Noncommercial
Joint Stock
Community

ALMATY UNIVERSITY OF POWER ENGINEERING AND TELECOMMUNICATIONS

Department of engineering cybernetics

TECHNOLOGIES OF PROGRAMMING

Lecture notes for specialty
5B070200 - Automation and control

Almaty 2017

COMPILED: I.T. Utepbergenov, Sh.N. Sagyndykova. Technologies of programming. Lecture notes for specialty 5B070200 - Automation and control. - Almaty AUPET, 2017. - 62 p.

The lecture notes are compiled to help the second-year students in the study of theoretical material on software engineering, and includes fifteen themes. At the end of each topic there are references to additional sources for more in-depth exploration of the subject. The annexes provide the necessary background and illustrations.

Lecture Notes for students of specialty 5B070200 - Automation and control.

Ill-5, table-0, Bibliography -12.

Reviewer: candidate of economic sciences, associate professor Tuzelbaev B.I.

Published according to the plan of publication of the non-profit joint-stock company "Almaty university of power engineering and telecommunications" for 2017.

©NJSC"Almaty university of power engineering and telecommunications", 2017.

Lecture №1. Programming Technologies. Concepts and approaches. Classification of the software

The goal - to get an idea of the modern classification of software, the term "software engineering" as well as the main stages of their development.

Software is a set of programs allowing to implement automated processing of information on a computer, designed for repeated use and application by different users, as well as policy documents necessary for their operation.

Shareware is classified as a systemic (general) and applied (special).

Programs running on the system level, provide a basic level of interaction with other systems and applications directly to the hardware.

Software application layer is a set of application programs, performing specific tasks at the workplace.

Meanwhile, the unit in the system and application is somewhat outdated. Modern division provides at least three gradations of software: system, middleware and application.

The trend in software is to reduce the volume of both the system and application programming. Most of the work is performed in intermediate programmer software. The decline in system programming due to modern concepts of operating systems, object-oriented architecture and the microkernel architecture, under which most of the functions of the system shall be made in the tools that can be attributed to the middleware. The decline in application programming due to the fact that modern middleware products offer a growing set of tools and templates to meet the challenges of its class. A significant part of the system software and virtually all application software is written in high level languages. This provides a reduction of their development / modification and portability.

Middleware is defined as a set of programs that manage secondary resources focused on the solution of a certain class of problems. Such software includes transaction management, database servers, communications servers and other application servers. From the point of view of tools middleware is closer to the application because it does not work directly with the primary resource, and uses the service provided by the system software. From the standpoint of the development of algorithms and technologies it is closer to the system, as there is always a complex of multiple software products and multi-purpose use, which uses algorithms similar to those used in the system software.[1]

Programming Technology (PT) is a set of methods and tools used in software development. PT is a set of technological instructions:

- an indication of the sequence of technological operations;
- identification of the conditions in which operations are performed;
- a description of the operations themselves with the original data, the results, instructions, regulations, standards, criteria and evaluation methods.

Technology defines a way to describe the designed system (models) used at a particular stage of development. There are techniques used at specific stages of

development (defined method), and technology covering several stages of development (basic method or methodology).

Among the main stages in the development of programming technologies emit "natural" programming, structure, object and component approach to programming [1].

The stage of "natural" programming. A characteristic feature of the period since the first computers until the mid- 60-ies of XX century is that formulated programming technology was virtually absent, as well as the art of programming itself.

The first program had a simple structure: a program in a machine language and data processing. The complexity of the software in machine codes limited the ability of the programmer to mentally trace the sequence of operations performed and at the same time follow the location of data in the process of programming. We used the intuitive programming technology. The emergence of assemblers instead of binary codes enabled use of symbolic names for the data and the mnemonic operation codes, and programs became more "readable". It was in this period that the concept of modular programming of PT originated which later became fundamental, focused on overcoming the difficulties of programming in native code. High-level programming languages (FORTRAN, ALGOL) were created which greatly simplified the programming of calculations, reducing the level of detail of the operations, which increased the complexity of the programs.

As a result of the appearance of tools allowing to operate with subroutines, there were created huge libraries of calculating and servicing subprograms. A typical program consisted of a main program, a global data area, and a set of subroutines, but with an increase in the number of subroutines, the probability of distorting some of the global data by a certain subprogram increased, so it was suggested that local data be placed in them (figure 1.1), and the emergence of new subroutine support tools allowed to develop software for several programmers in parallel.

In the early 60-ies of the XX century, the "programming crisis" erupted: developers of complex software disrupted all deadlines for the completion of projects: the project became obsolete earlier than it was ready for implementation, its cost increased, so many projects were never completed. All this was caused by the imperfection of the programming technology. First of all, the "bottom-up" development was used. In the absence of clear models for describing subprograms and design methods, the creation of subroutines turned into a difficult task. The interfaces of the subroutines turned out to be complex, and when assembling the software product, a large number of errors of agreement were detected, the correction of which required a serious change in the already developed subprograms, which often entailed new errors. As a result, the process of testing and debugging programs took more than 80% of the development time, if it ever ended. Analysis of the causes of errors allowed us to formulate a new approach to programming - *structural.*

The structural approach to programming, which developed at the second stage of technology development in the 60-70 years of the XX century, is a set of recommended techniques that cover the completion of all stages of software

development. It is based on the *decomposition* of complex systems with the purpose of subsequent implementation in the form of separate small (up to 40-50 operators) subprograms, later referred to as *procedural* decomposition. The structural approach required the representation of the problem in the form of a hierarchy of subtasks of the simplest structure, and the design was implemented *"from top to bottom*" and implied the realization of a general idea. There were introduced restrictions on the design of algorithms, formal descriptions were recommended, as well as a special method for designing algorithms - a method of *stepwise refinement.*

The principles of structural programming were used as the basis for *procedural* programming languages, which included the basic *"structural" control transfer operators*, supported the nesting of subroutines, localization and limitation of the "visibility" area of data (PL/1, ALGOL-68, Pascal, C). Further growth in the complexity and size of the software being developed required the development of data structuring, in languages it becomes possible to define user-defined data types [2].

The desire to differentiate access to global program data gave impetus to the emergence and development of modular programming technology (Figure A.1), which involved distinguishing of subprogram groups that used the same global data in separately compiled modules (libraries). Links between the modules were implemented through a special interface, while access to the implementation of the module (to the bodies of sub-programs and some "internal" variables) was prohibited. This technology is supported by modern versions of Pascal, C, C ++, Ada and Modula. The development of programs by several programmers has become much simpler: the modules were developed independently of each other and could be used without changes in other developments, and their interaction was provided through specially specified intermodule interfaces.

Structural approach in combination with modular programming allows obtaining reliable programs, the size of no more than 100,000 operators. Disadvantage: an error in the interface when calling a subroutine is detected only when the program is executed (due to separate compilation of modules), and when the program size increases, the complexity of the inter-module interfaces increases, so it is practically impossible to provide for the mutual influence of individual parts of the program. For the development of large-scale software, it was suggested to use the *object approach.*

At the third stage (80th - 90th years of the XXth century) an *object approach* to programming was formed. The technology of creating complex software, based on the representation of the program in the form of a set of interacting program objects, each of which is an instance of a particular class (and the classes form a hierarchy with inheritance of properties), was called *object-oriented programming* (figure A.2). The object structure of the program was first used in the language of imitation modelling of complex Simula systems, and then used in new versions of universal programming languages such as Pascal, C ++, Modula, Java.

The advantage of object-oriented programming is the "natural" decomposition of software, which greatly facilitates the development. This results in more complete

localization of data and integration with the processing subprograms, which allows a virtually independent development of individual program objects. In addition, the object approach offers new ways of organizing programs based on inheritance mechanisms, polymorphism, composition, filling, which allow constructing complex objects from simple objects. As a result, the code reuse rate is significantly increased and it becomes possible to create class libraries.

On the basis of the object approach, environments that support visual programming (Delphi, C ++ Builder, Visual C ++) were created, with which some of the future product is designed using visual tools for adding and configuring special library components. As a result, there appears a project of the future program, into which the codes have already been entered. The use of the object approach has many advantages, but its specific implementation in object-oriented programming languages, such as Pascal and C ++, has significant drawbacks: there are no standards for linking the binary results of compiling objects into a single whole, even within the same programming language; a change in the implementation of one program object is associated with the recompilation of the module and the reconfiguration of all the software that uses this object. Thus, the objective dependence of software modules on the addresses of exported fields and methods, as well as structures and data formats, is preserved. Module communications can not be broken, but their interaction can be standardized, on which the *component approach* to programming is based.

The fourth stage (90th years of the XX century - our time) – a period of the component approach and CASE-technologies. A component approach involves building software from individual components that interact with each other through standardized binary interfaces. Unlike ordinary objects, the component objects can be assembled into dynamically called libraries or executable files, distributed in binary form (without source codes) and used in any programming language that supports the corresponding technology [1]. The component approach underlies the technologies developed on the basis of COM (Component Object Model - the component model of objects, and technologies for creating distributed applications CORBA (Common Object Request Broker Architecture), which use similar principles and differ only in features of implementation.

In addition, distinctive features of the current stage of program development technology are the creation and implementation of automated technologies for the development and maintenance of software, which were called CASE-technology (Computer-Aided Software / System Engineering - Software Development / software systems with computer support). Without automation tools it would be difficult to develop sufficiently sophisticated software at the moment: the memory of a man is not able to capture all the details that must be considered when developing software. CASE-technology supports both structural and object (component) approach to programming [1, 2].

Microsoft's COM technology is the development of technology OLE I (Object Linking and Embedding - Object Linking and Embedding), which was used in earlier versions of Windows to create compound documents. It defines the general concept of the interaction of all types of programs (libraries, applications, operating system),

i.e. allows a single piece of software to use the functions (services) provided by other programs (figure A.4).

Modification of the COM, providing the transmission of calls between computers, is named DCOM (Distributed COM). Based on COM technology and its distributed version DCOM component technologies have been developed, to solve various problems of software development. COM is based on the ability of technology COM +, which provides support for distributed component-based applications, and is designed to support transaction processing systems [1, 9].

Technologies implementing component-based approach inherent in the COM include:

a) OLE-automation - technology for creating programmable application that provides programmable access to their internal services (e.g, MS Excel supports it by providing its services to other applications);

b) ActiveX - technology that is based on OLE-automation and designed to create both focused on the same computer software, and distributed in the network. Suggests the use of visual programming for creating components - ActiveX control elements, which are installed on the computer remotely from a remote server and applied to client parts used in Internet applications;

c) MTS (Microsoft Transaction Server) - technology that provides a safe and stable operation of distributed applications with large volumes of data transmitted;

d) MIDAS (Multitier Distributed Application Server) - a technology that organizes access to data of different computers with regard of the network load balancing.

Technology CORBA, developed by a group of companies OMG (Object Management Group), implements a similar approach on the basis of COM objects and interfaces CORBA.

CORBA software core is implemented for all major hardware and software platforms, therefore the technology can be used to create a distributed software in a heterogeneous computing environment. Organization of interaction between client and server objects in CORBA is carried out through the intermediary (VisiBroker) and specialized software.

Unfortunately, for various reasons, developers were forced to abandon this technology [1, 2]. Today, CORBA is mainly used to link components running inside corporate networks, where communication is protected by a firewall from the outside world. CORBA technology is also used in the development of real-time systems and embedded systems sector in which CORBA is really developing. Overall, however, CORBA is in decline, and now it can not be called otherwise than a niche technology.

Lecture № 2. Features of the development of complex software systems

The goal - to get an idea about the principles of the development of complex software systems and their life cycle; learn the basic steps of a software development and their peculiarities.

Most of today's software systems are quite complex. This complexity is caused by many reasons, chief among which is the logical complexity of their tasks.

Earlier computers were used in very narrow fields of science and technology, especially where problems were well determined and required considerable computing. Now, when there are created powerful computer networks, it is possible to delegate to them the solution of complex intensive tasks, computerization of which was earlier hardly thought of. The computerization process involves new subject areas, and for the developed areas are set more complex tasks. The complexity of the development of software systems increases due to the complexity of the formal definition of the requirements for these systems, lack of satisfactory means of describing the behavior of discrete systems with a large number of states in the non-deterministic sequence of input actions, collaborative development, the need to increase the degree of repeatability of codes. However, all these factors are directly related to the complexity of the object of development - software system [1, 9].

The vast majority of complex systems has a hierarchical internal structure. Links of elements of complex systems vary both in type and in strength, and that allows to consider the systems as a certain set of interdependent subsystems. Internal ties of elements of such subsystems is stronger than the ties among the subsystems. For example, the computer consists of a CPU, memory and external devices, and the solar system includes the Sun and the planets revolving around it.

Using the same difference of relations, each subsystem can be partitioned similarly to the "elementary" level. At this level, the system consists of a few types of subsystems combined and organized in various ways. Hierarchy of this type is called "whole-part".

In nature, there is another type of hierarchy - a hierarchy of "simple-complex" or hierarchy of development (complication) of systems in evolution. In this hierarchy, any functioning system is the result of development of a simpler system. It is this kind of hierarchy the inheritance mechanism of object-oriented programming is realized.

As a reflection of the natural and technical systems, software systems are hierarchical and have the above described properties. These properties of hierarchical systems make for the block-hierarchical approach to their study or construction. At first this suggests creation of parts of the object (blocks and modules), and then assembling from them the object itself.

The process of breaking up of a complex object into relatively independent parts is called *decomposition*. In the process of decomposition it should be taken into account that the links between separate parts should be weaker than the links of elements inside the parts. In order to assemble the object, in the process of decomposition, it is necessary to define all kinds of relationships between the parts themselves.

When creating complex objects decomposition process is repeatedly executed: each block is, in turn, decomposed into parts until blocks are received, which are relatively easy to develop. This development method called *stepwise refinement*. In the process of decomposition it is desirable to determine analogous blocks that could

be developed on a general basis. Thus, it provides increased degree of code repeatability and reduced development cost.

The result of decomposition is usually represented as a hierarchy diagram, the lower part of which shows relatively simple blocks, the top represents the object to be developed [3]. At each hierarchical level description of the blocks is performed to a certain degree of specification, abstracting from insufficient details. Typically, for the whole object, only general requirements can be successfully formulated, and the blocks of the lower level must be specified so that they could really be assembled into a working object. In other words, the larger the block, the more abstract should be its description.

Subject to this principle, the developer retains the ability to understand the project and make the most appropriate decisions at every stage, what is called local optimization (unlike global optimization of the specifications of the objects, which is not always possible for really complex objects).

So, the block-hierarchical approach is based on a hierarchical ordering and decomposition. An important role is played by the following principles:

a) consistency - control of agreement of elements;

b) completeness – control of the presence of extra elements;

c) formalization – strictness of the methodological approach;

d) repeatability - the need for the same blocks to reduce the cost and accelerate the development;

e) local optimization - optimization within the hierarchy level.

The set of languages of models, problem statements, methods of description of a certain hierarchical level is called the *level* of design. Different views on the subject of design are called design *aspects*.

The advantages of block-hierarchical approach:

- simplification of verifying operability of separate blocks and the system as a whole;
- ability to create complex systems;
- providing possibility of upgrading the systems.

Using the block-hierarchical approach applied to software systems became possible only after specifying the general provisions of the approach and making changes in the design process. This approach takes into account only the structural properties of the hierarchy, "whole - part", and the object approach additionally uses properties of the hierarchy "simple - complex".

Software life cycle is the period from the onset of the idea of creating a software until the end of its support by developer or company that performs maintenance [1, 3].

The composition of the processes of life cycle is regulated by international standard ISO / IEC 12207: 1995 «Information Technology - Software Life Cycle Processes» («Information Technologies - Processes of the software life cycle"). The standard describes the structure of the software life cycle, names and determines its processes, without specifying the details of their implementation.

The process of life cycle is a set of interrelated operations that transform input data into output data. The figure A.5 shows the process of the life cycle of this standard. According to this structure, the processes of software development and maintenance are the main processes [4]. Conventionally, the following basic stages of software development are distiguished:

a) formulation of the problem – the purpose of the software is stated, and the basic requirements to it (functional and operational) are defined. The results are the terms of reference (TOR), retaining the fundamental requirements and the adoption of the basic design decisions;

b) analysis of requirements and design specifications - requirements analysis is performed, meaningful statement of the problem is formulated, the domain model is constructed, subtasks are defined and methods for their solution are selected or developed, as well as tests for finding errors in the projected software with indication of the expected results are determined;

c) design – defining of the detailed specifications of the developed software product, carrying out of the overall structure of the design, decomposition of components and design of components. The result – a detailed model of the developed software product with specifications of the components of all levels;

d) implementation - the process of step-by-step writing program codes in the selected language of programming (coding), their testing and debugging.

Maintenance is the process of creation and implementation of new software versions. In accordance with the standard ISO / IEC 12207 the phase of maintenance was marked out as a separate lifecycle process.

The reasons for the release of new versions include: the need to correct errors encountered during the operation of previous versions; the need to improve the previous versions; changes in the operational environment (appearance of new hardware and / or software, which interacts with the software); to make the necessary software changes that may require a revision of the already adopted design solutions.

Changes in the software life cycle are made possible by the use of software development of CASE-technologies, which represent a set of methodologies for the analysis, design, development and maintenance of complex software systems based on structural and on the objective approach. The basis of any CASE-technology includes methodology, method, notation and tools [1, 4].

Among the facilities are distinguished:

1) CASE-tools for analyzing requirements, designing specifications and structure, editing interfaces. The first generation of CASE-I, mainly included tools for supporting graphic models, design specifications, screen editors and data dictionaries.

2) CASE-tools for source code generation and implementation of the integrated environment for supporting the full software development life cycle. The second generation of CASE-II differs substantially by greater opportunities, provision of control, analysis and linking of system information and information for managing the process of designing, prototyping, and system models, testing, verification and analysis of the generated programs.

The modern CASE-tools allow you to automate labor-intensive operations; improve productivity of programmers; improve the quality of produced software; reduce time of creation of the prototype system; provide automated generation of project documentation for all stages of the life cycle, in accordance with modern standards; partially generate program codes for various development platforms; support technology of reuse of the system components, etc.

However, modern CASE-tools are expensive, and their use requires highly skilled developers. Therefore, it makes sense to use them in complex projects, keeping in mind that the more complex is the software developed, the greater the benefits from the use of CASE-technologies. To date, almost all commercially produced sophisticated software is developed using CASE-tools.

Lecture № 3. Structural and non-structural programming. Basics of algorithms

The goal - to get an idea about the features of the structural and non-structural program; study the characteristics of the algorithm, as well as basic algorithmic structures.

One way to ensure high quality of developed software is structured programming.

At the core of any program is the algorithm. The term "algorithm" is derived from the name of the IX century mathematician Al-Khwarizmi, who formulated the rules for making arithmetic operations. Initially, under the rules of the algorithm we understand only performing four arithmetic operations on numbers. Later this term began to be used in general to describe the sequence of actions leading to the solution of any task. An algorithm for solving computing challenges - a formal description of the method of solving the problem by dividing it into a finite time-sequence (steps) that are understandable to the executor. This should be clearly identified as the content of each stage, and the execution order of steps. Different stages of the algorithm make the process of problem solution more simple, the algorithm should be developed earlier, be fairly simple and self-explanatory. The basic properties of the algorithm are:

a) certainty - every rule of the algorithm should be clear, unambiguous and not leaving room for arbitrariness. Due to this property, the execution of the algorithm is of a mechanical nature and does not require any additional instructions or information about the problem being solved;

b) effectiveness (finiteness) - the algorithm should lead to the solution of the problem in a finite number of steps;

c) mass - the algorithm for solving the problem is developed in a general way, that is, it must be applicable for a certain class of problems that differ only in the original data. In this case, the source data can be selected from a certain region, which is called the domain of applicability of the algorithm;

d) discreteness (discontinuity, separation) - the algorithm should represent the process of solving the problem as a sequential execution of simple (or previously defined) steps. Each action, provided by the algorithm, is executed only after the execution of the previous one has ended.

There are several ways to write algorithms: verbal, formula-verbal, graphic language of operator schemes, algorithmic language.

The most widely used, due to its clarity, is a graphical way to write algorithms using flowcharts.

The flowchart is a graphical view showing the logical structure of the algorithm, in which each step of processing information is represented as geometric symbols (blocks) having a particular configuration depending on the nature of the operations. Graphical symbols, their size and rules of construction schemes of algorithms identify a single system software documentation (SSSD), a state standard (SST). The list of symbols, their name displayed their functions are determined by the shape and size of the guest (table B.1). All formulas in the block diagram are written in the language of mathematics, and not a specific programming language.

There are three types of computing process implementing programs: linear, branched, and cyclic.

The linear structure of the computing process suggests that to get the result you need to perform operations in sequence.

At branched structure calculation process specific sequence of operations depends on the values of one or more variables. To get the result in the cyclic structure, some steps must be performed multiple times.

To implement these computational processes to programs appropriate control operators are needed.

Programs written using only the structural operators transfer control are called structural, to distinguish them from the programs developed with the use of low-level methods of transmission control.

After being proved in the 60-ies of the XX century that any complicated algorithm can be represented using three basic control structures, high level programming languages obtained control operators for their implementation [3, 4]. The basic ones include:

a) sequence - means the sequence of steps;

b) branching - select one of two options;

c) while-cycle - defines repetition of action until a certain condition is broken, the fulfillment of which is checked at the beginning of the cycle.

Besides the basic, procedural, high-level programming languages use three additional designs implemented via the base:

a) selection - selecting one of several options, depending on the value of a certain value;

b) do-cycle - a repetition of actions before the specified conditions, check which is carried out after the action in the ring;

c) for-cycle - with a specified number of repetitions (cycle counting) - repetition of some actions specified number of times.

These designs were used as the basis for structured programming. Programs written using only the structural operators of transfer control are called structural, to distinguish them from the programs developed with the use of low-level methods of transfer of control. Disadvantages of the schemes:

a) low level of refinement that obscures the essence of complex algorithms;

b) use of non-structural ways to transfer control which on the diagram look simpler than the equivalent structured.

Example 3.1 - Using a flowchart for describing the search algorithm in an array of A (n) element being given (figure 3.1).

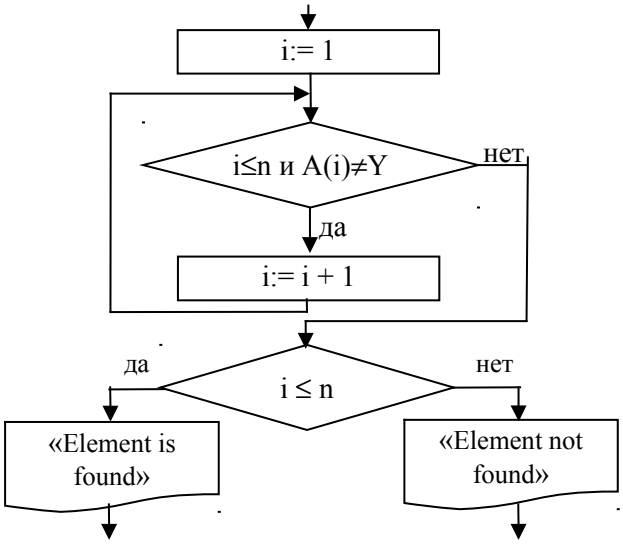

Figure 3.1 - Detail search flowchart

This example uses a structural variant of the algorithm (while-cycle). Array elements are moving and in turn compared with a predetermined value Y. As a result, a message is displayed.

The disadvantages of block diagrams are as follows:

a) a low level of detailing that obscures the essence of complex algorithms;

b) the use of non-structural ways to transfer control on the diagram looks simpler than the equivalent structure.

Besides schemes for describing algorithms, you can use pseudo-code, Flow-form and Nassi-Shneiderman charts, which are based on the same basic structures allow for different levels of detailing and make it impossible for non-structural description of algorithms [1, 3, 8].

The pseudo - formalized textual description algorithm. Initially, the designer focuses only on the structural modes. The pseudo-codes do not limit the level of detailing of the projected operations, but allow to balance the level of action refinement against the level of abstraction under consideration, and they are in good agreement with the method of stepwise refinement.

Example 3.2 - Using pseudo-code algorithm to describe the array A (n) element being given (fragment).

i: = 1
while i ≤ n, and A (i) ≠ Y
i: = i + 1
All -cycle
If i ≤ n
Display the "Element is found"
otherwise Output "Element not found"
After all, if

Flow-forms - graphical notation to describe the structural chart illustrating nested structures. Each Flow-shaped symbol corresponds to the control structure, shown as rectangles, and text containing mathematical notation or in natural language. To demonstrate the nesting structures symbol Flow-form fits into the corresponding area of the rectangle of any other character. Table B.1 shows the symbols Flow-forms corresponding to the main and additional control structures.

Example 3.3 - Using Flow-forms for describing the search algorithm in an array of A , (n) element being given (figure 3.2).

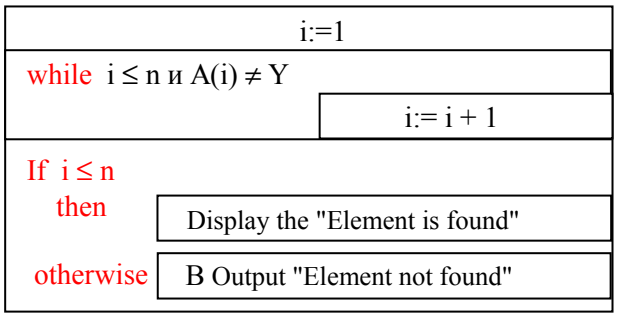

Figure 3.2

Nassi-Shneiderman diagrams are the development of Flow-form with the only difference that the area of designation of conditions and branching options are shown in the form of triangles, providing greater visibility of the algorithm.

Example 3.4 - Using Nassi-Shneiderman diagrams for describing the search algorithm in an array of A (n) element being given (figure 3.3).

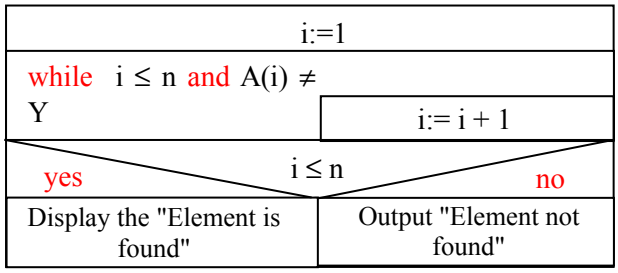

Figure 3.3 - Detail of Nassi-Shneiderman diagrams

Just like using pseudo-code to describe the non-structural algorithm, it is impossible to use Flow-chart form or Nassi-Shneiderman (no symbols). At the same time, as a graphic, the notation is better for displaying attachment structures than the

pseudo-code. The disadvantage: the difficulty of constructing character images complicates their practical use to describe the larger algorithms.

Lecture № 4. Algorithmic languages and requirements to them. Procedural languages

The goal - to get an idea about the basic characteristics of programming languages and their classification; explore the features of the use of procedural languages.

Programming languages are used when written algorithms have a number of characteristics that make it possible to categorize, compare, and choose them with the objectives of the development program. Such characteristics are of the level of power integrity [5].

The power of language is characterized by a variety of problems, algorithms that can be written using the language. Therefore, it is obvious that the most powerful processor language is because any problem is ultimately written in the language of the computer.

The level is determined by the complexity of the language problems to be solved using this language. The simpler solution is recorded, the more complex operations and concepts are expressed directly, the smaller the volume to obtain the original programs and, finally, the higher the level of the language.

The integrity of the language is related to the properties of economy, independence and uniformity of concepts.

Savings concept involves maximizing the power of language with the use of a minimum number of concepts. Independence concepts mean that the rules of use of the same concepts in different contexts should also be the same, except that between them there should be no mutual interference. Uniformity of concepts requires a single coherent approach to the description and use of all terms.

Traditionally, the classification of programming languages used by such a characterization, is called the level of language (figure 4.1). At the lower levels are placed machine-oriented languages, and on the upper - machine-independent.

Machine-oriented languages allow you to fully take into account the features of the processor and to receive the program with a high degree of speed. However, they are not able to provide mobility (portability) programs between heterogeneous computers. Mnemonic refers to assembly languages without macromedia, macro language refers to the assembly languages of macromedia.

Machine-independent languages are also called high-level languages. Procedural languages are algorithmic languages, which are intended to describe the procedures for solving the problem, that is, the programmer must specify a computer, how and what should be done to solve the problem. Among the procedural an isolated group of universal language is suitable for every application (such as Pascal, C / C + +, Ada). Non-procedural (problem) languages allow you to specify the computer what to do to solve the problem, and how to do it – the programming system decides

automatically. Among the problematic database languages there are: object-oriented, web programming, functional, logical, and others. Each of these groups is different from the others by level, and the principles of programming.

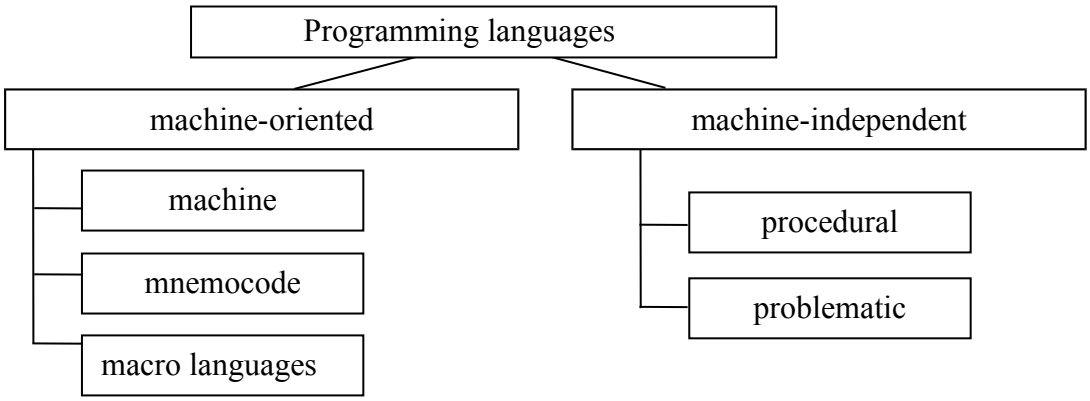

Figure 4.1 - Classification of programming languages by level

Thus, the classification which is shown in figure 4.1, contains 5 levels of language: 0 - machine, 1 - mnemonic, 2 - macro languages, 3 – procedural, 4 - problematic. But due to the trend of universalization of languages, this classification is no longer strict because the assembly language acquired abilities inherent to a high-level language, and C ++ language generally has features both of a high-level and of a low-level language. Therefore, it makes sense to define the class of language in terms of its underlying resources.

If you use any language you need to ensure that all written with the help of suggestions (lines) were correct in terms of alphabetical designs (syntactically) and had a particular meaning (semantics). The task of transferring the program from the original language in which it was written to the machine is performed by the program - compiler.

The source codes of the programs are typically used with comments, i.e. the explanatory text, designed in a certain way, which in no way affects the progress of the program. For identification (designation) of objects introduced into the program names are used (identifiers). An object is defined by variables, constants, data types, functions, and so on. For each language there are clearly defined rules under which designations are introduced. Key (service) words have a uniquely defined meaning and can only be used as it is defined in the language. Keywords can not be overridden, i.e. they can not be used as names entered by the programmer [1, 6, 7].

Lecture № 5. Introduction to C ++. The structure and the steps of creating a program in C ++. Standard C ++

The goal - to get an idea of the C ++ programming language, its features, the structure of the programs and the process of their creation.

High level programming language C ++ was developed in the US in the early 80s by an employee of Bell Laboratories Bjarne Stroustrup as a result of the expansion and additions of C means necessary for object-oriented programming. Among modern languages C ++ refers to the class of universal and is considered to be the dominant language used for commercial software development. Perhaps only a programming language like Java can be considered competitive. A variety of C ++ is C # - a new language developed by Microsoft for a network platform. Despite some fundamental differences, C ++ and C # are the same for about 90%. Especially effective is the use of C ++ to write system programs, compilers, operating systems, screen interfaces. This language combines the best features of the assembler and high-level languages. Programs made in C ++ for performance are comparable to programs written in Assembler, but more visual, easy to maintain, and easily portable from one computer to another. The main features of the language include the following:

1) C ++ offers a set of operations, many of which correspond to machine instructions and thus allow a live broadcast in the native code, and their diversity allows you to select different sets to minimize the resulting code.

2) Basic data types of C ++ coincide with data types of Assembler, and for conversions there are imposed minor restrictions.

3) The amount of C ++ is low because almost all of the functions performed are furnished in the form of a link library, and C ++ fully supports structured programming and provides a full set of corresponding operators.

4) C ++ widely uses pointers to variables and functions furthermore supports pointer arithmetic, and thus allows direct access to and manipulation of memory addresses; a convenient way to pass parameters are the links.

5) C ++ contains all the basic features of object-oriented programming languages: the presence of objects and data encapsulation, inheritance, polymorphism and abstraction types.

When writing programs in C ++, the following definitions are applied: alphabet, constants, identifiers, keywords, comments, directives [2, 5].

Alphabet can be represented as inherent in the language set of characters, all of which form the structure of the language. The C ++ language operates with the following set of characters: the Latin uppercase and lowercase letters (A, B, C, ..., x, y, z); Arabic numerals (0, 1, 2, ..., 7, 8, 9); an underscore ("_"); special characters (special characters, the list of C ++ is shown in table D.1); delimiter characters (whitespace, comments, line breaks, etc.).

With the help of these characters are formed names, key (service) words, numbers, character strings and tags.

Identifiers (names) must begin with a letter or an underscore "_", which may be followed by any combination of letters and numbers. C ++ distinguishes between uppercase and lowercase letters. Do not use special characters and delimiters for writing the names . For example,

_x, B12, Stack - right;

Label.4, Root-3 - is wrong.

There is some agreement on the use of uppercase and lowercase letters in identifiers. For example, the variable names contain only lowercase letters, constants, and macros - uppercase. With the underscore character names usually start the system of reserved variables and constants as well as the names used for the library routines. Therefore, in order to avoid possible conflicts and interconnectivity with a lot of library names it is not recommended to use an underscore as the first character name.

Some identifiers that have special meaning to the compiler are used as keywords. Their use is strictly defined, and they can not be used otherwise. The list of reserved words in C ++ is shown in table D.2.

The numbers are integers, and real values are written in decimal notation. Before it there can be any number of the "+" or "-". The real number integer part is separated from its fractional part of the point. *Real numbers containing a decimal point, should be before or after at least one digit*. The numbers are integers, and real values are written in decimal notation. Before there can be any number of the "+" or "-". The integer part of the real number is separated from its fractional part by the point. *Real numbers containing a decimal point, should be before or after the at least one digit*.

Jump to label name is a character-digital design, for example, label l, pass, *cross15, and the program is not announced.*

A string of characters - a string enclosed in quotation marks. For example, "character string".

There are two types of comments. Any sequence of characters enclosed in brackets limiting / * * /, in C / C ++ is viewed as a multi-line comment, for example,

/ * Main program * /.

In C ++, in addition, there is another kind of comment - a single line: all the characters following the sign // (double slash) before the end of the line are treated as comments, for example, // Main program.

Basically, C ++ style comments (//) and a comment of style C (/ * * /) are used to temporarily disable large parts of the program. Keep in mind that comments should explain not that is for the operators, but what they are used for.

A program written in C / C ++, typically comprises one or more functions. Function is an independent unit of a program designed to solve a particular problem that can manipulate data and return a value. The program structure is shown in figure 5.1.

Every program in C ++ preprocessor directive begins with #include, which connects the header file (* .h), containing the function prototypes that tell the compiler information about the syntax of functions, for example,

Include <iostream.h>

The header usually contains the definitions provided by the compiler to perform various operations. The header files are written in the format ASCII, their content can be displayed for viewing using any text editor from the catalog INCLUDE. The preprocessor scans the program before the compiler connects the necessary files, replaces the symbolic abbreviations in the program directives, and may even change the terms of the compilation.

Every C ++ program has at least one function - main (), which is automatically invoked when running, can cause other functions available in the program and usually takes the form:

```
void main ( )
#include <name.h>          →   Connection header files
function1 ( )              →   The definition of program functions
{
 the operator;;
}
function2 ( )
{
  the operator;
}
void main ( )              →   Determining the main function
{                          →   Начало тела главной функции
  the operator;1;          →   Operators function
    …
  the operator;N;
function1 ( );             →   Calling program functions
function2 ( );
…
}                          →   The end of the main body functions
```

Figure 5.1 - Structure of the program in C ++

Typically, the functionality you need to call the program (refer to it), during the execution of the code Main () function is called by the operating system, and referring to it from the program code is not possible. Void is an indication that the program does not return a specific value. In the case of the return of value of the operating system before the main () function word int is specified, and at the end of the body of this function the expression return () or return0 is placed.

After defining the main features of the program followed by the operators, which are enclosed in the grouping braces. Each statement ends with a semicolon (;) to indicate its completion. The program is carried out in rows in the order they appear in the source code, as long as meet the challenge of any function, then control is passed to the function lines. *After the function returns control of the line of code that follows the call to [2, 5, 10].*

Lecture № 6. Presentation of data in C ++. The assignment operator. Arithmetic operations. Preprocessor Directives

The goal - to get an idea of the standard data types, the order of operations, examine the features of the assignment operator and the use of the preprocessor.

By defining the data, it is necessary to provide the compiler with information on their type, and then it will know how much space you need to allocate (reserve) for storing information and what kind of value it will be. In C ++ there are distinguished five basic data types: character (*char*), integer (*int*), a real floating point (float), a real floating-point double-length (double), and a blank, with no type of value (void). On the basis of these types all the other ones are built.

The simplest technique is to use type modifiers that are put before the appropriate type: sign (*signed*), unsigned (*unsigned*), long (*long*) and short (short). table D.3 shows all possible combinations of the various types of modifiers with the range of change and takes the size in bytes. With repeated use of the program data types with various combinations of modifiers, for example, *unsigned short int*, it is easy to make syntax errors. In order to avoid them in C ++, you can create an alias synonymous with the keyword *typedef*. For example, the line *typedef unsigned short int USHORT*; it creates a new alias USHORT, which can be used anywhere one would write unsigned *short int*.

This name is associated with the region of memory that is reserved for the temporary accommodation of the stored value and subsequent recovery. For long-term (permanent) storage of variable values the database or file is used. In C ++, all variables must be declared before they are used. This ad requires a variable name and an indication of its type. However, it should be borne in mind that you can not create a variable of type void. The main form of variable declaration has the form

type < list of variables >;

In this declaration: *type* - one of the existing types of variables; < *list of variables*> may consist of one or more variables, separated by commas. For example,

int x, e, z; float radius; long double integral;

You can declare variables and assign them the initial values, i.e., initialize them. For instance, *int = 15 min; float p1 = 1.35;*

The variable is called *global* if it is declared outside any functions, including the function *main ().* This variable can be used anywhere in the program (except for the global static variables), and you run the program, assigned a value of zero. A variable declared in a function body (one block), is *local* and can be used only within that block. Outside the block it is unknown. It is important to remember that:

1) Two global variables can not have the same name.
2) Local variables of different functions may have the same name.
3) Local variables in one block can have the same names.

These programming languages may also be presented as constants. Constants are used in cases where the program is forbidden to change the value of a variable. To determine the constants in the traditional way using *#define*, which simply performs text substitution. For example,

#define StudentsOfGroup 15

In this case, the constant Students Of Group does not have a specific type, and each time the preprocessor encounters *Students Of Group* name, it replaces the literal 15. Because the preprocessor running before the compiler will never see a constant and will only see the number 15.

The most convenient way to determine the constants is as follows:

const type constant_name = value_constant

This method makes it easier to further support the program and prevents errors. Since the definition of constants contains a type, the compiler can trace its use only for its intended purpose (in accordance with the declared type). For example,

const int Diapazon = 20;

Literal constants (literals) - values that are entered directly in the text of the program. Because after compilation you can not change the values of literals, they are also called constants. For example, the expression *int MyAge = 19*; *MyAge* name is a variable of type int, and the number 19 - a literal constant that can not be assigned to any other value.

A *character* constant comprises a single character enclosed in single quotes: 'q', '2', '$'. For example,

const char month = 'December'; .

For symbolic constants are special characters (including managers, are listed in table D.1).

String constants consist of a sequence of characters of the code ASCII, enclosed in quotation marks, terminating zero byte. The end of the character string (zero bytes) is denoted by NULL ('\ 0').

Enumerated constants enable you to create new data types and then to define variables of those types whose values are limited to a set of values of the constants. To create an enumerated constants using the keyword enum, and the record is:

enum constant_name {list constant values};

The values of the constants in the list of values are separated by commas. For example,

enum COLOR {RED, BLUE, GREEN, WHITE, BLACK};

Each element of the enumerated constants corresponds to a certain value. By default, the first element is 0, and each subsequent - by one more. Each element can be assigned an arbitrary constant value, then the following are initialized to one more than the previous one. For example,

enum COLOR {RED = 100, BLUE, GREEN = 200, WHITE = 300, BLACK};

In this example, *BLUE = 101, BLACK = 301.*

There mechanism explicitly specifies types of constants using suffixes. For integer constants as suffixes can be used letters *u, l, h, L, H,* and for floating-point numbers - *l, L, f, F.* For example,

12h 34H - short int

23L -273l - long int

23.4f 67.7E-24F - float

89uL 89Lu 89ul 89 LU - unsigned short

Expression of C ++ is a certain allowable combination of operators and operands (constants, variables or functions). List of operations of C ++ is shown in table D.4. All these operations are carried out by conventional means, except for a division operation. The peculiarity of the division operation is that if both operands

are integer, then it will give a result of, for example, *3/2* gives *1*. To obtain a valid result, you must have at least one valid operand, for example, *3 / 2.0* gives *1.5*.

For each operation, the language defined by the number of operands:

a) one operand - *unary* operation, changes the sign, such as unary minus -*x*;

b) two operands - a *binary* operation, such as the operation of addition *x* + *y*;

c) three operands - operation *condition*?: it only.

Each operation can have only certain types of the operands. Each binary operation has a specific order of execution: from left to right or right to left. Finally, each operation has its own priority. Priority and order of operations are shown in table D.4.

Often terms of math functions are used in language C ++, which are located in the library of *math*. To take advantage of these functions in the beginning of the program there must be included the header file <*math.h*>. Basic math functions are listed in table D.5.

All expressions are *operators*, which are in the language used for describing the actions. Any operator may be marked with a label. Operators are separated by a semicolon (;). At any point in the program, which can be placed by one operator, you can place a compound statement, called the block. The block contains several statements that are executed as a single expression limited by braces {}, but does not end with a semicolon (;).

Declaring a variable in the program is only the allocation of space in the computer's memory for its placement. The program should allow the data to operate. In this process, the most important is the *assignment operator*, which looks like this: *variable* = *expression*. The assignment replaces the value of the operand to the left of the "=", a value calculated on the right. This may be an implicit type conversion. The sign "=" in the C / C ++ - is a sign of an assignment rather than equality.

Unlike other languages, where the assignment – operator is used by definition, in C / C ++, there are the concepts of "*assignment operation*" and "*assignment operator*". Operation is "*converted*" to the operator, if at the end of the expression there is a semicolon, for example, ++ *x* - is an expression, but ++ *x*; - an operator. The assignment operator is useful to initialize variables, for example, *j* = *k* ;. Furthermore, in C / C ++ in C / C ++ assignment operation can be used in terms that include the comparison operators or logical operators, for example, *if ((x = x + 5)> 0) count << "Conclusion";*.

Another feature of the use of the assignment operator in C / C ++ is the ability to assign multiple, which runs from right to left. For example, in order to assign the value of *2 * k* to several variables, it is possible to take advantage of the operation: *x = y = z = 2 * k.*

In C / C ++, there are *additional* assignment operation + = - = * = / =, and % =. The value on the right is added (subtracted, multiplied, divided, or divided in absolute value) to the value of the variable, which stands on the left. For example, instead of the operator *x = x + 5;* can write *x = 5 + ;*. Moreover, the operation *x + = 5* is faster than the operation *x = x + 5*.

Very often in the programs variables a block is added (or subtracted). Increasing the value of 1 is called an *increment* (++), and a decrease of 1 - *decrement* (-). For example, the operator *c = c + 1;* equivalent to the operator *of c ++ ;,* the operator *c = c-1;* equivalent to the operator *c-- ;.*

Increment and decrement operators exist in two forms: *prefix* and *postfix*.

Prefix operations increase (decrease) the value of a variable by one, and then use that value. For example, the operator *x = ++y;* is equivalent to the implementation of the two operators *y = y + 1; x = y ;.* In this example, first, an increase in the block value of the variable *y*, and then assigning the values of the variable *x*.

Postfix first operation using the variable value, then increase (decrease) it. For example, the operator *x= y--;* is equivalent to the implementation of the two operators *x = y; y = y-1 ;.* In this example, the variable *x* is assigned the value *y*, then the value *y* is decremented.

Generally, it is better to use operands of the same type, but C ++ allows *conversion of types*, that is, if the operands belong to different types, they are brought to a common type. Reduction is carried out in accordance with the following rules:

a) automatically performed are just the transformations that convert the operands with a smaller range of values in the operands with a large range of values, as this occurs without any loss of information;

b) expressions that make no sense (for example, floating-point number as an index), the compiler will not pass at the stage of transferring;

c) expressions that might lose information (for example, when you assign long integer values, shorter or actual values of the whole), can cause a warning, but they are valid.

Unlike other programming languages C ++, for any expressions can explicitly *convert its type* using the unary operator called a cast. The expression is the specified type of design rules.

(type name) expression;

For example, *(int) i = 2.5 * 3.2;* .

However, the operator can use this only if the purpose and consequences of such a conversion are understood [2, 5, 10].

Lecture №7. I / O functions. Basic design of C ++

The goal - to get an idea about the functions of "I/O", used in C ++, as well as get acquainted with the peculiarities of use of the basic structures of the language.

In any complicated program *linear* fragments can be provided. Fragmented programs have a *linear structure*, if all the operations therein are executed sequentially, one after another, and may comprise the assignment operators, mathematical function, arithmetic function "O" data and other operators do not modify the general order of the operators.

Rare program dispenses with operations "I/O". In C, it was implemented with two innovative ideas: the means of "I/O" are separated by language and made into a separate library *stdio* (standard library "I/O") and has implemented the concept of "input-output", regardless of the device . That is why the C ++ language, which inherited traits of its "ancestor", has a great feature set "I/O" different types of data.

In order to implement the aspect "O" two types are often used to *printf* and *scanf*, connected via the header file <*stdio.h*>. Printf function can be used to display any combination of characters, integers and real numbers, strings, unsigned integers, long integers and unsigned long integers. It is described as follows: *printf ("control_string ", a list of arguments)*.

The list of arguments - a sequence of constants, variables or expressions whose values are displayed in accordance with the format of the *control_string*, which determines the number and type of arguments, and typically includes the following objects: ordinary characters that appear on the screen without changes; *specifications conversion*, each of which causes display of the next argument values from the following list of arguments; *control character constants*. A conversion specification begins with the character%, and ends with a conversion , between which can be written:

- "minus" sign, indicating that the message text is left-aligned by default will align the right edge;
- a string of numbers that specifies the minimum amount of output fields;
- the point is the separator;
- a string of numbers that specifies the accuracy of the output;
- the symbol *l*, indicating that the corresponding argument is of type *long*.

For example, *printf ("\ nAge Eric -% d. His income $%. 2f", age, income);*

It is assumed that the integer variable *age* (Age) and the real variable *income* (income) assigned any value. The sequence of characters \ n moves the cursor to a new line, so the sequence of characters "*Eric Age*" will be displayed from the beginning of a new line. The symbols are symbols of the *% d* format conversion (specification) for the variable *age*. This is followed by a literal string "*His income $"* and the characters *% .2f* - specification for the value of the variable income, as well as an indication of the format to display only two digits after the decimal point.

Function Input function is similar to *scanf* described *printf*:

scanf ("control_string " , list of argument);

Scanf Function arguments must be pointers to the appropriate value (for more signs will be discussed later), which is written before the name of the variable *&* character. As in function *printf*, control line contains the conversion specification is used to determine the number and types of arguments. It is permissible to use spaces, tabs and newline, which are ignored on input. The same delimiter that you type with the keyboard must separate the control string conversion specification. For example, *scanf ("% d% f% c", & i, & a, & ch);*

C ++ program does not deal with any devices or files - they work with *streams*. *Type information from the input, the output is produced in the output stream, which is associated with the device or file.* In C ++, the concept of independence from the

device "O" has been further developed in the form of object-oriented library "I/O» *iostream*, which includes the object cout to output to the screen and the object cin, used to enter the information [7, 8].

Operator output *cout* is as follows:

cout << ... << << var1 variableN;

The "<<" is called insert, which inserts the characters in the output stream. To move the cursor to the next line in the *cout* statement is often used symbol *endl* (end of line), for example, *cout << xl << endl ;*. The operator display can use special characters that must be enclosed in single quotation marks if they are used on their own, such as sout << '\ a' << "Call". If special characters are used within double quotes, then further it is not necessary to enclose them in apostrophes.

For example,

cout << "output \ tx =" << x << endl;

To organize formatted output numbers using *cout* statement can be, for example, use the functions *cout.width, cout.fill, cout.put,* modifier *setw*, manipulator *setprecision*, include the header file <*iomanip.h*>.

Operator input *cin* is as follows:

cin >> ... >> >> variable1 variableN;

The ">>" is called the extraction operation (the operation extracts data from an input stream, assigning the value of a specified variable). The values of the input variables must match the types of variables specified in the declaration statement, and when you enter from the keyboard are separated by at least one space, for example, 1.5 2.15 -1.1 25.

The values of variables and character strings you type are recorded in the input stream without the apostrophe or quotation marks. For their input C ++ provides two additional functions: *cin.get* and *cin.getline*.

For the appearance of the program can be used the functions that are described in the header file <*conio.h*>.

To change the performance of the natural sequence of operators (transmission control) in C / C ++ provision is made with a number of special designs related to *constructions decision making* and in its meaning coinciding with similar structures of algorithms.

For operators to transmit control operators are *unconditional jump, conditional jump,* the operator of *choice (option*), which have counterparts in other programming languages (such as Pascal), as well as the *ternary conditional* operator.

The operator of unconditional transition is as follows:

goto label;

The label indicates the transition of the program operator, who should be referred to management. *When the goto statement transition occurs without checking any conditions.* Since these transitions destroy the relationship between structure and the structure calculation program that causes a loss of clarity and difficult task program verification, the operator unconditional jump should be used only in exceptional situations.

The operator conditional branch is to select one of two options of solving the problem, depending on the value of some verifiable conditions, and its full form is:

condition operator1; else operator2;

As a condition of using some arbitrary expression that specifies the selection condition executed by the operator *operator1* and *operator2* can be either simple or compound. If the condition is true (TRUE or any non-zero value), *operator1* is executed if the condition is false (FALSE or zero), then *the operator2*. Shorthand operator conditional transfer is:

condition operator;

In this case, if the condition is true, then the operator if the condition is false, then control is passed to the next program operator.

The most common condition is a logical expression composed of operands and operation symbols. As the operation in logical terms, primarily used comparison operators (==,! =, <,>, <=,> =). In addition to comparisons to the construction of logical expressions, you can use logical operators (*!, ||, &&).* The value of the logical expression is evaluated by performing operations specified in it with regard to their priority and arranged parentheses, for example,

(abs (x) <= 2) - *x* modulo value does not exceed 2;

((x> = 1) && (x <= 2)) - the point belongs to the interval [1,2];

*(x * x + y * y <1)* - he point with coordinates (*x, y*) belongs to the unit circle centered at the origin.

Because the C / C ++ is represented as truth nonzero and false as a zero, then another possible use of the conditional branch statement: *x = value*; *if (x) operator;*

In this case, the condition in the *if* statement is not logical expression and the variable to which the pre-assigned a value. If the variable is nonzero, then the condition is true if the value of the variable is zero, the condition is false.

Inside the *if* operator it is allowed the use of nested construction:

if (expression1) operator1;
else if (expression2) operator2;
...
else if (expressionN) operatorN;
else operator by default; // *optional part of*

To avoid ambiguous interpretation of the program it is necessary to use braces to highlight the nesting in a single unit since the else clause is similar to the nearest previous *if.*

If in the program you need to select one of many options, instead of nested *if* construct, it is more appropriate to use the operator-switch *switch*, called the operator of choice of options, which has the form:

switch (expression)
{case n1: operator1; break;
case n2: operator2; break;
case nK: operatorK; break;
default: operatorN; break;}

Execution of the operator option starts with calculating the value of the expression (selector). *The operator then transfers control to select the operator that faces constant coinciding with the calculated value of the selector switch.* If no match is found, the operator *appears* after the *default.* Construction of switch is allowed, in which the default statement is optional. Operator break, located in each branch operator option requires to complete the current operator and to transfer control to the next statement of the program. The absence of a break instructs continue executing until the first break statement or the end of the operator switch.

An alternative operator is a conditional branch ternary conditional *operator?:* - the sole operator, who works with three operands. This operator takes three expressions and returns a value:

(expression_1)? expression_2: expression_3;

The execution of the calculation begins with *expression_1*. If the expression is true, then the result will be the value *expression_2*, otherwise the result is *expression* _3. For example, the operator max = *(x> y)? x: y;* It determines the largest of the two numbers *x* and *y*.

An algorithm, which provides for repeated execution of the same sequence, an algorithm called a cyclic structure, and this sequence - *cycle*. Cyclic algorithm can significantly reduce the size of the program. In C ++, operators use cycles of three kinds: a prerequisite, followed by a condition and setting.

Operator cycle with a precondition refers to the basic design, perform repetitive actions as long as a specified condition is true, and has the form:

while (condition)
{ operator1;
operator2;
operatorN;},

where the condition is any simple or complex expression of C ++;

operator - any valid operator or block of statements (loop).

The loop is executed as long as the condition evaluates to TRUE. When false (FALSE) cycle is completed, the program transfers control to the next operator of the program. Since the condition is tested in a *while* loop there can be any valid expression, then, in principle, it is possible to use the expression *true*. However, in this case, the cycle is complete, and never becomes infinite, causing the computer to hang. Endless cycles should be used very carefully and thoroughly checked. If the analyzed condition is false from the start, none of the operators body forming the loop will be executed even once, i.e. the cycle will be skipped entirely.

If you want to ensure that the current loop occurs at least once, you should use the *cycle operator, followed by (a postcondition)*, the syntax of which is as follows:

do
{ operator1;
operator2;
operatorN}
while (condition);

If the loop contains only one statement, the written statement uses brackets to disambiguate, due to the presence of *while*. Actions defined by the operator, are performed as long as the condition is false or zero. Appointments *conditions* to continue the cycle and the operators are similar to the operator *while*. The loop body with postcondition, as in the case *while*, must contain actions that affect the result of the expression, which is a condition for input / output of the loop, or loop will be infinite.

Finally used in C / C ++ *operator cycle with a parameter* is:

for (initialization; test; increment)

operator;

The operator sets the initial value of the initialization counter. The operator checks - is any C ++ expression, the result of which is checked at each iteration: if the result is TRUE, the loop body is executed. After modifying the counter to *increment* value (by default, increase by 1), the actions are repeated. All these expressions are optional.

The initialization expression if it is, will always be executed. Calculation of the final expression (*check*) can not be performed if the condition is false to begin with. The law of the parameter loop usually determines the *increment*, but this is optional. Moreover, the *for* loop in C / C ++ is not a classical cycle with the option: cycle parameter can be a real value or change the specified law.

In a *for* loop, you can omit one or more expressions, but you can not drop the ";" symbol. It is only necessary to include in the body of the loop several operators that will lead to the completion of their work [7, 8].

If the cycle does not initialize the cycle parameter, does not include the control of expression and does not do anything, it is called *open*.

There are several ways to interrupt the loop or change the order of the statements of the loop body. Sometimes you need to go to the beginning of the cycle until the completion of all the operators of the loop body. For this purpose the operator *continue* is used, which makes a transition at the beginning of the loop, skipping all the remaining operators. In some cases, you want to get out of the loop body before checking the condition of the continuation of the cycle. For this purpose, use the operator *break*, which gets you out of the loop, skipping all the way to the closing brace.

Operators *continue* and *break* should be used very carefully, because their free application is able to confuse even a simple *while* loop and make it unreadable. This is the most dangerous commands after *goto* [10].

Lecture № 8. Complex data types: arrays. One-dimensional and multi-dimensional arrays. Organization of sorting algorithms

The goal - to get an idea of the complex data types, to explore methods of processing and especially the use of arrays.

In C / C ++, it is possible to build on the basic types complex data types. These include arrays, structures, unions, and files.

Array - an ordered set of elements of the same type, having a common name. All the elements of the array are numbered. The serial number of the element in the array is called the index. Arrays must always be described before using the program. There are one-dimensional (vectors), two-dimensional (matrix), and multidimensional arrays. General view of the array declaration:

Type of_elements,_the name of the array [N1] [N2] ... [Nk];

The number of indexes *[N1] [N2] ... [Nk]* determines the size of the array. When you declare the array shows the total number of array elements. Indexing of elements in the array in C / C ++, the default is zero-based. For example, the first element has index-dimensional array [0], a two-dimensional - [0] [0], etc. The size of the array can be set constant or a constant expression. You can not set an array of variable size used for this purpose a separate mechanism - *dynamic memory allocation*.

When you declare an array can be initialized. In this case, the initial values of the array elements are specified between the left and right curly brackets, following the equal sign. For example,

int a [3] = {10, 20, 30}, b [] = {10, 20, 30};

float c [2] [3] = {5, 10,15, 20, 25, 30};

If the array is declared without a dimension (array b), the compiler will determine the required number of elements in the array. When you declare an array with an unknown number of *elements vat size can not be decreed* only to the leftmost square brackets, for example,

float d [] [3] = {{5, 10, 15}, {20, 25, 30}};

If initialization is set smaller than the initial values of the array, then the total number of the remaining elements are initialized to zero.

The elements of the array apply two operations: *assignment* and *sampling*.

To sample the array elements are most commonly used *cycles*. For example, for input (output) and one-dimensional array of elements *a[100]* from the keyboard, you can use loop with a parameter

```
for (i = 0; i <100; i ++)
cin >> a [i];                    // operator is used in the derivation of cout << a [i];
```

For the organization of the input (output) pixel values multidimensional array uses nested loops (their number is equal to the number of indexes). For example,

```
for (i = 0; i <10; i ++)     // outer loop sampling organized by rows
for (j = 0; j <5; j ++)      // internal (nested) cycle by columns
cin >> a [i] [j];            // operator is used in the derivation of cout << a [i] [j];
```

In order to form an array of randomly used a random number generator. For example,

```
#include <stdlib.h>          // connection header file containing
... // The random
```

```
randomize ();                    // initialize the random number generator
...
for (i = 0; i <M; i ++)
{X [i] = random (100);      // the formation of an array of M elements,
cout << x [i]; }                 // Having integer values in the range 0 - 99
```

For equally likely positive and negative elements of the generated value must be subtracted the number equal to half of the argument of the function random. For example, *x [i] = random (100) -50;*

Using typical programming techniques when working with arrays cycles are also organized. In practice, the implementation of the basic techniques (table D.7) is reduced to the following steps: before the opening of the cycle is given an initial value - accumulated or expected value of the parameter; accumulation or search is carried out directly inside the loop.

For many tasks there is a need to rearrange the elements of the array so that they are arranged in ascending or descending order. Such arrays are called *ordered*, and the process for their preparation - *sorting*.

The simplest algorithm is to *sort a simple choice.* The sorted array is the minimum element that is exchanged places with the element at the beginning of the array. The remainder of the array is considered as a separate array in which also the smallest element is searched, and exchange with the first element of the array shortened. Actions are repeated as long as the array is not shortened by one element [7, 8].

Example 8.1 - Sort an array by sorting simple choice.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{Const int N = 6; int mas [N], i, j, i_min, j_min, min;
  randomize ();
  clrscr ();
  for (i = 0; i <N; i ++)
          {mas [i] = random (100) -50; printf ("% 4d", mas [i]); }
  printf ("\ n");
  for (i = 0; i <N; i ++)
          {mas = min {[i]; i_min = i;
         for (j = i + 1; j <N; j ++)            // search in a reduced array
                if (mas [j] <min) {min = mas [j]; j_min = j; }
          mas [j_min] = mas [i]; mas [i] = min; }
  for (i = 0; i <N; i ++)
          printf ("% 4d", mas [i]);
  printf ("\ n"); }
```

Under the *bubble* sort is understood a whole class of sorting algorithms. In its simplest embodiment, the bubble sort is carried out very slowly, so it is usually used

bubble sort with elements of optimization. All bubble sort algorithms have a common feature - the exchange of items made between two adjacent elements of the array.

Example 8.2 - Sort an array of 6 elements of the bubble method.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
 {const int N = 6; int mas [N], i, j, wrk;
  randomize ();
  clrscr ();
  for (i = 0; i <N; i ++)
       {mas [i] = random (100) -50; printf ("% 4d", mas [i]); }
  printf ("\ n");
  for (i = 0; i <N-1; i ++) // pass count
       for (j = 0; j <N-1; j ++)
            if (mas [j] <mas [j + 1])
            {wrk = mas [j]; mas [j] = mas [j + 1]; mas [j + 1] = wrk; }
  for (i = 0; i <N; i ++)
  printf ("% 4d", mas [i]);
printf ("\ n"); }
```

You can reduce the number of passes of the sort, they are not performing (N-1) 2, as long as the array is sorted. To determine this fact is quite simple: if the array is already sorted, the process of passage does not make any changes. Before you start, you need to install a sign of lack of permutations (*flag*). If at least one permutation is produced, the flag changes its value. If by the end of the passage of the initial value of the flag it is left unchanged, then the array is sorted, and further passes are not needed.

Example 8.3 - Sort an array bubble method with the optimization of the number of passes.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
 {const int N = 6; int mas [N], j, wrk, flag;
   randomize ();
   clrscr ();
   for (j = 0; j <N; j ++)
          {mas [j] = random (100) -50; printf ("% 4d", mas [j]); }
   printf ("\ n");
   flag = 1;                    // The operator needs to enter into a cycle
 while (flag! = 0)
       {flag = 0;
        for (j = 0; j <N-1; j ++)
           if (mas [j] <mas [j + 1])
```

```
            {flag ++; wrk = mas [j]; mas [j] = mas [j + 1]; mas [j + 1] = wrk; }}
    for (j = 0; j <N; j ++)
    printf ("% 4d", mas [j]);
  printf ("\ n"); }
```

To optimize the method according to the bubble sort execution time of each passage can be used that, after the first pass would be the largest element in the end of the array to its final destination. In carrying out the second pass the same will happen with the next largest element, and so *on. Therefore*, in subsequent passes *can be reduced,* the length of the view of the array that will significantly reduce the overall execution time of the algorithm**.** *If you combine this method with check feature to optimize the sorting is complete, it will exchange sorting algorithm is a sign of complete* [6, 10].

Example 8.4 - Sort an array by sorting feature of the exchange is complete.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main ()
{const int N = 6; int mas [N], j, wrk, flag, k;
   randomize (); clrscr ();
   for (j = 0; j <N; j ++)
         {mas [j] = random (100) -50; printf ("% 4d", mas [j]); }
  printf ("\ n");
  flag = 1;                    // The operator needs to enter into a cycle
  k = 1;                       // K -reduction length view
  while (flag! = 0)
        {flag = 0;
        for (j = 0; j <N-k; j ++)
              if (mas [j] <mas [j + 1])
                 {flag ++; wrk = mas [j]; mas [j] = mas [j + 1]; mas [j + 1] =
wrk; }
            k ++; }
    for (j = 0; j <N; j ++)
    printf ("% 4d", mas [j]);
  printf ("\ n"); }
```

Lecture № 9. Complex data types: character arrays. Processing character data

The goal - to get an idea about the features of the processing and use of character arrays.

A *string* of characters in a C / C ++ is an array of values of *char*, terminating null byte *'\ 0'* - a sign of the end of the line, which should be considered when

declaring a string, i.e. indicate not *N,* and *N + 1* elements. For example, a string of 10 characters will be declared: *char str [11] ;*.

During initialization of strings traditional methods ads are used. For example,

char My_symbols [] = "ABCD";

or equivalent *char My_symbols [] = {'A', 'B', 'C', 'D'};*

To determine the constants initialized to the length of the string variable, you can use the function *sizeof (),* for example,

char string_symb [] = "ABCDEFG";

#define Length_s (sizeof (string _symb))

When using string constants - letters, enclosed in double quotes - null byte '\ 0' at the end is not put.

Enter strings with the keyboard conveniently done using the functions *scanf ()*, which allows you to enter up to the first space character or *gets (),* which lets you enter strings that contain spaces. Entering ends by pressing Enter. Both functions are automatically put in the end of the line of zero bytes. The parameter is simply the name of the array.

The output produced by the functions printf () or puts (). If you use all of these functions require the header file <stdio.h>. Both functions output the contents of the array to the first null byte. Moreover, the puts () function adds to the end of the output string newline character, and when you use printf () function a newline should be provided in the format string.

To work with strings, a special library is used, which is described in the file <string.h>.

Most often the following functions are applied:

1) The function strlen (str); determines the length of the string str (in characters), *thus* terminating null byte is not taken into account.

2) Function strcpy (str1, str2).

3) Copies the string str2 in the string str1. Str1 The array must be large enough to fit in a string str2. This resulting string is automatically terminated with a null byte. If the place is small, the compiler does not issue instructions on the error or warning, it does not interrupt the execution of the program, but can cause damage to other data, or the program itself, and improper operation of the program in the future.

4) Function strcat (str1, str2); appends a string to a string str2 str1 and puts it into an array, where the string str1, string str2 in this case does not change. Null byte that terminates the string str1, will be replaced with the first character of the string str2. The result is written to the string str1, and the resulting string is automatically terminated with a null byte.

5) Function strcmp (str1, str2); compares the strings str1, str2; returns 0 if the strings are equal (ie, contain the same number of identical characters), the number is less than zero if str1 <str2; and a number greater than 0 if str1> str2. String comparison is performed character by character up to the first mismatched character, and that line is considered to be greater in which the first mismatched character has a larger code. (A <a, a <I).

6) Function strlwr (str); converts lowercase to uppercase characters (A -> a).

7) Function strupr (str); converts lowercase to uppercase (a -> A).

8) Function strset (str, '*'); organizes the filling of str specified by the symbol *.

9) Function strstr (ss, w); It allows you to find a substring in a string ss w.

10) The function int (a); allows to obtain a character code.

In addition to the functions of the input variables and character strings described above, C ++ provides the user with an additional two functions cin.get and cin.getline. In that case it is necessary to enter a single character from the keyboard, can use the cin.get, e.g., c1 = cin.get (). In that case you must enter the entire line, you can use the cin.getline. For example, cin.getline (s1,21) wherein c1 - a string name 21 - the number of characters stored in the character string.

When processing the symbol data use the same methods and typical devices as the processing and numerical arrays. String, like any other array can be treated either as a conventional method using an array indexing operation, either through pointers using pointer arithmetic operations.

When working with a string as an array it should be borne in mind that the length of the string is not known in advance, so it is advisable to organize a cycle not to count, but before the end-of-line [7, 8].

Lecture № 10. Complex data types: structures and unions

The goal - to get an idea about the features of the processing and use of structures and unions.

Complex data types except arrays are implemented by structures and associations.

Structure represents a group of related components (elements) of different types, the number of which is fixed.

Before you create the object structure, you must define the format, that is, to declare the structure. Variables that make up the structure are called its elements (members or fields). Name of structure - a type specifier. Struct keyword marks the beginning of a structure declaration. Thus, the description of the type struct has the following form:

struct name of structure
{type1 imya_elementa1;
...
Tip N name of element N; }
Name of changing type of structure;

The individual elements of the structure accessed via the operator "point". To refer to an element of structure, it is necessary to put its name and the name of the structure variable operator "point". The general format of the access is as follows:

Structure name variable. Element Name.

Consider an example that allows you to record in the structure the date of any event in the following way:

struct date

{int day; char month [9]; int year; } d;

In the example there is described the structure of the date and the variable d, referring to date type. The variable d is a structure consisting of three elements, each of which contains the following information:

- an element of day - an integer indicating the day (1 to 31);
- an element of month [9] - character string with the name of the month;
- item year - an integer that represents the year.

A variable of type structure d may be declared outside the description of the structure using the name, which is called a structural tag. Tag structure can begin with the word struct:

struct date d;

By structure elements it is applicable assignment operation.

For example, to send to the variable d the date of 14 April 1930, the program should contain the following snippet:

d.day = 14;
d.month = "April";
d.year = 1930;
The structure can be passed to a function name. For example,
d.day = 14;
strcpy (d.month, "April"); // use the copy function values
d.year = 1930;

Associations (unions) are different from other ways of information storage. At any given time the union stores the value of only one element. Memory storage distribution is the largest union member. Description union types are as follows:

union name
{type1element name1;
...
typeN element name N;
} name of changeable type of union;

For example, the association storage unit of time can be represented as follows:

union time
{int hour; long second;} t;

This example describes the association of time and variable t, it belongs to the type of time. Variable t association is described as consisting of two elements: hour and second, which contains the following data:

element hour - an integer type int, indicating the time in hours;
element second - an integer type long, indicating the time in seconds.

As the largest element (second) is of type long, 4 bytes of memory will be allocated for a union member.

Chance of association can be declared outside description of associations, for example:

union time
{int hour; long second;};

...

time t;

When accessing the member associations it is needed to specify the variable name and the name of the union member, separated by a point, the treatment is similar to the structure element:

t.hour = 10;

Using associations in the programs allows you to save memory. Even greater memory savings program¬mah in C / C ++ can be achieved by using anonymous association. The anonymous union has no name, the variable is not a declared association:

union {type1 name elementa1;

...

typeN name elementaN; };

The elements of an anonymous association referenced by name (without a dot), as ordinary variables in the program [6-8].

Consider the example of the use of anonymous unions in the program.

```
#include <iostream.h>
void main (void)
{
union {int hour; long second; };
hour = 720;
cout << "B month" << hour << "hours." << endl;
second = 2592000;
cout << "B month" << second << "seconds." << endl; }
```

In this example, the value of *a one-hour* anonymous unions will be destroyed at the time of assigning a value of element second.

Lecture № 11. Complex data types: files. File operations on arrays and structures

The goal - to get an idea about the features of the processing and use of files when used with other complex data types.

Another complex data type, which is often used in programs is presented by files. A file is a sequence of elements of one and the same type, having a common name. The number of elements (in the length-file) is not limited. The files are usually created in the computer's memory or on external devices.

To work with files, you can use the standard library functions, and the program must include header file <fstream.h>. Consider the basic functions.

Writing data to a file is possible in two modes:

- create a new file (or overwrite the existing, previously created file);
- adding data to an existing file.

To open a file creation mode using the operator

ofstream out_file ("filename.ext");

In parentheses operator ofstream, you can specify a second parameter from the list:

ios :: out - open the output file;

ios :: trunc - to overwrite the contents of an existing file;

ios :: noreplace - if the file exists, do not open the file;

ios :: nocreate - if the file exists, do not create the file;

ios :: ate - place the file pointer at the end of the file.

For example, the operator ofstream out_file ("file1.txt", ios :: out | ios :: noreplace);

opens the file to display the name file1.txt existing without overwriting in the current directory on the disk.

Output to a file using the operator inserts (").

To close a file, use the function close ().

To open a file in append mode is used operator

ofstream out_file ("filename. expansion", ios :: app);

Parameter ios :: app requires to open the file in extention, positioning the file pointer at the end of the file. For example,

```
#include <fstream.h>
void main (void)
{ofstream out_file ("file1.txt", ios :: app);
out_file << "mode add." << endl;
out_file.close (); }
```

As a result of this program in the current disk catalog file will be opened file1.txt, which will add the line "mode add." Thus, the file file1.txt will have three lines:

1) An entry in the file.

2) Creation mode.

3) Append mode.

Input (write) data can be performed not only with the keyboard, but also from the file. To open a file in input mode using the operator

ifstream in_file ("filename.ext");

The operator ifstream can specify a second parameter:

ios :: in - to open a file for input.

Input file using extractor operator (''). For example,

```
#include <iostream.h>
#include <fstream.h>
void main (void)
{
char a [64], b [64], c [64];
ifstream in_file ("file1.txt");
in_file >> a >> b >> c;
cout << "The first line of the file file1.txt:" << a << "" << b << "" << c << endl;
in_file.close (); }
```

Assume that the current directory on a disk file is created in advance file1.txt, which contains three lines:

1) An entry in the file.
2) Creation mode.
3) Append mode.

Then, by a decision of the program on the display screen will be displayed:

The first line of the file file1.txt: Writing to a file.

First will be read and displayed on the display screen the first line from the file file1.txt. The gaps between the words "Write to file." will be perceived by compiler as symbols of the end of the string, and the variables a, b and to get the value: "Record", "c" and "file." respectively.

To read from a file filel.txt all three rows, you can use the getline. For example,

```
#include <iostream.h>
#include <fstream.h>
void main (void)
char a [64], b [64], c [64];
ifstream in_file ("file1.txt");
in_file.getline (a, sizeof (a));
in_file.getline (b, sizeof (b));
in_file.getline (c, sizeof (c));
cout << a << endl;
cout << b << endl;
cout << c << endl;
in_file.close ();}
```

As a result of the program displayed on the screen there will be:

1) An entry in the file.
2) Creation mode.
3) Append mode.

In case the number of lines in the file is not known, it is possible to provide information to the reading until EOF is encountered. Reaching the end of the file is checked using the eof, which returns a value of 0 (false), until it encounters the end of the file, and a value of 1 (true) if the file meets the end. Consider the example of reading the contents of the file file1.txt up until the end of the file is encountered.

```
#include <iostream.h>
#include <fstream.h>
void main (void) {
char a [64];
ifstream in__file ("file1. txt");
while (! in file.eof ())
{In_file.getline (a, sizeof (a));
cout << a << endl; }
in_file.close (); }
```

To read the contents of the file file1.txt one word up until the end of the file is encountered in the body of the while loop instead of the operator should be in_file.getline (a, sizeof (a)); insert the operator in_file >> a;

To read the contents of the file file1.txt one character up until the end of the file is encountered, when declaring a variable instead of char a [64]; In the example above, use char a;

File operations can be effectively used when working with arrays and structures. *To write a file and read from a file is not character strings, and complex data types such as arrays and structures,* are used *the record and read.*

In the examples below, to write and read one of the parameters is used by the operator cast (char *), which is a pointer to a character string [6-8].

Example 11.1. Record file array.

```
#include <fstream.h>
void main (void)
{Int i, x [3] = {10,20,30};
ofstream out_file ("file2.dat");
for (i = 0; i <3; i ++)
out_file.write ((char *) & x [i], sizeof (x [i]));
out_file.close ();}
```

Example 11.2. Reading an array of file and display.

```
#include <iostream.h>
#include <fstream.h>
void main (void)
{Int i, x [3];
ifstream in_file ("file2.dat");
for (i = 0; i <3; i ++)
{In_file.read ((char *) & x [i], sizeof (x [i]));
cout << x [i] << "";}
in_file.close ();}
```

As a result of consistent implementation of these two programs, on the display screen will be displayed: 10 20 30

Example 11.3. Recording a file structure.

```
#include <fstream.h>
void main (void)
{Struct date {int day; char month [9]; int year;} d = {25, "November", 1998};
ofstream out_file ("file3.dat");
out_file.write ((char *) & d, sizeof (date));
out_file.close (); }
```

Example 11.4. Reading from a file structure and display.

```
#include <iostream.h>
#include <fstream.h>
void main (void)
{Struct date {int day; char month [9]; int year; } d;
ifstream in_file ("file3.dat");
in_file.read ((char *) & d, sizeof (date));
cout << d.day << "" << d.month << "" << d.year << endl;
in_file.close (); }
```

As a result of consistent implementation of these two programs on the display screen will be displayed: November 25, 1998.

Lecture № 12. The functions and their parameters. Recursion

The goal - to get an idea about the features of the development and use of user-defined functions.

Besides the standard functions, located in the header files, which are connected to the program with operator #include <imya.h> languages C / C ++ allows the user to form their own functions. This function is advisable to create, in solving problems if there is a need to carry out calculations on the same algorithm multiple times. Application functions allows you to split the program into simple, easily controlled side, as well as to conceal non-essential parts of the program to other parts of its implementation. Thus, the function is logically complete and designed in accordance with the requirements of the language program fragment.

The functions are placed in the program in any order and considered global to the program. When using the features necessary to distinguish description function and the function call operator. The structure of the function is similar to the structure of the program main. Description of the function contains a header functions ads of variables and operators:

```
Function_type function_name (the list of formal parameters)
{Variable declaration;
statement1;
...;
operatorN; }
```

Here: *Function_type* - the return type of the main program result; function_name - a unique name within the meaning of the relevant operations that perform the functions (for example, max definition maximum of two numbers); a list of formal parameters and their types.

Formal parameters - here is the variable through which information is transmitted from the main program to the function.

To call it is enough to write its name with a list of the actual parameters in any expression causing programs:

function_name (list of actual arguments);

Here: list of actual parameters. The actual parameters are names of variables whose values are passed to the function, assigned to the corresponding formal parameters.

To return the calculated value to the main function in a program is used the operator return (result);

Consider the examples of the use of functions in the programs.

Example 12.1 – shows that function does not contain parameters and returns to the main routine.

```
#include <iostream.h>
#include <conio.h>
void show (void)
{cout << "function show" << endl; }
void main (void)
{Clrscr ();
cout << "BY3OB functions show" << endl;
show ();
<< sout "Return to the main program" << endl;
getch ();}
```

Example 12.2 - max function contains the formal parameters and returns to the main program.

```
#include <iostream.h>
#include <conio.h>
void max (float x, float y) // x, y - the formal parameters
{if (x> y) cout << x << ">" << y << endl;
else cout << y << ">" << x << endl;}
void main (void)
{Clrscr ();
float a, b;
cout << "Enter a and b" << endl; cin >> a >> b;
max (a, b);
getch ();}
```

Example 12.3 - max function contains the formal parameters and returns to the main program value. The return value is of type float.

```
#include <iostream.h>
#include <conio.h>
float max (float x, float y)
{Float result;
if (x> y) result = x; else result = y;
return (result);}
void main (void)
{clrscr ();
```

```
float a, b;
cout << "Enter a and b" << endl; cin >> a >> b;
sout << "The largest of the two numbers" << a << "and" << b << ":" << max (a, b) << endl;
cout << " The largest of the two numbers 2.71 and 3.14:" << max (2.71,3.14) << endl;
getch ();}
```

In the examples preceded by the description of the function invocation. Otherwise, the beginning of the program you want to place a function prototype - ie object that contains the name information function return value type, quantity and type of formal parameters, for example, float max (float, float);

The values of formal parameters of functions can be used by default. For example,

```
#include <iostream.h>
show void (int a = l, b = 2 int, int c = 3)
{Cout << a << "" << b << "" << c << endl;}
void main (void)
show ();
show (4);
show (5,6);
```

In the title, the show function takes an assignment of values to the formal parameters a, b and c. At the first function call actual parameters are not available, the second - specified value for and, at the third - the values for the parameters a and b. The example shows, that, if the value of the parameter is omitted, it brings to lowered values of all subsequent parameters. As a result of the program on the screen will be displayed: 123 423 563. Thus, if the function call values are missing some actual parameters, the function uses default values of formal parameters defined by the user instead.

In order to improve the visibility and legibility of custom programs function overloading is used, which allows you to use the same name for several functions that solve the same problem, but with a different number and different types of formal parameters. In order to implement the function overloading in the program simply the functions of the same name are described. In addressing the compiler program independently, without any additional guidance required functions are determined on the basis of a list of the actual parameters to the function call operator. Consider the example of overloading the functions in the program.

```
#include <iostream.h>
int sum (int a, int b)
{Return (a + b);}
float sum (float a, int b, int c)
{return (a + b + c);}
void main (void)
{cout << sum (2, 3) << endl;
```

cout << sum (2.5,3,4) << endl;}

The example defines two functions sum with different quantity and types of formal parameters. As a result, the program *at first performs an operation call* to the first function sum, which adds the two values of the (2 + 3 = 5) of type int. Then it is addressed to the second function sum, which adds three values (2.5 + 3 + 4 = 9.5), and returns to the main program result of type float.

One feature of using program functions is the declaration of local and global variables. Local variables are declared within a function in the same way as the main inside of main. Local variables are valid only within the function in which they are declared. Global variables declared at the beginning of the program is a function. Global variables are available for any function in the program.

If the local and global variables have the same name, the variable in the function is perceived by the compiler C / C ++ as a local variable. If inside a function you need to use global variable, coinciding with the name of local variable, in this case, you must use global operator permission:

:: Variable_name

For example,

```
#include <iostream.h>
int xg = l; // declare a global variable
void show (int xg)
{Cout << "The local variable xg:" << xg << endl;
cout << global variable xg: "<<:: xg << endl;}
void main (void)
{Int y = 0;
show (y); }
```

As a result of the program on the screen of the display will be shown:

Local variable xg: 0

Global variable xg 1

Since the values of global variables can be easily modified any function using their programs is not recommended.

In solving computational problems recursive functions (functions that call themselves) are often used. Recursive functions that directly call themselves, are called recursive inclusive. If the two functions call each other, they are called mutually recursive.

Solution of recursive task is usually divided into several stages. One step is to solve the basic problem, i.e. the simplest tasks for which are written by the called function. If the task is more complicated than the base, it is divided into two sub-tasks: the basic allocation of tasks and all the rest. In this part, which is not a core task it should be easier than the original problem, or recursion can not be completed. The process continues until it is reduced to the solution of the basic problem [6-8].

Each call to a recursive function is called a recursive call or step recursion. It is advisable to use recursion to calculate the factorial, raising to a power, calculating Fibonacci and others. For example, consider the recursive calculation of factorial. By definition factorial: n! = N * (n-1) * (n-2) * ... * 2 * 1, and 0 = 1, 1 = 1.

The recursive solution is to repeatedly call the factorial function from the function. For example, when calculating the 5! 1 calculation! - The basic problem, as well as an indication of terminating the recursion. Thus, 1 = 1 - 1 returns

2 = 2 * 1 = 2 * 1 = 2 - returns 2

3 = 3 * 2 = 3 * 2 = 6 - 6 returns

4 = 4 * 3 = 6 * 4 = 24 - 24 returns

5 = 5 * 4 = 5 * 24 = 120 - 120 returns

Lecture № 13. Pointers and references

The goal - to get an idea about the features of variables such as pointers and references, as well as their use.

Indicators are considered to be an extremely powerful tool in programming. They can be used to simplify and improve the efficiency of programs. For example, using pointers, you can change the values of variables inside a function, with the variables parameters can be passed to the function. In addition, pointers can be used for dynamic memory allocation, which means that it is possible to write programs that can handle a virtually unlimited amount of data. Pointers are similar to tags that refer to locations in memory.

Imagine a bank safe deposit boxes with different sizes. Each cell has a unique number, which is linked only to the cell, so that you can quickly identify the desired cell. These numbers are similar to the cells of computer memory addresses. For example, the client keeps all your valuables in the safe. At the same time, to protect their savings, he decides to have a little safe, which will be based on a map showing the location of the large safe, which holds the real jewels and password. As a result, the safe with one card will store the location of the other safe. This organization is equivalent to a savings of jewelry pointer in C / C ++.

Pointers - variables that store addresses of other variables in the memory. Knowing the address of a variable, you can go to that address and retrieve the data stored in it (figure E.1). If you need to transfer large amounts of data to function it would be much easier to pass the address into memory where the data is stored than copy each element.

The principle of the cursor declaration is the same as the principle of variable declarations. Visually the only difference is that the name is placed before the *asterisk *: * datatype variable_name-pointer;*

When declaring pointers compiler allocates a few bytes of memory, depending on the type of data to be allocated to store certain information in memory. Furthermore, if one line declares multiple pointers, each of them must be preceded by an asterisk *.

Before you can use the pointer, it must be initialized in order to point to a specific memory address. Otherwise, the pointer will refer to whatever portion of memory that can lead to extremely unpleasant consequences. Moreover, the operating

system will prevent a program to access an unknown area of memory, because it knows that the program is not initialized the pointer, the result will lead to a crash.

For example, to put in a pointer to the variable var ptrvar address is required to initialize:

*int * ptrvar; // Declare pointer*

ptrvar = & var; // Initialize pointer

To get the value stored in a region referred to by the pointer, you must use a pointer dereference operation * or indirect addressing. To this end it is necessary to put an asterisk before the name and get access to a pointer value, for example,

** ptrvar.*

Pointers can be compared, because the address can be smaller or larger relative to one another, while using dereference operations can be compared and stored values of variables. In addition, over the pointers can be performed arithmetic operations such as addition, subtraction, increment and decrement.

Dereference of pointers to variables declared types can produce on them is the same as that of the corresponding types of variables.

The unary operator & returns the address-address of an object in memory, so it can be used to calculate the addresses of variables and functions, but it is impossible to evaluate expressions and symbolic constants declared using #define, since these entities do not have addresses.

Since the signs are limited to a given type of data checking is done at compile time type. For example,

*float * dPtr; char b;*

dPtr = & b; // will cause a compilation error.

In C ++, there are null pointers - pointers which currently do not address any allowable value in memory. The value of a null pointer is zero. This is the only address to which the pointer does not have access. Therefore, the program should be used kind of test: if (dPtr! = NULL) statement;

In this case, specified in terms of the operator will be executed only if dPtr, which addresses reliable data. Assigning dPtr = NULL or dPtr = 0 is initialized and can protect against mistakes.

In addition to zero in C ++, you can find pointers to type void - generalized pointers for addressing the data, without the need to pre-determine their type: void * SomethingPtr;

Thus pointer to void may be zero. Their main application - addressing of buffers, filling memory blocks, reading hardware registers.

Pointers can reference other pointers. In the memory cells, which will be referred to by the first pointer will contain no value, and second address pointers. * The number of characters in the cursor declaration show its order. To access the value referenced by a pointer, you must dereference the appropriate amount of time. For example,

int var = 123; // Initialization of the variable var number 123

*int * ptrvar = & var; // Pointer to the variable var*

*int ** ptr_ptrvar = & ptrvar; // Pointer to a pointer to the variable var*

Thus, the logic of the n-fold dereferencing is that the program iterates through all addresses until the pointer variable that contains the value.

Links - a special type of data, which is a disguised form of a pointer that is automatically dereferenced using. Link can be declared as a new name and an alias variable referenced. Ad links is as follows:

*/ * type * / & / * * / = / * variable_name * /;*

When you declare a reference to its name became a symbol of &, the link itself must initialize the variable name to which it refers. The type of data link could be anything, but must be the same object, which it is referred to, - the type of data reference variable. Any change to the values contained in the link, will entail a change in the value of the variable to which it refers. For example,

int value = 15;

int & reference = value; // Declaration and initialization links

The main purpose of the index - an organization of dynamic objects whose size can change (increase or decrease). While the references are for the organization of direct access to a particular object. The main difference is in the internal mechanism works. Pointers refer to the area in memory using its address. A reference link to an object by its name (the same kind of address).

Links are generally used in most cases in the function parameters as links or references argument. When there is a transfer of value, the transmitted data must first be copied, and when a large amount of data on the transfer of spent a lot of time and resources. In this case, you must use the transmission link, and there is no need to copy the data, as provided to them direct access. However, this violates the security of data stored in reference variables, since for them a direct access [1, 5, 7].

Lecture №14. Dynamic memory allocation. Using pointers in solving problems

The goal - to get acquainted with the functions of allocation and deallocation of dynamic memory and using them to get an idea of the possibilities of using pointers when working with complex data types (arrays, structures, files) and functions.

When working with a program it becomes necessary to allocate memory between objects of the accessible program. At the same time for global and local variables, which are called static and are declared directly in the program, memory is allocated at compile time for the duration of the program.

For objects whose size can not be determined in advance (for example, images, files, structures) dynamic allocation of memory is used. Dynamic memory is a collection of various types of memory that is allocated and freed on the orders of the programmer during program execution, in any place, in accordance with an algorithm for solving the problem. To access the dynamic memory pointers are used. With dynamic allocation of memory objects are placed in the "pile» (heap) - specially organized storage area of variable size. Moreover, the design of the object indicates

the size requested by the memory object, and, if successful, the allocated memory, so to speak, "withdrawn" from the "pile" is no longer available on subsequent allocation. Dynamic variables are also called addressable pointers. As you create new objects in the program, the amount of available memory decreases. Hence there is a need to constantly release previously allocated memory. In an ideal situation, the program must fully release all of the memory that is required for the job.

Incorrect memory allocation leads to its "leaks", i.e. situations when the allocated memory is not freed. Multiple memory leaks can lead to the exhaustion of the entire RAM and disrupt the operating system. Therefore, the operation opposite to the operation within the meaning of memory allocation object - is the liberation of occupied previously by an object memory frees up memory, so to speak, returns to "bunch" and is available with further operations allocation.

Because dynamic memory allocation memory is not reserved at compile time and run-time program, it gives an opportunity to allocate memory more efficiently, mainly in regard to arrays. With dynamic memory allocation is not necessary to pre-set the size of the array, the more that do not always know what size to be generated from the array.

For the allocation and deallocation of dynamic memory in C / C ++ using special functions.

The function malloc () - memory allocation - is defined in the header file alloc.h and is used to initialize the required memory pointers. Memory is allocated from the sector of RAM available to all programs running on the computer. The argument to malloc () function is the number of bytes of memory that should be allocated, the function returns - a pointer to the allocated block of memory. Since different types of data have different storage requirements, the amount of addressable memory is calculated by an operation sizeof (). For example, when you select the index ptrVar necessary amount of memory for storing an int you can use the

*int * ptrVar = malloc (sizeof (int));*

The size of allocated memory may be determined by passing a null pointer. For example, int ** ptrVar = malloc (sizeof (* ptrVar));*

Operation sizeof (* ptrVar) to estimate the size of the memory that is referenced by a pointer, as well as the ptrVar a pointer to the memory location of type int, the sizeof () returns the size of the integer.

Backup Operators can refuse if the bunch is already full or the remaining memory is less than the required number of bytes. In this situation, malloc () function returns a null value.

Automatically selected area of memory is no longer available for other programs. Therefore, after the allocated memory becomes unnecessary, it must be explicitly freed. Deallocation is executed by using the free () from the same library. For example, free (ptrVar);

Specify the size of the released memory is not necessary, because the He remembered a few bytes, each adjacent to a reserved block, and coincides with the amount initially reserved memory is associated with the pointer.

After the liberation of memory is good practice to reset the pointer to zero, that is, to assign * ptrVar = 0 ;. If the pointer to assign 0, it becomes zero, in other words, he has nowhere indicated. Otherwise, even after the release of memory, the pointer will still point to it, which means that you can accidentally harm the other programs that may be using this memory.

For backup memory instead of malloc () function, you can use the salloc (), the prototype of which was also declared in the header file alloc.h. This function also allocates memory on the heap, but it requires two arguments: the number of objects that must be placed, and the size of the object. For example, to reserve memory for 10 values of type long pointer and assign the first value sPtr address, you can use the long * sPtr = salloc (10, sizeof (long));

Function parameters salloc () can be specified in any order, ie, salloc (num, size) and salloc (size, num) reserve memory of num * size bytes. In addition to allocating memory, this function sets each byte reserved by it to zero, ie, initializes memory.

In C ++, operators also have to allocate and free heap memory, which was not in the C:

1) New - automatically creates a corresponding size in the memory and returns the address; if memory allocation has occurred, it returns a null pointer. For example, int * iPtr = new int;

In addition, when backing up memory via the new operator may be assigning a value created object at once:

*float * fPtr = new float (17.245);*

2) Delete - before the release of the reserved memory, the size of the memory liberated to ask is not required: delete iPtr;

When working with new and delete operators to connect the header file alloc.h not necessary.

Keep in mind, an attempt to release the same memory more than once, leading to system failure. Partially or fully these problems are solved creation of C ++ classes in the STL and Boost, implementing "smart pointers".

Pointers can address all kinds of variables - from simple integer variables of type int to complex types such as arrays and structures. Furthermore, they are not interchangeable when working with features.

In C / C ++ when working with arrays of pointers is very convenient to use because they can be used to organize an economical and quick access to any element of the array. Figure E.2 shows the relationship between the elements of arrays and pointers. It allows you to apply mPtr pointer to the element m [0], mPtr + 1 points to the next element, and mPtr + i on the i-element array M.

Using pointers when working with arrays saves memory. If the size of the array is not specified or required to use only part of the elements of an existing array, it is convenient to use dynamic memory using pointers. For this purpose it is necessary:

- to declare a pointer to the desired data type;
- later in the program to call malloc () function to allocate memory on the heap for an array or salloc () to initialize the values of an array of zeros;

- work with an array as usual, but after the release of its use with memory function free ().

For example, to allocate memory for an array of 100 elements of type double should perform these steps:

*double * BigArrayPtr;*

...

*BigArrayPtr = malloc (100 * sizeof (double));*

or BigArrayPtr = salloc (100, sizeof (double)); // Initialize the values of zero

...

for (int i = 0; i <100; i ++)

BigArrayPtr [i] = random (50);

...

free (BigArrayPtr);

To reserve a memory array and its subsequent release, you can use the operator new and delete:

*double * BigArrayPtr = new double (100);*

...

delete [] BigArrayPtr;

It should be noted that the removal of the temporary array delete operator must specify the brackets, since it is impossible to initialize a dynamic array and the results can be unpredictable.

When working with two-dimensional arrays, you can represent them in the memory as one-dimensional, and for allocating memory, use the following formula:

$$i = Nx * iY + iX,$$

where *i_* - the current value of the index in the one-dimensional array;

Nx - the number of columns in the original matrix;

iY, iX - index of the current element in the original matrix.

String pointers are addresses that specify the location of the first character string stored in the memory. String pointers are declared as char * or to the operators could not change them addressable data as const char *. Since the lines are character arrays, so over them is permissible perform the same actions as the above arrays. When you select a row of memory on the heap, you can use the functions malloc () and free ().

Budget accommodation in the memory of large structures is also advisable to use a bunch and not global or local variables. To do this, declare the structure and then a pointer to the structure type:

*struct name of structure * name of pointer;*

For example,

struct date

{Int day; char month [9]; int year; } d;

...

*struct date * dat_Ptr;*

Then use the malloc () function to allocate memory on the heap and assign the address of a pointer:

*Index = (struct tip_struktury *) malloc (sizeof (struct tip_struktury));*
*For example, dat_Ptr = (struct data *) malloc (sizeof (struct data));*

The addressable memory block pointer corresponds to the size of one structure of this type (in this example, the type of data). However, to use the usual way to access the structure fields can not, therefore, the organization of access to the program must be in place treatment

tip_struktury. ElementName = value;

to apply treatment

tip_struktury ¬-> ElementName = value;

Operator access to the field structure ¬-> points to the location in the memory with respect to the structure.

In normal function call the C compiler / C ++ creates a copy of the actual values of parameters and places the copy on the stack - the section of computer memory for temporary storage of information. The values of the parameters from the stack assigned to the formal parameters of the function. After completion of the function cleans the stack. Thus, the values of the actual parameters in the main memory of the computer after the completion of the function remain changed.

In case you need to change the values of the actual parameters, ie, Assignments of formal exercise options when the function actual parameters main program should use IP-address operator (& sign) and pointer variables (sign *).

The operator addresses for the transfer addresses meaning of actual parameters from the main program to the function. An operator address is:

function_name (& faktich_parametr1, ... & N);

In order to "explain" to the compiler C ++, that the values of the parameters fakti¬cheskih be transferred from the main program to the function using the address (the stack will be written to the address of the actual values, but not their values) are declared in the function header pointer variables (formal parameters) from * feature:

Function_type function name (* form_par1 type ... type * form_parN)

The variables (formal parameters) that uses operators in function must also be marked (*).

```
#include <iostream.h>
void change (int * x, int * y)
* {X = 100; * y = 200; }
void main (void)
{Int a, b; a = 10; b = 20;
cout << "The parameters a and b to access the function:" << a << "" << b << endl;
change (& a, & b);
cout << "parameters b after the call to the function:" << a << "" << b << endl;}
```

As a result, after the completion of the function parameters faktiches¬kie changed their values, and the screen displayed bu¬det:

The parameters a and b to access the function: October 20.

The parameters a and b after the function call: 100 200.

The values of actual parameters may be changed in the program and with the help of links that allow you to create aliases for the variables used as parameters to the function. To declare a reference used an ampersand (&) after the parameter type:

& type = imya_ssylki variable_name;

After the announcement of the reference in the program can use the variable name or the name of the link. As an illustration, change the values of the actual pa¬rametrov using the links look at an example:

```
#include <iostream.h>
void change (int & x, int & y)
{X = 100; y = 200; }
void main (void)
{Int a, b;
int & as = a; // declaration of links as - alias variable and
int & bs = b; // declare links bs - alias the variable b
a = 10; b = 20;
cout << "the parameters a and b to access the function:" << a << "" << b
<< endl;
change (as, bs);
cout << "the parameters a and b after the call to the function:" << a << ""
<< b << endl;}
```

As a result, the program will be displayed on the screen :

The parameters a and b to access the function: October 20.

The parameters a and b after the function call: 100 200.

Thus, the output of the program is fully coinside with a result of the program, considered in previous example. But we should not get used to frequent use of links, so it is difficult to understand how the program [1, 5, 7].

Pointers can also refer to the function. The function name, as well as an array name by itself is a pointer, that is, contains the address of entry:

*/ * data type * / (* / * pointer name * /) (/ * argument list function * /);*

The data type is determined according to the type returned by a function, which will link pointer. Symbol index and its name in parentheses are taken to show that it is - a pointer instead of a function that returns a pointer to a particular type of data. After the name of the index in parentheses, separated by commas lists all function arguments, as in the function prototype declaration.

And finally, a pointer to a file (file pointer) - a pointer to the information that determines its characteristics (name, status, current position), namely the structure of type FILE, which is defined in the header file stdio.h. To declare a pointer to the file is used by the operator FILE * fPtr;

Lecture №15. Using graphic language features

The goal - to explore the possibilities and characteristics of using graphics.

Package management functions screen is divided into two parts according to the capabilities of the computer: working in text mode (text mode), and work in graphical mode (graphics mode). All control functions screen in text mode have their prototypes in the header file conio.h. The screen in the graphic mode is done using a set of function prototypes are in the header file graphics.h.

Among the functions of setting the graphics mode should be the following:

1) Void far detectgraph (int far * gdriver, int far * gmode); - Is designed to define the type of graphic adaptor. This function returns a value to the addresses given first and second parameters. There gdriver - a pointer to an integer containing the number of the graphics driver. The second parameter is a pointer to detectgraph integer containing the number of graphics mode provides maximum screen resolution. If there is no graphics card, the function assigns detectgraph * gdriver = -2.

2) Void far initgraph (int far * gdriver, int far * gmode, char far * pathdriver); - Is designed to download the graphics driver, having a number * gdriver, and to install a graphics mode number * gmode. If at gdriver recorded zero (* gdriver = DETECT), the function accesses a function detectgraph, and then loads the driver, the number of which has been set function detectgraph. Needed memory to load the driver is available in the "heap". Depending on the graphics mode the maximum screen resolution is changed.

To upload graphics driver applies the function, declared as void close graph (void). This function frees the memory of the "heap", occupied by the graphics driver.

Initialize the graphics mode is as follows:

```
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
void main ()
{Int driver, mode; driver = registerbgidriver (EGAVGA_driver);
driver = VGA; mode = VGAHI;
initgraph (& driver, & mode, "");
closegraph ();}
```

After installing the graphics mode using initgraph monitor screen is a rectangular area, divided into equal squares - pixels, the sides of which are parallel to the top and bottom of the screen. A pixel (pixels) - this is the minimum element of the image on the screen, consisting of several (color) of the points covered by the program and as a single point of a certain brightness or color. Under the coordinates of the pixel refers to the integer coordinates of the centers of these rectangles, measured from the upper left coordinates of the center of the rectangle. To calculate the screen resolution in x and y, applied function int getmaxx (void), int getmaxy (void). The most commonly used graphic mode display, in which supports a resolution of 640 * 480 * 16. Here the 16 - the maximum number of colors which can be simultaneously present in the image.

The header graphics.h constants corresponding to the standard color palette. Changes in any of the standard color palette is made function

void setpalette (int index, int color);
where intindex - number of standard palette;
int color - color in the range from 0 to 63 (EGA palette).

Setting EGA palette function is performed

void setrgbpalette (int color, int red, int green, int blue);

where red, green, and blue are changed in the range from 0 to 255, where a small value corresponds to dark colors, large - more vivid.

Graphic function, which can be run from a graphical mode, are divided into three groups:

a) consists of functions that do not display on the screen, but set some parameters, for example, feature sets setcolor color number for further output lines;

b) consists of the functions that display, for example, to get the point of a given color, apply a function void putpixel (int x, int y, int color);

c) consists of functions that do not display on the screen, but provide information on the output image, for example, to read the color of a pixel is a function of unsigned getpixel (int x, int y).

Function names begin with the first group of words set (to put, put), while the third group - with the word get (get, get).

Graphics functions produce output in a page, which is called active.

The compression ratio is the ratio of the screen pixel width to its height. The compression ratio can be found by using the void getaspectratio (int far * xasp, int far * yasp);

The function writes at yasp number 10000 and at xasp - the product of the compression ratio to 10000. The compression ratio is taken into account in the derivation of circles, arcs, and sectors of a circle. The compression ratio is set at initialization graphic mode based on the mode corresponding to the maximum resolution of the screen, and can be changed by using the void setaspectratio (int xasp, int yasp) ;.

Group lines in the plane forms a contour shape (line segment, arc, circle, rectangle, ellipse, and so on. D.). Further forms, shapes can vary the color line (circuit), or the type of its thickness. By default, in graphics mode, the following settings: the current outline color - WHITE (white), thickness - one pixel type - solid line. Planar figures - are fragments of the plane of the screen bounded by a closed loop. They can be obtained from the contour by painting the area inside or outside a closed solid line forming. To display the most frequently used shapes, changes in the types of contour lines and their parameters you can use the functions of the standard graphics library graphics.

The output text in the graphics mode can be performed using the same prototype function library. Text information is displayed within the parameters: color, font type, font size and direction. Character Size (horizontal and vertical) is defined as the product of a standard size (8 × 8 pixels) on charsize option, that is, if the value is equal to charsize 3, each symbol is displayed on the screen, it will be inscribed in a square 24 × 24 pixels. Setting font, specifies the font style, connected to the program

files with the extension * .chr (non-standard "fonts"), so you must make these files available (the easiest way to copy them to the current directory).

To display text on the screen in graphics mode can be used for text-mode function, but they have several disadvantages. For example, using printf () function to display text on any colored background behind the label will be its "background" (black rectangle equal to the length of the displayed text). It is also not possible to change the appearance of the displayed text (font size, style, etc.) [5-8].

Appendix A

Features Block Diagram

Table A.1 - Basic elements of block diagrams

Nomination	Designation	Note
Terminator		It displays the output of the external environment or the input from the external environment. It used to indicate the beginning or end of the algorithm
Data		Displays the data storage medium is not defined. It used to indicate input and output operations
Process		Displays function data of any kind. It used to denote the assignment operation
Preparation		It used to indicate the cycle header
Decision		Used to designate the operator of conditional branch or select statement (variant)
Predefined process		Displays predetermined process consisting of one or more operations which are defined, for example, in the subroutine or module. Commonly used to mean routines
connector		Displays the output and input of the circuit from another part of the same scheme is used to break the line and continue it in another place
Line		Displays the data flow or control. From left to right and from the bottom up are indicated by arrows. Is used to connect symbols in the algorithm
Comment		Used to add a descriptive comment or explanatory records for the purpose of explanation or notes
Note - The symbols may be drawn in any orientation, but is preferably a horizontal. Inside the symbol is placed, or designations to describe operations		

Appendix B

Table B.1 – Special and control symbols

	Form	Name	Form	Name
Special symbols	+	Plus	|	Line
	++	Increment	||	Logical OR
	-	Minus	!	Exclamation point
	--	Decrement	->	Arrow
	*	Star	=	Assignment operation
	/	Oblique line	= =	Equality sign
	\	Reverse slash	!=	Not equal
	//	Double slash	>	More
	.	Point	<	Less
	,	Comma	<=	Less or equal
	:	Colon	<<	Left shift
	::	Permission	>>	Right shift
	;	Semicolon	< >	Angle brackets
	‘	Apostrophe	( )	Round brackets
	“	Quotes	[ ]	Square brackets
	^	«Cover»	{ }	Curly brackets
	%	Percent sign	/* */	Comment brackets
	&	Ampersand	#	Sign
	&&	Logical AND	~	Tilde
Control symbols	\a	Speaker signal	\t	Horizontal tabulation
	\b	BS, character drift	\v	Vertical tabulation
	\f	New page	\\	Backslash
	\n	New line	\0	Null character
	\r	Carriage return	\000	Octal constant
	\”	Double quote	\xhhh	Hexadecimal constant
	\’	Apostrophe	\?	Sign of question

Table B.2 – Reserved words in C++

and	char	false	int	private	Switch	virtual
and_eq	class	float	long	protected	Template	void
asm	compl	for	mutable	public	this	volatile
auto	const	else	namespace	register	throw	while
bitand	continue	enum	new	return	true	xor
bitor	default	explicit	not	short	try	xor_eq

bool	delete	friend	not_eq	signed	typedef	
break	do	goto	operator	sizeof	typename	
case	double	if	or	static	union	
catch	extern	inline	or_eq	struct	unsigned	

Table B.3 – Types of data with different

Types	Range of measuring		Size in bytes (bits)
	From	Till	
void	-	-	0
char (signed char)	-128	127	1 (8)
unsigned char	0	255	1 (8)
wchar_t	0	65535	2 (16)
bool	True	False	1 (8)
int (signed int, short int, signed short int)	-32768	32767	2 (16)
unsigned int (unsigned short int)	0	65535	2 (16)
long int (signed long int)	-2147483648	2147483647	4 (32)
unsigned long int	0	4294967295	4 (32)
float	3.4E-38	3.4E+38	4 (32)
double	1.7E-308	1.7E+308	8 (64)
long double	3.4E-4932	3.4E+4932	10 (80)
Note - The size in bytes and the range of variation may vary depending on the compiler, processor, and operating system (environment)			

Table B.4 – List of operations ,its priority and order execution

Level	Operator	Order	Level	Operator	Order
1	( ) . [ ] -> ::	⇒	9	&	⇒
2	* & ! ~ ++ -- + - sizeof new delete	⇐	10	^	⇒
3	. * -> *	⇒	11	|	⇒
4	* / %	⇒	12	&&	⇒
5	+ -	⇒	13	||	⇒
6	<< >>	⇒	14	?:	⇐
7	< <= > >=	⇒	15	= *= /= += -= %= <<= >>= &= ^= |=	⇐
8	== !=	⇒	16	,	⇒

Notes
1 The highest priority is given to level 1 operators, the lowest priority is level 16.
2 The symbol ⇒ indicates the execution of operations from left to right, and the sign ⇐ - execution of operations from right to left.
3 The unary operators (+) and (-) at level 2 have a higher priority than arithmetic (+) and (-) at level 5. The symbol & at level 2 is the address operator, and the & at the level 9 bit AND operator. The symbol * at level 2 is the operator for

accessing the pointer, and the symbol * on level 4 is the operator of multiplication. 4 In the absence of parentheses, operators at the same level are processed according to their location from left to right.

Table B.5 – Main mathematical functions

Name of function	Function	Type		Head file
		Of result	Of argument	
Absolute value	abs(x) cabs(x) fabs(x)	int double float	int double float	<stdlib.h> <math.h> <math.h>
Arccos	acos(x)	double	double	<math.h>
Arcsin	asin(x)	double	double	<math.h>
Arctg	atan(x)	double	double	<math.h>
Cos	cos(x)	double	double	<math.h>
Sin	sin(x)	double	double	<math.h>
Exponent e^x	exp(x)	double	double	<math.h>
Power function x^y	pow(x,y)	double	double	<math.h>
Natural logarithm	log(x)	double	double	<math.h>
Decimal Logarithm	log10(x)	double	double	<math.h>
Square root	sqrt(x)	double	double	<math.h>
Tangents	tan(x)	double	double	<math.h>

Table B.6 – Conversional characters in input and output functions

Form of output	Values	Form of input	Values
%c	Output symbols (char)	%c	Reading of symbols (char)
%d	Decimal integer (int)	%d	Decimal integer (int)
%i	Decimal integer (int)	%i	Decimal integer (int)
%e (%E)	number (float/double) in form x.xx e+xx (x.xx E+xx)	%e	Reading number in form float/double
%f (%F)	number float/double with fixed comma xx.xxxx	%h	Reading number in type short int
%g (%G)	Number in form f (F) or e(E) depending from value	%o	Reading of octal number
%s	Line of symbols	%s	Reading of line symbols

%o	Integer number (int) in octal form	%x	Reading in hexadecimal number (int)
%u	Unsigned decimal number(unsigned int)	%p	Reading of index
%x (%X)	Integer number (int) in hexadecimal form	%n	Reading of index in increased form
%p (%n)	Index		
Note - You can use the l and h modifiers for formats, for example,% ld (long in decimal),% lo (long in octal form),% lu (unsigned long),% lx (long in hexadecimal),% lf (long float with a fixed point),% le (long float in exponential form),% lg (long float as f or e depending on the value).			

Table B.7 – Characteristic methods of programming

methods of programming	Actions in cycle	Actions in cycle
Collection of *sum*	S = 0;	S+=*element of array*;
Collection of *multiplication*	P = 1;	P*=*element of array*;
Collection of *number*	k = 0;	k++;
Search of *maximum* value	max=*supposed_value*;	if (*present_element*>max) max=*present_element*;
Search of *minimum* value	min=*supposed_value*;	if (*present_element* <min) min=*present_element*;

Bibliography

1 Ivanova, G.S. Technology of programming. Technologiya programmirovaniya. - M.: ed. MSTU named after N. Bauman, 2002.

2 Ivanova, G.S. Basics of programming. - M.: ed. MSTU named after N. Bauman, 2001.

3 Terekhov, A.N. Technology of programming. - M.: BINOM Knowledge laboratory - Intuit.ru, 2006.

4 Wirth, N. Algorithms and data structures. – M.: Mir, 1989.

5 Van Tassel, D. Style, efficiency, development, adjustment and testing of programs. – M.: Mir, 1985.

6 Sommerville, Ian. Software engineering. - M.: Williams publ. house, 2002.

7 Cantor, M. Managing programming projects. “A Guide to Successfull Software Development”. - M.: Williams publ. house, 2002.

8 Bachman, P., Frenzel, M., Hanzschman, K., et al. Software systems. Programmnye systemy. – M.: Mir, 1988.

9 Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide. – M.: DMK Press, 2001.

10 Kaner, C., Falk, J., Nguyen, H. Computer software testing - Kiyev: «DiaSoft», 2000.

11 Гримм С.Дж. Как писать руководства для пользователей. – М.: Радио и связь, 1985.

12 Ашарина И.В. Основы программирования на языках С и С++.- М.: Горячая линия - Телеком, 2002.

Content

Lecture №1. Programming Technologies. Concepts and approaches. Classification of the software……. 3
Lecture №2. Features of the development of complex software systems……. 7
Lecture №3. Structural and non-structural programming. Basics of algorithms…... 11
Lecture №4. Algorithmic languages and requirements to them. Procedural languages…… 15
Lecture №5. Introduction to C ++. The structure and the steps of creating a program in C ++. Standard C ++…… 16
Lecture №6. Presentation of data in C ++. The assignment operator. Arithmetic operations. Preprocessor Directives…… 19
Lecture №7. I / O functions. Basic design of C ++……. 23
Lecture №8. Complex data types: arrays. One-dimensional and multi-dimensional arrays. Organization of sorting algorithms……. 28
Lecture №9. Complex data types: character arrays. Processing character data……. 32
Lecture №10. Complex data types: structures and unions……. 34
Lecture №11. Complex data types: files. File operations on arrays and structures... 36
Lecture №12. The functions and their parameters. Recursion……. 40
Lecture №13. Pointers and references……... 44
Lecture №14. Dynamic memory allocation. Using pointers in solving problems….. 46
Lecture №15. Using graphic language features……. 51
Appendix A……. 55
Appendix B……. 56
Bibliography…… 61

Additional plan 2017, pos. 14

Irbulat Turemuratovich Utepbergenov
Sholpan Nazarovna Sagyndykova

TECHNOLOGIES OF PROGRAMMING

Lecture notes for specialty
5B070200 – Automation and control

Editor L.Y.Korobeinikova
Specialist for standardization N.K. Moldabekova

Signed into print __. __. __.
Edition 60 copies
Volume 3,9 quires

Format 60x84 1/ 16
Typographical paper №1
Order _____ Price 1950

Duplicating copying bureau of
noncommercial joint-stock company
«Almaty University of Power Engineering & Telecommunications»
Almaty,126, Baitursynov str., 050013