

```
struct < structure name >
{
type <field name 1>;
type <field name 2>;
};
```

Accessing structure fields:

1. The direct access operator (`.`) allows you to access the elements of the structure:

```
StructuralVariableName.NameFields
```

2. The indirection operator (`->`) is used to access elements of a structure that is stored in dynamic memory (for more details, see Section 3) and is referenced by a pointer addressing an object.

```
StructuralVariableName->FieldName
```

Listing. Organizing data entry into the "Employee" structure, sorting by full name, searching for the maximum value in the salary field and calculating its amount

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

typedef struct
{
 int number;
 string fio;
 string section;
 float salary;
} TSotrudnik;

void main()
{
 const int N = 3;
 TSotrudnik S[N], s_temp; // creating objects
 int i,j;
 cout << "Vvedite nomer, fio, otdel, zarplata : \n";
 for ( i = 0; i < N; i++ ) {
      cin >> S[i].number;
      getline(cin, S[i].fio);
      cin >> S[i].section;
      cin >> S[i].salary;
 }

 //sorting
 for ( i = 0; i < N - 1; i++ )
      for ( j = N - 2; j >= i; j-- )
           if ( S[j].fio > S[j+1].fio) // sort by key "fio"
           {
                s_temp = S[j];
                S[j] = S[j+1];
                S[j+1] = s_temp;
           }
           // print sorted structures
           for ( i = 0; i < N; i++ )
```

```
                    cout << S[i].fio << " " << S[i].salary << " RUR.
"
                    <<endl;
               // calculate the amount
               float sum = 0;
               for ( i = 0; i < N; i++ )
                    sum = sum + S[i].salary;
               cout << "summa = "<< sum << endl;

               // search for maximum salary
               float max = S[0].salary; int imax=0;
               for ( i = 1; i < N; i++ )
                    if (S[i].salary > max)
                    {
                         max = S[i].salary;
                         imax = i;
                    }
                    cout << "MAX zarplata = " << max << " y sotrudnika
s nomerom "<< imax<< endl;

                    system("pause");
    }
```

In the example discussed above, structures are sorted, and they are moved in memory. If there are quite a lot of array elements (10^3 or more), this may take quite some time. It would be more expedient to rearrange pointers to these elements and use the indirect access operator (`->`) to the fields of structures.

Another option for creating custom data types is to use *enumerations* . This is a user-defined type consisting of a set of integer constants (enumerators).

Listing. Announcement of "transfer"

```
#include <iostream>
using namespace std;
enum months {
 january = 1, february,
 March, April, May,
 June, July, August,
 september, october, november,
 December
```

```
};

void main()
{ setlocale(LC_ALL, "rus");
cout << "January – " << january << "th month"<< "\n";
cout << "February – " << february << "th month"<< "\n";
cout << "March – " << March << "th month"<< "\n";
system (" pause ");
}
```

You can also assign arbitrary element numbers to each of the values

The next user-defined type is *union* , in which all members share the same memory area. This means that at any given time a union cannot contain more than one object from its member list. Unions are used to save memory when there are many objects and/or limited memory.

Syntax :

```
union [type name] { list of union variables };
```

When a union is declared, the compiler creates a variable of sufficient size to hold the largest variable present in the union.

Listing. Announcement of "unification"

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

union UnionType
{
 int i;
 float f;
 char ch;
};

void main()
{
 UnionType t;
 t.i = 1; // t contains type int
 t.f = 1.25; // now t contains float type
 cout << t.i << endl; // incorrect output (in fact, an attempt to
interpret " float " data into " int ")
```

```
 cout<< t.f << endl; // correct output
 system("pause");
}
```

Test questions and general tasks:

1. What is structure? How is a structure different from an array?
2. In what ways can you declare a structure and initialize its fields? How to access structure fields?
3. What is an enumeration? How to access enumeration fields?
4. What is a union? How to access join fields?

TOPIC 9. PROCESSING EXTERNAL FILES

In C / C ++ c, standard input/output is implemented using library functions. So, for example, library `< stdio . h >` contains input/output facilities for data exchange with both devices and external files. From the point of view of language design, there is no difference whether data is exchanged with devices or files on the disk, which ensures the hardware independence of the language.

Working with files is used in the following cases:

- when a large amount of data is processed (there is not enough RAM memory);
- when data must be stored on external devices in order to be accessed by third-party programs.

The following types of files are distinguished:

1) *Text files* : contain text broken into lines (special newline character). Such files are viewed and edited in a text editor and store data in symbolic representation. But in them it is impossible to access certain characters until all those in front of it have been read.

2) *Binary files* : contain any data and codes (pictures, audio, video, etc.) in an internal representation. The size of a data block is determined by its type. To access the desired data block, you can move the pointer directly to it.

3) *Directories (folders)* : contain links to other files.

Working with external files in C

To work with a file, a special variable of type `FILE` (`<stdio.h>`) is used, which is called *a file pointer* . In fact, this is the address of a data block in memory in which all information about the open file is stored.

File declaration:

```
FILE *logical file_name;
```

This pointer can be called *a logical file name* for use in subsequent operations.

Opening a file is necessary to associate the logical file name with a file that physically exists on disk.

```
logical_file_name =fopen("Physical_file_name","mode");
```

The physical file name is specified using absolute (for example, `"c:\\ admin \\ file_1.txt ")` or relative (for example, `" file_1.txt ")` paths. In the latter case, the file must be in the same directory as the executable program (or in a project with a . cpp file).

File access modes:

- mode `"r"` (from the English read) – reading from a file. The file must already exist on disk.

```
f = fopen (" file_1.txt ", " r ") ;
```

- mode `"w"` (from the English write) – writing to a file. If there was data in the file, it will be deleted, and if the file does not exist, it will be created.
- mode `"a"` (from English append) – adding to the end of the file. If the file is created, it will be opened for writing, and if the file was not created, it will be created. The cursor position for recording is set before the “end of file” sign.

The “+” mode after the file type (`r +, w +, a +`) expands its capabilities – the file is available for both reading and writing:

- `" r +"` mode – opens an existing file for modification with the ability to write and read anywhere in the file. Writing to the end is not allowed because the file size increases;
- `“w+”` mode – creating a new file for recording and subsequent modification (if a file with the same name already exists, it is replaced with a new one). Subsequent write and read operations are allowed anywhere in the file, including at the end, i.e. The file size may increase;
- `“a+”` mode – the file is opened or created and becomes available for changes. Unlike `w +` , when opening an existing file, its contents are not destroyed, and unlike r +, you can add an entry to the end of the file.

After finishing working with the file, you must close it.

```
fclose(filename);
```

To work with *text files* , functions of the library `<stdio.h> are used` , the distinguishing feature of which is the letter `f` at the beginning of the function name.

For formatted data:

- function – reading data from a file;

- `fprintf()` function – writing (output) data to a file.

Listing. Reading data from a file into an array and writing from an array to another file

```
#include <stdio.h>
#include <conio.h>
const int N = 5;
void main()
{
int i, A[N];
FILE *f1, *f2;
f1 = fopen("file1.txt", "r");
for (i = 0; i < N; i++ )
fscanf(f1,"%d",&A[i]); // reading from file into array
fclose(f1);

for (i = 0; i < N; i++ )
 printf("%d",A[i]); // output to console

f2 = fopen("file2.txt", "w");
for (i = 0; i < N; i++ )
fprintf(f2,"%d",A[i]); // output (write) data to a file
fclose(f2);

getch();
}
```

For string data, it is rational to use the following functions:

- `fgetc()` function – reads a character from the input stream;
- `fputc()` function – outputs a symbol to the output stream;
- `fgets()` function – reads characters into the string s (char *) from the input stream stream;
- `fputs()` function – outputs the string s to the output stream.

Listing. Reading a line from a file into a character array

```
#include <stdio.h>
#include <conio.h>
```

```
const int N = 10;
void main()
{
char s[N]; int i;
FILE *f;
f = fopen("file1.txt", "r");
fgets(s,N,f);
fclose(f);

for (i = 0; i < N; i++ )
 printf("%c", s[i]);

getch();
}
```

To work with *binary files* in "mode", the second character is used, which determines the file type (data storage type). The symbol t is a text file (default), b is a binary file. For example,

- mode `"rb"` – read binary (reading data);
- mode `"wb"` – write binary (data recording);
- mode `"ab"` – append binary (adding data to the end).

Working with external files in C ++

To work with files in C ++, you need to include the `header file <fstream>` . It defines classes such as:

- `ifstream` – file input (open for reading);
- `ofstream` – file output (open for writing).

As an example of working with files, consider the task of calculating the sum of numbers located in one file and writing the result to another file.

Let's create class objects `ifstream` and `ofstream`

```
ifstream f1;
ofstream f2;
```

Next, you need to associate the class object with a physical file on disk (`open () method`):

```
f1.open (" file 1.txt " );
```

Similarly, you can perform this operation when creating a class object

```
ifstream f1(" file 1. txt ");
```

Now you need to organize a cycle to read data from the file. Typically, we don't know the amount of data in a file, so there is a special end marker in the file. `The eof ()` function returns 1 if the end of the file has been reached; otherwise it returns 0. The code will be as follows:

```
 while ( !f1.eof () ) // until the end of the file
 {
      f1 >> temp ; // temporary variable for uploading data from
their file
      S = S + temp ;
 }
```

After that, close the file:

```
f1.close();
```

To write the result (sum of numbers) to another file, you need to open it and apply the write operation to the stream:

```
f2 << S;
```

Listing. Sum of numbers from file

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

```
void main()
{
 ifstream f1;
 ofstream f2;
 f1.open("file1.txt");
 int S, temp;
 S = 0;
 while( ! f1.eof() ) // until the end of the file
 {
      f1 >> temp;
      S = S + temp;
 }
 f1.close();
 f2.open("file2.txt");
 f2 << S;
 f2.
close();
}
```

Listing. Reading data from a file into an array and back

```
#include <iostream>
#include <fstream>
using namespace std;
const int N = 5;
void main()
{
 int M[N], i = 0;;
 ifstream f1("file1.txt");
 ofstream f2("file2.txt");
 while (!f1.eof() )
 {
      f1 >> M[i];
      i++;
 }
 f1.close();

 for ( i = 0; i < N; i++ )
      f2 << M[i] << endl; // entry in column
 f2.close();
}
```

Reading character lines from a file can be conveniently done using the function `getline()`.

Listing. The source file contains information about employees: in each line full name, year of birth and city: "IvanovI.I. 1970 Moscow", etc. Output information about employees with a year of birth less than 1985 into another file.

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

void main()
{
 string s, s_temp;
 int pos, gr; // space position, year of birth
 ifstream f1( "file1.txt" );
 ofstream f2( "file2.txt" );
 while ( !f1.eof() ) {
      getline(f1, s);
      pos = s.find( ' ' );
      s_temp = s.substr ( pos+1 );
      gr = stoi(s_temp); // function to convert "string" to "int"
      if ( gr < 1985 )
           f2 << s << endl;
 }
 f2.close(); f1.close();
}
```

Test questions and general tasks:

1. What is the difference between text and binary files?
2. What is a file variable?
3. How to determine the end of a file?
4. General task. Develop a console application that implements sorting an array of numbers from a file. Write the sorted array to a new file.

SECTION 3. DYNAMIC DATA STRUCTURES

TOPIC 10. DYNAMICALLY ALLOCATED MEMORY. POINTERS

When a program is executed, all instructions and data must be loaded into RAM. If there are too many objects in memory needed to execute the program and the RAM cannot accommodate them, the system will freeze. With static variable declarations, memory is allocated at compile time, and the objects remain in memory while the program runs. So, for example, if a modern computer game (with large-scale special effects) used such a static method of working with data, then the player would have to reboot the system after a few seconds of running the game. If you do not destroy unused objects, very soon they will fill the entire amount of PC resources.

In the programs discussed above, we used static data structures, for example:

```
int a = 5, b;
float M[5];
```

When the compiler processes a static variable declaration (for example, `int a = 5`), it allocates memory according to the type (int) and initializes it with the specified value (5). All references to a variable by its name(s) are replaced by the compiler with the address of the memory region (for example, `0x003CFE28`) in which the value of the variable is stored. The programmer can also define his own variables to store the addresses of memory areas; they will be discussed below.

Translators of the C/C++ programming languages form, among others, 3 logical types of memory: static, stack and dynamic (Fig. 3.1), which differ in their operating methods. At the physical level, there are no differences in memory: RAM is an array of numbered (address) bytes, starting from zero.

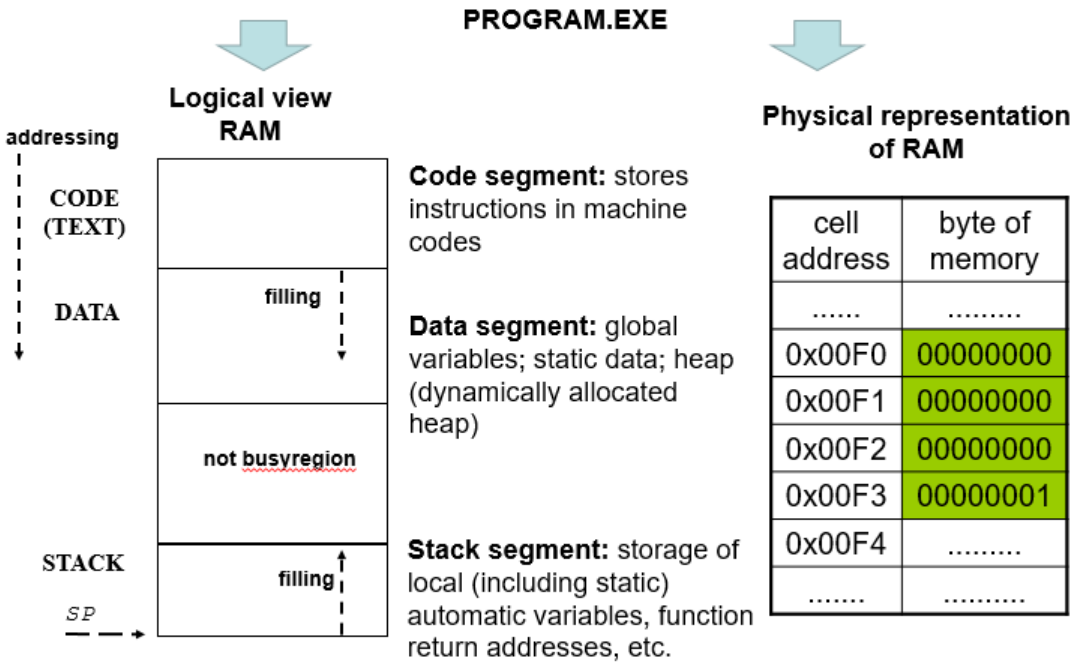


Fig. 3. 1. Process memory model

The program memory is not uniform. It is divided into blocks, which can be stored in physically different places in RAM. Let's take a closer look at these types of logical representation of process (task) memory.

1. *The code area (code segment)* stores instructions in machine codes (binary codes), which are executed by the computer system (sequential reading of commands and their execution is implemented). As is known, according to von Neumann's principles, the program code is stored along with the data in RAM.

2. Local variables are stored on *the stack (stack segment)* . It may fill up, and the OS will indicate that it is full. But what if the program needs a lot of memory? In order for an application to request memory from the OS, it needs a mechanism to interact with it – these are system calls (system interrupts are generated by the process of stopping the execution of the application and transferring control to the OS). In programming languages, system calls are wrapped in library functions, for example, `malloc()`, `printf()`, `scanf()` , etc. The last two functions are also system calls, since only the OS can interact with I/O devices.

3. *The data area (data segment)* is a block of global static data that appears in memory at the time of loading (initial values of constants, variables). As a general rule, the number of global variables should be kept to a minimum (they live outside functions,

including outside main ()). The executable file (.exe) at the time of loading into RAM is decomposed into several areas (*code segment* and *data segment*).

Let's consider dynamically allocated memory. When calling the memory allocation function, the OS issues memory addresses in a special area called *the heap* (dynamic memory area), and the variables are called dynamic. Dynamic memory allocation is a method of allocating RAM for objects in a program, in which memory allocation for an object is carried out during program execution, rather than translation. In fact, the OS hides from the developer the real messed-up view of memory paging, which is much more complex. If, for example, the malloc() function requests a memory area, then we can assume that the OS will allocate a virtual contiguous chunk. The size of the memory area must be a multiple of the size of the data type (for this, the sizeof() operator is used, which calculates the size of the area at compile time). The malloc() function will return the beginning of this memory area. We save this address in the index. The OS may also issue a negative response to a request for memory allocation (for example, request several GB). If the call is unsuccessful, NULL is returned:

```
if (p==NULL) printf ("Memory not allocated")
```

disadvantages when declaring static structures :

- the size of the data structures must be known in advance (it cannot be increased while the program is running);
- memory is allocated by the OS when structures are declared.

The solution is as follows – to use *dynamic data structures*, the characteristics of which include the following:

- the size may be unknown in advance and determined while the program is running (allows you to create data structures of variable size);
- memory is allocated while the program is running;
- no name (logical).

Accordingly, it is necessary to access data without a logical name using an address in memory. To store RAM cells, special variables are used – *pointers* .

A pointer is a variable that stores the address of a RAM cell (variable address). That is, the contents of the pointer are a memory address.

A pointer is not an independent type; it is always associated with some other specific type whose address it can store (the asterisk refers specifically to the type).

```
type* name;
```

To access variables located in memory, hexadecimal addresses are used. The pointer refers to a block of data from a memory area, and to its very beginning. The address of a value is the number of the first byte of the allocated memory block in which the value is located (the size of the block is determined by the type). To find out the address of a variable, *the operation of taking the address* & is used. To access the value referenced by a pointer, the *pointer dereference operator* (*) is used.

Note that static variables are placed on the "stack", and dynamic objects are placed on the "heap" (Figure 3.2).

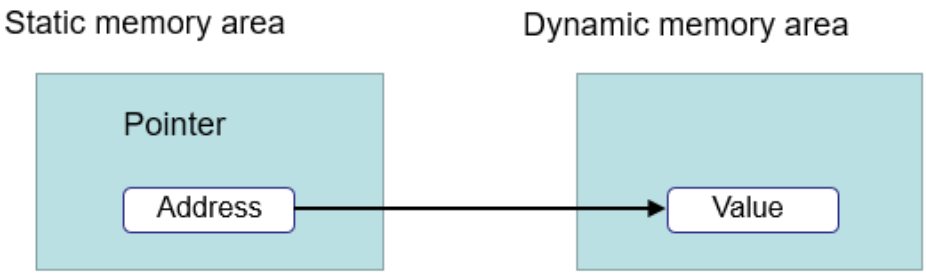


Fig. 3.2 . Location of the pointer in memory

The pointer contains the address of a field in dynamic memory (although it can also be static) that stores a value of a certain type. The pointer itself is located in static memory. A section of RAM that is allocated statically is called static memory, a section of memory that is allocated dynamically is called dynamic memory (dynamically allocated memory). The lifetime of dynamic objects is from the point of creation to the end of the program or until memory is explicitly freed.

Dynamic objects themselves do not require description in the program, since memory is not allocated for them during compilation. During compilation, memory is allocated only for static values, but pointers are static values and require a description.

Scope of pointers:

- storing addresses of memory areas;
- working with dynamic quantities;
- working with dynamically allocated memory ("heap" from the English heap);
- transferring data to a subroutine "by reference" (if the amount of data is large, this speeds up their processing, since they do not need to be copied).

Listing. Declaring and initializing pointers

```
    int a = 5; // static variable of integer type
    int* p ; // declaration of a pointer (from English Pointer )
    p = &a ; // the address of an existing static variable is written
to the pointer
    // The above two lines can be replaced with int* p = &a; or int*
p(&a);
    cout << *p << endl ; // =5 output value by address (pointer con-
tents)
    cout << *p+1 << endl; // =6
    cout << p << endl ; // contents of the pointer – address of variable
a (003CFE28)
    cout << &a << endl; // address variable a (003CFE28)
    cout << &p << endl ; // pointer address in RAM (003CFE1C)
```

You can also store the start address of the data structure

```
    int* p ;
    int M[3] = { 7, 8, 9 };
    p = &A[2]; // address of array element A [2]
    p = NULL ; // zero address
```

Pointers to pointers (N order pointers)

When declaring multidimensional program structures, N-order pointers (pointers to pointers) may be required. Pointers can refer to other pointers, and the RAM memory cells to which the pointers will refer will contain the addresses of other pointers. The number of asterisks when declaring a pointer indicates its order. To obtain the value it refers to, the pointer must be "dereferenced" an appropriate number of times.

The question arises of how to obtain the value referenced by an Nth -order pointer. It is necessary to implement the following chain of steps: by the value of the N order pointer, obtain the address of the N-1 order pointer, by the value of the N -1 order pointer, obtain the address of the N-2 order pointer, etc. And finally, at the address of the variable, access its value.

It can be seen (Fig. 3.3) that the pointers are connected to each other through addresses. So, for a 3rd order pointer `p_p_p_var` the number `008FF95C` is the address, and for the pointer `p_p_p_p_var` it is the value. As you can see, the address values are located relative to each other by the value C_{16}

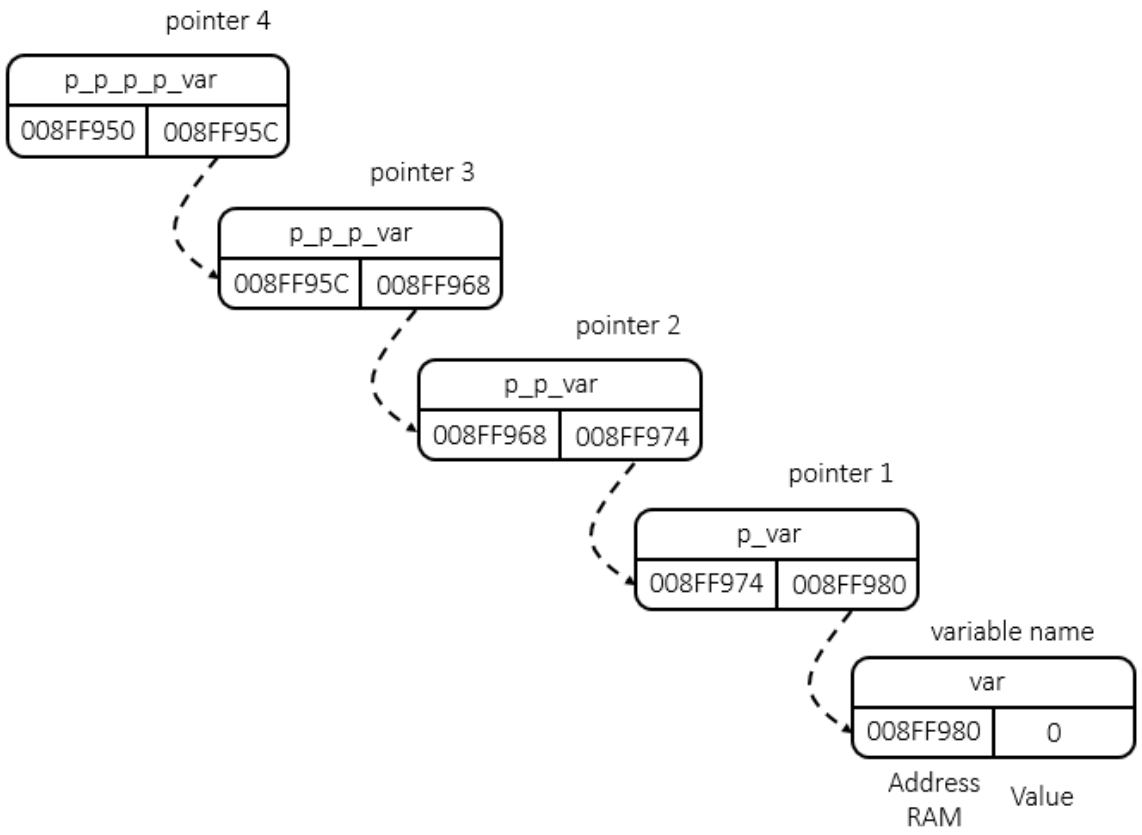


Fig. 3.3. Visualization of links between 1st and 4th order pointers

Listing. Pointers to pointers

```
    # include < iostream >
    using namespace std;
    int main()
    {
    int var = 0; // initialization of the static variable var
    int *p_var = &var; // declare a 1st order pointer to the variable
var
    int **p_p_var = &p_var; // declare 2nd order (“pointer to pointer”)
    int ***p_p_p_var = &p_p_var; // declare 3rd order ("pointer to
pointer to pointer")
    int ****p_p_p_p_var = &p_p_p_var; // declare 4th order ("pointer
to pointer to pointer to pointer")

     cout << "var              " << var << endl; // 0
```

```
     cout << "*p_var               " << *p_var << endl ; // print: 0
- dereference the pointer
     cout << "** p_p_var           " << ** p_p_var << endl ; // print:
0 - 2 times dereference the pointer, because second order pointer
     cout << "***p_p_p_var    " << *** p_p_p_var << endl ; // print:
0 - 3 times dereference the pointer, because third-order pointer
     cout << "****p_p_p_p_var " << **** p_p_p_p_var << endl ; // print:
0 - 4 times dereference the pointer, because fourth-order pointer

     cout << "adress p_p_p_p_var= "     << &p_p_p_p_var << endl; //
4th order pointer address 008FF950
     cout << "adress p_p_p_var= "<< &p_p_p_var << endl; // 3rd order
pointer address 008FF95C
     cout << "adress p_p_var= "<< &p_p_var << endl; // 2nd order
pointer address 008FF968
     cout << "adress p_var= "<< &p_var << endl; // 1st order pointer
address 008FF974

     cout << "adress var= "<< p_var << endl; //variable address =
008FF980
     cout << "adress var= "<< &var << endl; //variable address =
008FF980

     system("pause");
     return 0;
    }
```

Program operation protocol:

```
var 0
*p_var 0
**p_p_var 0
***p_p_p_var 0
****p_p_p_var 0
address p_p_p_p_var= 008FF950
address p_p_p_var= 008FF95C
address p_p_var= 008FF968
address p_var= 008FF974
address var= 008FF980
address var= 008FF980
```

Dynamic memory allocation

To allocate a dynamically allocated memory area, the C language provides a number of functions, for example, `malloc ( memory allocation )` . Its syntax is:

```
pointer = (type*) malloc ( sizeof (type))
```

Used memory can be freed when the data is no longer needed, allowing you to increase the amount of data processed. Function for freeing occupied memory:

```
free (pointer);
```

The C++ language provides the `new operator for these purposes` . Its syntax is:

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

Operator `delete pointer` is used to free memory.

So, a section of memory is allocated in a dynamically allocated memory area ("heap"), and the return value is the address of the first cell of the allocated section. If the system call is not successful (for example, a very large size was requested from the OS), the NULL value is returned. Therefore, the following pointer check is appropriate:

```
if ( p == NULL ) printf ("No memory allocated")
```

The notation `*p` can be deciphered as follows: there is a dynamic object that is referenced by the pointer `p` . Memory for the pointer (of a size sufficient to accommodate only the pointer) is allocated at the compilation stage in a static memory area. If a dynamic value loses its pointer, then it becomes "garbage" (the data takes up memory, but is no longer needed).

The disadvantage of this method of declaring structures is that working with dynamic data slows down program execution, since access to a value occurs in two steps: first the pointer is looked for, then the value is searched for. Thus, the gain in memory capacity is compensated by the decrease in operating speed.

Listing. Using the malloc() function

```
    ...
    int *p1, *p2;
    p1 = (int*) malloc (sizeof(int));
    p2 = (int*) malloc (sizeof(int));
    // Space has been allocated in the dynamic memory area for two integer variables and the pointers have received the corresponding values of their addresses
    *p1 = 3; *p2 = 5; // Dynamic variables are assigned values
    p2 = p1; // Write the address from the pointer p 1 to p 2; pointers P and D began to refer to the same value, equal to 3.
    cout << *p2 << " " << *p1 << endl ; // Number 3 is printed twice
    cout << p2 << " " << p1 << endl ; // address of dynamic variable (00A34790, 00A34790)
    cout << &p2 << " " << &p1 << endl ; // pointer addresses () (003CFDC4, 003CFDD0)
    int a = *p1; // assigning a value to a static variable
    cout << a << endl ;
    free (p1); // procedure that allows you to free memory from data that is no longer needed
    ...
```

The dynamic value equal to 5 lost its pointer and became inaccessible ("garbage" has formed in memory), although it still occupies space in memory.

Example: Using the `new operator`

```
    int *n = new int; // declaring a pointer, allocating a section of dynamic memory sufficient to accommodate a value of type int, and writing the address of the beginning of this section in n.
    int *m = new int (10); // initialize the allocated memory with the value 10.
    int *q = new int [10]; // allocating memory for 10 int values (an array of 10 elements) and writing the address of the beginning of this section to q.
```

Links

A reference is a reference type that stores the address of an object (like a pointer) and represents an alternative name for the object (in terms of the object's logical name, not its physical implementation). You need to understand that a reference is not the “address operator &”, which is used on an already created object in order to obtain its address (i.e., the address of the memory area where the values are stored).

The difference between a “link” and a “pointer”:

- Once a reference is created, it cannot be translated to another object (that is, the reference cannot be redefined);
- links cannot be null (i.e. point to nowhere), pointers can;
- references cannot be uninitialized (for example, writing "int & r "; is an error).

Listing. Links

```
    int a = 1; //variable " a " is located at address 0x0028F998
    int & ra = a ; //an alternative name ( ra ) is defined for the
variable at address 0x0028F998
    cout << &a << endl << &ra << endl ; // print 0x0028F998 0x0028F998
    cout << a << endl << ra << endl; // print 1 1
    ra++; cout << a << endl << ra << endl; // print 2 2
    // i.e. " a " and " ra " are bound to the same address
```

Using pointers

Let's return to the problem about employees, which describes the `Tsotr type` and an array of elements of this type `S [ N ] is created.`

```
    ...
    typedef struct
    {
     int number ;
     string fio ;
     string otdel ;
     int zarplata ;
    } TSotr ;
    ...
    TSotr S [ N ];
```

When sorting large structures (> 1000 records), it can take a long time due to multiple data copying processes. It is more convenient to use pointer sorting.

will need an additional array of pointers (`TSotr * p [ N ]`) and a temporary variable (`TSotr * p_temp ) to swap` them.

Let's arrange the pointers so that the i-th pointer is associated with the i-th structure

```
for ( i = 0; i < N ; i ++ )
  p [ i ] = & S [ i ];
```

To navigate to the fields of the object referenced by the pointer, use the indirection operator `(->)`. As a result of sorting, only pointers are rearranged, but structures are not moved:

```
 for ( i = 0; i < N - 1; i ++ )
      for ( j = N - 2; j >= i ; j -- )
           if ( p [ j ]-> fio > p [ j +1]-> fio ) // sort by key
" fio "
           {
                p 1 = p [ j ];
                p [ j ]= p [ j +1];
                p [ j +1]= p 1;
           }
```

The complete program code for the task is presented below.

Listing. Pointers to pointers

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

typedef struct
{
 int number;
 string fio;
 string section;
 int salary;
} TSotr; // type Employee
```

```
    void main()
    {
     const int N = 3;
     TSotr S [ N ] , s_temp ; // creation objects
     int i,j;
     TSotr *p[N], *p1;
     S[0].number = 1;
     S[0].fio = "IvanovI.I.";
     S[0].section = "Sklad";
     S[0].salary = 40000;

     S[1].number = 2;
     S[1].fio = "SidorovS.S.";
     S[1].section = "Proizvodstvo";
     S[1].salary = 35000;

     S[2].number = 3;
     S[2].fio = "PetrovP.P.";
     S[2].section = "Sklad";
     S[2].salary = 45000;

     for ( i = 0; i < N ; i ++ )
          p [ i ] = & S [ i ]; // arrangement pointers

     // sorting
     for ( i = 0; i < N - 1; i ++ )
          for ( j = N - 2; j >= i ; j -- )
               if ( p [ j ]-> fio > p [ j +1]-> fio ) // sorting By
key " fio "
               {
                    p1 = p [ j ];
                    p [ j ]= p [ j +1];
                    p [ j +1 ]= p1;
               }
               // output sorting
               for ( i = 0; i < N ; i ++ )
                    cout << p[i]->fio << " " << p[i]->salary << " RUR.
"
                    << endl ;

               // calculate the amount
               int sum = 0;
               for ( i = 0; i < N; i++ ) sum = sum + p[i]->salary;
```

```
                cout << " summa = "<< sum << endl ;

                // search for maximum salary
                int max = p [0]-> salary; int imax =0;
                for ( i = 1; i < N ; i ++ )
                     if ( p [ i ]-> salary > max ) {
                          max = p [ i ]-> salary ;
                          imax = i ;
                     }
                     cout << " MAX salary = " << max << " y sotrudnika
s nomerom "<< imax << endl ;

                     system (" pause ");
    }
```

Dynamic arrays

Previously, when declaring a static array (for example, `int M [5]`) memory for it was allocated during broadcast. But, for example, when extracting data from external sources, the volume may not be exactly known in advance. Therefore, it is advisable to allocate memory while the program is running (dynamically). As noted earlier, dynamic data structures allow you to vary their size in memory and remove them when they are no longer needed.

So, memory can be allocated:

1. At the translation stage (the required amount of static memory must be known before the start of program execution, i.e. it is specified as a constant and is maintained throughout the program).

2. During program execution using system calls (new operation, malloc function, etc.). But the required size must also be known before allocating memory in the dynamic area (i.e., now the size of the data structure can be set from the keyboard `cin >> N`).

In both cases, a contiguous piece of memory is allocated. There are mechanisms for allocating non-contiguous memory areas. This applies to dynamic data structures: lists (unidirectional (simply linked), bidirectional (doubly linked), cyclic), stack, deck, queue, binary trees. They differ in the ways in which individual elements are connected and in the operations allowed. These structures will be discussed in the next section.

So, to create a dynamic array, you need to declare a pointer and then use the `new operation:`

```
int* M ; // pointer declaration, no memory allocated
M = new int [ N ]; // memory allocation
```

Next, it is advisable to organize a check of the fact of memory allocation:

```
if ( A == NULL ) {
  printf ("Failed to allocate memory");
  return ;
}
```

Syntactically, using a dynamic array is no different from a static one:

```
for (int i = 0; i < N ; i ++ )
  cin >> M [ i ];
...

for (int i = 0; i < N ; i ++ )
{
  M [ i ] = i * i ;
cout << M [ i ] << " ";
  }
```

Freeing memory:

```
delete [] A ;
```

Listing. Creating a dynamic array and filling it

```
#include <iostream>
using namespace std;
void main()
{
int N;
cout << "Vvedite size: ";
cin >> N;
int *mass = new int[N]; // or separately int *mass; mass = new
int[N];
for (int i = 0; i < N; i++) {
mass[i] = i*i;
cout << " znachenie " << i << " elementa = " << mass[i] << endl;
```

```
}
delete [] mass; // cleanup memory
system("pause");
}
```

Dynamic matrices

In the traditional sense of the term, *a matrix* (in programming) is an array of arrays. Then the dynamic matrix is a pointer to an array of pointers. Since the dimensions of the matrix are unknown in advance, there are two mechanisms for allocating space in memory for a dynamic matrix.

Mechanism No. 1. Allocating a separate memory block for each row of the matrix:

a) with the declaration of a new data type

```
typedef int* ptrInt ; // new data type “pointer to type int ”
...
ptrInt* mas ; // address of the pointer array
...
mas = new ptrInt [ rows ]; // allocating memory for an array of
pointers
for ( i = 0; i < rows ; i ++ )
  mas [ i ] = new int [ cols ]; // allocating an array for each
line
```

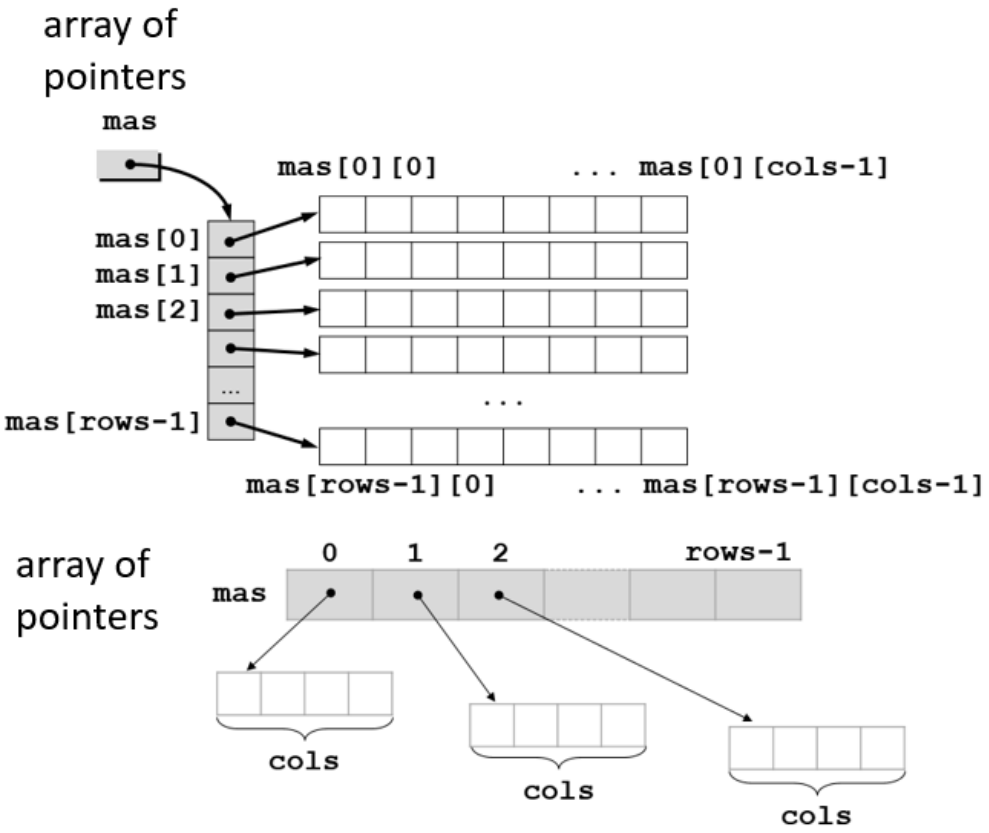


Fig. 3.4. Dynamic Matrix Structure

b) with the declaration of a second-order pointer

```
    int** mas = new int*[ rows ]; // declaration of a dynamic matrix
(allocation of memory for an array of pointers)
    ...
    for ( int i =0; i < rows ; ++ i )
        mas [ i ] = new int [ cols ]; // allocate memory for each line
    ...
    delete [] mas [ i ]; // free up memory for strings
    delete [] mas ; // free the array of string addresses
```

Mechanism No. 2. Memory allocation for the entire matrix at once.

```
    typedef int *pInt;
    ...
    ptrInt *mas;
    ...
    mas = new pInt[rows];
    mas [0] = new int [rows*cols]; // selection memory under one block
```

```
    for (int i = 1; i < cols; i ++ ) mas[i] = mas[i-1] + cols; //
arrangement pointers
    ...

    delete mas [0]; // release memory of one block
    delete mas ;
```

Listing. Filling a dynamic matrix and sorting the desired row

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

    void main()
    {
     setlocale(LC_ALL,"rus");
     int rows, cols ; // rows and columns
     int k, temp ; // number of the sorted string

    cout <<"Set array size:\ n ";
    cout <<"Number of lines: "; cin >> rows ;
    cout <<"Number of columns: "; cin >> cols ;
    cout <<"Line number to sort: "; cin >> k ; k --;

    int** mas = new int *[ rows ]; // declaration of a dynamic matrix

    cout <<" Original array : "<< endl ;
    for ( int i =0; i < rows ; ++ i )
    mas [ i ] = new int [ cols ]; // allocate an array for each line

    for ( int i =0; i < rows ; ++ i )
    {
        for ( int j =0; j < cols ; ++ j )
    {
            mas [ i ][ j ]= rand () %100;
            cout << mas [ i ][ j ]<<"\ t "; // tab
        }
    cout << endl ;
    }

    // Sort only one row
    for (int i=0; i<cols; i++)
    {
```

```
for (int j=0; j<cols-1; j++)
{
if (mas[k][j]>mas[k][j+1])
        {
temp=mas[k][j];
mas[k][j]=mas[k][j+1];
mas[k][j+1]=temp;
}
}
}

cout<<" Processed array : "<<endl;
for (int i=0; i<rows; ++i)
{
for (int j=0; j<cols; ++j)
cout<<mas[i][j]<<"\t"; cout<<endl;
}

for (int i=0; i<rows; ++i)
    delete [] mas [ i ]; // free up memory for strings
delete [] mas ; // free the array of string addresses

system (" pause ");
}
```

Test questions and general tasks:

1. What are dynamic data structures? Where is the memory allocated for them?

2. Name ways to work with dynamic memory (memory allocation and clearing) in C / C ++ languages.

3. General task. Develop an application for working with dynamic one-dimensional arrays (function `malloc()` || operator `new`). In the application, create three dynamic one-dimensional arrays (set the size from the keyboard) with random numbers (organize filling in a separate subroutine). Sum the corresponding values of two arrays into a third. Clear memory.

TOPIC 11. STL CONTAINER CLASSES. CONTAINER VECTOR

STL library (Standard Template Library , from English. "standard template library") in the C ++ language consists of a set of ready-made classes and serves to increase the reliability of developed programs, their portability and versatility, as well as reduce their creation time.

The main disadvantage of using it is the reduction in program performance (depending on the compiler implementation).

There are 5 categories of objects in STL:

- Container – objects for storing a set of similar elements in memory. There are methods for processing elements, implemented using class templates.
- Adapter – adaptation of components to provide a different interface.
- Iterator (English iterator) – objects for universal access to elements stored in a container.
- Algorithm – generalized procedures for processing container elements
- Functional object (eng. functor) – hiding a function in an object for use by other components.

Here is the classification of containers:

I. *Sequential containers* provide storage of a finite number of values of the same type in a continuous sequence.

- Basic containers:
  - vectors (vector),
  - lists (list),
  - two-way queues (deque).
- Adapter containers (based on base containers, with a modified interface):
  - stacks
  - queues (queue),
  - priority queues (priority_queue).

II. *Associative containers* provide access to data by key. They are built on the basis of balanced trees. Scroll:

- associative arrays (dictionaries) (map), dictionaries with duplicates,
- (multlmap), sets (set), sets with duplicates (multiset) and bitsets (bitset).

Next we will look at some container classes.

Variable size array. Container «Vector»

A vector is a variable-sized array. Objects of this class have many standard processing methods (delete, insert, sort, etc.). This is a replacement for a standard dynamic array, for which memory is allocated manually (using the new operator). Language developers recommend using `vector` instead of manually allocating memory for an array. This avoids memory leaks and makes the programmer's work easier.

To use objects of this type, you need to include the corresponding header file:

```
# include < vector >
```

Next, we create an object indicating the type of elements.

```
vector < float > V ; // declaration of an empty vector
vector <int> V (10 ) ; // reserve 10 int elements
vector <int> V (10, 0 ) ; // initialize 10 elements of type int and fill them with zeros
```

Ready-made functions have been created for it (methods of the `vector` class):

- `capacity()` method – total volume of the vector,
- `size()` method – number of filled elements,
- `resize ()` method – changes the number of elements in the vector,
- `push_back ( )` method – adding a new element to the end of the vector ,
- `pop_back ( )` method – remove the last element ,
- `clear ()` method – remove all elements of the vector,
- `empty ()` method – check the vector for emptiness,
- `reserve ()` method – memory reservation (parameter – number of elements),
- method `shrink _ to _ fit ()` – frees unused memory (C++11),
- and etc.

Listing. Filling and outputting the contents of a vector

```
for (int i = 0; i < N; i++ )
V.push_back(i);

for (int i = 0; i < V.size(); i++ )
```

```
cout << V[i] << " ";
```

Please note that accessing the elements of a vector is implemented in exactly the same way (`[]`) as in a classic array.

ARRAY container

`An Array` is a sequential container containing a contiguous sequence of fixed-size elements. New elements cannot be added to the `array container` , nor can existing ones be deleted.

To use objects of this type, you need to include the corresponding header file:

```
# include <array>
```

Next, we create an object indicating the type of elements.

```
std::array<int,10> A{}; // declaration of an empty array
```

If we did not use curly initialization, but its elements are still initialized to zero. The template <T, S> takes two parameters: y the contained type T and the fixed size of the array S.

The set of built-in functions can be found in the official documentation.

Test questions and general tasks:

1. How to declare a variable-sized array and set its size?
2. How to expand an array?
3. General task. Develop a console application in which:

a) describe an empty vector: `vector <int> <name>` (previously `#include <vector>`)

b) programmatically write the squares of numbers from 1 to 10 into a vector: `push_back()`

c) determine the size of the vector: `size()`

d) remove last element: `pop_back()`

e) remove all elements: `clear()`

```
#include <algorithm>
#include <iterator>
```

f) search element And his numbers : `find(first, last, value), binary_search (first, last, value)`

g) search for maximum and minimum elements: `max_element(), min_element()`

h) searching for a sequence in array form: `search(first1, last1, first2, last2)`

i) counting the number of required elements: `count()`

j) sorting: `sort()`

TOPIC 12. LIST. LIST CONTAINER

A list is a set of elements of the same type for which addition (insertion) and deletion operations are introduced. Nodes can be located in different memory areas, but access to elements is only sequential. The connection of elements is provided through pointers.

Classification of lists:

1. By the number of index fields:
   - singly linked linear list (one pointer to the next element);
   - doubly linked linear list (two pointer fields – to the next element and to the previous one).
2. According to the method of connecting the elements:
   - linear (last element points to NULL);
   - cyclic (the last element is connected to the first):
     - singly linked cyclic list;
     - doubly linked cyclic list.

A list as a data structure can be modeled:

1. “manually” using a chain of connected nodes;
2. using container `list` .

Let's consider the first option (*modeling a “list” with a chain of connected nodes*).

For simplicity, consider a singly linked linear list. The structure must contain a data field and a field with a pointer to the next element.

```
struct TNode // “node” structure
{
  int Data ; // data field
  TNode * Next ; // pointer to next element
};
```

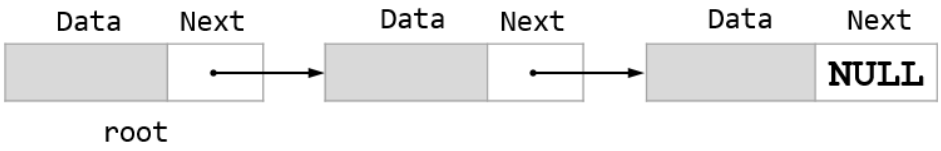


Fig. 3.5. Singly linked linear list

Listing. Implementation of the list model and functions for working with it

Problem: modeling a unidirectional list with a chain of connected nodes (TNode). Develop a program in which to describe a one-way list whose nodes contain a data field and a pointer field to the next node. Implement:

1) declaration of the TNode node structure with a data field (int Data) and a pointer to the next element (TNode *Next) and the function of its creation (initialization);

2) the function of adding new nodes;

3) the function of removing a node from the list;

4) the function of printing list elements;

5) a function for rearranging two nodes of the list.

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

struct TNode // list node structure
{
intData; // data field
TNode *Next; // pointer to next element
};
```

It was also possible to define a new name for the type:

```
typedef TNode* PNode ;
```

1. The function of creating the root node (root) of a list (in it, the pointer field to the next element contains a NULL value).

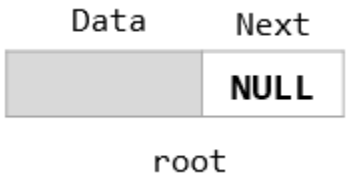


Fig. 3.6. List root node

```
TNode *create_root(int n) // argument "n" – value for the Data field
of the root node
{
TNode* list;
list = (TNode*)malloc(sizeof(TNode)); // allocate memory to the root
of the list
```

```
  list->Data = n; // OR (*list).field = n;
  list->Next = NULL; // for now this is the last node of the list
return(list);
}
```

2. Function adding a node.

To implement this function, you must: create the node to be added (temp), move the node pointer from the previous node to the one being added, and set the pointer of the added node to the next node (the one that the previous node pointed to) (Fig. 3.7).

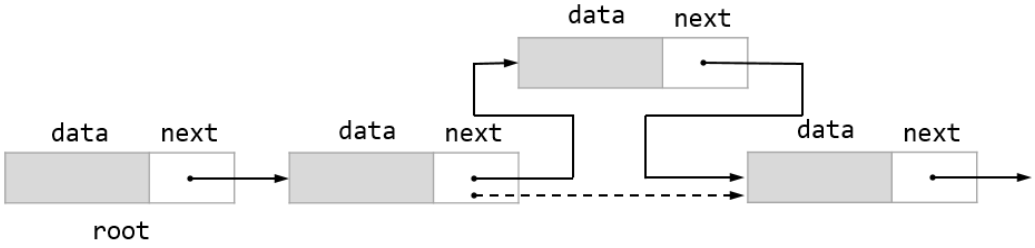


Fig. 3.7. Scheme for adding a node

```
TNode *add_node(TNode *list, int n) // Takes 2 arguments: – Pointer
to the node after which the addition occurs; – Data for the node to
be added (argument "n").
{
TNode *temp, *p;
temp = (TNode*)malloc(sizeof(TNode)); // allocate memory for a new
element
p = list->Next; // from the received argument (i.e. the node after
which the addition occurs) we extract a pointer to the next node
list->Next = temp; // the address of the created element is written
into the received argument (previous) (i.e. now the previous node
points to the one being created)
temp->Data = n; // write data for the new node
temp->Next = p; // write the address for the new node (now the cre-
ated node points to the next element)
return(temp); // function return value is the address of the added
node
}
```

3 a). Node removal function.

To implement this function, you need to: move the pointer of the node being deleted to the next element and free the memory from under the node being deleted (Fig. 3.8).

Fig. 3.8. Node removal scheme

```
TNode *delete_node (TNode *list, TNode *root) // arguments: pointer
to the node to be deleted; pointer to the root of the list
{
TNode *temp;
temp = root;
while (temp->Next != list) // look for the node to be deleted (see
list starting from the root)
{ // the node preceding list has not yet been found
    temp = temp -> Next ; // go to the next node (we move temp to the
address written in the Next field, i.e. to the next element)
}
  temp -> Next = list -> Next ; // rearrange the pointer (from the
Next field of the node being deleted to the Next field of the previ-
ous node)
free(list); // free the memory of the node being deleted
return(temp); // return a pointer to the node before the one to be
deleted
}
```

3 b). Root node removal function.

```
TNode *delete_root(TNode *root)
{
  TNode* temp;
  temp = root -> Next ; // rearrange the Next field data (pointer to
the next node) from the root to a temporary element (new root)
free(root); // freeing memory from the old root
return(temp); // the return value will be the new root of the list
}
```

4. Function of printing node data.

To implement this function, it is necessary to sequentially traverse all nodes and display values in the Data field.

```
void list_print(TNode *root) // argument is a pointer to the root of
the list
{
```

```
TNode *p;
p = root;
do {
     printf("%d ", p->Data);
     // displaying the field value by node p

     p = p->Next; // move to the next node (since they point to each
other from the end) (repositioning the pointer from the Next field to
the current address of the list element)
} while (p != NULL);
// until we reach the last element, which has ->Next = NULL
}
```

5. Function of rearranging two nodes.

Rearranging nodes is done by changing pointers to them, and not by moving the data structures themselves. To do this, it is necessary to determine the previous and subsequent nodes for each replaced one. In this case, two situations are possible:

a) rearranged nodes – neighbors (Fig. 3.9 a);

b) the rearranged nodes are not neighbors (Fig. 3.9 b).

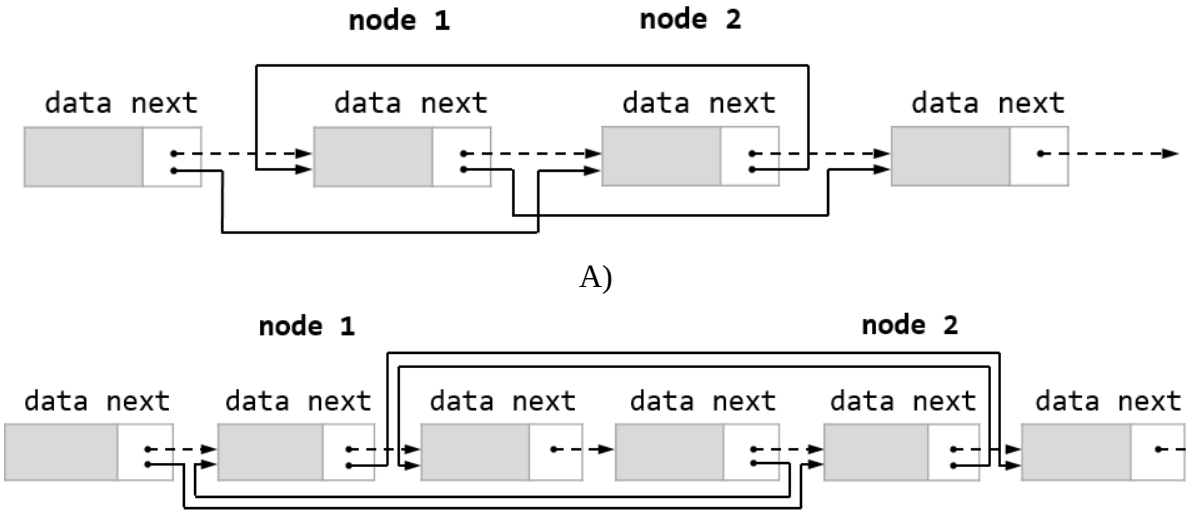


Fig. 3.9. Scheme of rearranging two nodes:
a) rearranged nodes – neighbors; b) rearranged nodes – removed

```
    TNode * swaping ( TNode* node1, TNode* node2, TNode* root ) //
parameters: pointers to the nodes to be swapped, and a pointer to the
root of the list
{
```

```
TNode *prev1, *prev2, *next1, *next2; // declaration of previous
(prev) and subsequent (next) pointers
prev1 = root; // place the root element in temporary previous nodes
prev2 = root;

// next we determine whether the exchanged nodes are located at the
root of the list. When permuting pointers, it is necessary to check
whether the node being replaced is the root of the list, because in
this case there is no node preceding the root one.

if (prev1 == node1) // if the 1st node being exchanged is at the be-
ginning (i.e. is the root)
    prev1 = NULL;
else
while (prev1->Next != node1) // search node previous node1
      prev1 = prev1->Next;

if (prev2 == node2) // if the 2nd node being exchanged is at the be-
ginning (i.e. is the root)
    prev2 = NULL;
else
while (prev2->Next != node2) // search node previous node2
      prev2 = prev2->Next;

next1 = node1->Next; // node next to node1 (remember the link to the
next element after the 1st exchanged node)
next2 = node2->Next; // node next to node2 (remember the link to the
next element after the 2nd exchanged node)

if (node2 == next1) // if the nodes are neighboring (i.e. the address
of the 2nd coincides with the ->Next field of the 1st)
{ // neighboring nodes exchange
node2->Next = node1;
    node1->Next = next2;
if (node1 != root)
prev1->Next = node2;
}
else
if (node1 == next2)
    {
// neighboring nodes exchange
node1->Next = node2;
      node2->Next = next1;
```

```
if (node2 != root)
        prev2->Next = node2;
}
else
{
// non-neighboring nodes exchange
      if (node1 != root)
prev1->Next = node2;
node2->Next = next1;
if (node2 != root)
prev2->Next = node1;
node1->Next = next2;
}

if (node1 == root) return(node2);
if (node2 == root) return(node1);
    return(root); // function returns the address of the root ( root
) of the list
}

void main()
{
setlocale(LC_ALL,"RUS");
TNode *ptrroot = create_root(0); // creation root node list with
value = 0 and Next = NULL; ptrroot – address root node
TNode *PNode1 = add_node(ptrroot, 1); // adding value = 1 to the list
and storing its address in the declared PNode pointer
TNode *PNode2 = add_node(PNode1, 2); // ...
TNode *PNode3 = add_node(PNode2, 3); // ...
TNode *PNode4 = add_node(PNode3, 4); // ...
cout << " original list : ";
list_print(ptrroot); // 0 1 2 3 4 – output

cout << endl << "print element addresses: ";
cout << endl;
cout << ptrroot << endl; // print the address of the root element
cout << PNode1 << endl; // print addresses of elements (addresses do
not have to be sequential)
cout << PNode2 << endl; // ...
cout << PNode3 << endl; // ...
cout << PNode4 << endl; // ...

// rearrangement of PNode2 and PNode4 values
```

```
swaping(PNode2, PNode4, ptrroot);
cout << "after rearrangement : ";
list_print(ptrroot); // 0 1 4 3 2 – output
cout << endl;

cout << " delete element PNode1: ";
delete e _node(PNode1, ptrroot); // deleted element By address (
pointer ) PNode1
list_print(ptrroot); // 0 4 3 2 – output
cout << endl;

cout << " delete root element : ";
TNode *p_new_root = delete_root(ptrroot);
list_print(p_new_root); // 4 3 2 – output

system (" PAUSE ");
}
```

It is also possible to create a *doubly linked linear list model* . The structure must contain a data field and two fields with pointers (to the next and previous elements).

```
struct TNode // "node" structure
{
intData; // data field
TNode *Next; // pointer to next element
TNode *Prev; // pointer to previous element
};
```

If necessary, you can create a Pointer Node data type.

```
typedef TNode* PNode;
```

Container `list`

The above functions are implemented in the library container (class) `list` . To use objects of `the list type` , you need to include the corresponding header file:

```
# include < list >
```

Next, we create an object indicating the type of elements in angle brackets

```
list <int> L ;
```

Operations (methods) of the `list class` :

- `push_back ( )` – adding a new element to the end;
- `push_front ( )` – adding a new element to the beginning;
- `insert ()` – insertion anywhere in the list (at the iterator position);
- `sort()` – sorting;
- `unique()` – unique elements;
- `merge()` – merging lists;
- and etc.

To iterate over the elements of containers in the STL library, a generalized abstraction – an “iterator” – is used as an intermediary to access elements.

An iterator (or cursor) is a special object (“pointer”) that allows you to iterate over the elements of a container from the STL library . It is a smart pointer that "knows" how to access elements. Each container supports its own iterator type. Using an iterator, it is possible to separate algorithms from the storage method (for example, you can write a universal search algorithm that takes 3 arguments: 2 iterators for the beginning and end of the region and the desired value).

To use objects of this type, you need to include the corresponding header file:

```
# include < iterator >
```

Listing. Creating an iterator for list and looping through elements

```
list<double> x;
list<double>::iterator it;

// brute force elements ( via for)
for(it=x.begin(); it!=x.end(); it++)
{
  sum +=(*it ); //access elements by iterator
}

// iterate over elements (via while )
it = x.begin ();
while(it!=x.end())
{
sum+=*it;
    it++;
}
```

Listing. Methods for working with lists

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;
void main()
{
 setlocale(LC_ALL,"rus");
 list<int>L;
 for(int i = 0; i<10; i++) L.push_back(rand()%20);

 cout << " List #1: " << endl;
 copy(L.begin(), L.end(), ostream_iterator<int>(cout," "));
 // copy the sequence L . begin ( ) , L. end () to the output
stream

 cout << endl << "First element of list 1: " << L . front () <<
endl ;
 cout << "Last element of list 1: " << L . back () << endl ;

 L.sort ();
 cout << endl << "Sorted list #1: " << endl ;
 copy(L.begin(), L.end(), ostream_iterator<int>(cout," "));

 L.unique ();
 cout << endl << "List #1 with unique elements: " << endl ;
 copy(L.begin(), L.end(), ostream_iterator<int>(cout," "));

 list<int> L2;

 for(int i=0; i<10; i++) L2.push_back(rand()%20); // add V list
new elements

 cout << "\n List No. 2: " << endl;
 copy(L2.begin(), L2.end(), ostream_iterator<int>(cout," "));

 L2.sort () ; cout << endl << "Sorted list #2: " << endl ;
 copy(L2.begin(), L2.end(), ostream_iterator<int>(cout," "));

 L2.merge ( L ); // merge () method works if both lists are ordered
 cout << endl << "Combining lists #1 and #2: " << endl ;
 copy(L2.begin(), L2.end(), ostream_iterator<int>(cout," "));
```

```
 system (" pause ");
}
```

Test questions and general tasks:

1. What is a "list" and what operations does it allow?
2. List the types of lists.
3. How can a “list” be modeled as a chain of connected nodes?
4. General task (modeling a unidirectional list with a chain of connected TNode nodes). Develop a program in which to describe a one-way list whose nodes contain a data field and a pointer field to the next node. Implement:

a) declaration of the TNode node structure with a data field (int Data) and a pointer to the next element (TNode *Next) and the function of its creation (initialization);

b) function of adding new nodes;

c) function for removing a node from the list;

d) function for printing list items;

e) function of rearranging two nodes of a list.

5. General task (using the container class <list> of the STL C++ library). Develop a console application in which:

a) create an object of class <list> (previously #include <list>);

b) describe and fill two lists of natural numbers with values: push_back() method;

c) display the first and last elements of lists: methods front(), back();

d) sort both lists: sort() method;

e) select unique elements: unique() method;

f) merge both sorted lists: merge() method.

TOPIC 13. STACK. STACK CONTAINER

A stack is a special case of a unidirectional linear list in which elements are added and removed from only one end, called the top of the stack . This model uses the “LIFO (Last In – First Out)” principle. “last to come, first to leave” (Fig. 3.10).

a) vertical model b) horizontal model

Fig. 3.10. Stack structure model

When programs are executed, a certain area of memory is allocated to the program stack. Moreover, the processor has a special cell (register) that stores the address of the top of the stack.

Scope: The stack principle is used when executing programs in the OS. The system (hardware) stack stores:

- return addresses from subroutines (addresses to which the program goes after executing the subroutine);
- parameters passed to subroutines;
- local variables of subroutines.

To prevent the system stack from overflowing, it is better not to pass large structures (for example, arrays) to subroutines. You can pass its address or use pass-by-reference.

Methods for software implementation of the stack model in programming languages:

- using a one-dimensional array;
- using a linked list;
- using the class.

Let's consider modeling the “stack” structure using a one-dimensional array.

```
#define NMAX 5
struct stack {
 int elem[NMAX];
 int top ;
};
```

where NMAX is the maximum number of elements in the stack, elem is an array of NMAX int numbers intended to store stack elements, top is the index of the element located at the top of the stack.

Listing. Implementation of the stack model and functions for working with it

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

#define NMAX 5 // maximum number of elements in the stack
struct stack {
 int elem[NMAX]; // array for storing stack elements
 int top; // index of the element at the top of the stack
};

//Function for adding (“push”) an element to the top of the stack
(parameters: 1. stack, 2. element to be added;)
void push(struct stack *s, int data) {
 if(s->top < NMAX) {
      s->elem[s->top] = data; // room element on position s->top
      s->top++; // in the last addition this increment will shift
the real counter by one; therefore, in pop() we will first do a dec-
rement (s->top--)
 } else
      cout << "Stack is full, number of elements: " << s->top <<
endl;
}

//Function for removing (pop – from the English “pull out”) an
element from the top of the stack
float pop(struct stack *s) {
 int elem;
 if((s->top) > 0) { // if stack Not empty
      s -> top --; // immediately decrement the vertex index
(since the last operation in the push() function was s->top++, i.e.
the index is one more than needed)
      elem = s -> elem [ s -> top ]; // assign to a temporary
variable for output (it could have been done without it)
      return elem; // extracted element (1 piece )
 } else {
      cout <<" Stack empty " << endl;
      return 0;
 }
}
```

```
    //Function for getting the "content" of the top element without
removing it (this is equivalent to the operations i=pop (s); push (s,
i);)
    int getTopElem(struct stack *s) {
     if((s->top) > 0) {
          return( s->elem[s->top-1]);
     } else {
          cout <<" Stack empty " << endl;
          return 0;
     }
    }

    //Function for getting the “number” of the top element of the
stack without removing it (it is also the “number” of remaining ele-
ments)
    int getTopIndex(struct stack *s) {
     return(s->top);}

    //Function to check if the stack is empty (returns "true" if the
stack is empty and "false" otherwise)
    int empty(struct stack *s) {
     if((s->top) == 0) return 1;
     else return 0;
    }

    // Function print everyone elements stack
    void printStack(struct stack *s) {
     int i;
     i=s->top;
     if(empty(s) == 1) return; // if stack let it be the way out
     do {
          i--;
          cout << s->elem[i] << endl;
     }
     while(i > 0);
    }

    void main() {
     setlocale(LC_ALL,"RUS");
     struct stack *s; // announcement pointer
     int i, n, elem;
     s = (struct stack*)malloc(sizeof(struct stack)); // selection
memory under stack type
     s->top = 0; // initial value for the number
     cout << "Enter the number of elements in the stack: ";
```

```
     cin >> n;

     for(i = 0; i < n; i++) {
          cout << " Enter element No. " << i << " ";
          cin >> elem;
          push(s, elem); // push ("push") the element to the top of
the stack
     }

     cout << "On stack " << getTopIndex(s) << " elements" << endl; //
"number" of elements on the stack
     printStack(s); // print all elements
     cout << "Element at top " << getTopElem(s) << endl;;
     do {
          cout << "Retrieving element " << pop(s) << " ";
          cout << "there are " << getTopIndex(s) << " elements left
on the stack" << endl; // "amount of elements
     }
     while(empty(s) == 0);

     system("pause");
    }
```

Stack container

The above functions are implemented in the library container (class) `stack`. To use `stack objects` , you need to include the appropriate header file:

```
#include <stack>
...
stack<int> S;
```

Operations (methods) class `stack`:

- `push ()` – "push" (add) an element to the top of the stack;
- `pop ()` – "pull" (remove) an element from the top of the stack;
- `top ()` – return the "top" element from the top of the stack (without removing);
- `empty ()` – return true if the stack is empty, false otherwise.
- and etc.

Listing. Writing from one file to another in reverse order

```
#include <iostream>
```

```
#include <fstream>
#include <stack>
using namespace std;
void main()
{
ifstream f1; // input stream
ofstream f2; // output stream
stack <int> s; // object of type stack
int x;

f1.open("in. txt "); // numbers written in the column
while ( f1 >> x ) // reading data
s.push(x); // "push" (add) to the top of the stack
f1.close();

f2.open("out.txt");
while ( ! s.empty() )
  {
f2 << s.top() << endl; // show the element from the top of the
stack (without removing)
s.pop(); // "pop" (remove) an element from the top of the stack
}
f2.close(); // numbers written into the column in reverse order
system("pause");
}
```

A striking example of using the `stack container` is the implementation of algorithms for calculating arithmetic expressions and parsing parenthetical expressions (see the computer science textbook for grades 10–11, authors K.Yu. Polyakov, E.A. Eremin) .

Test questions and general tasks:

1. What is a "stack" and what operations does it allow?
2. How is the system (OS) stack used when running a program? What problems can arise when using a stack?
3. How to model a "stack" structure with a chain of connected nodes?
4. General task. In the source file, integers are written in a column. Using the stack container, move them in reverse order to another file.

TOPIC 14. QUEUE. QUEUE CONTAINER

A queue is a special case of a one-way list, adding elements to one end and fetching from the other end. The queue implements the FIFO (First In – First Out) service principle. When fetched, the element is removed from the queue. In fact, two basic operations are defined:

- adding an element to the end;
- removing the first element.

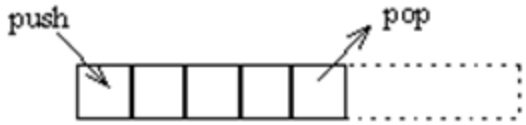


Fig. 3.11. Model of the "Queue" structure (FIFO)

Areas of application of the queuing principle:

- queues for processing messages in the OS;
- queues of input and output requests;
- queues of data packets in routers;
- and etc.

When working with a queue, the delete operation is time inefficient, since it is necessary to move all elements to the beginning of the structure. Another approach can be proposed, in which the elements in the queue will not move. Let us have a static array and two variables to store the first (head, English Head) and last (tail, English Tail) elements of the queue. Then removing an element from the queue only amounts to increasing the Head variable . When an element is added to the end of the queue, the Tail variable is incremented. One array element must remain unoccupied to recognize whether the queue is empty or full.

Methods for software modeling of a queue:

- based on a one-dimensional array

```
#define NMAX 10
struct queue {
int q[NMAX];
  int front , last ; // position of the first and last elements in
the queue
};
```

- linked list based

```
struct TNode {
intData;
struct TNode *ptr;
};
struct TQueue {
struct TNode *front, *last;
};
```

- class based

```
class TQueue {
static const int NMAX =10;
int *queue;
int front , last ;
public:
// operations (class methods
} ;
```

Listing. Implementation of the queue model (based on a linked list) and functions for working with it

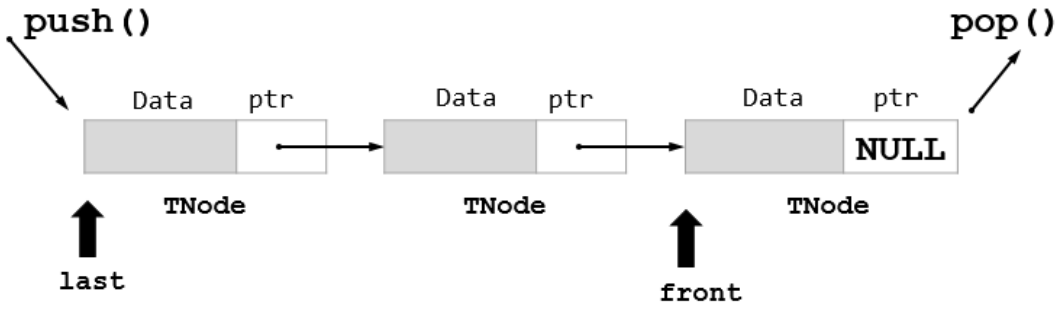


Fig. 3.12. Model of the "Queue" structure based on a linked list

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

// Model of the "Queue" structure based on a singly linked list
struct TNode { // queue element
int Data;
struct TNode *ptr;
};
struct TQueue { // queue
```

```
    struct TNode *front; // address of the first element in the queue
    struct TNode *last; // address of the last element in the queue
    };

    // Initialization function
    struct TNode* init(int a)
    {
    struct TNode* temp = (struct TNode*)malloc(sizeof(struct TNode));
    temp->Data = a;
    temp->ptr = NULL;
    return(temp);
    }

    // Function for adding a new element
    struct TNode * addelem(TNode *lst, int number)
    {
    struct TNode *temp, *p;
    temp = (struct TNode*)malloc(sizeof(TNode));
    p = lst->ptr; // saving a pointer to the next node
    lst->ptr = temp; // the previous node points to the one being
created
    temp->Data = number; // saving the data field of the added node
    temp->ptr = p; // the created node points to the next element
    return(temp);
    }

    // Function to check if the queue is empty
    int isempty(struct TQueue *q) {
    if(q->front==NULL) // if the address of the first element in the
queue is empty (i.e. the queue is empty)
    return(1);
    else
    return(0);
    }
    // Push function: add an element to the end of the queue
    void push(struct TQueue *q, int x) {
    if((q->last==NULL) && (q->front==NULL)) { // if we “push” the very
first element
    q->last = init(x);
    q->front = q->last;
    } else
    q->last = addelem(q->last, x);
    }
```

```
    // Function to print queue items
    void print(struct TQueue *q) {
    struct TNode *h;
    if(isempty(q)==1) {
    cout << "Queue empty!!!\n";
    return;
    }
    for(h = q->front; h!= NULL; h=h->ptr)
    cout << " " << h->Data;
    return;
    }

    // Pull function: removes the first element from the queue and
returns the removed value
    int pop(struct TQueue *q) {
    struct TNode *temp;
    int x;
    if(isempty(q)==1) {
    cout << "Queue empty!!!\n";
    return(0);
    }
    x = q->front->Data; // save the value of the "pulled" element
    temp = q->front; // save the address of the “pulled out” element
    q->front = q->front->ptr;
    free(temp); // clear memory
    return(x); // return the value of the "pulled out" element
    }

    // Get the first element in the queue without removing it
    int front(struct TQueue *q) {
    int x;
    x = q->front->Data;
    return(x);
    }
    int main() {
    system("chcp 1251"); system("cls");
    struct TQueue *q;
    int a;

    q = (TQueue*)malloc(sizeof(TQueue*)); // allocating memory for a
queue based on OLS
    q->front = 0;
    q->last = 0;
```

```
    print(q);
    // Push 5 elements into the queue
    for(int i=0; i<5; i++) {
    cout << "Enter queue element: ";
    cin >> a;
    push(q, a);
    }
    cout << "\nSource: ";
    print(q);

    // Remove all queue elements one by one
    while(q->front!= NULL) {
    a = pop(q);
    cout << "\n Element removed from queue: "<< a;
     cout << "\n q->front: "<< q->front; // the address of the first
element in the queue will be shifted (since the elements are decreas-
ing)
     cout << "\n q->last: "<< q->last; // the address of the last
element will be constant
    cout << "\nRemaining elements in queue: "; print(q);
     cout << endl << "-----";
    }

    system("pause");
    return 0;
    }
```

Program operation protocol:

```
    The queue is empty!!!
    Enter queue item: 1
    Enter queue element: 2
    Enter queue element: 3
    Enter queue element: 4
    Enter queue element: 5
    Source: 1 2 3 4 5
    Element removed from queue: 1
    q->front: 00E94B58
    q->last: 00E94E50
    Remaining elements in queue: 2 3 4 5
    -----
    Item removed from queue: 2
    q->front: 00E94BA0 // A0-58=48
    q->last: 00E94E50
```

```
Remaining elements in queue: 3 4 5
-----
Item removed from queue: 3
q->front: 00E94BE8 // E8-A0=48
q->last: 00E94E50
Remaining elements in queue: 4 5
-----
Item removed from queue: 4
q->front: 00E94E50
q->last: 00E94E50
Remaining items in queue: 5
-----
Item removed from queue: 5
q->front: 00000000
q->last: 00E94E50
Remaining items in the queue: The queue is empty!!!
```

Queue container

To use objects of this type, you need to include the corresponding header file:

```
#include <queue>
...
queue <type> Q ;
```

Operations (methods) of the `queue class` :

- `push ()` – "push" an element to the end of the queue;
- `pop ()` – "pull" (remove) the first element in the queue;
- `front ()` – return the first element in the queue (without removing);
- `empty ()` – return true if the queue is empty, false otherwise;
- and etc.

An interesting example of using the `queue container` is an algorithm for filling an area (matrix) in graphic programs (see the computer science textbook for grades 10–11, authors K.Yu. Polyakov, E.A. Eremin) .

Test questions and general tasks:

1. What is a "queue" and what operations does it allow?
2. How can a "queue" be modeled as a chain of connected nodes?

TOPIC 15. BINARY TREES. ASSOCIATIVE MAP CONTAINER

Trees are designed to store hierarchical data structures. They consist of *nodes* and connections between them (*arcs*). The very first node at the top of the tree is called *the root* . The final nodes of the lower level are *leaves* . *The height of a tree* is the largest number of arcs from the root to the leaf (in Fig. 3.12, the height of the tree is 2). Each node in a tree is itself the root of the nodes emanating from it, making that node and its children a subtree.

A binary tree is a root and two separate binary trees associated with it ("left" and "right" subtrees). It follows that a tree is a recursive data structure.

Scope of application of tree data structures:

- data search in a large information array (databases), indexes;
- data sorting;
- optimal data compression (Huffman method);
- calculation of arithmetic expressions;
- organization of abstract data structures of the language;
- and etc.

As a more general case of a tree, there is the concept of a graph, which describes network models. Graph theory algorithms are used to solve problems of choosing the optimal path, routing problems on the Internet, etc.

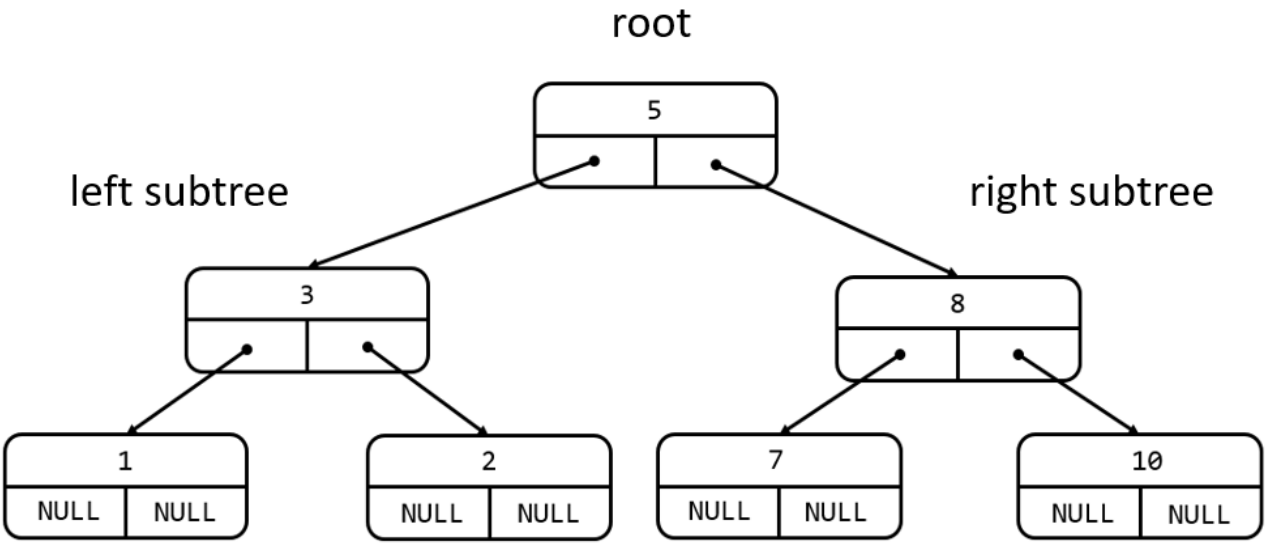


Fig. 3.13. Binary tree model

Let's assume that we need to find an element in the tree (let's call it *target*). *The node key* is the values of the nodes that are searched for (theoretically, in addition to the key, a node can contain other data, i.e., the information representing each node is a record, and not a single data field).

A binary search tree is a tree that has the following properties:

- to the left of the node are nodes with smaller or equal keys;
- to the right of the node are nodes with greater or equal keys.

If the element precedes the root element, you only need to search the left half of the tree. If the target element follows the root element, then the search must be performed only in the right subtree of the root node.

For example, you need to find a node whose key is 2. We start the search from the root, whose key is 5 (greater than the search value), and, therefore, the search needs to continue in the left subtree, etc.

Thus, one comparison eliminates half the tree from the search (the number of comparison operations is proportional to $log_2 N$ and, accordingly, has an asymptotic complexity of O ($log_2 N$)) . In linear search, 1 element is cut off per comparison.

A tree is a nonlinear structure, so a dynamic array is inconvenient for its construction. The most straightforward way to implement a binary search tree involves using dynamically allocated nodes linked together via pointers.

A tree node can be described as a structure:

```
struct node_tree
{
int data; // data field
struct node_tree *left; // left child
struct node_tree *right; // right child
};
```

Fig. 3.14. Model of a tree structure node

Tree Traversal Methods

Traversing a tree means “visiting” all nodes once.

1. KLP – “root-left-right” (in direct order, “depth first traversal”, prefix form): root, deep into the tree along the left subtrees (until we reach the leaf), traverse the right subtree.

The tree traversal will look like:

```
void print_tree (node_tree *tree) {
  if (tree!=NULL) { // Until an empty node is encountered
```

```
    cout << tree -> data << " "; //Output the root of the tree
       print_tree ( tree -> left ); //Recursive function for the left
subtree
       print_tree ( tree -> right ); //Recursive function for the
right subtree
      }
    }
```

2. LCP – “left-root-right” (in symmetrical order, infix form): traverse the left subtree, visit the root, traverse the right subtree.

The tree traversal in infix form will look like:

```
    void print_tree(node_tree *tree)
    {
    if (tree != NULL) { //Until an empty node is encountered
    print_tree(tree->left); //Recursive function for printing the
left subtree
    cout << tree->data << " "; //Output the root of the tree
    print_tree(tree->right); //Recursive function for displaying the
right subtree
    }
    }
```

3. LPK – “left-right-root” (in reverse order, postfix form): traverse the left subtree, traverse the right subtree, visit the root.

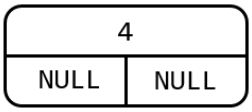
Traversing the tree in postfix form will look like:

```
    void print_tree(node_tree *tree)
      if ( tree != NULL ) { //Until an empty node is encountered
       print_tree ( tree -> left ); //Recursive function for the left
subtree
       print_tree ( tree -> right ); //Recursive function for the
right subtree
    cout << tree -> data << " "; //Output the root of the tree
    }
    }
```

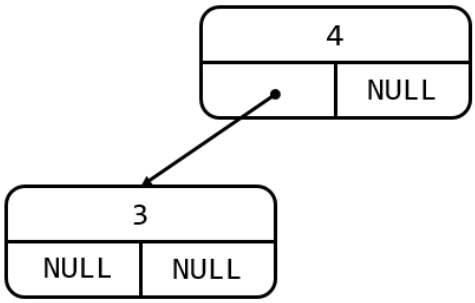
4. Walk around in width (gradually go down one level) – “root-sons-grandsons”.

Example: building a binary tree.

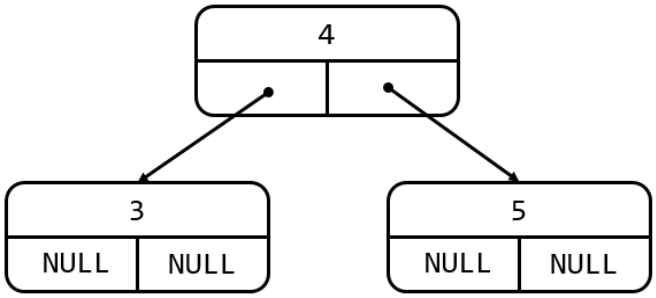
Formulation of the problem. The root of the tree is the first element of the input sequence (4, 3, 5, 1). After this, elements smaller than the root are located in the left subtree, and elements larger than the root are located in the right (the rule is observed at all levels of the tree). This way, all elements will be placed in the tree. To access data (for example, when displaying it on the screen), we will use the infix form (LKP – “left-root-right”).



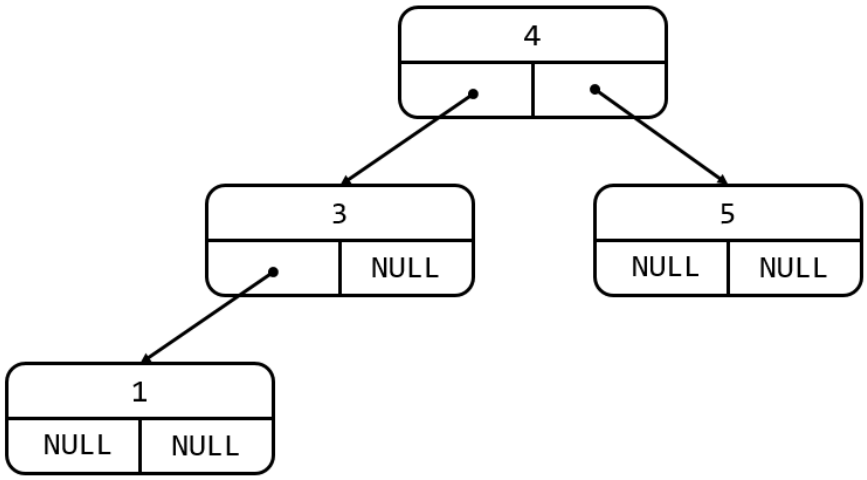
A)



b)



V)



G)

Fig. 3.15. Creating tree structure nodes

After the first iteration of the loop, the root node is created in the `main () function ( data field = 4 )`. The second iteration adds a node on the left since the `data field = 3` , which is less than the root node. The third iteration adds a node to the right (`data field = 5`), etc. Each time `the data` field is analyzed from the root node, gradually going down the tree.

Listing. Implementation of a binary tree model and functions for working with it

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

// Model node tree
struct node_tree
{
int data; // field data
struct node_tree *left; // left descendant
struct node_tree *right; // right descendant
};

// Print data on LCP tree nodes – "left-root-right";
// Implement tree traversal in infix form
void print_tree(node_tree *tree)
{
if (tree != NULL) { // until an empty node is encountered
print_tree(tree->left); // print the left subtree
```

```
        cout << tree -> data << " "; // print root
        print_tree ( tree -> right ); // print the right subtree
    }
    }

    // Adding nodes
    struct node_tree * addnode(int x, node_tree *tree) {
      if (tree == NULL)
      {
        tree = new node_tree ; // allocate memory for a new node
        tree -> data = x ; // filling the data field
        tree -> left = NULL ; // filling branches
        tree->right = NULL;
      }
      else
        if ( x < tree -> data ) // from the 2nd iteration of the loop
in main (); if the incoming element x is less than the root element,
move to the left
          tree -> left = addnode ( x , tree -> left ); // recursively
adding an element along the left branch
        else // otherwise we go to the right
          tree -> right = addnode ( x , tree -> right ); // recur-
sively adding an element
    return(tree);
    }
    //Clearing the memory of the entire tree
    void delete_tree(node_tree *tree)
    {
    if (tree != NULL) // if the tree is not empty
    {
    delete_tree(tree->left); // recursive deletion of the left branch
    delete_tree(tree->right); // recursively delete the right branch
    delete tree; // remove root
    }
    }

    void main()
    {
    setlocale(LC_ALL,"RUS"); // Russian language in the console
    struct node_tree *root = NULL; // declare the tree structure
    int temp ; // current node value
    // data input for 5 tree nodes
```

```
      for (int i = 0; i< 5; i++)
    {
    cout << " Enter data For node : " << i + 1 << ": ";
    cin >> temp;
        root = addnode ( temp , root ); // place the entered node on
the tree; from the second iteration of the loop, the second parameter
sent to root is the return result from the previous call to add-
node(), etc.
    }
    print_tree(root); // display tree elements, get a sorted array
    delete_tree(root); // delete allocated memory
    system("pause");
    }
```

More examples of building binary trees:

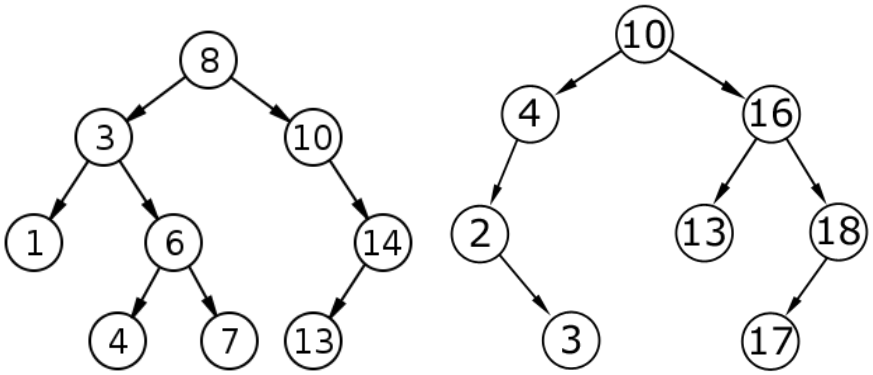


Fig. 3.16. Examples of binary trees

With the help of such structures it is easy to sort data: just traverse the tree according to the LCP scheme. Ideally, all vertices should have two sons, and all leaves should be located at the same depth.

MAP container

MAP (“mapping”) is an associative array (dictionary), element indexes are any data.

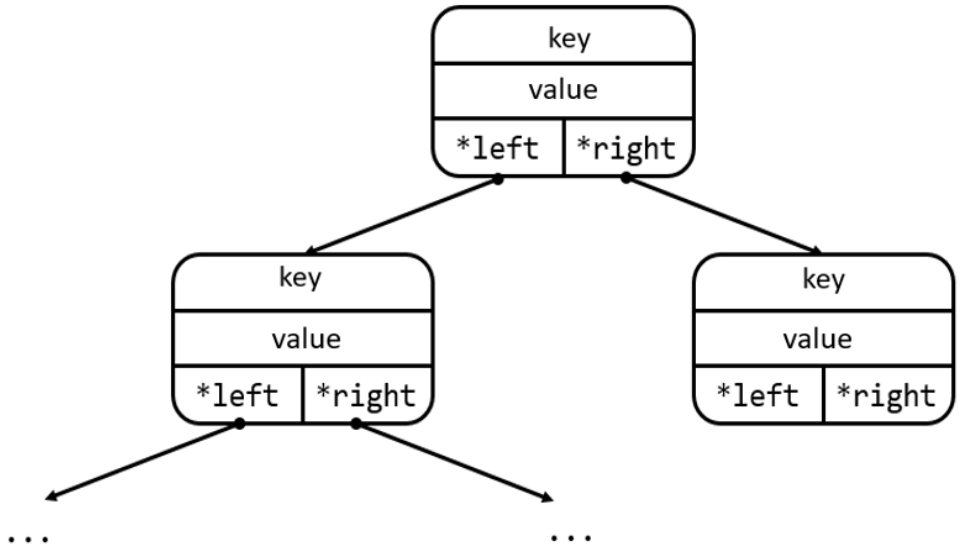


Fig. 3.17. MAP Container Internal Model

To use objects of this type, you need to include the corresponding header file:

```
#include <map>
...
map <string,int> M;
```

Operations (methods) class `map` :

- `begin()` – iterator to the first element;
- `end()` – iterator to the last element;
- `empty()` – true if the container is empty;
- `count ( key )` – the number of elements with a given key (in map 1 or 0);
- `find ( key )` – iterator to the element with the specified key;
- `erase ( it ), erase ( start , end )` – deletes an element with a given iterator or between given ones;
- `size()` – number of elements;
- `clear()` – complete cleaning of the container;
- `insert()` – insert;
- `printmap()` – print;
- `swap()` – rearrangement;
- and etc.

To iterate over the elements of a container, you need an iterator of the appropriate type:

```
map <string,int>::iterator it;
```

For example, the output of container elements looks like this:

```
for ( it = L.begin(); it != L.end(); it++ )
  with out << it->first << ": " << it->second << endl;
```

Listing. Finding an entry in a MAP container

```
#include < iostream >
#include <string>
#include <map>
using namespace std;
typedef map<string, int> TMap; // declare type data
typedef TMap::const_iterator iter; // declare type iterator
void main(){
TMap a; // an object
a["Ivanov II"] = 1;
a["Petrov PP"] = 2;
a [" Sidorov S. _ S. "] = 3;
iter i = a . begin (); // set to the beginning
for ( i = a.begin(); i != a.end(); i++ )
cout << i->first << ": " << (*i).second << endl;

string name;
cout << "Enter name: "; getline(cin, name);
i = a.find(name);
cout << "Number for name " << name << " is " << i->second << endl;

 system (" pause ");
}
```

Test questions and general tasks:

1. Give a definition of the concept “tree”.
2. Where is the tree structure used?
3. List ways to traverse a tree.
4. What is an associative MAP container?
5. In what ways can you access the elements of a MAP container?

6. General task. Develop an application in which to describe (using the MAP container) telephone codes of countries around the world. Initially, country codes are written in the source file (.txt), in the application, sort by country and write the result to a MS Excel file (.xls). Source file data (.txt) :

```
Russia 7
Belgium 32
Switzerland 41
Syria 963
Taiwan 886
Tunisia 21
Iraq 964
Israel 972
Turkey 90
Turkmenistan 993
United_Arab_Emirates 971 _ _ _
Uruguay 598
US A 1
China 86
Croatia 385
Germany 49
Greece 30
Greenland 299
Hungary 36
Iceland 354
India 91
Japan 81
Kazakhstan 7
Italy 39
Mexico 52
Netherlands 31
South_Africa _ _ 27
Spain 34
Sweden 46
Brazil 55
Vietnam 84
Iran 98
United Kingdom 44
```

SECTION 4. TASKS FOR INDEPENDENT WORK

Practical work No. 1. Assignments for independent work on the topic “Linear Algorithms”

Option 1

Find the distance between two points with given coordinates (x1, y1) and (x2, y2).

Option 2

Find the circumference and area of a circle of given radius R.

Option 3

Two numbers are given. Find the arithmetic mean of their squares and the arithmetic mean of their modules.

Option 4

Write a program to calculate the area of a circle and the circumference of a circle.

Option 5

Write a program to calculate the area of a parallelogram and a trapezoid.

Option 6

Write a program to calculate the area of the lateral surfaces of a parallelepiped, the volume of a pyramid and a sphere.

Option 7

Write a program to calculate the volume of a truncated pyramid and the volume of a tetrahedron.

Option 8

Write a program to calculate the lateral surface area and volume of a cube and octahedron.

Option 9

Write a program to calculate the area and volume of a spherical segment.

Option 10

Write a program to calculate the volume and lateral surface of a truncated cone.

Option 11

Write a program to calculate the volume and area of a sphere.

Option 12

Write a program to calculate the area and volume of a spherical sector.

Option 13

Write a program to calculate the volume and lateral surface of a cylinder and cone.

Option 14

Write a program to calculate the area and volume of a spherical belt.

Option 15

Write a program to calculate the area of a square, circle, or ellipse.

Option 16

Write a program to calculate the area of a circular ring and the volume of a prism.

Option 17

Find the roots of the quadratic equation $Ax^2 + Bx + C = 0$, given by its coefficients A, B, C (coefficient A is not equal to 0), if it is known that the discriminant of the equation is non-negative.

Option 18

A three-digit integer number is given. Find the sum of its digits.

Option 19

Write a program to calculate the area of a right triangle and an equilateral triangle.

Option 20

The coordinates of the three vertices of the triangle are given (x_1, y_1), (x_2, y_2), (x_3, y_3). Find its perimeter and area.

Practical work No. 2. Assignments for independent work on the topic "Conditional operator"

Option 1

a) Write a program that finds the smallest and largest among three given numbers.

b) Write a program that, given the sides of a triangle, checks it for rectangularity; if the triangle is not rectangular, then calculate the size of the angles of this triangle.

Option 2

a) Write a program that, given three numbers *a* , *b* , *c*, determines which of them is early *d* ; if neither is equal *to d* , then find

$$\max(d-a, d-b, d-c)$$

b) Write a program that, given four values, finds

$$\max(\min(a,b), \min(c,d))$$

Option 3

a) Write a program that finds solutions to the equation

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

for any given values *of a* And *b* .

b) Write a program that squares odd numbers and cubes even numbers.

Option 4

a) Write a program that, depending on a given *x*, displays the values of the expressions in ascending order: *cos(x), sin(x)/x, ln(x)* . If for some value of *x* the expression does not make sense, display a message about this and compare the values of only those that make sense.

b) Write a program that, depending on the requested type of translation, converts a number from a radial measure to a degree measure or vice versa.

Option 5

a) Write a program that searches for the largest of four numbers.

b) Write a program that counts the number of negative numbers among given three numbers.

Option 6

a) Write a program that checks given three values x , y , x and calculates the following expression:

$$u = \frac{\max^2(x, y, z) - \min^2(x, y, z)}{(\max(x, y, z) + \min(x, y, z))^2}$$

b) Write a program that counts the number of negative numbers among the given three numbers; if there are no negative numbers among the given numbers, then report this.

Option 7

a) Write a program that, given the coordinates of three points on the coordinate plane, determines which points are at the same distance. If there are none, the program should report this.

b) Write a program that calculates the product of negative and positive given four numbers, and then tells which product is greater than the other in absolute value.

Option 8

a) Write a program that squares odd numbers and cubes even numbers.

b) Write a program that displays on the screen those numbers that are less than k and the sum of the digits of which is less l . Three three-digit numbers are specified.

Option 9

a) Write a program that prints in the order: a) decreasing and b) increasing three given numbers.

b) Write a program that calculates the product of negative numbers among four given ones. Then it reports whether the resulting product is negative or positive.

Option 10

a) Write a program that displays on the screen those numbers that are greater than k and the sum of their digits is greater than l . Three three-digit numbers are specified.

b) Write a program that counts the number of positive numbers among the given three numbers; if there are no positive numbers among the given numbers, then report this.

Option 11

a) Write a program that finds the smallest and largest absolute value among three given numbers.

b) Write a program that, among the given four numbers, finds a different number from the others and displays its number on the screen. It is assumed that three equal numbers and one different number are given.

Option 12

a) Write a program that, depending on a given x, displays the values of the expressions in ascending order: $cos(x)/2$, $sin(x)$, $ln(x-2)$. If for some value of x the expression does not make sense, display a message about this and compare the values of only those that make sense.

b) Write a program that displays odd numbers whose sum of digits is greater than a given value m . Initially, three three-digit numbers are specified.

Option 13

a) Write a program that displays even numbers whose sum of digits is greater than a given value m . Initially, three three-digit numbers are specified.

b) Write a program that finds among three different numbers a , b , c one that is greater than one but less than the other and doubles that number. Moreover, the program should report which number was doubled.

Option 14

a) Write a program that prints the floor number where the elevator should stop if it is known that in N The storey building has one entrance, with 5 apartments per floor. When entering the elevator, the apartment number is entered. If you enter an apartment number by mistake, the program should report this.

b) Write a program that checks whether the sum of the last digits of three given numbers is an even number; If yes, then check the last digit of the amount for evenness.

Option 15

a) Write a program that checks whether the sum of the last digits of three given numbers is an odd number; if yes, then check the last digit of the amount for oddness.

b) Write a program that, given four values, finds
$\max(\min(a,b),\min(c,d))$.

Option 16

a) Write a program that, based on a given time in a clock, determines the time of day (morning, afternoon, evening, night).

b) Write a program that checks a triangle with sides a , b , c for squareness.

Option 17

a) Write a program that determines equal numbers among given three numbers.

b) Write a program that finds the number of numbers (among three given ones) that end in an odd digit.

Option 18

a) Write a program that, given the sides of two triangles, checks whether they are equal in size (equal areas).

b) Write a program that finds the number of numbers (among three given ones) that end in an even digit.

Option 19

a) Write a program that checks whether the sum of the last digits of three given numbers is an odd number.

b) Write a program that determines which of two points, specified on the plane by their coordinates, is closer to the OX axis, and which is closer to the OY axis.

Option 20

a) Write a program that finds the smallest of four numbers entered.

b) Write a program that counts the number of positive numbers among the given three numbers.

Practical work No. 3. Assignments for independent work on the topic “Cycles”

Option 1

Given a sequence of n integers. Find the arithmetic mean of this sequence. Don't use an array data structure.

Option 2

Given a sequence of n integers. Find the sum of the even elements of this sequence. Don't use an array data structure.

Option 3

Given a sequence of n integers. Find the sum of even-numbered elements from this sequence. Don't use an array data structure.

Option 4

Given a sequence of n integers. Find the sum of the odd elements of this sequence. Don't use an array data structure.

Option 5

Given a sequence of n integers. Find the sum of odd-numbered elements from this sequence. Don't use an array data structure.

Option 6

Given a sequence of n integers. Find the minimum element in this sequence. Don't use an array data structure.

Option 7

Given a sequence of n integers. Find the number of the maximum element in this sequence. Don't use an array data structure.

Option 8

Given a sequence of n integers. Find the number of the minimum element in this sequence. Don't use an array data structure.

Option 9

Given a sequence of n integers. Find the maximum element in this sequence. Don't use an array data structure.

Option 10

Given a sequence of n integers. Find the sum of the minimum and maximum elements in this sequence. Don't use an array data structure.

Option 11

Given a sequence of n integers. Find the difference between the minimum and maximum elements in this sequence. Don't use an array data structure.

Option 12

Given a sequence of n integers. Find the number of odd elements of this sequence. Don't use an array data structure.

Option 13

Given a sequence of n integers. Find the number of even elements of this sequence. Don't use an array data structure.

Option 14

Given a sequence of n integers. Find the number of elements of this sequence that are multiples of the number K. Do not use an array data structure.

Option 15

Given a sequence of n integers. Find the number of elements of this sequence that are multiples of its first element. Don't use an array data structure.

Option 16

Given a sequence of n integers. Find the number of elements of this sequence that are multiples of K1 and not multiples of K2. Don't use an array data structure.

Option 17

Given a sequence of n integers. Determine which numbers are more in this sequence: positive or negative. Don't use an array data structure.

Option 18

Given a sequence of integers followed by 0. Find the arithmetic mean of this sequence. Don't use an array data structure.

Option 19

Given a sequence of integers followed by 0. Find the sum of the even elements of this sequence. Don't use an array data structure.

Option 20

Given a sequence of integers followed by 0. Find the sum of the even-numbered elements from this sequence. Don't use an array data structure.

Option 21

Given a sequence of integers followed by 0. Find the sum of the odd elements of this sequence. Don't use an array data structure.

Option 22

Given a sequence of integers followed by 0. Find the sum of the odd-numbered elements from this sequence. Don't use an array data structure.

Option 23

Given a sequence of integers followed by 0. Find the minimum element in this sequence. Don't use an array data structure.

Option 24

Given a sequence of integers followed by 0. Find the number of the maximum element in this sequence. Don't use an array data structure.

Option 25

Given a sequence of integers followed by 0. Find the number of the minimum element in this sequence. Don't use an array data structure.

Option 26

Given a sequence of integers followed by 0. Find the maximum element in this sequence. Don't use an array data structure.

Option 27

Given a sequence of integers followed by 0. Find the sum of the minimum and maximum elements in this sequence. Don't use an array data structure.

Option 28

Given a sequence of integers followed by 0. Find the number of negative elements of this sequence. Don't use an array data structure.

Option 29

Given a sequence of integers followed by 0. Find the difference between the minimum and maximum elements in this sequence. Don't use an array data structure.

Option 30

Given a sequence of integers followed by 0. Find the number of even elements of this sequence. Don't use an array data structure.

Practical work No. 4. Assignments for independent work on the topic “Functions”

Option 1

Describe the procedure Func Swap (X, Y), which swaps the contents of the variables X and Y. Call the described subroutine for three different sets of numbers.

Option 2

Describe the function Func Quart (X, Y), which determines the number of the coordinate quarter in which the point with coordinates (X, Y) is located. Call the described subroutine for three different sets of numbers.

Option 3

Describe the procedure Func PowerTwoThreeFour (N,X , Y , Z), which calculates the second, third and fourth powers of the number N and returns them to the variables X, Y, Z (N is the input parameter , X , Y , Z are the output parameters). Call the described subroutine for three different sets of numbers.

Option 4

Describe a boolean function Func Even Number (N) that returns TRUE if parameter N is even and FALSE otherwise. Call the described subroutine for three different numbers and find the number of even ones.

Option 5

Describe the procedure Func Mean(X, Y, M), which calculates the arithmetic mean of two positive numbers X and Y (X and Y are input, Mean is the output parameter of real type). Call the described subroutine for three different sets of numbers.

Option 6

Describe a boolean function Func Sqr(N) that returns TRUE if the integer parameter N (>0) is the square of an integer, and FALSE otherwise. Call the described subroutine for three different numbers.

Option 7

Describe a function FuncArea Sphere(R) of real type that finds the area of a sphere of radius R. Call the described subroutine for three different numbers.

Option 8

Describe a boolean function Func Power Two (N) that returns TRUE if the integer parameter N (>0) is a power of 2, and FALSE otherwise. Call the described subroutine for three different numbers.

Option 9

Describe the procedure Func Rect P erimeter S pace (X1, Y1, X2, Y2), which calculates the perimeter and area of a rectangle with sides parallel to the coordinate axes. Call the described subroutine for three different sets of numbers.

Option 10

Describe a function Func SumRange (X , Y) of integer type that finds the sum of all integers from X to Y inclusive (X and Y are integers). Call the described subroutine for three different sets of numbers.

Option 11

Describe the procedure Func DigitCountSum (N , C, S), which finds the number C of digits of a positive integer N , as well as their sum S (N is the input, C and S are the output parameters of the integer type). Call the described subroutine for three different sets of numbers.

Option 12

Describe the function FuncP erimeterTriangle (A, H), which finds the perimeter of an isosceles triangle based on its base A and height H drawn to the base. Call the described subroutine for three different sets of numbers.

Option 13

Describe the procedure Func Digits Reverse (N), which reverses the order of the digits of a positive integer N. Call the described subroutine for three different numbers.

Option 14

Describe a function Func CircleS pace (R) of real type that finds the area of a circle of radius R. Call the described subroutine for three different numbers.

Option 15

Describe the procedure Func MinMax (X, Y), which records the smallest of the values of X and Y in the variable X, and the largest in the variable Y (X and Y are parameters that are both input and output). Call the described subroutine for three different sets of numbers.

Option 16

Describe the procedure Func QuadraticEquation (A, B, C) of integer type, which determines the roots of the quadratic equation $AX^2 + BX + C=0$ (A, B, C are real parameters). Call the described subroutine for three different sets of numbers.

Option 17

Func SortedThree (X , Y , Z) procedure that changes the contents of the parameters so that their values are sorted in ascending order. Call the described subroutine for three different sets of numbers.

Option 18

Describe the function Func Ca1c (X , Y , Op) of real type, which performs one of the arithmetic operations on non-zero real numbers X and Y and returns its result. The type of operation is determined by the integer parameter Op: 1 – addition, 2 – subtraction, 3 – multiplication, 4 – division. Call the described subroutine for three different sets of numbers.

Option 19

Describe the procedure Func PowerThree (X , Y), which calculates the third power of the number X and returns it in the variable Y (X is the input parameter, Y is the output parameter; both parameters are real). Call the described subroutine for three different sets of numbers.

Option 20

Describe the procedure Func DigitSum (N , S), which finds the sum of digits S of a positive integer N (N is an input parameter, S is an output parameter of an integer type). Call the described subroutine for three different sets of numbers.

Practical work No. 5. Assignments for independent work on the topic “One-dimensional arrays”

Option 1

Given a one-dimensional array of integers. Print in the same order all even numbers from the given set and the number K of such numbers.

Option 2

Given a one-dimensional array of integers. Print in the same order the numbers of all odd numbers from the given set and the number K of such numbers.

Option 3

Given a one-dimensional array of real numbers. Print the numbers of those numbers in the set that are less than their left neighbor, and the number K of such numbers.

Option 4

Given a one-dimensional array of real numbers. Print the numbers of those numbers in the set that are greater than their right neighbor, and the number K of such numbers.

Option 5

Given a one-dimensional array of integers. Check whether the given set forms an increasing sequence. If it does, then print True; if not, print False.

Option 6

Given a one-dimensional array of real numbers. If this set forms a decreasing sequence, then output 0; otherwise, print the number of the first number that violates the pattern.

Option 7

Given a one-dimensional array of integers containing at least two zeros. Print the sum of the numbers from the given set located between the first two zeros (if the first zeros are consecutive, then print 0).

Option 8

Given a one-dimensional array of integers. If this set forms a decreasing sequence, then output 0; otherwise, print the number of the first number that violates the pattern.

Option 9

Given a one-dimensional array of integers containing at least two zeros. Print the sum of numbers from the given set located between the last two zeros (if the last zeros are consecutive, then print 0).

Option 10

Given a one-dimensional array of integers. Print even numbers among positive elements.

Option 11

Given a one-dimensional array of real numbers. Print non-zero array elements and their product.

Option 12

Given a one-dimensional array of integers. Swap the maximum and minimum elements of the array.

Option 13

Given a one-dimensional array of real numbers. Swap the maximum and last elements of the array.

Option 14

Given a one-dimensional array of integers. Replace with zeros all elements of the array up to the maximum.

Option 15

Given a one-dimensional array of integers. Write a program that displays on the screen those elements of a given array that are located after the maximum element of the entire array.

Option 16

Given a one-dimensional array of real numbers. Replace all array elements after the minimum with zeros.

Option 17

Given a one-dimensional array of integers. Write a program that moves the zero elements of a given array one element forward.

Option 18

Given a one-dimensional array of integers. Write a program that prints the part of the array up to the element whose value is zero. An array is only allowed one element whose value is zero.

Option 19

Given a one-dimensional array of real numbers. Write a program that swaps the maximum and minimum elements of an array.

Option 20

Given a one-dimensional array of integers. Write a program that displays on the screen those elements of a given array that are located before the minimum element of the entire array.

Practical work No. 6. Assignments for independent work on the topic "Multidimensional arrays"

Option 1

Display the numbers of those rows of an $n \times m$ *integer matrix* that coincide with a given linear array consisting of m elements .

Option 2

In a 4x4 integer square matrix, duplicate the row || matrix column containing its minimum || maximum element. Fill the array with random values.

Option 3

In an integer rectangular matrix, define:

a) the number of columns containing at least one zero element;

b) the number of the line containing the longest series of identical elements.

Option 4

In a 4x5 integer matrix before || after the line || matrix column with the given number k, insert a row || a column of zeros.

Option 5

In an integer rectangular matrix:

a) implement sorting of all rows of the array;

b) determine the minimum among the sums of the moduli of the elements of the diagonals parallel to the secondary diagonal of the matrix.

Option 6

In an integer rectangular matrix, determine: the row and column numbers of all “saddle points” (a matrix element that is simultaneously the minimum element in the corresponding matrix row and the maximum element in the corresponding matrix column).

Option 7

In a 5x10 integer matrix, remove row || column containing minimal || maximum element of the matrix. The matrix is filled with random values.

Option 8

In an integer square matrix:

a) determine the sum of the minimum elements from each line;

b) rearranging its columns, arrange them in accordance with the growth of the sums of the modules of its negative even elements.

Option 9

In a real square matrix:

a) implement its smoothing (a new matrix of the same size is obtained, each element of which is calculated as the arithmetic mean of the existing neighbors of the corresponding element of the original matrix);

b) in the smoothed matrix, find the sum of the modules of the elements located above the main diagonal.

Option 10

In an integer square matrix:

a) count the number of local minima (elements are strictly less than all its neighbors);

b) find the sum of the elements located below the main diagonal.

Option 11

In an integer square matrix:

a) compact its elements by removing rows and columns filled with zeros;

b) find the number of the first line containing at least one positive element.

Option 12

In an integer square matrix, implement a cyclic shift of the elements of a rectangular matrix by *k* elements to the right || left || up || down.

Option 13

In a 4x4 integer rectangular matrix, find k such that the kth row of the matrix coincides with the kth column.

Option 14

In a 5x10 integer matrix, remove the first || the last columns containing only positive elements. The matrix is filled with random values.

Option 15

In an integer rectangular matrix, define:

a) the number of columns that do not contain a single zero element;

b) rearranging the rows of a given matrix, arrange them in accordance with the growth of the sums of its positive even elements.

Option 16

In an integer square matrix, order the rows in increasing order of the number of identical elements in each of them.

Option 17

In an integer square matrix, by rearranging its elements, ensure that its maximum element is in the upper left corner, the next largest is in position (2,2), the next largest is in position (3,3), etc. thus filling the entire main diagonal.

Option 18

In an integer rectangular matrix, define:

a) the line with the smallest sum of elements;

b) the maximum among the sums of diagonal elements parallel to the main diagonal of the matrix.

Option 19

In an integer square matrix, define:

a) the sum of row elements that do not contain negative elements;

b) the maximum among the sums of elements of diagonals parallel to the main diagonal.

Option 20

In an integer rectangular matrix, define:

a) the number of lines that do not contain a single positive element;

b) the maximum of the numbers that appear more than once in a given matrix.

Practical work No. 7. Assignments for independent work on the topic “Processing of characters and strings”

Option 1

Given a text string S, which contains several consecutive digits. Get the number formed by these digits.

Option 2

Given a text string S, consisting of 10 digits. Find the sum of all numbers in it.

Option 3

Given text strings S and S0. Remove substrings from the string S that match S0. If there are no matching substrings, report this.

Option 4

Given a text string S, consisting of 5 words. Fill an array of 5 elements of string type with them.

Option 5

Given text strings S and S0. Remove substrings from the string S that match S0. If there are no matching substrings, report this.

Option 6

Given a text string S. Swap the first and last word in it.

Option 7

Given text strings S, S1, and S2. Replace occurrences of string S1 in string S with string S2.

Option 8

Given a text string S. Find the lengths of the shortest and longest words in the original string.

Option 9

Given a text string S. Print the substring located between the first and second space of the original string. If there are no gaps, report it.

Option 10

Given a text string S. Words are separated by one space. Determine the number of words and sentences in the text, as well as the number of vowels.

Option 11

Given a text string S and a character C. Find the positions where the character C appears in the string S. Insert a new string S0 at these found positions.

Option 12

Given a text string S. Print its words in descending order of their lengths.

Option 13

Given a text string S, consisting of 10 digits separated by spaces. Find the maximum and minimum numbers in it.

Option 14

Given text strings S and S0. Remove the second substring from S that matches S0. If there are no matching substrings, report this.

Option 15

Given a text string S. Print it in the reverse order of the words (for example, “Hello, dear friends” – “Hello, dear friends”).

Option 16

Given text strings S, S1 and S2. Replace all occurrences of string S1 in string S with string S2.

Option 17

Given text strings S and S0. Find the positions of the occurrence of the substring S0 in the string S. Find the number of such occurrences.

Option 18

Given a text string S. Find the number of words that begin and end with the same letter.

Option 19

A text string S is given in the format: last name, first name, patronymic, city and date of birth (for example, “Petrov Petrovich, Perm, April 12, 1961”). Convert the string to a new format with last name and initials (for example, “Petrov P.P.; Perm; 04/12/1961”).

Option 20

Given a text string S. Count the number of punctuation marks it contains.

Practical work No. 8. Assignments for independent work on the topic “Structures”

Option 1

Describe the “Employee” structure with the fields: pass number, full name, department name, salary. Enter employee information. Sort them alphabetically (by any key) Calculate the amount of salaries of all employees. Display the full name of the employee with the maximum salary.

Option 2

Describe the “Applicant” structure with the following fields: full name, year of birth, Unified State Exam scores (3), average certificate score. Organize data entry. Sort fields alphabetically (by any key). Remove items with scores below average. If the required records are not available, please report this.

Option 3

Describe the structure of the “Price List” with the fields: article number, product name, supplier, cost of the product. Organize data entry. Sort them alphabetically (by any key). Calculate the total cost of goods. Search for all products from a specific supplier. If the required entry is not available, please report it.

Option 4

Describe the “Data” structure with the fields: medium, volume, author. Organize data entry. Sort fields alphabetically (by any key). Remove the first element with a specified amount of information. If the required records are not available, please report this.

Option 5

Describe the structure of the “Notebook” with the fields: full name, phone number, address. Organize data entry. Sort them alphabetically (by any key). Search for a phone number using the entered query. If the required entry is not available, please report it.

Option 6

Describe the structure of the “Book” with fields : title, author, year of publication, number of pages. Organize data entry. Sort fields alphabetically (by any key). Implement a book search by author. If the required records are not available, please report this.

Option 7

Describe the “Order” structure with fields: order number, supplier, buyer, order contents (array). Organize data entry. Sort fields alphabetically (by any key). Search for all products included in an order from a specific customer. If the required entry is not available, please report it.

Option 8

Describe the structure “Complex number” with fields: real and imaginary parts. Organize data entry. Write functions that perform the operations of addition, subtraction and multiplication by a real constant.

Option 9

Describe the structure of the “Car” with the fields: make, registration number, year of manufacture. Organize data entry. Sort fields alphabetically (by any key). Display elements whose year of manufacture is less than the specified one. If the required records are not available, please report this.

Option 10

Describe the “Cartesian coordinates” structure with x, y fields. Organize data entry. Create a function to find the distance between two points.

Option 11

Describe the “Student” structure with the following fields: student ID number, full name, group, array of grades for 3 exams. Organize data entry. Sort them alphabetically (by any key). Calculate the grade point average for each student. Display the name of the student with the best grades.

Option 12

Describe the structure of the “State” with fields : name, capital, area, population. Organize data entry. Sort fields alphabetically (by any key). Output elements whose number is less than the specified one. If the required records are not available, please report this.

Option 13

Describe the structure of the “Vehicle Fleet” with the fields: flight number, destination, type of vehicle. Organize data entry. Sort them alphabetically (by any key). Search for the desired flight using the entered query. If the required entry is not available, please report it.

Option 14

Describe the “Patient” structure with the fields: full name, address, insurance policy number. Organize data entry. Sort fields alphabetically (by any key). Delete an item with a given insurance policy number. If the required records are not available, please report this.

Option 15

Describe the structure of the “Depot” with the fields: route number, array of car numbers, year of manufacture. Organize data entry. Sort them alphabetically (by any key). Search for the desired route using the entered query. If the required entry is not available, please report it.

Option 16

Describe the structure of a “Video file” with fields: film title, director, duration, price. Organize data entry. Sort fields alphabetically (by any key). Remove items with a price higher than the specified one. If the required records are not available, please report this.

Option 17

Describe the structure of the “Sports Team” with the fields: name, city, sport, number of victories. Organize data entry. Sort fields alphabetically (by any key). Remove all elements with a specific sport. If the required records are not available, please report this.

Option 18

Describe the structure of “Schoolchild” with the fields: full name, class, grades in subjects (mathematics, physics, computer science). Organize data entry. Sort fields alphabetically (by any key). Remove items that have an “unsatisfactory” rating in at least one subject. If the required records are not available, please report this.

Option 19

Describe the structure of "Organization" with the fields: name, city, tax identification number, income. Organize data entry. Sort fields alphabetically (by any key). Display elements whose income is less than the specified one. If the required records are not available, please report this.

Option 20

Describe the "Date" structure with fields: year, month, day. Organize data entry. Create a "days until the end of the month" function that calculates the number of days until the end of the month. Write a "is the date correct" function that checks the correctness of the entered date.

Practical work No. 9. Assignments for independent work on the topic “Processing text files”

Option 1

The text file contains information about employees: each line contains the last name, year of birth and city (for example, Petrov 1985 Perm). Output lines with a birth year greater than 1970 to another file.

Option 2

An unknown number of numbers are written in a column in a text file. Output them to another file, sorted in ascending order. Write down their sum in the last line.

Option 3

The text file contains data about goods and their costs. In each line, the first 5 positions are allocated for text (product name), and the remaining part is allocated for a real number. Create two new files: a string file named FILE 1, containing the text part of the source file (product names), and a file of real numbers named FILE 2, containing the numbers from the source file. The last line in FILE 2 is to write down their sum.

Option 4

The text file contains data (no more than 20 integers). Output the same numbers, sorted in descending order, as well as their product into another text file.

Option 5

Create a text file programmatically containing 2 tables of values of the functions sin(x), cos(x) on the interval [1, 20] with step 1.

Option 6

The data is stored in a text file. Replace all consecutive spaces with one space.

Option 7

The text file contains data about the goods: each line contains the name, production date and city. Output information about products with a production year older than the entered value into another file.

Option 8

Given two files named FILE 1 and FILE 2. Create a new text file named FILE 3, which is a concatenation of the contents of the original files. Also in FILE3 delete the first and last lines.

Option 9

Given a text file FILE 1. Copy into another file FILE 2 those lines in which the length is more than 10 characters.

Option 10

In a text file, delete the line with the given number N. If there is no such line, then report an error.

Option 11

In a text file, remove the first N characters from each line (N is user-defined).

Option 12

Given two text files named FILE 1 and FILE 2. Add the corresponding line FILE 2 to the end of each line of FILE 1.

Option 13

Given a text file. Write all even lines to the second file, and odd lines to the third file.

Option 14

Given a text file. Count the number of lines and characters in it. Count the number of characters in each line.

Option 15

Given a text file and a C symbol . Output all words starting with the character C to another file .

Option 16

Given a text file FILE 1. Create a new file FILE 2 containing all the punctuation marks from FILE 1 (in the same order).

Option 17

Given a text file FILE 1. Create a new file FILE 2 containing all the characters from FILE1 (without repetitions).

Option 18

Given a text file FILE 1. Find the number and length of the longest line, and print it. If there are several such lines, then print the first one.

Option 19

Given a text file FILE1 containing 5 lines. Rewrite each of its lines into an array.

Option 20

Given a text file FILE1 containing zeros and ones. Rewrite all its lines into another file FILE2, replacing the character 0 in them with character 1 and vice versa.

Practical work No. 10. Assignments for independent work on the topic “Array Container”

Option 1

Given a one-dimensional dynamic array consisting of N real elements. Calculate: number of array elements equal to 0; the sum of the array elements located after the minimum element. Arrange array elements in ascending order of element modules. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 2

Given a one-dimensional dynamic array consisting of N real elements. Calculate: product of positive array elements; the sum of the array elements located up to the minimum element. Separately arrange in ascending order the elements in even-numbered places and the elements in odd-numbered places. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 3

Given a one-dimensional dynamic array consisting of N integer elements. Calculate: the minimum absolute element of the array; the sum of the modules of the array elements located after the first element, which is equal to zero. Transform the array so that the first half contains elements in even positions, and the second half contains elements in odd positions. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 4

Given a one-dimensional dynamic array consisting of N real elements. Calculate: number of the minimum element of the array; the sum of the array elements located between the first and second negative elements. Transform the array so that all elements whose modulus does not exceed 5 are placed first, and then all the rest. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 5

Given a one-dimensional dynamic array consisting of N real elements. Calculate: number of array elements greater than K ; the product of the array elements located after the element with the largest modulo element. Transform the array so that all negative elements are placed first, and then all positive ones (elements equal to 0 are considered positive). Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 6

Given a one-dimensional dynamic array consisting of N real elements. Calculate: product of negative array elements; the sum of the positive array elements up to the maximum element. Perform inversion of array elements. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 7

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the number of the maximum modulo element of the array; the sum of the array elements located after the first positive element. Transform the array in such a way that first all elements are located, the whole part of which lies in the interval [a, b], and then all the rest. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 8

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the maximum modulo element of the array; the sum of the array elements located between the first and second positive elements. Transform the array so that elements equal to zero are placed after all others. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 9

Given a one-dimensional dynamic array consisting of N real elements. Calculate: number of array elements less than K ; the sum of the integer parts of the array elements located after the last negative element. Transform the array so that all elements that differ from the maximum by no more than 20% are placed first, and then all the rest. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 10

Given a one-dimensional dynamic array consisting of N integer elements. Calculate: the product of array elements with odd numbers; the sum of the array elements located between the first and last zero elements. Transform the array so that all positive elements

are placed first, and then all negative ones (elements equal to 0 are considered positive). Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 11

Given a one-dimensional dynamic array consisting of N integer elements. Calculate: the number of positive array elements; the sum of the array elements located after the last element, which is zero. Transform the array so that all the elements whose integer part does not exceed 1 are placed first, and then all the rest. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 12

Given a one-dimensional dynamic array consisting of N integer elements. Calculate: number of the maximum array element; the product of the array elements located between the first and second zero elements. Transform the array so that the first half contains elements in odd positions, and the second half contains elements in even positions. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 13

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the sum of negative array elements; the product of array elements located between the maximum modulo and minimum modulo elements. Sort array elements in descending order. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 14

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the number of the minimum absolute value of the array element; the sum of the modules of the array elements located after the first negative element. Compress the array by removing from it all elements whose value is in the interval [a, b]. Fill the vacated elements at the end of the array with zeros. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 15

Given a one-dimensional dynamic array consisting of N real elements. Calculate: minimum array element; the sum of the array elements located between the first and last positive elements. Transform the array so that all elements equal to zero are placed first, and then all the rest. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 16

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the number of array elements lying in the interval [a, b]; the sum of the array elements located after the maximum element. Sort array elements in descending order of element modules. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 17

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the sum of array elements with even numbers; the sum of the array elements located between the first and last negative elements. Compress the array by removing from it all elements whose modulus does not exceed 5. Fill the elements freed at the end of the array with zeros. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 18

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the number of negative array elements; the sum of the modules of the array elements located after the element with the minimum modulus. Replace all negative elements of the array with their squares and order the elements of the array in ascending order. Functional parts of the program are organized into subroutines; implement a dialog menu for selecting a demonstration of the operation of subroutines.

Option 19

Given a one-dimensional dynamic array consisting of N real elements. Calculate: the sum of positive array elements; the product of the array elements located between the maximum and minimum elements. Sort array elements in ascending order. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Option 20

Given a one-dimensional dynamic array consisting of N real elements. Calculate: maximum array element; the sum of the array elements up to the last positive element. Compress the array by removing from it all elements whose modulus is in the interval [a, b]. Fill the vacated elements at the end of the array with zeros. Functional parts of the program are organized into subroutines; implement a dialog menu to demonstrate the operation of subroutines.

Practical work No. 11. Tasks for independent work on the topic "Vector Container"

Option 1

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the maximum positive element.
- Calculate the product of array elements.
- Bring positive elements to the screen.

Option 2

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum element.
- Calculate the sum of even array elements.
- Display negative elements in reverse order.

Option 3

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the minimum positive element.
- Calculate the sum of even array elements.
- Display the array in reverse order.

Option 4

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum negative element.
- Calculate the product of non-zero array elements that are multiples of 3.
- Display negative elements in reverse order.

Option 5

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum negative element.
- Calculate the arithmetic mean of even array elements.
- Display non-zero elements in reverse order.

Option 6

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the minimum positive element.
- Calculate the product of odd array elements.
- Display negative elements on screen.

Option 7

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the minimum positive element.
- Calculate the sum of positive array elements that are multiples of 3.
- Display non-null elements on the screen.

Option 8

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum element.
- Calculate the product of non-zero array elements.
- Display positive elements in reverse order.

Option 9

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum element.
- Calculate the arithmetic mean of negative array elements.
- Display the array in reverse order.

Option 10

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum element.
- Calculate the arithmetic mean of array elements.
- Display the array in reverse order.

Option 11

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum element.
- Calculate the sum of array elements.
- Bring positive elements to the screen.

Option 12

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum negative element.
- Calculate the product of negative array elements.
- Display non-zero elements in reverse order.

Option 13

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum negative element.
- Calculate the arithmetic mean of the positive elements of the array.
- Bring positive elements to the screen.

Option 14

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the maximum element.
- Calculate the arithmetic mean of odd array elements.
- Display negative elements on screen.

Option 15

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum element.
- Calculate the arithmetic mean of the positive elements of the array.
- Display negative elements in reverse order.

Option 16

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum positive element.
- Calculate the product of non-zero array elements.
- Display non-zero elements in reverse order.

Option 17

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Enter the array from the keyboard.
- Find the maximum negative element.
- Calculate the sum of negative array elements.
- Bring positive elements to the screen.

Option 18

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the minimum element.
- Calculate the sum of the positive odd elements of an array.
- Bring positive elements to the screen.

Option 19

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the maximum positive element.
- Calculate the sum of array elements.
- Display non-zero elements in reverse order.

Option 20

Given a vector (variable size array). Using, whenever possible, operations (methods) of the type (class) Vector, do the following:

- Fill the array with random numbers.
- Find the maximum element.
- Calculate the arithmetic mean of negative array elements.
- Display positive elements in reverse order.

Practical work No. 12. Assignments for independent work on the topic "List"

In the job variants below, the `TNode type` has the following structure:

```
struct TNode // "node" structure
{
intData; // data field
TNode *Next; // pointer to next element
TNode *Prev; // pointer to previous element
};
```

If necessary, you can create a Pointer Node data type.

```
typedef TNode* PNode;
```

Option 1

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). Display the values of the Data fields of the subsequent and previous record specified at address P, as well as the addresses of the subsequent and previous record.

Option 2

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There are pointers to the first and last elements of a doubly linked list. Duplicate the first and last elements in the list (add new elements before existing elements with the same values) and display a pointer to the first element of the transformed list.

Option 3

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Duplicate all elements with odd numbers in the list (add new elements before existing elements with the same values) and display a pointer to the first element of the transformed list.

Option 4

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Remove all elements with odd values from the list and print a pointer to the first element of the transformed list (if, as a result of removing elements, the list turns out to be empty, then print NULL). After removing elements from the list, free the memory they occupied.

Option 5

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a number D and a pointer to one of the elements of a non-empty doubly linked list. Insert a new element with the value D after the given list element and display a pointer to the added list element.

Option 6

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to one of the elements of a non-empty doubly linked list. Move the given element to the end of the list and print pointers to the first and last elements of the transformed list. Do not use memory allocation and release operations, do not change Data fields.

Option 7

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a number D and a pointer to one of the elements of a non-empty doubly linked list. Insert a new element with the value D before this list element and display a pointer to the added list element.

Option 8

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the beginning of a non-empty chain of record elements of type TNode, interconnected using the Next field. Print a pointer to the middle and last element of the chain.

Option 9

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There are pointers to the first and last elements of a doubly linked list containing at least two elements. Duplicate the first and last elements in the list (add new elements after existing elements with the same values) and display a pointer to the last element of the transformed list.

Option 10

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to one of the elements of a non-empty doubly linked list. Print the number N – the number of elements in the list, as well as pointers to the first and last elements of the list.

Option 11

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a doubly linked list containing at least two elements. Remove all odd-numbered elements from the list and display a pointer to the first element of the converted list. After removing elements from the list, free the memory they occupied.

Option 12

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Duplicate all elements with odd values in the list (add new elements before existing elements with the same values) and display a pointer to the first element of the transformed list.

Option 13

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Duplicate all elements with odd values in the list (add new elements after existing elements with the same values) and display a pointer to the last element of the transformed list.

Option 14

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There are numbers D ata 1 and D ata 2 and a pointer to one of the elements of a non-empty doubly linked list. Add a new element with the value D ata 1 to the beginning of the list , and a new element with the value D ata 2 to the end. Print the addresses of the first and last elements of the resulting list.

Option 15

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a positive number K and a pointer to one of the elements of a non-empty doubly linked list. Move this element back in the list K positions (if there are less than K elements before this element, then move it to the beginning of the list). Print pointers to the first and last elements of the converted list.

Option 16

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to one of the elements of a non-empty doubly linked list. Remove this element from the list and print two pointers: to the element preceding the deleted one, and to the element following the deleted one (one or both of these elements may be missing; for missing elements, print NULL) . After removing an element from the list, free the memory occupied by this element.

Option 17

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.).
There is a pointer to one of the elements of a non-empty doubly linked list. Move the given element to the beginning of the list and display pointers to the first and last elements of the transformed list. Do not use memory allocation and release operations, do not change Data fields.

Option 18

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a positive number K and a pointer to one of the elements of a non-empty doubly linked list. Move this element forward in the list K positions (if there are less than K elements after this element, then move it to the end of the list). Print pointers to the first and last elements of the converted list. Do not use memory allocation and release operations, do not change Data fields.

Option 19

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Rearrange its elements by moving all odd-numbered elements to the end of the list (in the same order) and output a pointer to the first element of the rearranged list.

Option 20

Describe the TNode type as an element of a chain of connected nodes (provide the necessary functions: adding nodes, printing list elements, etc.). There is a pointer to the first element of a non-empty doubly linked list. Duplicate all elements with odd numbers in the list (add new elements after existing elements with the same values) and display a pointer to the last element of the transformed list .

Practical work No. 13. Assignments for independent work on the topic “Stack”

In the job options below, `the Stack type` can have the following structure (modeling using a one-dimensional array):

```
#define NMAX 5
struct TStack {
 int elem[NMAX];
 int top;
};
```

Where

`NMAX` – maximum number of elements in the stack;

`elem` – an array of NMAX numbers of type int , intended for storing stack elements;

`top` – index of the element located at the top of the stack (the first element of the chain).

Option 1

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of the stack. Pop all elements from the stack and print their values. Also print the number of elements retrieved N (for an empty stack print 0). After popping elements from the stack, free the memory they occupied.

Option 2

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). Create 3 new stacks, moving all elements of the original stack with odd values into the first one, with even values (except zero ones) into the second one, and zeros into the third one. Print the addresses of the vertices of the resulting stacks (for an empty stack, print NULL).

Option 3

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of the stack. A one-dimensional array has been created. Add elements of a one-dimensional array to the stack and print the address of the new top of the stack.

Option 4

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of a stack containing at least ten elements. Pop the first nine elements from the stack and print their values. Also print the address of the new top of the stack. After popping elements from the stack, free the memory they occupied.

Option 5

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There are pointers to the tops of two non-empty stacks. Move all elements from the first stack to the second (as a result, the elements of the first stack will be located on the second stack in the reverse order of the original one) and print the address of the new top of the second stack.

Option 6

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of the stack. Using the TStack type, describe the functions StackIsEmpty (S) of a logical type (returns TRUE if the stack S is empty, and FALSE otherwise) and Peek(S) of an integer type (returns the value of the top of a non-empty stack S without removing it from the stack). In both functions, the variable S is an input parameter of type TStack. Using these functions, as well as the Pop function, extract 3 elements from the original stack (or all elements contained in it, if there are less than five) and print their values. Also print the value of the StackIsEmpty function for the resulting stack and, if the resulting stack is not empty, the value and address of its new top.

Option 7

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There are pointers to the tops of two non-empty stacks. Move elements from the first stack to the second until the top of the first stack is even (the moved elements of the first stack will be placed on the second stack in the reverse order of the original). If there are no elements with even values in the first stack, then move all elements from the first stack to the second. Print the addresses

of the new vertices of the first and second stack (if the first stack is empty, print the constant NULL for it).

Option 8

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of a non-empty stack. Create two new stacks, moving all the even-valued elements of the original stack into the first one, and all the odd-valued elements into the second (the elements in the new stacks will be arranged in the reverse order of the original; one of these stacks may be empty). Print the addresses of the vertices of the resulting stacks (for an empty stack, print NULL).

Option 9

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). Using the pop function, pop 10 elements from the original stack and print their values. Also output a pointer to the new top of the stack.

Option 10

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There are N numbers. Create a stack containing the original numbers (the last number will be the top of the stack), output a pointer to the top and a pointer to the middle element.

Option 11

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of a non-empty stack. Pop the first (top) element from the stack and print its value D ata , as well as the address of the new top of the stack. If after retrieving the element the stack is empty, then put the value NULL in the address . After popping an element from the stack, free the memory occupied by that element.

Option 12

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack, print the address of the new top of the stack. Describe the function for finding the maximum value on the stack.

Option 13

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack, print the address of the new top of the stack. Describe the function for finding the minimum value on the stack.

Option 14

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack, print the address of the new top of the stack. Describe a function that implements a search for even stack elements and writes them into a one-dimensional array.

Option 15

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack, print the address of the new top of the stack. Describe a function that implements the search for stack elements larger than K and copying them into a one-dimensional array.

Option 16

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack, print the address of the new top of the stack. Describe a function that implements a search for stack elements greater than the average value and writes them into a one-dimensional array.

Option 17

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of a non-empty stack. Pop the bottom and penultimate elements of the stack from the stack and print their D ata values and addresses.

Option 18

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of a non-empty stack. Pop the maximum and middle elements of the stack from the stack and print their D ata values and addresses.

Option 19

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a pointer to the top of the stack. A one-dimensional array has been created. Write elements from the stack into a one-dimensional array. The stack should remain empty.

Option 20

T Stack type (provide the necessary functions: adding an element to the top of the stack, removing an element from the top of the stack, etc.). There is a number D ata and a pointer to the top of the stack. Add an element with the value D ata to the stack and print the address of the new top of the stack.

Practical work No. 14. Tasks for independent work on the topic “Queue”

Option 1

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. Using the TQueue type, describe a boolean function Empty(Q) that returns TRUE if the queue is empty and FALSE otherwise – an input parameter of type TQueue). Using this function to check the

state of the queue, as well as the function to append elements to the end, extract five elements from the original queue and print their values. Also display the result of calling the Empty() function for the resulting queue and the new addresses of its beginning and end.

Option 2

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are pointers to the beginning and end of a non-empty queue. Retrieve elements from the queue until the value of the starting element of the queue is even, and print the values of the retrieved elements (if the queue does not contain elements with even values, then retrieve all its elements). Also print the new addresses of the beginning and end of the queue.

Option 3

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are a set of 10 numbers. Create two queues: the first should contain numbers from the original set with odd numbers (1, 3, 5, ..., 9), and the second – with even numbers (2, 4, 6, ..., 10); the order of the numbers in each queue must match the order of the numbers in the original set. Print pointers to the beginning and end of the first and second queues.

Option 4

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues, the start and end addresses of the first and second. Move all elements of the first queue (in order from start to end) to the end of the second queue and print the new addresses of the beginning and end of the second queue.

Option 5

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front)

is the first element of the chain, the end (last) is its last element. There is a number K and two queues. Move the K starting elements of the first queue to the end of the second queue. If the first queue contains less than K elements, then move all elements from the first queue to the second. Print the new addresses of the beginning and end of the first and then the second queue.

Option 6

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a set of 10 numbers. Create a queue containing the given numbers in the specified order (the first number will be placed at the beginning of the queue, the last at the end), and display pointers to the beginning and end of the queue.

Option 7

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues. Queues contain the same number of elements. Combine two queues into one, in which the elements of the original queues are interleaved (starting with the first element of the first queue). Print pointers to the beginning and end of the resulting queue.

Option 8

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues, pointers to the beginning and end of a queue containing at least 10 elements. Extract five initial elements from both queues and write them into a 2 by 5 matrix.

Option 9

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues. The elements of each queue are ordered in ascending order. Combine queues into one

while maintaining the order of elements. Print pointers to the beginning and end of the resulting queue.

Option 10

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a set of 10 numbers. Create two queues: the first should contain all odd numbers, and the second – all even numbers from the original set (the order of the numbers in each queue must match the order of the numbers in the original set). Print pointers to the beginning and end of the first and then the second queue (one of the queues may be empty; in this case, print two nil constants for it).

Option 11

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a number K and pointers to the beginning and end of the queue. Add an element with value K to the end of the queue and print the new addresses of the beginning and end of the queue.

Option 12

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a number K and pointers to the beginning and end of a queue containing at least two elements. Add an element with value K to the end of the queue and remove the first (starting) element from the queue. Print the value of the extracted element and the new addresses of the beginning and end of the queue.

Option 13

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a number K and pointers to the beginning and end of the queue. Extract K initial elements from the queue and display their values (if the queue contains less than K elements, then extract all its

elements). Also print the new addresses of the beginning and end of the queue (for an empty queue, print NULL twice).

Option 14

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues. It is necessary to move elements from the beginning of the first queue to the end of the second until the value of the initial element of the first becomes even (if the first queue does not contain even elements, then move all elements from the first queue to the second). Print the new addresses of the beginning and end of the first and then the second queue.

Option 15

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are three queues. The elements of each queue are not ordered. Combine queues into one and sort its elements. Print pointers to the beginning, middle, and end of the resulting queue.

Option 16

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is a number K and three queues. Move K initial elements of the first queue to the end of the second and third queue. If the first queue contains less than K elements, then move all elements from the first queue to the second and third. Print the new start and end addresses of all queues.

Option 17

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There is an array of 5 numbers. Create a queue and write the elements of this array there in the specified order (the first number will be placed at the beginning of the queue, the last at the end) and display pointers to the beginning, middle and end of the queue.

Option 18

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues. Queues contain different numbers of elements. Combine two queues into one, in which the elements of the original queues alternate (starting from the first element of the first queue), placing the remaining elements at the end. Print pointers to the beginning and end of the resulting queue.

Option 19

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are three queues (5 elements each), pointers to the beginning and end of the queues. Extract five initial elements from the queues and write them into a 3 by 5 matrix.

Option 20

There is a queue structure (type TQueue), which is modeled by a chain of connected nodes (an element with a Data field and a Next pointer). The beginning of the queue (front) is the first element of the chain, the end (last) is its last element. There are two queues. The elements of each queue are ordered in descending order. Combine queues into one while maintaining the order of elements. Print pointers to the beginning, middle, and end of the resulting queue.

Practical work No. 15. Assignments for independent work on the topic “Binary trees”

Option 1

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Find, using the CLP tree traversal method (root-left-right), the number of elements with a given key.

Option 2

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Find, using the LKP (left-root-right) tree traversal method, the maximum element in the tree.

Option 3

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LKP (left-root-right) tree traversal method, find the number of leaves in the tree.

Option 4

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the RLP (root-left-right) tree traversal method, find the minimum element in the tree.

Option 5

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LKP (left-root-right) tree traversal method, find the height of the tree.

Option 6

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LKP (left-root-right) tree traversal method, find the arithmetic mean of the tree elements.

Option 7

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LPK (left-right-root) tree traversal method , find the number of tree elements starting with a given symbol.

Option 8

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the CLP (root-left-right) tree traversal method, find the number of elements with a given key.

Option 9

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LKP (left-root-right) tree traversal method, find the maximum element in the tree.

Option 10

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LRC (left-right-root) tree traversal method, find the number of leaves in the tree.

Option 11

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the RLP (root-left-right) tree traversal method, find the minimum element in the tree.

Option 12

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Information field type char. Using the LKP (left-root-right) tree traversal method, find the height of the tree.

Option 13

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Information field type int. Using the LPK (left-right-root) tree traversal method, find the arithmetic mean of the tree elements.

Option 14

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Information field type char. Using the CLP (root-left-right) tree traversal method, find the number of elements with a given key.

Option 15

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Information field type string. Using the LKP (left-root-right) tree traversal method, find the number of tree elements starting with a given symbol.

Option 16

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LRC (left-right-root) tree traversal method, find the maximum element in the tree.

Option 17

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the root-left-right tree traversal method, find the number of leaves in the tree.

Option 18

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LKP (left-root-right) tree traversal method, find the minimum element in the tree.

Option 19

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the LPK (left-right-root) tree traversal method , find the arithmetic mean of the tree elements.

Option 20

Describe a model of the structure of a binary tree. Implement functions for adding, deleting and printing elements. Using the CLP tree traversal method (root-left-right), find the number of elements with a given key

BIBLIOGRAPHY

1. Algorithms and programs. Language C ++ / E.A. Konova, G.A. Pollak. – St. Petersburg: Lan, 2017. – 384 p.
2. Algorithms: development and application. Classic Computers Science / J. Kleinberg, E. Tardos. – St. Petersburg: Peter, 2016. – 800 p.
3. Androsova E.G. Methodological and content aspects of constructing a programming course based on an object-oriented approach (for physics and mathematics specialties of pedagogical universities): dis. ...cand. ped. Sci. – M., 1996.
4. Bordyugova T.N. Methodological approaches to the formation of competencies in the field of programming based on the implementation of an individual learning path: dis. ...cand. Ped.Sc. – M., 2011. – 141 p.
5. C ++ programming language [Electronic resource] / Stanley Lippman, Josi Lajoye. – Saratov: Vocational Education, 2017. – 1104 p. // EBS IPRbooks – URL : http://www.iprbook shop.ru/63964.html (access date: 09/01/2017).
6. C programming . – M.: Eksmo, 2013. – 624 p.
7. C++ programming language. Basic Course / Stanley B. Lippman, Josie Lajoie, Barbara E. Mu. – M.: Williams, 2017. – 1397 p.
8. Clemens B. Language C in the 21st century. – M.: DMK Press, 2015. – 376 p.
9. Computer science (basic level): 2 hours, grades 10–11. Part 2: textbook. / ed. N.V. Makarova. – M.: BINOM. Knowledge Laboratory, 2019. – 368 p.
10. Computer science. 10–11 grades. Basic level: textbook: in 2 hours. Part 1 / ed. prof. N.V. Makarova. – M.: BINOM. Knowledge Laboratory, 2016. – 384 p.
11. Computer science. 10th grade: textbook. / L.L. Bosova, A.Yu. Bosova. – M.: BINOM. Knowledge Laboratory, 2016. – 288 p.
12. Computer science. Advanced level: textbook. for 10th grade / I.A. Kalinin, N.N. Samylkina. – M.: BINOM. Knowledge Laboratory, 2013.
13. Computer science. Advanced level: textbook. for 11th grade / I.A. Kalinin, N.N. Samylkina. – M.: BINOM. Knowledge Laboratory, 2013.
14. Computer science. Advanced level: textbook. for grade 10: at 2 hours. Part 1 / K.Yu. Polyakov, E.A. Eremin. – M.: BINOM. Knowledge Laboratory, 2013. – 344 p.
15. Computer science. Advanced level: textbook. for grade 10: at 2 hours. Part 2 / K.Yu. Polyakov, E.A. Eremin. – M.: BINOM. Knowledge Laboratory, 2013. – 304 p.
16. Computer science. Advanced level: textbook. for grade 11: at 2 hours. Part 1 / K.Yu. Polyakov, E.A. Eremin. – M.: BINOM. Knowledge Laboratory, 2013. – 240 p.

17. Computer science. Advanced level: textbook. for grade 11: at 2 hours. Part 2 / K.Yu. Polyakov, E.A. Eremin. – M.: BINOM. Knowledge Laboratory, 2013. – 304 p.

18. Computer science. Grade 11. Basic level: textbook. / L.L. Bosova, A.Yu. Bosova. – M.: BINOM. Knowledge Laboratory, 2016. – 256 p.

19. Davis S. C++ for dummies. – M.: William c , 2003. – 336 p.

20. Demonstration version of control measurement materials of the Unified State Exam (USE) 2019 in computer science and ICT [Electronic resource] / Federal. institute of ped. measurements. – URL: http://fipi.ru/ege-i-gve-11/demoversii-specifikacii-kodifikatory

21. Demonstration version of control measuring materials for conducting the main state exam (OGE) in computer science and ICT in 2019 [Electronic resource] / Feder. institute of ped. measurements. – URL: http://fipi.ru/oge-i-gve-9/demoversii-specifikacii-kodifikatory

22. Federal state educational standard of secondary (complete) general education (dated May 17 2012 г, No. 413). – 45 s.

23. Flenov M.E. Programming in C ++ through the eyes of a hacker. – St. Petersburg: BHV-Petersburg, 2009. – 352 p.

24. Informatics and ICT: textbook. for 10th grade general education institutions: basic and profile. levels / A.G. Gein, A.B. Livchak, A.I. Senokosov and others – 2nd ed. – M.: Education, 2012. – 272 p.

25. Ivanov V.B. Application programming in C / C ++. From scratch to multimedia and network applications [Electronic resource]. – M.: SOLON-PRESS, 2008. – 240 p. // EBS IPRbooks. – URL : http://www.iprbookshop.ru/8727 (access date: 09/01/2017).

26. Kaufman V.Sh. Programming languages. Concepts and principles [Electronic resource]. – Saratov: Vocational Education, 2017. – 464 p. // EBS IPRbooks. – URL : http://www.iprbook shop.ru/64055.html (access date: 09/01/2017).

27. Kirillov A. G. Formation of professional competencies of a future computer science teacher in the process of teaching programming: abstract. dis. ...cand. ped. Sci. – Ekaterinburg, 2005. – 22 p.

28. Kolpien J. Programming in C ++. Classic CS . – St. Petersburg: Peter, 2005. – 479 p.

29. Koltsov D.M. 100 examples in C. – St. Petersburg: Science and Technology, 2017. – 256 p.

30. Kostyukova N.I. Programming in C language [Electronic resource]. – Novosibirsk: Sibir. university. publishing house, 2017. – 160 p. // EBS IPRbooks. – URL : http://www.iprbook shop.ru/65289.html (access date: 09/01/2017). –

31. Kugel L.A. Teaching students algorithmization and programming based on a structural-algorithmic approach to the formulation and implementation of problems: using the example of the direction of bachelor's training "Applied Informatics": abstract of thesis. dis. ...cand. ped. Sci. – M., 2015. – 22 p.

32. Laforet R. Object-oriented programming in C ++. – St. Petersburg: Peter, 2004. – 922 p.

33. Meshcheryakova N.A. Formation of information competence of students of economic specialties of universities when teaching object-oriented programming: abstract of thesis. dis. ...cand. ped. Sci. – Omsk, 2005. – 22 p.

34. Pavlovskaya T.A. C/C++. Programming in a high level language. – St. Petersburg: Peter, 2010. – 461 p.

35. Perry Gr. C Programming for Beginners. – M.: Eksmo, 2015. – 368 p.

36. Prata St. C programming language . Lectures and exercises. – M.: Williams, 2015. – 928 p.

37. Prata St. C++ programming language. Lectures and exercises. – M.: Williams, 2012. – 1248 p.

38. Programming language C / B. Kernighan, D. Ritchie. – M.: Williams, 2009. – 304 p.

39. Rao S. Master C ++ on your own in 21 days. – M.: Williams, 2013. – 651 p.

40. Retabouil S. Android NDK. Android application development using C / C. – M.: DMK Press, 2012. – 496 p.

41. Shildt G. C++: basic course. – M.: Williams, 2010. – 624 p.

42. Sofronova N.V. Theory and methodology of teaching computer science. – M.: Higher. school, 2003. – 186 p.

43. Spirin I.S. Electronic training course as a means of activating educational and cognitive activity when teaching programming to future computer science teachers: dis. ...cand. ped. Sci. – Shadrinsk, 2004. – 179 p.

44. Standard for Programming Language C++. ISO/IEC . – 2017. – No. 4687 . – 1647 rub .

45. Stolyarov A.V. Introduction to the C language. – M.: MAKS Press, 2018. – 136 p.

46. Stroustrup B. Design and evolution of the C++ language [Electronic resource]. – M.: DMK Press, 2008. – 448 p. // EBS IPRbooks. – URL : http://www.iprbookshop.ru/7784 (access date: 09/01/2017).

47. Stroustrup B. Programming: principles and practice using C++. – M.: Williams, 2016. – 1328 p.

48. Theory and methodology of teaching computer science / M.P. Lapchik, I.G. Semakin, E.K. Henner; under general ed. M.P. Lapchika. – M.: Academy, 2008. – 592 p.

49. Tolstova N.S. Bimodularity as a condition for building adaptive methodological systems for teaching programming: abstract of thesis. dis. ...cand. Ped.Sc. – Ekaterinburg, 2005. – 24 p.

50. Vasiliev A.N. C ++ tutorial with examples and tasks. – St. Petersburg: Science and Technology, 2016. – 480 p.

51. Vasiliev A.N. Object-oriented programming in C ++ [Electronic resource]. – St. Petersburg: Science and Technology, 2016. – 544 p. // EBS IPRbooks. – URL : http://www.iprbook shop.ru/60648.html (access date: 09/01/2017).

52. Vasiliev A.N. Programming in C ++ with examples and problems. – M.: E, 2017. – 368 p.

53. Vstavskaya E.V. Electronic training course “Programming” [Electronic resource]. – URL : https://prog-cpp.ru (access date: 09/01/2017).

54. Williams E. Parallel programming in C++ in action. Practice of developing multi-threaded programs. – M.: DMK Press, 2012. – 672 p.

55. Zhuzhalov V.E. Specifics of teaching programming in the preparation of computer science students // Vestn. MSLU. Ser. Informatization of education. – 2004. – No. 1. – P. 56–61.

56. Ziborov V.V. MS Visual C++ 2010 in the .NET environment. Programmer's library. – St. Petersburg: Peter, 2012. – 320 p.

Учебное издание

Ильин Иван Вадимович

Fundamentals and methodology of programming. Procedurally oriented programming in C++

Учебное пособие

Издается в авторской редакции
Компьютерная верстка: *И. В. Ильин*

Объем данных 1,97 Мб
Подписано к использованию 25.01.2024

Размещено в открытом доступе
на сайте www.psu.ru
в разделе НАУКА / Электронные публикации
и в электронной мультимедийной библиотеке ELiS

Управление издательской деятельности
Пермского государственного
национального исследовательского университета
614068, г. Пермь, ул. Букирева, 15