

TOMSK POLYTECHNIC UNIVERSITY

S. A. Lopatkin, V. I. Reizlin

**COMPUTER TECHNOLOGIES
IN SCIENCE AND EDUCATION**

Textbook

Tomsk Polytechnic University Publishing House
2008

UDC 681.3.06(075.8)
BBC 32.973.я73
L89

Lopatkin S. A.

L 89 Computer Technologies in Science and Education: Textbook /
S. A. Lopatkin, V. I. Reizlin. – Tomsk: TPU Publishing House, 2008. –
235 p.

ISBN 5-98298-203-2

The concept of information technology, its stages of development, problems of applying, types of information technologies are considered in this textbook. The basics of programming in a high-level programming language C++ are set out. The manual contains the following parts: program structure, data type, operators, statements, functions, storage class and variables visibility, operations with arrays, classes, and basics of object-oriented programming.

The textbook is developed in the framework of Innovative Educational Programme of TPU on the direction “Power-saving, basic, special, and industrial discharge, radiation, and plasma-beam technologies”. The manual is prepared at the High Voltage Technology and Electrophysics and Informatics and System Design departments of TPU. The manual is intended for students of Master’s programme on a specialty 140200 “Technology and Physics of High Voltage”.

UDC 681.3.06(075.8)
BBC 32.973.я73

Reviewer

Doctor of Physics and Mathematics,
Professor of Tomsk State University

A. I. Potekaev

ISBN 5-98298-203-2

© Lopatkin S. A., Reizlin V. I., 2008
© Tomsk Polytechnic University, 2008
© Design. Tomsk Polytechnic University
Publishing House, 2008

CONTENTS

INTRODUCTION	9
--------------------	---

UNIT 1

1. INFORMATION TECHNOLOGIES	11
1.1. A DEFINITION OF INFORMATION TECHNOLOGY	11
A New IT	11
The Toolbox of IT	12
The Way IT Correlates with Information System	13
1.2. THE STAGES OF IT DEVELOPMENT	13
The Type of Tasks and Information Processing.....	13
Problems Facing the Society Informatization	14
The Advantage of Computer Technology	14
The Types of Technology Toolbox	15
1.3. THE PROBLEMS OF IT USE	16
The Methodology of IT Use	16
Variants of IT Implementation into an Enterprise.....	18
1.4. TYPES OF IT	19
IT of Data Processing	19
Management IT.....	21
Office Automation IT	22
Decision Making IT.....	27
Expert Systems IT.....	32
QUESTIONS	36

UNIT 2

2. THE FUNDAMENTALS OF C++ LANGUAGE	39
2.1. THE FIRST PROGRAMS.....	39
2.2. THE LANGUAGE ALPHABET	40
2.3. COMMENTS	41
2.4. DATA TYPES	42

2.5. INTEGER DATA TYPE	43
<i>char</i> or Symbol Type.....	43
<i>int</i> Type (Equivalent <i>short int</i>)	44
<i>unsigned int</i> Type	44
<i>long (long int)</i> Type.....	44
2.6. FLOATING-POINT DATA TYPES	45
2.7. STRING CONSTANTS, OR LITERALS.....	45
2.8. A PREPROCESSOR STATEMENT <i>DEFINE</i>	46
2.9. DECLARATIONS.....	46
2.10. <i>CONST</i> MODIFIER.....	47
3. EXPRESSIONS	48
3.1. ASSIGNMENT OPERATOR AND EXPRESSION	48
3.2. ARITHMETIC OPERATORS	49
3.3. RELATION OPERATORS	50
3.4. LOGICAL OPERATORS.....	50
3.5. BITWISE OPERATORS	51
3.6. SHIFTS	52
3.7. INCREMENT AND DECREMENT OPERATORS ++ AND --	53
3.8. TERNARY OR CONDITIONAL OPERATOR	54
3.9. SEQUENCING OF OPERATORS	54
3.10. OPERATOR PRECEDENCE AND CALCULATION ORDER	55
4. STATEMENTS.....	57
4.1. EMPTY STATEMENT	57
4.2. COMPOUND STATEMENT	57
4.3. DECLARATIONS.....	57
4.4. CONDITIONAL STATEMENT	57
4.5. THE <i>SWITCH</i> STATEMENT	58
4.6. THE <i>WHILE</i> STATEMENT.....	61
4.7. THE LOOP <i>DO-WHILE</i> WITH POST-CONDITION	61
4.8. THE <i>FOR</i> LOOP STATEMENT.....	62
4.9. THE STATEMENT OF UNCONDITIONAL JUMP	64
4.10. THE <i>BREAK</i> STATEMENT	64
4.11. THE <i>CONTINUE</i> STATEMENT	64
4.12. THE <i>RETURN</i> STATEMENT.....	65
5. THE <i>sizeof</i> OPERATOR.....	66
6. DECLARATIONS AND DEFINITIONS	67

7. NAME RESOLUTION.....	68
7.1. NAME RESOLUTION IN A LOCAL SCOPE (BLOCK).....	68
7.2. NAME RESOLUTION IN A FUNCTION	69
7.3. NAME RESOLUTION IN FUNCTION TEMPLATE DEFINITIONS (FUNCTION PROTOTYPE).....	69
7.4. NAME RESOLUTION IN A FILE	69
7.5. NAME RESOLUTION IN CLASS SCOPE	69
8. VISIBILITY SCOPE	70
9. MEMORY CLASSES	72
10. OBJECT AND TYPE DECLARATION.....	74
11. TYPE NAMES.....	75
12. A SYNONYM OF A TYPE NAME	76
13. TYPE CONVERSION RULES	77
13.1. EXPLICIT CONVERSIONS.....	77
13.2. IMPLICIT CONVERSIONS OF A STANDARD BASE TYPES	78
13.3. THE CONVERSION OF DERIVED STANDARD TYPES.....	79
14. POINTERS	80
14.1. DEFINITION OF POINTERS	80
14.2. POINTERS AND ARRAYS	81
14.3. ADDRESS ARITHMETIC.....	83
14.4. SYMBOL ARRAYS AND STRINGS	83
14.5. MULTIDIMENSIONAL ARRAYS.....	85
14.6. POINTERS AND MULTIDIMENSIONAL ARRAYS.....	86
15. DYNAMIC MEMORY MANAGEMENT OPERATORS.....	88
15.1. THE OPERATOR <i>NEW</i> FOR MEMORY ALLOCATION.....	88
15.2. THE OPERATOR <i>DELETE</i> FOR MEMORY DEALLOCATION	89
ADVICE.....	91
EXERCISES	93

UNIT 3

16. FUNCTIONS	97
16.1. FUNCTION DEFINITION AND CALL	97
16.2. ARGUMENT PASSING	99
16.3. MULTIDIMENSIONAL ARRAYS PASSING	101
16.4. THE POINTERS TO THE FUNCTIONS	103
16.5. REFERENCES	105
16.6. REFERENCES AS FUNCTION PARAMETERS	106
16.7. THE ARGUMENTS BY DEFAULT	107
16.8. FUNCTION OVERLOADING	107
16.9. FUNCTION TEMPLATES	109
ADVICE.....	112
EXERCISES	113

UNIT 4

17. CLASSES	115
17.1. DECLARATION OF CLASSES.....	115
17.2. CONSTRUCTORS	118
17.3. DESTRUCTORS	121
17.4. STATIC MEMBERS OF THE CLASS	122
17.5. <i>THIS</i> POINTER	123
17.6. STATIC MEMBER FUNCTIONS.....	124
17.7. THE POINTERS TO THE CLASS MEMBERS	126
17.8. INITIALIZING OF DATA MEMBERS OF THE CLASS	126
Initializing of the Members of the Abstract Types.....	126
Initializing of the Constants.....	127
17.9. THE COPY CONSTRUCTOR AND ASSIGNMENT OPERATOR.....	128
17.10. FRIENDLY FUNCTIONS	130
17.11. CONSTRUCTOR AND OPERATOR <i>NEW</i>	133
18. INHERITANCE.....	134
18.1. CONSTRUCTION OF A DERIVED CLASS	134
18.2. PROTECTED CLASS MEMBERS	135
18.3. CONTROL OF THE ACCESS LEVEL TO THE MEMBERS OF THE CLASS	136

18.4. A SEQUENCE OF CONSTRUCTOR AND DESTRUCTOR CALL DURING CONSTRUCTING OF THE DERIVED CLASS BASED ON ONE BASE CLASS	137
18.5. TYPE CONVERSION.....	138
19. POLYMORPHISM.....	140
19.1. EARLY AND LATE BINDING	140
19.2. VIRTUAL FUNCTIONS	141
19.3. ABSTRACT CLASSES	146
20. ENUMERATIONS	148
ADVICE.....	151
EXERCISES	153

UNIT 5

21. OVERLOADING OF STANDARD OPERATORS	156
21.1. THE BASIC DEFINITIONS AND PROPERTIES.....	156
21.2. THE OPERATORS <i>NEW</i> AND <i>DELETE</i> FOR THE ABSTRACT TYPES	160
The Use of <i>new</i> when Creating a Dynamic Object of the Abstract Type	160
The Operator <i>delete</i>	161
21.3. TYPE CONVERSION.....	161
22. SOME PECULIARITIES OF OVERLOADED OPERATORS.....	165
22.1. OPERATOR =	165
21.2. OPERATOR [].....	167
ADVICE.....	173
EXERCISES	174

UNIT 6

23. CLASSES AND TEMPLATES	177
24. DYNAMIC DATA STRUCTURES.....	180
24.1. LISTS.....	180

24.2. THE OPERATIONS OVER UNIDIRECTIONAL LISTS	183
Implementation of the List	184
24.3. DOUBLE-LINKED AND CIRCULAR LISTS	186
24.4. THE CIRCULAR LIST OPERATIONS	188
Element Insertion.....	188
Element Deletion	188
Element Search.....	188
24.5. STACKS	189
Stack Realization through the Array	189
Stack Realization through the Dynamic Chain of Links.....	191
24.6. BINARY TREES	194
Definition and Construction	194
24.7. TABLES	197
ADVICE.....	202
EXERCISES	203
C++ GLOSSARY	206
APPENDIX	
PUNCTUATION MARKS AND SPECIAL SYMBOLS.....	227
CONTROL SEQUENCES	228
DATA TYPES	229
OPERATORS PRECEDENCE AND THE EXECUTION ORDER	230
C++ KEYWORDS.....	231
STANDARD FUNCTIONS	232
REFERENCES	233

INTRODUCTION

New information technologies are wider and wider used in all fields of life, including science and education. One of the important aspects of information technologies is programming in high level languages. The greatest part of this manual is devoted to programming in one of the modern programming language C++.

C++ was developed on the basis of language C by Bjarn Stroustrup. The authorship of language C belongs to Denis Ritchy, the research worker of AT&AT Bell Laboratories (1970). At first the programming language C was created for setting up and supporting an operational system UNIX. Before all the programs of this system were written either in Assembler language or B language, designed by Ken Tompson, a creator of UNIX. C is a language of general-purpose. It can be used for designing various programs, but its popularity was associated with operational system UNIX. It was necessary to design programs in C language to support UNIX system. C language and UNIX system fitted each other so well, that soon almost all commercial programs for UNIX proved to be written in C. This language gained such popularity that it was adjusted to other operational systems. Nevertheless, C language had some imperfections.

The peculiarity of C language is that it has retained many features of low level languages. These all present its weak and strong points at the same time. Programming language C can control the computer memory like assembler. On the other hand, C possesses the features of high level language, so designing and reading programs in it much easier than in the assembler language. C is outstanding for designing system programs, but understanding programs in C for other purposes often more difficult than in other high level languages. C language also has fewer capabilities for program autocheck than some other languages of high level.

To overcome disadvantages of C, B. Stroustrup (AT&AT Bell Laboratories) in 1980 developed programming language C++ on its basis. The greater part of C is a subset of C++ language, thus, most of C programs

are also C++ programs. The text in C language can be used in programs in C++. Also, it is possible to refer to functions library of C language. The main difference of C++ and C is realization of object-oriented technique of programming, which is extremely powerful modern programming method.

In fact, the specification of AT&T C++ release 2.0 became a language standard after several years of using C++. It was also designed under the direction of B. Stroustrup. Today there is a committee in the National Institute of Standards (NIST) that deals with C++ language (X3J16). The description of the language with comments, edited in 1990, was accepted by NIST committee as a basic material for normalization of C++. The last version of the standard is dated on 26th of May 1994. At present a realization of the language in Visual C++, Borland C++ x.x, Borland C++ Builder x and other systems have become widespread. The manual is focused on Borland C++ 3.1 – 5.x. version.

UNIT 1

1. INFORMATION TECHNOLOGIES

1.1. A DEFINITION OF INFORMATION TECHNOLOGY

The term “*technology*” has a root “*techne*” from Greek that means “art, mastery, skill”, meaning nothing but processes. “*Process*” implies a definite combination of actions, directed to achieve a certain aim. The process must be defined by a strategy, chosen by a person, and must be realized by means of the set of different means and methods. “*Material fabrication technology*” implies a process, defined by the aggregate of treatment means and methods, fabrication, change of a state, properties, form of a material or raw. Technology changes the quality and initial state of a material in an effort to obtain a material product. The aim of the material fabrication technology is a product release that satisfies the demand of people and system.

Information is one of the most valuable resources of the society together with such traditional material resources as oil, gas, minerals etc. It means that the process of information treatment by analogy with that treatment of material resources can be described as technology. According to this, it is possible to give the following definition for information technology (IT).

Information technology is a process, using the system of means and methods for collecting, processing and transmission of data (initial information) in order to obtain the information of a new quality about the state of an object, process or phenomenon (information product).

A New IT

IT is the most essential constituent of the information resources use of the society. By now, it has passed several evolutionary stages. The stages change was mainly due to scientific-and-engineering progress, appearing of new engineering devices of information organization. In modern society, a personal computer (PC) has become the main engineering means of information organization technology. PC influenced greatly the concept of construction and use of processing as well as the quality of obtained information.

The introduction of PC into information domain and application of telecommunication facilities have determined a new stage of IT development. Consequently, one of the following synonymous attributes – “**new**”, “**computer**” or “**modern**” – have changed IT name when joining to it. The definition “**new**” highlights not an evolutionary but innovative character of this technology. Its introduction changes considerably the content of different kinds of activity in organizations. The notion of “**new**” **IT** includes communication technologies, providing information transmission by different means – telephone, telegraph, telecommunication, fax, etc.

Thus, **new IT** is an information technology with user-friendly interface, including PC and telecommunication facilities. The definition “**computer**” highlights the idea that the main engineering means of its realization is a computer. There are three basic principles of a new (computer) IT:

- real time (conversational) mode of operating with computer;
- integration (splicing, interrelation) with other software products;
- flexibility of the data changing process as well as problem definition changing process.

Apparently, the most precise notion must be “**new IT**” not “*computer IT*”, as *new IT* is shown in its technology structure, which is based not only on the use of computers, but also other engineering means, especially those, providing telecommunication.

The Toolbox of IT

The realization of engineering process of material manufacturing is carried out by means of different engineering facilities. They include: equipment, engineering tools (lathes), tools, pipelines and others.

The engineering devices for information fabrication are **hardware**, **software**, and **mathematical support** of this process. They process the initial information into the information of a new quality. Let’s pick out software products in these context and call them as a **toolbox**, or more exactly **IT software tools**.

IT software tools is one or several interrelated software products for a definite computer type, which work technique allows to achieve the object, set by a user

It is possible to use widespread software product types for PC as a toolbox. They are as following: word processor (editor), desktop publishing (DTP), electronic worksheets, data manager, electronic notebooks, electronic calendars, information systems of functionality (financial, bookkeeping, marketing, etc.), expert systems and so on.

The Way IT Correlates with Information System

IT is intimately connected with information systems, which are the basic medium for IT. At first sight it may seem that the introduced IT and systems definitions are too similar. However, this is not the case. **IT** is a process, consisting of well-regulated rules of performance of operations, actions, stages of different degrees of complexity with data, storing in computers. The main purpose of IT is to obtain necessary information for a user as a result of task-oriented actions in initial information processing.

Information system is a medium, which constituents are computers, multicomputer systems, software products, databases, people, engineering and software means of all kinds and so forth. The main purpose of information system is organization of information storage and transmission.

Realization of information system functions is **not possible** without knowing IT which is oriented at it. IT **can exist** beyond information system domain.

So, IT is a more capacious notion, reflecting the contemporary idea about the process of information transformation in society.

Summing up all above-mentioned, we suggest some narrower definitions of an information system and IT than those, introduced earlier. **IT** – is a collection of well-defined purposeful actions of the staff in information processing on computers. **Information system** is a person-computer system for support of decision making and software product manufacturing, which uses computer IT.

1.2. THE STAGES OF IT DEVELOPMENT

There are several viewpoints at the IT development with the use of computers that determine its stages based on different features. A common approach is that a new stage of IT development has begun when computers appear. The main purpose is satisfaction of information needs of a person both in professional and domestic field. Therefore, there are several criteria for division of IT development into stages. These criteria are as discussed below.

The Type of Tasks and Information Processing

The first stage (1960–1970s) is data processing in a computer center in a state of multiple-access. The main tendency of IT development is automation of operating routine actions of a person.

The 2nd stage (since 1980s) is a creation of the IT which are focused on solution of strategic tasks.

Problems Facing the Society Informatization

The first stage (up to the end of 1960s) is characterized by the problem of the processing the data of great volumes when there are computer hardware of limited capacities.

The 2nd stage (up to the end of 1970s) is connected with distribution of IBM/360 series computers and is characterized by the problem of software lagging behind the level of hardware development.

The 3rd stage (beginning with 1980s) is remarkable for computer becoming a tool of an amateurish user, and information systems have become the means of supporting when making his/her decisions. The problems are ultimate satisfying the user's needs and creation of the appropriate interface when working in computer medium.

The 4th stage (beginning with 1990s) concerns the current technology creation of interorganizational connections and information systems. The problems of this stage are numerous. The most essential are:

- elaboration of agreements and establishment of standards, protocols for computer communication;
- organization of the access to strategical information;
- organization of the information security and safety.

The Advantage of Computer Technology

The 1st stage (beginning with 1960s) is characterized by quite efficient information processing when performing traditional operations oriented toward centralized resource sharing of computer centres. The main criterion of efficiency evaluation of created information systems was the difference between funds spent on designing and those saved as a result of implementation. The major problem at this stage was psychological, that is, poor interaction of users, for whom these information systems had been created, and computer designers because of difference of opinions and understanding of the current problems. Consequently, the systems that users conceived poorly, had been created, and in spite of their quite great capability, were not used in full power.

The 2nd stage (in the middle of 1970s) is connected with the advent of personal computers (PC). The method of information system creation changed. The orientation is shifted to the individual user's side for support his/her decision making. The user is interested in the current development, the contact with the designer is established, and mutual understanding appears between both groups of specialists. At this stage decentralized data processing, based on the solution of local tasks and on the operation with data bases at user's workplace, is used together with centralized data processing, typical for the 1st stage.

The 3rd stage (beginning with 1990s) is connected with the notion of the analysis of strategical advantages in business and is based on the achievements of telecommunication technology of distributed information processing. Information systems are aimed not only at efficiency increase of data processing and assistance to a manager. The appropriate information technologies must help organization to withstand competitive activity and to score an advantage.

The Types of Technology Toolbox

The 1st stage (up to the second half of XIX century) is “*handheld*” IT, which toolbox consisted of a pen, ink-pot, book. The communication was fulfilled manually by means of sending letters, bags, and dispatches by post. The main purpose of technology is representation of the information in a required form.

The 2nd stage (since the end of XIX century) is a “*mechanical*” technology, which toolbox consisted of a typewriter, telephone, dictating machine and delivery of mail, instrumented by enhanced means. The main technology purpose is presentation of the information in the required form by more convenient means.

The 3rd stage (1940–1960s of XX century) is an “*electric*” technology, which toolbox consisted of large computers and appropriate software, electric typewriters, Xeroxes, portable dictating machines.

The purpose of technology changes and the accent in IT starts to shift from the form of information presentation to formation of its content.

The 4th stage is “*electronic*” technology, which toolbox is large computers and automated control system (ACS), created on their base and information storage and retrieval system (IRS), equipped with wide range of common- and special-purpose software complexes. The technology focus shifts even greater to forming of intensional side of information for management of different spheres of life, especially to organization of analytical work. A great number of objective and human factors didn't allow to solve problems, facing a new IT concept. However, the experience of formation of intensional side of management information was obtained and professional, psychological, and social bases were prepared for transition to a new stage of technology development.

The 5th stage is “*computer*” (“new”) technology, which basic toolbox is a personal computer with wide range of standard software products of different assignment. At this stage the process of ACS personalization takes place, which becomes apparent when creating expert support systems of decision making for certain specialists. Such systems have integrated elements

of analysis and intelligence for different levels of management and are realized in PC and use telecommunication. Owing to the transition to the microprocessor basis, the technique of domestic, cultural and other assignments are subjected to considerable changes. Global and local computer networks are widely used in different fields.

1.3. THE PROBLEMS OF IT USE

It is quite natural that IT **become out of date** and they are replaced by the new ones. For instance, the work technique on PC at user's workplace took the place of batch processing technique on a large computer in computer centre. Telegraph handed down near all its functions to a telephone. The telephone is forced out gradually by Internet service.

When introducing a new IT into organizations, it is necessary to assess the risk of being behind the rivals as a result of IT inevitable decay with time, as software products, in comparison with other kinds of material products, have an extremely high replacement by new versions or types. The periods of replacement vary from several months up to one year.

In case this factor is not considered in the process of new IT introduction, it is probably that by the moment of completion of an enterprise transition to a new IT, IT will have already become outdated, and measures of its upgrading will have to be assumed. Such misfortunes of IT introduction are usually connected with imperfection of hardware. Whereas the main reason of failures is the absence or weak work up of IT use methodology.

The Methodology of IT Use

Centralized information processing in computer centres was the first historically existing technology. The prominent computer centres (CC) of multiple-access were created and were equipped with big computers (in Russia – electronic computers ES). The application of such electronic computers allowed to process data bulks of input information and to obtain different types of software products on this basis, which were afterwards transmitted to users. Such processing is caused by insufficient equipping enterprises and organizations with computing machinery in 1960–1970s.

The advantages of methodology of centralized information processing are:

- the possibility of user to apply to data bulks in the form of databases and to software products of extensive nomenclature;
- relative simple to introduce the methodological solutions in development and improvement of IT owing to their centralized acceptance.

The disadvantages of such methodology are evident:

- limited responsibility of inferior staff, who do not assist online obtaining information by a user. Thus they impede the accuracy of making management decisions;
- limitation of user's capabilities in the process of information obtaining and use.

Decentralized information processing is connected with the advent of PC in the 1980s and development of telecommunication means. It pressed considerably the preceding technology as it gives the user a lot of opportunities at work with information without limiting his/her initiative.

The advantages of such methodology are:

- structure flexibility, providing freedom to the user's initiatives;
- responsibility raising of inferior staff;
- reduction of the need in using a central computer and correspondingly in monitoring computer centre;
- better realization of constructive potential of a user due to the means of computer connection.

However, this methodology has its disadvantages:

- the complexity of standardization because of the great number of unique designs;
- user's psychological aversion of standards and finished software products, recommended by the computer centre;
- inequality in development of the local IT level, that first of all is determined by the skill level of a definite employee.

The described advantages and disadvantages of centralized and decentralized IT led to the necessity of reasonable application of this or that approach. Such approach can be called rational methodology and the way of responsibilities distribution will be the following:

- a computer centre must be responsible for elaboration of the general strategy of IT use, help users both at work and in education, set up the standards and specify the policy of software engineering means application;
- a staff, using IT, must follow the instructions of the computer centre, implement the development of its local systems and technologies in accordance with the general plan of organization.

Rational methodology of IT use allows to achieve greater flexibility, to support general standards, fulfil the compatibility of local software products, reduce the activity duplication and so on.

Variants of IT Implementation into an Enterprise

When applying IT into this or that enterprise, it is necessary to choose one of two basic concepts of organization structure and the role of computer information processing in it.

The first concept is oriented at *existing* structure of an enterprise. IT is adjusted to organization structure, and only enhancement of operation modes takes place. Service lines are weakly developed, only working places are rationalized. The distribution of functions takes place among technical workers and specialists. The degree of risk of new IT introduction is minimal, as the expenses are insignificant and organizational structure of an enterprise does not change.

The main disadvantage of such strategy is necessity of continuous variation of the information presentation form, adapted to specific technological methods and hardware. Any operative decision “gets stuck” at different stages of IT.

The advantages of the strategy are a minimal degree of risk and expenses.

The second concept is oriented at the *future* structure of an enterprise. It is assumed that the existing system must be upgraded.

A given strategy is connected with maximal communication development and designing of new organizational interrelationships. The productivity of the organizational structure of an enterprise increases as database records are distributed rationally, the volume of information, circulating in system channels, reduces and the equilibrium among current tasks is achieved.

The main disadvantages of this approach:

- considerable expenses at the first stage, connected with development of general conception and inspection of all enterprise subdivisions;
- the presence of psychological tensivity, caused by assumed changes in an enterprise structure and as a consequence by changes in staff list and job responsibilities.

The advantages of a given strategy:

- rationalization of enterprise organizational structure;
- maximal employment of all workers;
- high professional skill;
- integration of professional functions due to the use of computer networks.

A new IT at enterprises must be such that the information levels and subsystems, processing this information are connected by one information array. At the same time, two demands raise. **First**, the system structure of

information organization must correspond to the distribution of authority at an enterprise. **Second**, the information inside the system must function in a way to reflect the levels of management as fully as possible.

1.4. TYPES OF IT

IT of Data Processing

Description and Function

IT of data processing is intended for solution well structured tasks, for which there are necessary input data, and algorithms and other standard procedures of their processing are well known. This technology is applied at the level of operating (executive) activities of the staff of low qualification in an effort to automatize some routine repetitive operations of administrative work. That is why IT and system implementation at this level will considerable increase working efficiency of the staff, relieve people of routine operations, and possibly, even lead to the necessity of reducing the number of workers.

At the level of operational activity, the following problems should be solved:

- data processing in the operations made by an enterprise;
- making of periodical monitoring reports about the state of enterprise work;
- obtaining the answers to various current queries and their registration as paper documents or reports.

The examples of routine operations are:

- the operation of correspondence test of reserve level of indicated goods at a warehouse with the standard. When the reserve level lowers, the order is given to the supplier with indication of necessary goods and the delivery date;
- the operation of goods selling by the enterprise as a result of which is an outgoing document for the customer in the form of a cheque or receipt.

The example of monitoring report is:

- everyday report about the income and payment in cash by a bank, formed to monitor the balance of available cash.

The example of query:

- the query to the staff database, which allows to obtain the data about the requirements to candidates for taking a definite post.

There are several peculiarities, connected with data processing, distinguishing the given technology from others:

- performing the tasks of data processing, necessary for the certain enterprise. The law charges every enterprise to have and store the data of its activity, which can be used as means of providing and supporting of supervision. Therefore all enterprises must have a data processing system and corresponding IT;
- solution of well-structured tasks only, for which an algorithm can be developed;
- performing the standard processing procedures. The existing standards determine typical data processing procedures and assign their observance by the organizations of all types;
- performing the main processing procedures automatically with minimal participation of a human;
- the use of the detailed data. The records of the enterprise activity have detailed nature, admitting the audit. In the process of audit the enterprise activity is checked chronologically from the beginning of the period to the end and vice versa;
- the accent is on the chronology of the events;
- minimal assistance from the other levels specialists is required in solution the problems.

Basic Components

The basic components of data processing IT can be represented in the following way:

Data collection. As an enterprise manufactures production, each its activity is accompanied by corresponding data records. Usually the enterprise activities, concerning external environment, are marked particularly as the operations, performed by the enterprise.

Data processing. For producing from the incoming data the information reflecting the enterprise activity, the following typical operations are used:

- classification or grouping. The initial data usually have the types of codes consisting of one or several symbols. These codes, representing definite object features, are used for identification and grouping of records;
(**The example.** During salary accounting, each record includes a code (clock number) of the worker, subdivision code, where he/she works, position, and so on. In accordance with these codes different grouping can be formed.)
- sorting out, by means of which the sequence of records is organized;
- calculations, including arithmetic and logical operations. These operations, performed with data, give opportunity to obtain new data;
- enlargement or aggregation, serving for reduction of data quantity and is realized in the form of computation total or average values.

Data storage. Many data at the level of operational activity are necessary to store for subsequent use either at this level or at different one. For their storage, databases are created.

The reports (documents) creation. In IT of data processing, documents must be created for the directors and the workers of the enterprise, as well as outside partners. At the same time, the documents can be created on demand or due to performed operation by the enterprise, as well as at the end of each month, quarter, and year.

Management IT

Description and Function

The purpose of management IT is meeting information needs all employees of the enterprise without exceptions, who deal with decision-making. It can be useful at any level of management. This technology is oriented at work in the environment of management information system and is used at worse structured problems being solved in comparison with those, solved by means of data processing IT.

Information systems (IS) of management are ideal for satisfaction of similar information needs of employees at different functional subsystems (subdivisions) or enterprise management levels.

The information delivered by IS contains data about the past, present and probable future of the enterprise. This information looks like regular or special management reports.

For decision making at the level of supervising management, the information must be represented in aggregated form in order that the tendencies of data changes, reasons of aroused deviations, and possible solutions are viewed. At this stage, the following tasks of data processing are solved:

- assessment of the planned state of managed object;
- assessment of the deviations from the planned state;
- identification of the deviation reasons;
- the analysis of possible solutions and actions.

Basic Components

Management IT is directed at the making of different types of reports.

Regular reports are made in accordance with the fixed schedule, determining the time of its creation (for example, monthly analysis of sales of the company).

Special reports are made on demands of managers or when something unplanned happened.

These or those types of reports can have the form of summing, comparing and emergency types of reports.

In *summing* reports, data are combined into separate groups, sorted out and represented as intermediate and final results in separate areas.

Comparing reports contain data, obtained from different sources, classified according to different features, and used in the purposes of comparison.

Emergency reports contain the data of exceptional (extraordinary) character.

The use of the reports for management support turns out to be very effective at realization of so-called “management according to deviations.”

The management according to deviations means that the main content of the obtained by a manager data must be deviations of the state of the enterprise economical activity from some fixed standards (for example, from its planned state). When using at the enterprise the principles of management according to deviations, the following demands raise to the reports:

- the report must be made only in case this deviation took place;
- the data in the report must be sorted out according to the significance of the index critical for this deviation;
- it is useful to represent all the deviations together for the manager to catch the relation among them;
- it is necessary to show a quantitative deviation from the norm.

Office Automation IT

Description and Function

Historically automation began at plants and then it spread to the office, having the purpose of automation of routine secretary work. In the course of communication means development automation of office technologies awoke the interest of specialists and managers, who saw here the possibility to increase their labour productivity.

The aim of the office automation is not to replace existing traditional communication system of the staff (with their meeting, phone calls and orders), but only to supplement it. Automated office is attractive for managers of all management levels at the enterprise not only because of its support company inner contacts of staff, but also due to new communication means with external environment.

IT of the automated office is arrangement and support of communication processes inside the company as well as with external environment on the basis of multicomputer systems and other modern means of transmission and operation with information.

Office automated technologies are used by managers, specialists, secretaries and clerks; especially they are attractive for team decision of problems. They allow to raise the labour productivity of secretaries and clerks and give them opportunity to deal with the increasing volume of work. However, this advantage is secondary in comparison with the possibility of automation use in the office as a tool for problem solving. Improvements of the decisions, made by the managers as a result of their more perfect communication, are capable of providing the economic growth of an enterprise. Nowadays there are several tens of software products for computers and non-computerized engineering means, providing office automation technology (word processor, table processor, E-mail, electronic calendar, audio-mail, computer and teleconferences, videotext, image storage), as well as special-purpose programs of management activity (documentation running, supervision of order fulfillment, etc.) Also, non-computerized means are widely used (audio- and video conferences, facsimile posting, Xerox and other means of office equipment).

Basic Components

Database. A compulsory component of any technology is database. In automated office, database concentrates the data about the enterprise production system itself as well as in the technology of data processing at the operation level. The information can enter the database from the outside of the enterprise. The specialists must know the basic technological operations of the work in database medium.

For example, the information about daily selling or raw material delivery is accumulated in the database and is transmitted to the host computer by commercial agents of the enterprise. The information about the rate of exchange or quotations of stock, including the stocks of the enterprise can be received from stock exchanges every day by E-mail. This information is updated every day in the corresponding data base array. The information from data base comes to the input of computer applications (programs) such as word processor, table processor, E-mail, computer conferences and so on. Any computer application of the automated office provides the connection among employees and with other enterprises. The information received from the databases can be used in non-computerized technical means for information transmission, replication, and storage.

Word processor. It is a type of application software, intended for creation and processing of textual documents. It allows to add and remove words, move sentences and paragraphs, adjust the format, manipulate the text elements and modes and so forth. When the document is ready, the worker rewrites it to the external memory and then prints and if necessary passes it by the network. Thus, a manager has at his/her disposal effective type of

written communication. Regular receiving of letters and reports by means of word processor gives a manager a chance to assess the situation constantly.

Electronic mail. Electronic mail (E-mail), based on network use of computers, gives the user a chance to receive, store and send messages to their partners by network. In this case only one-way communication takes place. This limitation, in the judgment of many researches, is not too important as in the half of cases the office negotiations on the phone are aimed at receiving information. For providing two-way communication, the messages have to be sent and received repeatedly by E-mail or by use of another type of communication. E-mail can provide the user with different opportunities depending on the type of the software used. To make the message available for all users of E-mail it should be put on the computer public *bulletin board*, at will it is possible to mark that this is a private correspondence. You can also send the message with the notification of its receiving by the addressee. When an enterprise decides to introduce E-mail, it has two opportunities. The first one is to buy personal hardware and software and create its own local multi-computer system, realizing the function of E-mail. The second opportunity is connected with buying of the E-mail use service, which is provided by specialized communication organizations.

Audio-mail. This is the mail for message transmission with voice. It resembles E-mail, excepting that instead of the typing the message on computers it is transmitted by the phone. Also the messages are received by the phone. The system includes a special device for conversion of the audio signals into digital code and vice versa, and a computer for audio messages storage in digital form. Audio-mail is also realized in the network. The mail for the audio message transmission can be successfully used for team decision of problems. For it, a sender of the message must indicate the list of people, who this message is sent to. The system will occasionally ring up all the indicated employees for delivering the message to them. The main advantage of audio mail in comparison with electronic one is that it is simpler and it is not necessary to key the data.

Table processor. Along with word processor, it is a basic constituent of informational culture of any employee and automated office technology. Without knowing the basics of work technique in it, it is impossible to use PC at full extent. The functions of modern software environments of table processors allow to perform a lot of operations with data represented in table form. Combining these operations by the common features, it is possible to list the most numerous and widely applied groups of production operations:

- data input from both the keyboard and databases;
- data processing (sorting, resume automated formation, data copying and transfer, different groups of calculating operations, data aggregation and so on);

- information output in a printed form, in the form of imported files into other systems, directly into the database;
- qualitative design of table forms of data representation;
- multipronged and qualitative data design in the form of diagrams and graphs;
- performing engineering, financial, statistics calculations;
- making the mathematical simulation and a number of other auxiliary operations;

Any modern environment of a table processor have the means of data transmission by network.

Electronic calendar. It provides one opportunity to use a network computer for storage and manipulation of managers and other employees working timetable. A manager (or his/her secretary) fixes the date and time of the meeting or other arrangements, looks through the arranged timetable, makes changes by means of a keyboard. Hardware and software of an electronic calendar fully corresponds to similar E-mail components. In addition, the system provides the opportunity to gain an access to the calendars of other managers. It can automatically coordinate the meeting time with their personal timetables.

The use of an electronic calendar proves to be very efficient for the managers of the high management level, whose working days are filled for long ahead.

Computer conferences and teleconferences. *Computer* conferences use multi-computer systems for information interchange among the conferees, solving a definite problem. Naturally, the circle of people, having the access to this technology, is limited. The number of conferees of the computer conference can be many times greater than those of audio and video conferences. It is possible to meet in the literature the term “*teleconference*”. A teleconference includes three types of conferences: audio, video, and computer ones.

Videotext is based on the use of a computer for receiving an imaging of text and graphic data on the screen of the visual display unit (VDU). There are three possibilities of information receiving in the videotext form for people decision making:

- to create videotext files on their own computers;
- to set up a contract with specialized companies to obtain the access to the videotext files developed by them. Such files, specially intended for sale, can be stored on the servers of the company, which fulfilling such services, or delivered to the client on magnetic or optical disks;
- to set up the contract with other companies for obtaining access to videotext files.

The exchange by the catalogues and pricelists of production among companies in videotext form gains now greater popularity. As for companies that specialize in videotext selling their services began competing with such printed matters as newspapers and magazines. Thus, in many countries it is possible to order a newspaper or magazine in videotext form, without mentioning current summaries of stock information.

Image storage. At any enterprise, it is necessary to store a great number of documents over a long period. Their number can be so great that storage in the file form causes serious problems. That is why the idea to store not the document itself, but only its image appeared, moreover, to store in a digital form. The image storage is a perspective office technology and is based on the use of a special device – an optical image recognizer, allowing to transform a document or a film image into digital form for further storage in the external computer memory. The image saved in a digital format can be printed out or displayed at any moment in its real appearance. For image storage, optical disks are used, which have vast memories (capacities). This way, 200 thousand pages can be recorded on a five-inch disk. It is necessary to keep in mind, that the idea of image storage is not new. There was an attempt to realize it on the base of microfilms and microfiches. The advent of a new engineering solution – an optical disk in combination with digital image recording – facilitated the creation of a given technology.

Audio conferences. They use audio-communication for supporting connection between workers who are in the distant areas or between the subdivisions of an enterprise. The simplest technical means of audio conference realization is a telephone communication, equipped with additional devices, providing the participation in the conversation for more than two people. The creation of audio conference does not necessitate the presence of a computer, but it only supposes the use of two-way audio communication between its conferees. The use of audio conferences facilitates decision making, it is cheap and convenient. The efficiency of audio conferences increases when meeting the following conditions:

- the employee who organizes an audio conference must provide the opportunity of participation for all interested parties preliminary;
- the number of the conferees should not be too great (no more than six) in order to restrain the discussion in the limits of the problem under discussion;
- the conferees must be informed about the conference program beforehand, for example, by means of facsimile telegraph;
- every conferee has to introduce her/himself before starting his/her speech;
- the recording of the conference and its storage must be organized;
- the recording of the conference must be printed and sent to all its conferees.

Video conferences. They are designed for the same purposes as audio conferences, but with the use of the video equipment. Their holding does not need a computer. In the process of the conference its conferees, remote from each other at a considerable distance, can see on the screen themselves and other conferees. The sound accompaniment is transmitted simultaneously with television image. Although video conferences can cut down transport and travel expenses, most of the enterprises use them not only due to this reason. These enterprises see in them the opportunity to involve maximal number of managers and other employees who are geographically remote from the office into the problem solution. The most popular three configurations of video conference design:

- *one-way video and audio communication.* In this case audio and video signals move in one direction, for example, from the head of the project to the executors;
- *one-way video and two-way audio communication.* Two-way audio communication gives the conferees, receiving the image, a chance to exchange the audio information with the conferee who transmits the video signal;
- *two-way audio and video communication.* In this more expensive configuration a two-way audio and video communication is used among all the conferees, usually having the same status.

Facsimile communication. This communication is based on the use of a fax, capable of reading a document on one end of the communication channel and reproducing its image on the other. Facsimile communication contributes into decision making owing to quick and simple distribution of documents to the conferees, solving a definite problem, independently of their geographical position.

Decision Making IT

Description and function. The systems of decision making support and corresponding IT, appeared due to the efforts mainly of American scientists at the end of 1970s – at the beginning of 1980s. The wide spread of PC, a standard application package, and a successful creation of artificial intelligence facilitated it. The main peculiarity of *decision making support IT* is a qualitatively new method of organization of the interaction between a person and a computer. Elaboration of the solution, being the main purpose of this technology, occurs as a result of an iterative process, where the following participate:

- decision making support system in the role of a computing section and a controlled object;
- a person as a managing section, setting the input data and assessing the obtained result of calculation on a computer.

The completion of an iterative process occurs on a person's will. In this case it is possible to speak about the capability of an information system together with a user to create new information for decision making. In addition to this peculiarity of IT of decision making support, it is possible to point out a number of its distinguishing characteristics:

- orientation toward badly structured (formalized) tasks;
- combination of traditional methods of computer data access and processing with capabilities of mathematical models and methods of task solution on their base.
- orientation at amateurish computer users;
- high adaptability, providing the possibility to adjust to the peculiarities of the available hardware and software, as well as to the user's needs.

IT of decision making support can be used at any level of managing. Besides, the decisions made at different levels are often have to be coordinated. So the important function either of a system or technology is a coordination of people, making decisions either on different or one level of management. Three main components belong to the system of decision making support: database, model base and software subsystem, which consists of data base control system (DBCS), model base control system (MBCS) and interface control system between a user and a computer.

Database

It is an important part of IT of decision making support. Data can be used directly by a user for calculation by means of mathematical models. Let's consider the data resources and their peculiarities.

1. A part of the data arrives from information system of the operational level. These data must be preliminary processed to use them efficiently. There are two possibilities:

- to use for data processing the enterprise DBCS that belongs to the structure of the system of decision making support;
- to make processing outside the system of decision making support, having created a special data base for it. This variant is more preferable for the enterprises making a great number of commercial operations. Processed about the enterprise operations data form files, which are stored beyond the system of decision making support, for increasing reliability.

2. In addition to enterprise operation data, for functioning the system of decision making support other inside data are required, for example, the data about the staff movement, engineering data and so on, which must be in good time collected, set in, and supported.

3. The data from external sources have a great importance, especially for supporting of decision making at high management levels. It is important to point among necessary external data the data about the competitors, national and world economy. In contrast to internal data, external data are usually bought in the organizations which specialize in their collection.

4. At present, the problem of including one more data source – documents (letters, contracts, orders and so on) into database is widely explored. If the content of these documents is recorded in the memory and then processed using some key characteristics (suppliers, consumers, dates, kinds of services etc.), the system will obtain a new powerful source of information.

Data management system must possess the following capabilities:

- formation of data combinations, received from different sources by means of using the aggregation and filtering procedures;
- fast addition or eliminating this or that data source;
- construction of logical data structure in user's terms;
- using and manipulating unofficial data for experimental verification of user's operational alternatives;
- providing of complete logical independence of this database on other operational databases, functioning within the enterprise.

Models Base

The aim of model creating is to describe and to optimize some object or process. The use of models provides the analysis realization in the systems of decision making support. Models based on mathematical interpretation of the problem by means of definite algorithms assist the information discovery, useful for making proper solutions.

The use of the models in the structure of informational systems was initiated when statistics and financial analysis methods were being applied. These methods were first realized by the commands of common algorithmic languages. Later special languages have been created, allowing to model the situations like “what will happen if..?” or “what to do in order that ..?”. Such languages, created specially for model construction, give a chance to construct the models of a definite type, providing the solution finding at flexible variation of variables.

There are a lot of types of models and the ways of their classification, for example, according to the target use, field of possible applications, the way of variable evaluation and so on.

According to the purpose of use, the models are subdivided into *optimization*, aimed to finding the maxima and minima of indices (for example, managers often want to know which of their actions can cause maximizing of profits

or minimizing the expenses), and *descriptive*, describing the behavior of a system and not intended for management purposes (optimization).

According to the way of assessment, models can be classified as *deterministic*, using the variable assessment by one number at definite values of initial variables, and *stochastic*, assessing the variables using several parameters, as initial data are specified by probabilistic characteristics.

Deterministic models are more popular than stochastic ones as they are cheaper and they are easy to use and construct. In addition, quite sufficient information for decision making is often obtained by means of deterministic models.

According to the field of possible application, models are grouped into *specialized*, intended for the use by only one system, and *universal* – for the use by several systems.

Specialized models are more expensive; they are usually applied for description of unique systems and have great accuracy.

In the systems of decision making support, the data base consists of strategic, tactical, and operational models in the form of collection of model blocks, modules and procedures, used as elements for their constructions.

Strategic models are used at the highest levels of management for setting the purposes of the enterprise, a volume of resources necessary for their achievements, and the policy of obtaining and application of these resources. They can be efficient when choosing the variants of enterprise location, predicting the competitors' policy and so on. A considerable scope width, variety of variables, data representation in a condensed aggregate form is typical for strategic models. These data are often based on external sources and can have subjective nature. Planning horizon in strategic models as a rule is measured in years. These models are usually deterministic, descriptive, specialized for the use in one definite enterprise.

Tactic models are applied by the managers of a medium level for distribution and control of using existing means. Among possible fields of their use it is necessary to point out financial planning, planning of requirements to the employees, planning of gaining in sales, scheming of enterprise lay-out. These models are applicable only to separate parts of an enterprise (for example, to the system of manufacturing and marketing) and can include aggregated indices. Time horizon, covered by tactic models, is from one month to two years. The data from external sources can also be demanded here, but a great attention at realization of given models must be paid to internal data of an enterprise. Usually tactic models are realized as deterministic, optimization, and universal.

Operational models are used at lower levels of management for supporting of operative decision making with the horizon, measured by days and weeks. The possible applications of these models include maintenance of debtor accounts and credit settlements, production scheduling, inventory management, etc. Operative models are usually used for calculation of company data. They are generally deterministic, optimization and universal (i.e. they can be used in different organizations).

Mathematical models consist of a collection of model blocks, modules, and procedures, realizing mathematical methods. It can include the procedures of linear programming, statistic analysis of time series, regressive analysis and so on beginning with the simplest procedures up to the complex applied software packages. Model blocks, modules and procedures can be used either one by one or in complex for construction and support of the models.

The control system of model base must possess the following capabilities: it must create new models or change the existing ones, support and update the model parameters, manipulate models.

Control System of Interface

The efficiency and flexibility of IT depends greatly on the interface characteristics of support system of decision making. Interface determines the user's language, the language of computer messages, organizing dialogue on the monitor, user's knowledge.

User's language is those operations, which a user performs in relation to the system by using the keyboard capabilities, wand for writing on the screen, joystick, "mouse", verbal instructions and so forth. The simplest form of the user's language is a creation of forms for input and output documents. Having received an input form (document), a user fills in it with necessary data and set them into a computer. The support system of decision making makes the necessary analysis and issues the results in the standard form of the output documents.

Visual interface has gained considerable popularity lately. By means of the "mouse" manipulator a user chooses objects and commands in the form of images presented on the screen, thus realizing his/her operations.

Computer controlling using a human voice is the simplest and thus the most desirable form of a user language. It is not developed sufficiently and that is why not very popular. The existing versions demands serious restrictions of the user: a definite set of words and phrases; special structure taking into account the peculiarities of the user's voice; control in the form of discrete commands not in the usual smooth speech. The technology of this method is

improving and in the nearest future the advent of the support systems of decision making can be expected. These systems use a verbal input.

Message language is that a person can see on the monitor (symbols, graphics, color), data received on the printer, beep output signals, etc. The important gauge of efficient interface use is a chosen form of the dialogue between a system and a user. At present, the following dialog forms are most widely spread: interactive query mode, command mode, menu mode, gaps-in-the-phrases filling mode, offered by a computer.

Each form, depending on the type of the task, peculiarities of the user, and sort of decision making, can have its advantages and disadvantages.

For a long time, the only realization of message language was printed or displayed *report* or *message*. Now a new possibility of presenting output data – *computer graphics* – has appeared. It gives a chance to create color graphic three-dimensional view on the screen. The use of computer graphics enhances visualization and interpretability of output data considerably, it becomes more popular in IT of decision making support.

For the last several years, a new tendency developing computer graphics is outlined. It is *animation*. Animation proves to be especially efficient for output data interpretation of support system of decision making, connected with modeling of physical systems and objects.

Within the next few years, the use of human voice as a message language is to be expected. Now this form is applied in the support system of decision making in the field of finance, where in the process of emergency report generation, the voice explains the reasons of singularity of this or that position.

The user knowledge is things a user must know when working with a system. Not only an action plan, which is in the mind of the user, belongs to it, but also textbooks, instructions, reference data, yielded by a computer.

The improvement of support system interface of decision making is affected by the success in development of each of three pointed out element. Interface must have the following capabilities:

- to manipulate different dialog forms, changing them in the process of decision making at user's option;
- to transmit data to the system using different means;
- to obtain the data from different system devices in various format;
- to support (to assist on demand, to prompt) user's knowledge plially.

Expert Systems IT

Description and Function

The greatest progress among computer information systems is marked in the field of *expert system* development, based on the use of artificial intel-

ligence. Expert systems give a manager or a specialist a chance to get an expert's opinion on any problem, about which these systems accumulated knowledge.

Artificial intelligence is capability of computer systems to such operations which can be called intellectual, if they come from a person. Frequently in this case the capability, connected with human thinking, are meant. The work in the field of artificial intelligence is not limited by expert systems. It also includes creation of robots, systems that simulate a human nervous system, his hearing, eye-sight, learning capability.

The solution of specific problems requires definite knowledge. However, not every company can afford having the experts in different problems connected with its work, or even invite them every time a problem arises. The main idea of expert system technology is to use the expert knowledge every time when it is necessary, having obtained and loaded it into the computer memory. Being one of the basic applications of artificial intelligence, expert systems represent computer programs, transforming the expert experience in some field of knowledge into the form of heuristic rules (heuristics). Heuristics do not guarantee the obtaining of the optimal result with the same confidence as common algorithms, used for task solution in the context of support technology of decision making. But they often supply acceptable enough solutions for practical application. All these make the use of expert system technology successful.

The similarity of IT, used in expert systems and support systems of decision making, is that both of them provide high level of support of decision making. However, there are three essential differences. The first one concerns the fact that a problem solution within the support system of decision making reflects the level of its comprehension by the user and its capability to obtain and conceive the solution. On the contrary, the expert system technology offers the user to make a decision, exceeding his/her abilities. The second distinguishing feature of mentioned technologies is a capability of expert systems to explain their reasoning in the process of decision making. Very often these explanations turn out to be more important for the user, than the decision itself. The third distinction is connected with the use of a new component of IT – knowledge.

Basic Components

The basic components of IT, used in expert system, are user's interface, knowledge base, interpreter, system creation unit (Fig. 1).

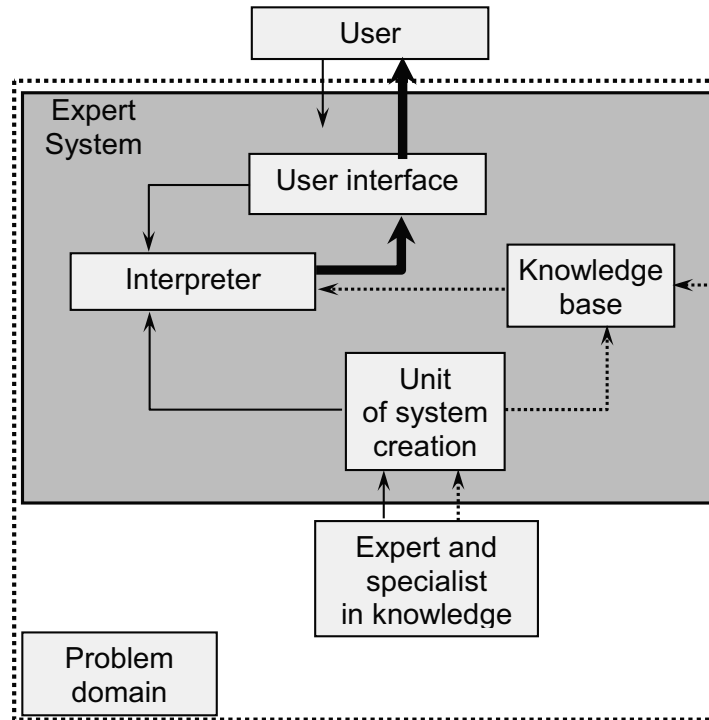


Fig. 1. Sketch of expert system

User's interface. A manager (specialist) uses interface for input information and commands into the expert system and for obtaining the output information from it. The commands include the parameters, guiding the process of knowledge processing. The information usually is yielded in the form of the values, being bound to definite variables. A manager can use four methods of information input: menu, commands, natural language and native interface. Expert system technology provides the possibility to obtain not only decisions, but also necessary explanations as *output* information. There are two types of explanations:

- the explanation, yielded on demands. The user can demand explanation of his/her operations from expert system at any moment;
- the explanation of the obtained problem solution. After obtaining the solution, the user can demand the explanation of the way it has been obtained. The system must illustrate every step of its reasoning leading to the solution of the task. Although the work technique with expert system proves to be difficult, the user's interface of these systems is user-friendly and usually does not cause any problems when put into the dialog.

Knowledge base. It contains the facts describing the problem domain, as well as logical connection of these facts. A central place in knowledge base belongs to the rules. A *rule* determines what to do in each specific situation, and consists of two parts: a condition, which is either met or not, and an action, which must be performed if the condition is met. All rules, used in the expert

system form a *rule system*, which may consist of several thousands of rules even in a comparatively simple system. All types of knowledge, depending on the specific character of the knowledge domain and the proficiency level of a designer (with engineering skills), can be represented with one or another degree of adequacy using one or several semantic models. Logical, productional, framebased, and semantic networks belong to the most widely spread.

Interpreter. This is a part of the expert system, processing the knowledge (reasoning), being in knowledge base, in a specific order. The technology of interpreter operation consists in successive considering the rule collection (rule by rule). If the condition in the rule is met, a definite action is performed and the user is granted the variant of the problem solving. Besides, in many expert systems additional blocks are introduced: database, calculation unit, input, and data correcting unit. Calculation unit is necessary in the situations, connected with management decision making. At the same time the database, which contains planned, physical, reporting and other constant or on-line indices, takes an important place. Input and data correcting unit is used for on-line and in-time reflection of current changes in data base.

System creation unit. It is used for creation of the rules collection (hierarchy). There are two approaches, which can be a base of system creation unit: the use of algorithmic languages and expert system shells. For presentation of knowledge base the languages Lisp and Prolog are specially developed, although any well known algorithmic language can be used. *Expert system shell* represents a complete program medium that can be adapted to the solution of a definite problem by means of creation of a proper knowledge base. In most cases, the use of the shells allows to create expert systems faster and easier in comparison with programming.

QUESTIONS

1. What is an Information System?
2. How can you interpret the term “Information Technology”?
3. What is the difference between computers and information systems?
4. How the processes happen in an information system can be interpreted?
5. How have the information systems been developed?
6. Why information systems are the strategic development means of the firm? What are their contribution?
7. Tell about the pyramid of management levels in a firm.
8. Why is it important to consider environmental effects when setting up an information system?
9. Give examples of information systems, which support firm activity.
10. What are the problems when setting up an information system?
11. Tell about main functional information systems.
12. Give examples of information systems, which provide efficiency of work.
13. How can you interpret the structure of information system?
14. Tell about dataware, hardware, software, mathematical means, management means, and legislative regulation.
15. What is the meaning of data flow diagram?
16. What is the main principle of database construction methodology?
17. Why the structureness of the problem is an important factor when developing information system?
18. How does the structureness of a problem influence the classification of information systems?

19. What are particular features of information systems, which create management reports?
20. What are features and forms of information systems, which develop alternative solutions?
21. What is the main point of functional characteristic when classifying information systems?
22. What is the management level characteristic when classifying information systems?
23. Tell about information systems pyramid in a firm when functional characteristic of classification is used.
24. What is the role and functions of operative level IS, IS for specialists, for middle level managers, strategic IS?
25. Give the classification of information systems in the terms of information usage, degree of automation, application field.
26. What are the similarity and the difference of information technology and technology of material production?
27. Show the information technology as a hierarchical structure and give examples of its components.
28. Tell the requirements, which the information technology must satisfy.
29. What is the toolbox of information technology?
30. What does “New information technology” mean?
31. How do information technology and information system correlate?
32. What is the history of information technology development?
33. Describe the methodology of information technology usage.
34. Give a general idea of data processing information technology, management IT, office automation, decision-making IT. Mention their main components.
35. Tell about computer and non-computer office technologies.
36. What is the model bank? What models do you know? Give examples.
37. Tell about an information system interface and its components.
38. What is the difference between information and data?
39. What information measures do exist? When the one should use them?
40. Tell about syntactic, semantic, pragmatic information measures.

41. What information quality indexes do you know?
42. What is the information classification system?
43. Tell the main ideas of hierarchic, facet and descriptor classification methods. Give examples.
44. What is information-encoding system? Give classification of encoding methods.
45. What are classification and registration encoding? Give examples.
46. Compare classification system and encoding system purposes.
47. Tell about classification of information circulating in an organization.
48. Tell about information revolutions in the civilization history.
49. Compare processes of the computer evolution history and the last information revolution.
50. Define the main point of information technologies and telecommunications.
51. How can you describe information-oriented society?
52. What does information crisis mean?
53. What is the idea of informatization process?
54. What is the difference between informatization and computerization processes?
55. Give a definition for a term of information culture. How does it show?
56. What defines informational potential of the society?
57. Tell about resources kinds. Characterize information resource, information product, information service. Give examples.
58. How can you describe the term "Database"?
59. What is the legislative control of the information market?

UNIT 2

2. THE FUNDAMENTALS OF C++ LANGUAGE

2.1. THE FIRST PROGRAMS

Write the simplest program:

```
#include<iostream.h>
void main()
{
cout<<"Hello, World!\n";
}
```

The word “**#include**” in the first row is a so-called **preprocessor statement** (including directive). Performing of this directive leads to inserting of the content of **iostream.h** file (files with the extension *.h* are called **headers** or header files; they contain the text in C++ language) into the program instead of the first row. After it, a compiler will process the new received text of the program. In the second row of the initial text of the program, there is a title of the function with the name “**main**”. Empty round brackets suggest that this function doesn’t have arguments, and a **keyword** “**void**” means that the function **main()** doesn’t return any value. The curly brackets contain a unit that is often called a program body.

In the forth row, there is a statement, which operation is output into stream **cout** (it sounds as **see-out**), which is here associated with a monitor. This statement displays the row

```
Hello, World!
```

in a monitor.

A symbol ‘\n’ at the end of the text in quotation marks informs the compiler that after the text printing it is necessary to pass to a new row.

Now it is necessary to give an example of a simple program and a dialogue, which is displayed when the user starts this program and sets corresponding data. Hereinafter a person who uses a program will be called

a **user**. The data input by the user are marked boldface. An author of this program will be called a **programmer**.

```
#include <iostream.h>
void main()
{
int m, n, sum;           //descriptions
cout<<"For input of numbers key in two numbers\n";
cout<<"(over spaces) and press ENTER.\n";
cin>>m>>n;             //numbers input
sum = m + n;
cout<<"at m = "<<m<<" and n = "<<n
<<" their sum is equal to "<<sum<<".\n";
                        //result output
}
```

When running this program, the following dialog will appear on the screen:

```
For input of numbers key in two numbers
(over spaces) and press ENTER.
20 45
at m = 20 and n = 45 their sum is equal to 65.
```

Comments are anything after the symbols `//` to the end of the line in this program. In the fourth row, the integer variables with the **names** *n*, *m*, and *sum* are **described**.

The operator, beginning with the word **cin**, informs the compiler that the values input by the user, which are equal to 20 and 45 here, should be placed accordingly into variables **m** and **n**. **cin** object (is read as **see-in**) is an input stream – here is understood as a keyboard, and the arrows `<<` and `>>` indicate the direction of data movement.

Let's go on to the description of C++ language and its possibilities.

2.2. THE LANGUAGE ALPHABET

The C++ alphabet includes:

- Latin caps A...Z;
- Lower case characters a...z;
- Arabic figures 0...9;
- Underlining symbol `_` (is considered as a letter).

All these symbols are used for the language key words and names formation. The name is a sequence of letters and figures, beginning with a letter and being not a key word (it is not recommended to put an underlining symbol `_` at the beginning of the name).

In C++ capital and lower case characters are different, therefore the names ARG1, Arg1, and arg1 are different.

- The punctuation marks and special symbols given in the Table 1.
- White spaces.

Space, tab, line feed, carriage return, and form feed belong to this group. These symbols separate language **lexemes** from each other. Any sequence of white spaces is considered at compilation as one space.

Table 1

Punctuation Marks and Special Symbols

Symbol	Name	Symbol	Name
,	comma	{	left curly bracket
.	point	}	right curly bracket
;	semicolon	<	greater
:	colon	>	less
?	question mark	[left square bracket
'	apostrophe]	right square bracket
!	exclamation mark	#	number or grid
	vertical bar	%	percent
/	slash	&	ampersand
\	backslash	^	logical NOT
~	tilde	-	minus
*	asterisk	=	equals sign
(left round bracket	“	quotation mark
)	right round bracket	+	plus

2.3. COMMENTS

Comments serve as an aid to the human readers of the programs; they are a form of engineering etiquette. They may summarize a function's algorithm, identify the purpose of a variable, or clarify an otherwise obscure seg-

ment of code. Comments do not increase the size of the executable program. They are stripped from the program by the compiler before code generation.

There are two comment delimiters in C++.

The first comment delimiter, indicated by a double slash (`//`), serves to delimit a single-line comment. Everything in the program line to the right of the delimiter is treated as a comment and ignored by the compiler:

```
// symbols up to the end of the line.
```

The comment pair (`/*,*/*`) is the second comment delimiter used in the C++ language. The beginning of a comment is indicated by a `/*`. The compiler will treat everything that falls between the `/*` and a matching `*/` as part of the comment. A comment pair can be placed anywhere a tab, space, or newline is permitted and can span multiple lines of a program. For example:

```
/* symbols */
```

or

```
/* symbols  
...  
symbols */
```

or

```
void main()          /* symbols */  
                    /* symbols */.
```

Comment pairs do not nest – that is, one comment pair cannot occur within a second pair. One way to fix the problem of the nested comment pairs is to put a space between the asterisk and the slash:

```
/* * /.
```

The asterisk-slash sequence is treated as a comment delimiter only if the two characters are not separated by white space.

In comments, symbols are not only letters from C++ language alphabet but also any possible symbols including Russian, Arabian letters, and any other symbols.

2.4. DATA TYPES

All types of data can be divided into two categories: scalar and aggregate (Fig. 2).

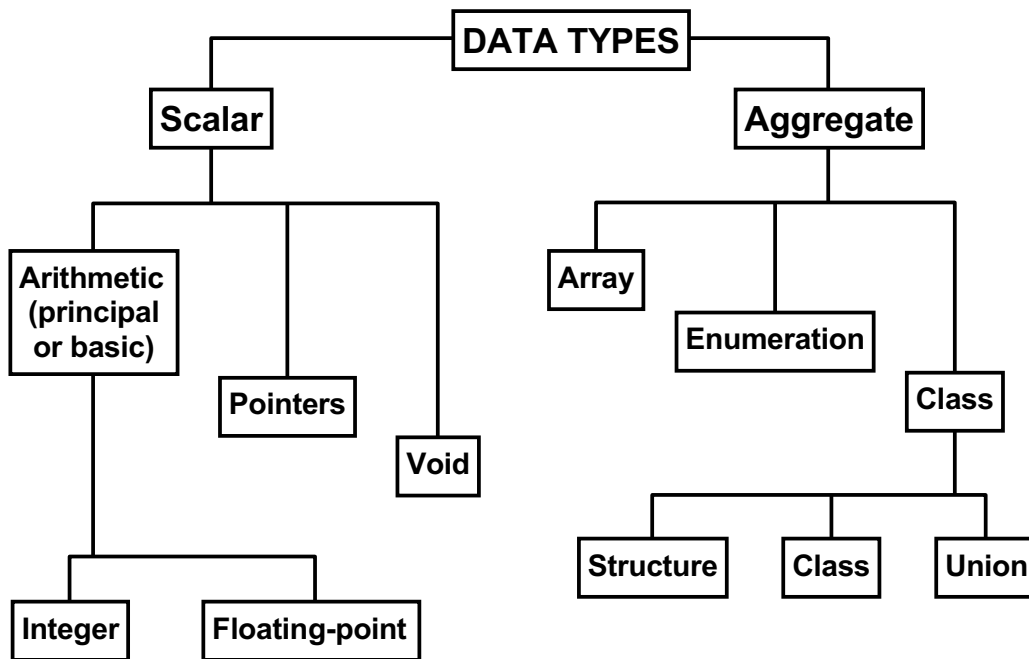


Fig. 2. Types of C++ language data

The key words, used for declaration of the basic types of data are:

- for integral types: **char, int, short, long, signed, unsigned**;
- for floating-point types: **float, double, long double**;
- for classes: **struct, union, class**;
- for enumeration: **enum**;
- for void type: **void**.

2.5. INTEGER DATA TYPE

char or symbol type

The data of **char** type are different symbols, where the value of these symbols is numerical value in inner coding of computer.

A symbolic constant – is a symbol, enclosed in apostrophe, for example: '&', '4', '@', 'a'. For example, the symbol '0' has value 48 in ASCII coding.

There are two modifications of this type: **signed char** and **unsigned char**.

Data of **char** occupy one byte and change in the range:

- signed char (or simply char) $-128 \dots 127$;
- unsigned char $0 \dots 255$.

Let's note that if it is necessary to deal with variables, possessing the value of Russian, Arabian letters, and some special symbols, their type must be **unsigned char**, since the codes of such letters are greater than 127 (in ASCII coding). Symbols, including nongraphic, can be represented as symbolic constants with the help of so called control sequences. A control

sequence is special symbolic combinations, beginning with \ (backslash), that is followed by a letter or a combination of figures (see Table 2).

Sequences '\ddd' and '\xddd' allow representing any symbol from the computer charset as a sequence of octal digits or hexadecimals accordingly. For example, carriage return symbol can be set as: '\r' or '\015' or 'x00D'.

Table 2

Control Sequences

Control sequence	Name
\a	Ring
\b	Step back
\t	Horizontal tabulation
\n	Line feed
\v	Vertical Tab
\r	Carriage return
\f	Form feed
\"	Quotation marks
\'	Apostrophe
\\	Backslash

int type (equivalent short int)

The data of int type occupy two bytes and possess integer values from the range:

-32 768 ... 32 767.

unsigned int type

The data of such type occupy two bytes, and their range:

0 ... 65 535.

long (long int) type

Such data occupy four bytes and change in the range:

0 ... 4 298 876 555.

We should note that if an integer constant exceeds **int**-range, it automatically becomes the constant of **long** type or even **unsigned long**.

Thus, 32768 is of **long** type, 2676768999 is of **unsigned long type**.

It is possible to set the type of a constant with the help of suffixes 'U' (unsigned) and 'L' (long):

-6L; 6U; 33UL.

Actually, in the language standard it is only determined that `sizeof(char) = 1` and `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)`.

Here **sizeof(type)** is an operation, determining the size of **type** kind in bytes.

The integer constant, which begins with `0`, is an octal constant, and the one that begins with `0x` – a hexadecimal constant, for example:

`031; 0750; 01`

are octal constants;

`0x17; 0xA9; 0xFF`

are hexadecimal constants.

2.6. FLOATING-POINT DATA TYPES

The information about floating-point data types, which represent real numbers in computer, is given in Table 3.

Table 3

Floating-point types

Type	Length, bytes	Range	Decimal digits
float	4	3.4e-38 ... 3.4e38	7
double	8	1.7e-308 ... 1.7e308	15
long double	10	3.4e-4932 ... 1.1e4932	19

By default, floating constants are of **double** type, if they do not exceed appropriate range:

`1.0; .3; -6.; 2.3e-6 (means 2.3·10-6); -3e-19`

are constants of **double** type.

Suffix *l* indicates that a floating constant is of **long double** type:

`3l, 3e8l, 1.6e-19l, 1.3e-200l`

are constants of **long double** type.

2.7. STRING CONSTANTS, OR LITERALS

String constant is a sequence of symbols, enclosed in quotes:

`"row"; "a + b = c\" - is an equality \".`

A string literal is represented in memory as an array of **char**-type elements.

During compilation null symbol '\0' is added automatically at the end of each string for convenience. This is an indication of the end of the string. Thus, literal “four” occupies not four but five bytes in memory.

The value of a string is the address of its beginning.

2.8. A PREPROCESSOR STATEMENT *DEFINE*

The **define** statement of a preprocessor can be represented as

```
#define name substitution_text,
```

for example

```
#define nmax 1000
```

or

```
#define km (nmax*3+1).
```

The “name” that is indicated in `#define`, in the range of its visibility is replaced in the program with the “substitution_text”. Thus, **1000** appears everywhere in the text instead of **nmax** name (in the range of its visibility), and **(1000*3+1)** instead of **km**. Let’s note that there is one more form of the `#define` directive (with parameters). Statement `#define` could be used to create constants through text substitution, but it is better to use `const` to allow type checking. Statement `#define X Y` has the effect of replacing symbol `X` with arbitrary text `Y` before compiling.

2.9. DECLARATIONS

All names (in particular, variable names) must be declared before their use. The syntax is a type name followed by a list of objects with possible modifiers and initializers applying to each object:

```
int i, j, k, pmax;  
float radix, a, b, s_m;  
double k, kr;  
char ch, chl;  
char symbol;
```

During declaration, the variable can be initialized by some value, for example:

```
char t = 't', BACKSLASH = '\\';  
int i = 0, j, k, s = 1;  
float ro, eps = 1e-6;
```

2.10. **CONST** MODIFIER

If **const** modifier presents in the name of declaration, the object, which the given name is referred to, is considered in the domain of existence of this name as a constant. For example:

```
const int i = 50;    // the same as const i = 50;  
const double pi = 3.14159;
```

Such named constants cannot be changed in the program. It is possible to use these constants as conventional.

3. EXPRESSIONS

An expression is a combination of different operands and operators. For example:

```
a + b;    a / b;    c << d;
```

and so on.

3.1. ASSIGNMENT OPERATOR AND EXPRESSION

The operator of assignment is indicated by symbol ‘=’

The simplest kind of assignment operator:

```
v = e.
```

Here **v** is any expression that can possess the value, **e** is an arbitrary value.

Assignment operator is performed right-to-left, i.e. first, the value of expression **e** is calculated, and then this value is assigned to the left operand **v**. The left operand in the assignment operator must be so called **address expression**, which can be also called ***l*-value**. The example of address or designational expression is the name of the variable. For example, expression **a + b** is not *l*-value. Constants can never be address expression. In C++ language the assignment operator forms the assignment expression, i.e.

```
a = b
```

means not only sending of **b** value into **a**, but also that **a = b** is an expression, which value is the left operand after assignment. It follows that it is possible to write the following:

```
a = b = c = d = e + 2;
```

Therefore, the result of assignment expression is its left operand. If the type of the right operand doesn't coincide with type of the left one, the value on the right is transformed to the type of the left operand (if it is possible). At the same time, the value loss can take place, for example:

```
int i; char ch;
```

```
i = 3.84; ch = 777;
```


Here **i** obtains value 3, and 777 value is too large to be represented as **char**, that is why **ch** value will depend on the way a specific implementation performs transformation from greater to less integer type.

There is also a so called combined assignment operator (**ao**):

```
a ao= b;
```

where **ao** is one of the binary operators:

+, -, *, /, %, >>, <<, &, |, ^, &&, and ||.

Assignment of

```
a ao= b
```

is equivalent to **a = a ao b**, except for the address expression is calculated only once. Examples:

```
a += 2      means      a = a + 2;
```

```
bottom_count[2*i+3*j+k] *= 2
```

means

```
bottom_count[2*i+3*j+k] = bottom_count[2*i+3*j+k]*2;
```

```
s /= a      means      s = s/a.
```

The result of assignment operator is its left operand; consequently, its result is an address expression, and that is why the following recording is possible:

```
(a = b) += c;
```

It is equivalent to the following two expressions:

```
a = b; a = a + c;
```

3.2. ARITHMETIC OPERATORS

Binary arithmetic operators are **+ - * / %**. (There are also unary **+** and **-**.) A fractional part is truncated during division of integers. Thus **10/3** gives **3**, at the same time **10/3.0** gives **3.33333...** The **a % b** operator is applied only to integer operands and yields remainder from division **a** by **b**, so

```
10 % 3 gives 1,
```

```
2 % 3 gives 2,
```

```
12 % 2 gives 0.
```

3.3. RELATION OPERATORS

Relation operators are `=>`, `>`, `<=`, and `<`. All of them have equal precedence. According to the level of precedence, **equality** and **inequality** operators follow directly after them: `==` (equal) and `!=` (not equal) with the same precedence.

Relation operators are of lower order than arithmetic one so that the expressions like `i < lim + 3` are understood as `i < (lim + 3)`.

Comparison operator means some expression. The value of this expression equals integer 1 (TRUE), if the condition is true and equals 0 (FALSE), if not.

3.4. LOGICAL OPERATORS

Logical operators include:

unary operator logical NOT,	!	(negation);
binary operator logical AND,	&&	(conjunction);
binary operator logical OR,		(logical addition, disjunction).

Operands of logical operators can be of integer, floating types, and some other types, at the same time operands of different types may be involved in every operator.

Operands of logical expressions are calculated from left-to-right. The result of a logical operator is 0 (FALSE) or 1 (TRUE) of `int` type.

!operand (NOT-logical) operator yields 0 (FALSE), if the **operand** is nonzero, and it results in 1 (TRUE) if the **operand** equals zero.

&& operator (AND-logical, logical multiplication) results in 1 (TRUE), if both operands are nonzero. If one of the operands equals zero, the result is zero (FALSE) as well. If the value of the first operand equals zero, the second operand is not computed.

|| operator (OR-logical, logical addition) yields zero (FALSE), if both operands are equal to zero. If at least one of the operands not equals zero, the result of the operator is 1 (TRUE). If the first operand is nonzero, the second operand is not computed.

According to priority these operators are arranged in the following way: `!`, `&&`, `||`.

3.5. BITWISE OPERATORS

Bitwise operators include:

AND bitwise operator &;
 OR bitwise operator |;
 exclusive OR bitwise operator ^;
 unary operator of bitwise negation (complement) ~.

Besides, shift operators are considered: << and >>.

Operands of bitwise operators can be of any integer type.

& operator compares every bit of the first operand with the corresponding bit of the second operand. If both compared bits equal 1, the corresponding bit of the result is also set to 1, otherwise it is set to zero.

| operator compares every bit of the first operand with the appropriate bit of the second operand. If any of them or both bits equal 1, the appropriate bit of the result is set to 1, otherwise it is set to zero.

^ operator. If one of the compared bits equals zero, and the other is 1, the appropriate bit of the result is set to 1, otherwise, i.e. when both bits equal 1 or if both are 0, the bit of the result is set to zero.

~ operator changes 0 into 1 and 1 into 0 in the bit representation of the operand.

& bitwise operator is often used for masking of some bit aggregate, for example expression

`C = N&0177`

transfers seven lower bits of N into C, supposing the rest equal zero. (Octal constants begin with the first zero in C++; hexadecimal constants – with 0x)

Let N = 642. Let's equate bitwise representation of N, octal constant of 0177 and result of C:

N	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
0177	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

| operator is used for bit gain. Thus, operator **C = N|MASK** sets in 1 those bits of N, which equal 1 in MASK.

More examples:

```
int a = 0x45ff, b = 0x00ff;
int c;
c = a ^ b;      // c: 0x4500;
c = a | b;      // c: 0x45ff;
c = a & b;      // c: 0x00ff;
c = ~ a;        // c: -0x3a00;
c = ~ b;        // c: -0x7f00.
```

This program segment can be represented as follow:

a	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1
b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
c = a ^ b	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0
c = a b	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1
c = a & b	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
c = ~b	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0

It is possible to determine the residue of operand division of **unsigned int** type by 2, 4, 8, 16 and so on by means of the operator **&**. It is enough to apply **&** operator to the dividend with masks of 0x01, 0x03, 0x07, 0x0f, 0x1f and so on.

For example:

7&0x03 results in 3.

In other words, the lower bits of a variable are picked out, and the rest are set to zero.

3.6. SHIFTS

Shift operators **<<** and **>>** shift the bits of the left operand left and right, accordingly. The number of bit positions to shift is specified by the right operand. Thus, $X \ll 2$ shifts X left by 2 positions, filling right bits with zeros, that is equivalent to the multiplying by 4. The shift of the value without sign right is accompanied by addition of zeros to left bits. The right-shift of such value by n -bits is equivalent to integer division of the left operand by 2 in the n^{th} power.

Thus,

5 << 3 yields 40;

7 >> 2 yields 1.

Let's note that the right operand must be a constant expression, i.e. the expression, including only constants. If the right operand is negative or it is greater or equal to the number of bits of the left operand, the result of the shift is not specified.

3.7. INCREMENT AND DECREMENT OPERATORS ++ AND --

These operators are unary operators of assignment. They accordingly increase and decrease the operand value by 1. The operand must be of an integer or floating type (or a pointer-type) and it must not be a constant address expression (i.e. without word "const" in the description). The type of the result corresponds to the type of the operand.

A prefix form of the operator:

`++operand` and `--operand`.

Postfix form:

`operand++` and `operand--`.

If a sign of the operator is before an **operand**, the result of the operator is an **increased** or **decreased** value of the **operand**. At the same time the result is an address expression (*l-value*).

If a sign is **after** an operand, the value of the expression is the value of the operand. **After** using this result the value of the operand increases or decreases. The result of postfix form of these operators is not *l-value*.

Examples:

```
int i = 0, j = 0, k, l;
k = ++i;           // here k = 1 and i became = 1;
l = j++;           // l = 0 , and j became = 1;
--k;               // k = 0;
++j;               // j became = 2.
```

In other words, the result of assignment

```
k = ++i;
```

is the same as in the sequence of the operators

```
i = i + 1; k = i;
```

And the result

```
k = i++;
```

is the same as

```
k = i; i = i + 1;
```

3.8. TERNARY OR CONDITIONAL OPERATOR

A ternary operator, i.e. the operator with three operands, has the form:

```
operand1 ? operand2 : operand3.
```

The first operand can be of an integer or floating type (as well as pointer, reference or an element of enumeration). It is important whether the value of the first operand is zero (FALSE) or not for this operator. If **operand1** is not equal to **zero**, **operand2 is computed** and its value is a result of the operator. If **operand1** equals **zero**, **operand3 is computed** and its value is the result of the operator. Let's note that either **operand2** or **operand3** is calculated, but not both of them.

Example:

```
max = a <= b ? b : a;
```

A peak value of variables **a** and **b** is assigned to variable **max**.

If in the conditional operator **operand2** and **operand3** are the address expressions, a ternary operator can be on the left of the sign of assignment:

```
a < b ? a : b = c * x + d;
```

Here the value of expression **c*x+d** is assigned to the less of two variables **a** and **b**.

3.9. SEQUENCING OF OPERATORS

A symbol for sequencing of operators is “,” (comma). The expressions separated by this symbol are performed left to right strictly in the order they are enumerated.

The result of this operator is a result of the last expression. If it is an address expression, the result of the operator is also an address expression.

Examples:

```
int a = 3, b = 8; c; // here a comma is a separator, not an operator;
```

```
c = a++, a + b; // here a will be equal to 4, and c will be 12;
```

```
(b--, c) *= 3; // here b will be equal to 7, and c will be 36.
```

The sequence operator is often used in the operator **for**. It is possible to include several expressions into different parts of this operator, for example, for parallel alteration of two indices. This is illustrated by the function REVERSE(S), which arranges S string in a reverse order at the same place.

```
void REVERSE(char S[ ])
{
int C, I, J;
for(I = 0, J = strlen(S) - 1; I < J; I++, J--)
    {
    C = S[I]; S[I] = S[J]; S[J] = C;}
}
```

In this example **strlen(S)** is a function calculating the number of symbols in S string (without symbol '\0').

Commas, which separate the arguments of functions, variables in descriptions, and so on are not referred to operator **comma** and do not provide left to right calculations.

3.10. OPERATOR PRECEDENCE AND CALCULATION ORDER

The information about all C++ operators is represented in the Table 4.

The first priority is the highest. Most operators are performed left-to-right. For example, expression **a + b + c** can be interpreted as **(a + b) + c**. The exception is unary operators, assignment operators and operator **? :**, which are performed right-to-left: **a = b = c** is performed as **a = (b = c)**.

If it is necessary to change the order of actions, round brackets are applied.

Expression

$7.*a + b/-c$

is interpreted as

$(7.*a) + (b/(-c))$.

It is possible to change this order:

$7.*(a + b)/(-c)$.

We should note that the expressions containing one of the binary operators *** + & ^ |**, can be regrouped by a compiler even if they are enclosed in round brackets. It is possible to use explicit intermediate calculations for providing a necessary order of computing. The order of operand calculation is not fixed in the expression in C++. For example, in

```
c = sin(a * x + b) + fabs(x);
```

firstly the first operand may be computed and then the second one, and may be vice versa. In simple cases, it doesn't make sense. However, if a definite order is necessary, intermediate variables can be introduced.

Table 4

Operators precedence and the execution order

Priority	Operator	Note	Execution order
1	:: -> .	context resolution, extraction	left-to-right
	[]	array indexing	left-to-right
	()	function call	left-to-right
	()	type conversion	left-to-right
2	++ -- ~ !	increment, decrement, complement, not	right-to-left
	- +	unary - unary +	right-to-left
	&	address of	right-to-left
	*	pointer resolving	right-to-left
	new, delete	create and destroy	right-to-left
	sizeof	size of object	right-to-left
3	*	multiplication	left-to-right
	/	division	left-to-right
	%	remainder	left-to-right
4	->* .*	extraction	left-to-right
5	+	binary addition	left-to-right
	-	binary subtraction	left-to-right
6	<< >>	shifts	left-to-right
7	< <= > =>	comparison	left-to-right
8	== !=	equal not equal	left-to-right
9	&	bitwise AND	left-to-right
10	^	XOR (excluding OR)	left-to-right
11		bitwise OR	left-to-right
12	&&	AND-logical	left-to-right
13		OR-logical	left-to-right
14	? :	ternary operator	right-to-left
15	= *= /= %= += so on	assignment operators	right-to-left
16	,	sequencing	left-to-right

4. STATEMENTS

4.1. EMPTY STATEMENT

A **semicolon (;)** in C++ is an indicator of the end of the statement. The simplest form of program statement is the **empty**, or **null** statement that consists of **;**. It is used in the place where a statement must be according to the rule of the language, but due to the logic of the program there is nothing to do. Any expression, which is followed by **;**, is a statement. Such statement is called a **statement-expression**. For example:

```
i++;  
a = b + c;  
c += (a < b) ? a : b;  
x + y; // The result here is not used  
// and a warning will appear.
```

4.2. COMPOUND STATEMENT

A compound statement is differently called a **block**. It is a sequence of statements enclosed by a pair of curly braces. A composite statement must be used at the place where language syntax requires the presence of only one operator, and program logic requires several operators at once:

```
{i = 5; c = sin(i * x); c++;} // It is a block.
```

4.3. DECLARATIONS

In C++, the declarations are the language statements and can stand at the place where any other statement C++ is possible:

```
s = 0.3; d /= s; intk = 5;  
d = s + 2 * k;  
double f = s + d; f *= k;
```

4.4. CONDITIONAL STATEMENT

There are two forms of a conditional statement:

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

Statement1 is calculated in case the **condition** possesses a nonzero value. If the **condition** is zero (or indicator NULL), the **statement2** is calculated.

Examples:

```
if (a > b) c = a - b; else c = b - a;
if (i < j) i++; else {j = i - 3; i++;}
```

When using nested statements **if**, current **else** is always belongs to the latest **if**, with which no **else** has been matched yet.

```
void main() {
int a = 2, b = 7, c = 3;
if(a > b)
    {if(b < c) c = b;}
else c = a;
cout<<"c="<<c<<".\n";
}
```

Here the result will be the output of the string **c=2**.

If you omit the curly braces at the **if**-statement, the program will have the form:

```
void main() {
int a = 2, b = 7, c = 3;
if(a > b)
    if(b < c) c = b;
    else c = a;
cout<<"c="<<c<<".\n";
}
```

Here **else** belongs to the second **if**. As a result the following string will be drawn: **c=3**.

4.5. THE SWITCH STATEMENT

This statement allows to pass control to one of several labeled statements depending on the value of an integer expression. The labels of **switch** operator have special view:

case integer_constant:

The form of **switch** statement:

```
switch(integer_expression) {
    [declarations]
    [case constant_integer_expression1:]
        [statements]
    ...
}
```

```

[case constant_integer_expression2:]
    [statements]
...
[case constant_integer_expression m:]
    [statements]
[case constant_integer_expression n:]
    [statements]
[default:]
    [statements] }

```

Here [] mean optional part of the statement, and ... indicates that a specified construction may be applied ad libitum. A block after **switch()** is called a statement body of **switch**.

The sequence of statement performance is as follow:

First the **integer_expression** in parentheses is calculated (let's call it a selector).

Then the calculated value of the selector is sequentially compared with the constant expressions, following **case** keywords.

If a selector is equal to any of such a constant expression, the control is transferred to the statement, labeled by appropriate statement **case**.

If a selector does not coincide with any of the labeled variant, the control is transferred to the statement, labeled as **default**.

If **default** is absent, the control is passed to the statement following **switch** statement.

Let's note that after control transfer according to one of the labels, the further statements are performed in succession. That is why if it is necessary to perform only a part of them, it is important to provide the exit from **switch**. It is usually performed with the help of **break** statement, which leads to immediate exit from the body of **switch** statement.

Example 1:

```

int i, d;
cout<<"Set an integer value i\n";
cin>>i;
switch(i){
    case 1: case2: case3: cout<<"i="<< i <<"\n";
    case 4: cout<<"i="<< i <<" i^2= " <<i*i<<"\n";
        d=3*i - 4; cout<<"d=" << d <<".\n";
        break;
    case 5: cout<<"i=5.\n";
        break;
    default:
        cout<<" i value is less than 1 or greater than 5.\n";
}

```

If the number 2 is entered, the following will be displayed

```
Set an integer value i
2
i=2
i=2 i^2= 4
d=2.
```

If i equals 4, the following will be displayed

```
Set an integer value i
4
i=4 i^2= 16
d=8.
```

When i = 5, it will be

```
Set an integer value i
5
i=5.
```

With all the rest values of i it will be displayed

```
Set an integer value i
7
i value is less than 1 or greater than 5.
```

Example 2:

```
char sign;
int x, y, z;
cout<<"set the sign of the operator + - * / \n";
cin>>sign;
cout<<"Set x and y \n";
cin>>x>>y;
switch(sign) {
    case '+': z = x + y; break;
    case '-': z = x - y; break;
    case '*': z = x * y; break;
    case '/': if(y == 0){cout<<
        "it is forbidden to divide by zero!\n";
        exit(1);}
        else z = x / y; break;
    default: cout<<"Unknown operator!\n"; exit(1);
}
```

Here **exit(1)** – is a call of a function, which result in the immediate stop of a program run.

4.6. THE *WHILE* STATEMENT

A **while** loop statement with **pre-condition** looks like the following:

```
while(condition) statement
```

The **statement** is called a body of the loop. First, the value of **condition** is computed when performing such a statement. If it is equal to 0, the **statement** is not performed and the control is transferred to the statement following it. If the value of the **condition** is non-zero, the **statement** is performed and then the **condition** is computed again, and so on. It is possible that the loop statement will never be performed, if the **condition** will be equal to 0.

Example 1:

```
char c;
while(cin.get(c)) cout<<c;
```

Here symbol copying takes place, including whitespaces, from **cin** stream (in this case from the keyboard buffer) into **cout** stream (in this case to monitor screen). Here **get(c)** function (a member of the class) extracts one symbol from input stream, including whitespaces. It yields zero value until it reach the end of the file (EOF) (the indicator of EOF – **ctrl-z**).

Example 2:

```
while(1){ statements ... }
```

This is an infinite loop.

Example 3:

```
char c;
while((c = cin.get()) == ' ' || c == '\n' || c == '\t');
```

This loop statement skips from **cin** stream so called whitespaces. Here **get()** is another function form, reading one symbol from the stream. It returns an integer number – symbol code or a number –1, if the indicator of EOF is met.

4.7. THE LOOP *DO-WHILE* WITH POST-CONDITION

This loop statement verifies the condition of ending at the end (after every pass through the loop body) that is why the loop body is performed once at least.

Statement appearance:

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

First, the **statement** is performed, then the **condition** is calculated and if it is non-zero, the statement is calculated again, and so on. If the **condition** equals zero, the loop is completed. Such a loop is very convenient to use when checking data, keyed by the user:

```
int input = 0;
int minvalue = 10, maxvalue = 150;
do {cout <<"key the input value \n"; cin >>input;
    cout <<" input=" << input << "\n";}
while(input < minvalue || input > maxvalue);
```

4.8. THE FOR LOOP STATEMENT

This loop statement can be represented as:

```
for(init_statement; expression1; expression2) statement2
```

Init_statement can be a declaration, an empty statement or an expression statement. The most widespread case is when **init_statement** and **expression2** are assignments or function references, and **expression1** is a conditional expression. This loop is equivalent to the following construction:

```
init_statement;
while(expression1){statement2 expression2;}
```

Sometimes **init_statement** is called a loop initializer and **expression2** is a reinitializer.

Any of three parts can be omitted, although it is obligatory that a **semicolon** remains. If the condition, i.e. **expression1**, is absent, it is considered that **expression1** is non-zero, thus:

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

is an infinite loop and it is necessary to abort it.

Example 1:

```
int n = 20, s = 0;
for(int i = 1; i <= n; i++) s += i*i;
```

Here the sum of squares of integers from 1 to 20 is calculated.

Example 2:

```
double s, sum, den = 0.85, eps = 1e-10;
for(s = 1, sum = 0; s > eps; s *= den) sum += s;
```

Here the sum of geometric progression 1, $1*0.85$, $1*0.85*0.85$, etc. is calculated, until its regular member becomes less than 10^{-10} .

Let's calculate and display the table of $y = \sin(x^2)$ function for $x \in [0, \pi/2]$ with step $\pi/20$ in the next example.

```
#include <iostream.h>
#include <math.h>
#include <conio.h>

void main(){
int n = 10;
double x0 = 0, xk = M_PI_2, y,
h =(xk - x0)/n, xt = xk + h/2;
clrscr();
cout<<" x y\n"
for(double x = x0; x < xt; x += h){
    y = sin(x*x);
    cout.width(4); cout.precision(2);
    cout<<x;
    cout.width(10); cout.precision(4);
    cout<<y<<'\n';
}
}
```

In this program **M_PI_2** constant, which represent value $\pi/2$ and is specified in header file **math.h**, is used. The reference to the function **cout.width(k)** sets the width of field of the next output into **k** positions which allows to align the table appearance. The function **cout.precision(k)** assign the number of digits, produced after a decimal point. The function **clrscr()**, which prototype is in **conio.h**, clears the screen.

The use of **while** and **for** loops is basically a matter of taste. **For** loop is more preferential at the place where there is simple initialization and reinitialization, as during this the statements, controlling the loop, visually appear together at the beginning of the loop. It is the most evident in the following construction:

```
for(i = 0; i < n; i ++),
```

which is applied for processing the first **n-th** elements of the array, similar to loop statement **for** in Pascal. The analogy however is not absolute, as the limits of the loop can be changed inside of the loop, and a control variable saves its value after loop termination, whatever reason of its termination is.

4.9. THE STATEMENT OF UNCONDITIONAL JUMP

The statement of unconditional jump can be represented as **goto label**.

Label is a name, which is followed by ‘:’. This statement passes the control to the statement, marked by an indicated label. With its help it is convenient to exit from several embedded loops at once:

```
for( i = 0; i < n; i++)
for( j = 0; j < m; j++)
for( k = 0; k < l; k++){
...
statements;
...
if(condition) goto lab;
statements;
}
lab;; . . .
```

With the help of **goto** statement it is possible to jump from the outside into the block body, if at the same time the control is not passed through the declaration of names, which are present in this block.

4.10. THE *BREAK* STATEMENT

This statement implements the exit from the body of loops **for**, **while**, **do-while** or statement **switch**, in which it appeared. The control is passed to the first statement after loop.

The statement cannot provide the exit at once from two or more embedded loops.

4.11. THE *CONTINUE* STATEMENT

This statement implements the pass to the point straight away after the last statement of loop body without exit from the loop, thus further iterations in the loop will continue.

The example of output of evens:

```
for(int num = 0; num < 100; num++){
    if(num % 2) continue;
    cout << num << "\n";
}
```

When **num** is an odd number, the expression **num % 2** obtains value 1 and the statement is satisfied, which jumps to the next iteration of **for** loop without output performing.

4.12. THE *RETURN* STATEMENT

This statement completes the function performance, in which it is set, and returns the control into calling function.

The control is passed to the calling function to the point, following directly the call.

If **return** is present at the function **main()**, it activates program operation breakdown.

5. THE *sizeof* OPERATOR

This operator is performed at the stage of compilation. The result of this operator is a number of bytes, necessary for location of the object in the memory. There are two variants of the syntax of this operator. In the first one, a single operand of the operator determines some type name of the language, and it must be enclosed in brackets:

```
sizeof(float);  
sizeof(int).
```

In the second one, the operand sets an expression and here the use of brackets is no necessary:

```
sizeof a;  
sizeof *ip;  
sizeof array[ i ];
```

Let's note that when obtaining the sizes of arrays, in spite of the name of the array being a pointer, the result of the operator

```
sizeof array,
```

where **array** is the name of some array, is a length of this array in bytes. This property may be used for calculation of the element number in the array:

```
const n = 20;  
int array[n];  
...  
int num = sizeof array / sizeof(int)           // num = 20.
```

Application of the **sizeof** operator on a **reference** type returns the size of the memory necessary to contain the referenced object, i.e. **sizeof(double &)** and **sizeof(double)** are equivalent.

6. DECLARATIONS AND DEFINITIONS

Any name, excluding the names of labels, must be declared in the program:

```
int i, j;
double d=7.3;
extern int m;
typedef unsigned int size_t;
int add(int a, int b){return a+b;}
void func(char*, int);
```

After such declarations a compiler knows that **i**, **j**, **m** are the names of the variables of **int** type, **d** is a name of the variable of **double** type, **size_t** is a name of a type, and **add** and **func** are the names of the functions (about functions see part 16).

In declarations not only some type can be associated with the name, but also certain element, which identifier is the name. For example, when declaring **int i, j;** memory is allocated for variables **i, j;** the areas of memory of 2 bytes, which can be used for storage of the variables, are associated with these names. The name of the specific type (unsigned int) is associated with the name **size_t**, which synonym is **size_t** now; the code of the function is associated with the name **add**. All such declarations are called definitions. However not all declarations can be definitions. Two of the above mentioned declarations are not definitions:

```
extern int m;
void func(char*, int);
```

In the first one, it is said that the variable **m** has **int** type; the memory for it must be allocated somewhere else. The key word **extern** indicates it.

The second declaration indicates that **func** is a name of the function with two arguments, the first of which is a pointer to **char**, and the second one is **int**, and the function itself does not return any value. Such declaration is called a **prototype** of a function. The function **func** itself must be defined together with its body somewhere else.

In the program for each name there must be only one definition, while there can be as many as possible declarations, which are not definitions.

7. NAME RESOLUTION

Name resolution is the process by which a name used in an expression is associated with a declaration. It is the process by which the name is given meaning. This process depends on how the name is used and on the scope in which the name is used. The name, as a rule, can be used only in some part of the program.

Scopes and name resolution are compile-time notions; they apply to some portion of program text. These notions give meaning to the program text in a source file. The compiler interprets the program text it reads according to the scope rules and name resolution rules.

The text of the program can be placed in one file or in several different files, each of which contains one or several functions as a whole. For combining into one program these files are compiled simultaneously. The information about all these files is placed in so called project file (with extension **".prj"**). A compiler creates an object code (a file with extension **".obj"**) for each source file. Then all object files (along with library ones) are combined by the linker into executable or load module, which has a name of the project file and extension **".exe"**. The name resolution is necessary for the compiler for generation of the exact computer code.

There are 5 types of name resolution.

7.1. NAME RESOLUTION IN A LOCAL SCOPE (BLOCK)

Each compound statement (or block) represents its own local scope. It should be kept in mind that a block is a program segment, enclosed in curly braces { }, for example

```
if(a != 5){  
int j = 0;  
double k = 3.5;  
a++;  
...  
}
```

The name declared in the block can be used from the point, where its declaration is, and up to the end of the block.

Let's note that a body of any function is a block, so the names in the function definitions have the same name resolution:

```
int f1(int i){ return i; }
```

The name **i** has a "block" as a domain of existence. The domain of existence "block" covers the nested blocks.

7.2. NAME RESOLUTION IN A FUNCTION

Only the names of transition labels, used by the statement **goto**, are resolved in a function:

```
void f(){...
...
goto lab;
...
{... lab: ...}
...
}
```

7.3. NAME RESOLUTION IN FUNCTION TEMPLATE DEFINITIONS (FUNCTION PROTOTYPE)

A function prototype is a function declaration, which is not its definition and has, for example, the following view:

```
int F(int a, double b, char* str);
```

The name resolution in such a case is enclosed between closing and opening round brackets. In other words, the names **a**, **b**, and **str** in the example are defined only inside the round brackets. Consequently, it is possible to use any names for the arguments or not to use them at all in the prototypes:

```
int F(int, double, char*);
```

7.4. NAME RESOLUTION IN A FILE

The names, declared beyond any block or class are called as global ones. Global names are defined from the point of their declaration up to the end of the file, where their declaration is met. The example of such names is function names:

```
#include <iostream.h>
int a, b, c[40];           // global names;
int f1()                  // local name f1;
{int i;                   // local name;
...
}
int count;                // global name;
void f2(){ ... }         // global name f2.
```

7.5. NAME RESOLUTION IN CLASS SCOPE

The names, declared in classes, are defined in all the class, where they are declared, independently of the point of their declaration.

8. VISIBILITY SCOPE

If, using a name, it is possible to obtain the access to the element, which this name is associated with, the given name is said to be in the scope of its visibility. A visibility scope is a subregion of the domain of name resolution. If the element of the language, which name is in the domain of its resolution, nevertheless, is not accessible using this name, this name is supposed to be latent or masked. Global names are visible from the point of their declaration up to the end of the file, if they are not masked by the local names. The variables from enclosing blocks, as well as global ones, are visible in the internal blocks. If a variable, declared inside the block, has the same name as the name of the variable of enclosing level, the name of the enclosing level is masked and definition of the variable in the block replaces the definition of the enclosing level over the whole block. The visibility of the masked variable is restored at the transfer out of block. The labels in the function are visible in the whole function body.

```
int i = 3;
{int c = i;    // c becomes equal to 3;
...
int i = 0;    // name i masks external name i;
cout <<"c = "<< c <<" , i= "<< i <<"\n";
}            // the end of the domain of names i and c resolution at the
            // block; name i, declared before the block, is visible again
cout <<" i = "<< i <<"\n";
...
```

Here the following will be printed:

```
c=3, i=0.
i=3.
```

If a global name of the function or the object is a masked name, then it is possible to call it, using the operator **access permission**, or **context permission ::**, for example:

```
int i=5;          // Global variable;
void main() {
int i=1;         // local variable.
i++;
::i++;
cout<<"i="<<i<<" , global i= " << ::i << ".\n";
}
```

Here the following will be typed

```
i=2, global i=6.
```

Using the operator `::` it is impossible to call the masked local object.

9. MEMORY CLASSES

There are 3 memory or storage classes in C++:

- 1) static memory is static data, located in data segment;
- 2) automatic data, located in a special stack (stack segment) or as a special case, in processor registers;
- 3) dynamic data, which are evidently located in dynamic memory with the help of **new** and **delete** operators.

Static objects exist during the whole period of program performance. Global and local variables, declared with accessory word **static**, refer to them:

```
int i=3, j; // global variables. Memory class – static;
void main(){
int a; // Automatic variable;
static float b[1000], c=2.3; // static variables;
...
}
int f(){
int d; // automatic variable;
static int m=2, k; // static variables m, k.
...
}
```

Static and global variables, if they are not initialized evidently, are initialized by zero values. **In any case** initialization of static variables is performed **only once**.

Local variables, which are not declared as static, are automatic. Such object starts its existence when its name is declared in the block and finish it when this block is completed. If the automatic object is not initialized evidently, its value is not determined until assignment.

```
void f();
void main(){
for(int i = 3; i > 0; i--) f();
}
void f(){static int i; int j = 0;
cout<<"i ="<<i++<<" j ="<<j++<<"\n"; }
```


Here the following will be typed:

```
i = 0      j = 0
i = 1      j = 0
i = 2      j = 0
```

Let's note that if an accessory word **static** is applied to a global variable or to the function name, it has different meaning. In this case both a global variable and a function become visible only in the limits of the file, where they are determined, and are not visible from other files.

10. OBJECT AND TYPE DECLARATION

When declaring, it is possible to use more than one modifiers (it is * [] and ()) simultaneously. It allows to create unlimited variety of complicated type descriptors. At the same time, some combinations are not allowed:

- functions cannot be the array elements;
- functions cannot return an array or function.

When interpreting complicated descriptors, square and round brackets (to the right of the identifier) have priority before * (to the left of the identifier). Square or round brackets have the same priority.

A type qualifier is considered at the last stage. It is possible to use round brackets to change the order of the interpreting into necessary one. For correct interpreting of the complicated descriptors it is possible to follow the rule below (“inside – outside”).

Start with the identifier and look to the right, whether there are square or round brackets.

If they are present, interpret this part of the descriptor and then look to the left in the search of *.

If a closing round bracket is met at any stage on the right, it is necessary to apply all these rules inside round brackets and then proceed with interpreting.

Interpret a type qualifier.

For example, when using the construction

```
int *(*array[10]) ( );
```

the name **array** is declared as the array, consisting of 10 pointers to the function without arguments, returning the pointer to the integer value.

11. TYPE NAMES

The type names are obviously used in some language constructions (operators **sizeof**, **new**, **cast**). The name of the type is simply an object declaration of such type where the name of the object is omitted.

The examples of the type names are:

`int` – integer;

`double` – floating with double accuracy;

`int *` – a pointer to an integer;

`int *[10]` – the array of the pointers to an integer;

`int(*)[10]` – a pointer to the array of 10 integers;

`int *(void)` – a function without arguments, returning the pointer to the integer;

`int*(*)(void)` – a pointer to the function without arguments, returning the result of the integer type.

12. A SYNONYM OF A TYPE NAME

A synonym of the type name is formed with the help of the key word **typedef**. The expression, where this key word is present, is a description of some name. The presence of the word **typedef** indicates that a declared identifier becomes not the name of the object of some type but a synonym of the name of this type.

```
int INTEGER;           // INTEGER – the name of int type variable
typedef int INT;       // INT – a synonym of int type
typedef unsigned size_t;
typedef char string[255];
typedef void(*FPTR) (int);
```

The last two lines determine string as a synonym of the type – “a row consisting of 255 symbols”, and **FPTR** is a synonym of the type – “a pointer to a function, possessing one argument of **int** type, which doesn’t return any result”. After declaration with the help of **typedef**, a new name turns to a full name of the type, given below:

```
string array;         // array – array containing 255 symbols
FPTR func_pointer;   // void(*func_pointer)(int);
typedef string STRING;
```

The use of **typedef** can simplify the understanding of complicated type names. Thus, the type **int(*(void))[]** can be formed as following:

```
typedef int intarray[];           // the type called «integer array»
typedef intarray * ptrtointarray; // the type of the pointer to the
                                  // integer array
typedef ptrtointarray TYPE(void);
```

Now the name **TYPE** can be used, for example, in prototypes:

```
double fun(int, int, TYPE);
```

13. TYPE CONVERSION RULES

In any cases two conversions are performed:

- the name of the array is converted to the pointer to its first element;
- the function name is converted to the pointer to that function.

13.1. EXPLICIT CONVERSIONS

Any conversion of one standard type to another one is allowed. When converting a longer type to a shorter type, a loss of the accuracy occurs. During conversion of a shorter integer type to a longer one, vacant bits are filled with 0 (if a short type is unsigned), or sign propagation occurs (for the type with a sign).

Any conversion of pointers and references into each other is allowed. An explicit conversion of the types is performed by means of **cast** operator, which has two forms:

```
(type_name) operand           // traditional form;
```

or

```
type_name (operand)         // functional form.
```

Here `type_name` specifies the type and **operand** is a value, which must be converted to the specified type.

Let's note that in the second form the `type_name` must be a common identifier, for example, obtained with the help of **typedef**.

Examples:

```
double d =(double)5;
int i = int(d);
int *ip = &i;
float fp =(float*) ip;
typedef float* FP;
fp = FP(ip);
```

13.2. IMPLICIT CONVERSIONS OF A STANDARD BASE TYPES

For standard base types, a compiler can perform any conversion of one type into another:

```
int i='A';           //i = 65;
char c=256;         //8 high-order bits are lost; c will be equal to '\0';
int j=-1;
long l=j;
long m=32768;       // A binary notation of 32768
                   //contains a single unit in 15th bit.
int k=m;           //k=-32768, as 15th bit for int is sign bit.
unsigned u=m;      //u = 32768
double d=0.999999;
long n=d;          //n = 0.
```

When performing arithmetic operations, an implicit conversion of types also occurs. The rules here are the following:

a) the types **char**, **short**, **enum** are converted to the type **int**, and **unsigned short** to **unsigned int**; **float** type is converted to **double**;

b) then if one of the operands has **long double** type, the second one is converted to **long double** also;

c) otherwise, if one of the operands has **double** type, the second one is converted to **double**;

d) otherwise, if one of the operands has **unsigned long** type, the second one is converted to **unsigned long**;

e) otherwise, if one of the operands has **unsigned** type, the second one is converted to **unsigned**;

f) otherwise, if one of the operands has **long** type, the second one is converted to **long**;

g) otherwise both of the operands have **int** type.

Example 1.

```
int g = 10, t = 5, t2 = t*t/2;
double s = g*t2;           // s will be equal to 120;
double s0 = g*t*t/2.0;     // s0 will be equal to 125.
```

Example 2.

A simplified function **atoi**, which converts a string of digits into its numeric equivalent:

```
int atoi( char s[ ] ){
int i, n = 0;
for(i = 0; s[i] >= '\0' && s[i] <= '9'; ++i)
n = 10*n + s[i] - '\0';           // Conversion of char to int.
return n;
}
```

13.3. THE CONVERSION OF DERIVED STANDARD TYPES

An implicit conversion of the pointer to any type to the pointer to **void** type is allowed for pointers. All other conversions must be explicit.

```
int *ip;
void *vp=ip;
ip=vp;           // Error!
ip=(int*)vp;    // now it is correct.
float *fp=ip;   // error.
fp=(float*)ip;  // correct.
```

A constant 0 can be implicitly converted into the pointer to any type. In this case, it is guaranteed that such pointer will not refer to any object. The value of a standard constant NULL equals 0 for all types of the pointers.

14. POINTERS

14.1. DEFINITION OF POINTERS

A pointer is a variable, containing the address of some object, for example, another variable, exactly the address of the first byte of this object. It allows the indirect access to this object through the pointer. Let **x** is a variable of **int** type. Let's label a pointer as **px**. Unary operator **&** yields the address of the object, so the statement

```
px = &x;
```

assigns the address of **x** variable to **px** variable. It is said that **px** "points" to **x**. The operator **&** is applicable only to the address expressions, so the constructions like **&(x-1)** and **&3** are not valid.

Unary operator ***** is called an operator of address resolution. This operator considers its operand as an address and refers to this address to extract the object, available in this address.

Consequently, if **y** also has **int** type,

```
y = *px;
```

assigns **y** the content of that, at what **px** points to. Thus, sequence

```
px = &x;  
y = *px;
```

assigns to **y** the same value as operator

```
y = x;
```

does. All these variables must be described:

```
int x, y;  
int *px;
```

The latter is a pointer description. It can be considered as mnemonic. It indicates that ***px** combination has **int** type or, in other words, **px** is a pointer to **int**. It means that if **px** appears in the form of ***px** it is equivalent to the variable of **int** type.

It is evident from the pointer description that it can point to only definite type of an object (in this case **int**). The address-resolved pointer is valid in any expressions where the object of the type, to which this pointer refers, appears. Thus, statement


```
y = *px + 2;
```

assigns **y** value, greater by 2 than the value of **x** .

Let's note that the priority is that that unary ***** and **&** operators are connected with their operands more firmly than arithmetic, thus expression

```
y = *px + 2
```

takes the value at which **px** points to, adds 2 and assigns **y** the result of calculation. If **px** points to **x** then

```
*px = 3;
```

supposes **x** equals 3 and

```
*px += 1;
```

augments **x** by 1 as well as the expression

```
(*px) ++
```

Braces are necessary here, otherwise this expression (e.g. ***px ++**) increases **px**, not the variable it is pointing to, as unary operators, similar to ***** and **++**, are performed right-to-left.

If **py** is another pointer to **int**, the following assignment can be performed

```
py = px;
```

Here the address from **px** is copied to **py**. As a result **py** points to the same as **px**.

14.2. POINTERS AND ARRAYS

An array is a set of elements of one type, which are successively arranged in the computer memory, one after another.

The indicator of array declaration is square brackets. It is possible to declare the array consisting of 10 elements of **float** type in the following way:

```
float a[10];
```

To refer to the element of this array, it is necessary to use the operation of indexing **a[ind]**. The integer-type expression, which is called index, should be placed inside the square brackets. Numbering of the elements of the array **starts with 0**, therefore the description mentioned above indicates that computer storage contains space, reserved for 10 variables of **float** type and these variables are **a[0]**, **a[1]**,. . ., **a[9]**.

Here is an example for the use of array.

Let's make the program for calculation of the digits, whitespaces, and all other symbols appearance frequency.

The frequency of whitespaces will be kept in **nwhite**, other symbols – in **nother**, and the number of digits appearance – in **ndigit** array:

```
# include <iostream.h>
void main( ){
int c, i, nwhite = 0, nother = 0;
int ndigit[10];
for( i=0; i<10; i++) ndigit[i]=0;
while(( c=cin.get( ) )!=EOF)
if(c>='0' && c<='9') ++ndigit[c - '0'];
else if(c == ' ' || c == '\n' || c == '\t') ++nwhite;
else ++nother;
cout<<" digit \n";
for( i=0; i<10; i++)
cout<<i<<" entered "<<ndigit[i]<<" times \n";
cout<<" whitespaces - "<< nwhite <<" other symbols - "
<< nother <<"\n";
}
```

When declaring the array, it can be initialized:

```
int c[ ] = { 1, 2, 7, 0, 3, 5, 5 };
char array[ ] = { 'h', 'e', 'l', 'l', 'o', '\n', '\0' };
```

The last initialization is allowed to be performed in a simpler way:

```
char array[ ] = "hello\n";
```

Such syntax of initialization is allowed only for strings. A compiler itself calculates the necessary storage size considering symbol '\0' with code 0, automatically added to the end of the string, which is the indicator of the string end.

In C++ language, the name of the array is a constant pointer to the first element of this array:

```
int mas[20];
int *pmas;
pmas = &mas[0];
```

The last statement can be written as: **pmas = mas;**

The operation of array indexing [] has two operands – the name of the array, i.e. the pointer, and index, i.e. integer: **a[i]**. In C++ language, any expression **pointer[index]** is treated as:

```
*(pointer + index)
```

and is converted to such expression by compiler automatically.

Thus, **a[3]** is equivalent to ***(a + 3)**. Moreover, it can be written even as **3[a]**, as it will in any case be interpreted as ***(3+a)**. Here pointer **a** and

integer 3 are summed. In this connection, let's consider so called address arithmetic.

14.3. ADDRESS ARITHMETIC

A pointer can be added to the integer. If integer increment **i** is added to pointer **pa**, the increment is scaled by the storage size, occupied by the object, which pointer **pa** points to. Thus, **pa+i** is an address of **i-th** element after **pa**, where it is considered that the size of all these **i** elements equals the object size, which **pa** points to. So if **a** is an array, **a+i** is an address of **i-th** element of this array, i.e. **&a[i]** equals **a+i** and **a[i]** is equal to ***(a+i)**.

```
float b[10];
float *pb=b;
pb++;           // It is equivalent to pb=pb+1.
               // Here pointer pb will point to the element of array b[1].
pb+=3;         // Here pb points to the element of array b[4].
```

Let's note that it is impossible to write **b++** or **b = b+i**, as the name of the array **b** is a constant pointer; it must not be changed.

The pointers can be compared. If **p** and **q** point to the elements of the same array, such relation as **<** **>** **=**, etc. works properly. For example,

```
p<q;
```

is true, that is **== 1**, if **p** points to the earlier element of the array, than **q**. Any pointer can be compared using equality and inequality to so called zero indicator **NULL**, which doesn't point to anything. However, it is not recommended to compare pointers, pointing to different arrays.

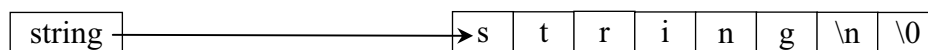
Pointers can be subtracted. If **p** and **q** point to the elements of the same array, **p-q** gives the number of the elements lying between **p** and **q**.

14.4. SYMBOL ARRAYS AND STRINGS

A string is a symbol array. The value of the string is the pointer to its first symbol:

```
char *string = "string\n";
```

Here the pointer to symbols **string** will contain the address of the first symbol – 's' of the string "**string\n**", which is placed at some storage area, beginning with this address:

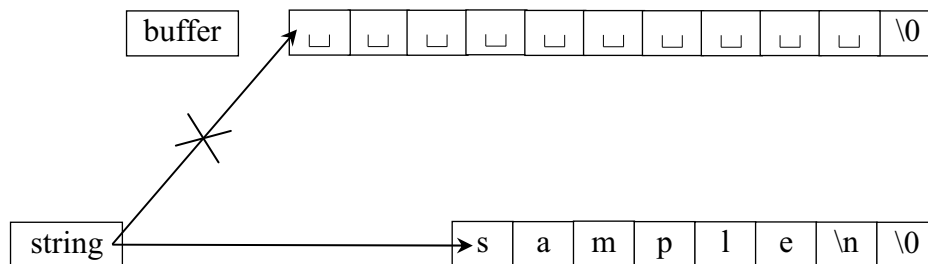


Here `string[3] == 'i'`.

Let's consider the program segment:

```
char buffer[ ] = "    \n"; // Initialization
// string consisting of 10 spaces.
char *string = buffer; // string points to the
// beginning of the buffer.
string = "sample\n"; // Assignment !
```

When initializing, string **buffer** is created and the symbols (here are 10 spaces) are placed into it. Initialization `char *string = buffer` adjusts pointer **string** to the beginning of this string. The assignment statement at the last row does not copy a given string "sample\n" into the array **buffer**, but it changes the value of the pointer **string** in such a way that it starts to point to the string "sample\n"



To copy the string "sample\n" into **buffer**, it is possible to do the following:

```
char *p = "sample\n";
int i =0;
while( ( buffer[i] = p[i] ) != '\0' ) i++;
```

Or otherwise:

```
char buffer[ ] = "    \n";
char *p = "sample\n";
char *buf = buffer;
while(*buf++ = *p++ );
```

Here, first, `*p` is copied into `*buf` i.e. symbol 's' is copied to the address **buf**, which coincides with the address **buffer**, i.e. `buffer[0]` is getting equal to 's'. Then the increment of the pointers **p** and **buf** takes place, which leads to movement along the strings "sample\n" and **buffer** accordingly. The last copied symbol will be '\0', its value equals 0 and **while** statement will end the loop.

It is simpler to use library function, which prototype is in the file **string.h**:

```
strcpy( buffer, "sample\n");
```

When copying, it is necessary to provide that the storage size, reserved for **buffer**, was sufficient for storage of the copied string.

14.5. MULTIDIMENSIONAL ARRAYS

A two-dimensional array is considered as an array of the elements, each of which is one-dimensional array. A three-dimensional array is an array, which elements are two-dimensional arrays, and so on.

After declaration

```
int a[5][6][7];
```

the following expressions may appear:

```
a[i][j][j]    // an object of the type int;
a[2][0]       // an object of the type int* is
               // one-dimensional array of 7 integers;
a[1]          // two-dimensional array of 6*7 = 42 integers;
a             // three-dimensional array itself.
```

As the element of the array **a** is a two-dimensional array with the size $6*7$, a displacement at a value of the element of the array **a** takes place when performing the expression **a + 1**. It means the transfer from **a[0]** to **a[1]**. The value of the address in this case is increased by $6*7*\text{sizeof(int)} = 84$.

For two-dimensional array **mas** the expression **mas[i][j]** may be interpreted as $*(*(\text{mas}+\text{i})+\text{j})$. Here **mas[i]** is a constant pointer at *i*-th row of the array **mas**.

The arrays are stored by rows in memory, i.e., when addressing the elements in the order of their arrangement in the memory, the very right index changes faster.

Thus, for the array **c[2][3]**, its six elements are arranged in memory in the following way:

```
c[0][0]  c[0][1]  c[0][2]  c[1][0]  c[1][1]  c[1][2]
```

Multidimensional arrays are also can be initialized, when declaring:

```
int d[2][3]={ 1, 2, 0, 5 };
```

In this case, the first 4 elements of the array obtain the indicated values, and the rest two will be initialized with zeros.

If a multidimensional array is initialized, the very first dimension can be unspecified. In this case a compiler itself calculates the size of the array:

```
int f[ ][2] = { 2, 4, 6, 1 };           // array f[2][2];
int a[ ][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8 }; // array a[2][2][2].
```

An initializing expression may have the structure, reflecting the fact that the array is, for instance, two-dimensional:

```
int c[2][3]={{1, 7},{-5, 3 } };
```

In this case, in matrix **c** a zero and first columns are initialized, and the second column, i.e. elements `c[0][2]` and `c[1][2]`, are initialized with zeros.

14.6. POINTERS AND MULTIDIMENSIONAL ARRAYS

Let's consider the difference between the objects **a** and **b**, described in the following way:

```
int a[10][10];
int *b[10];
```

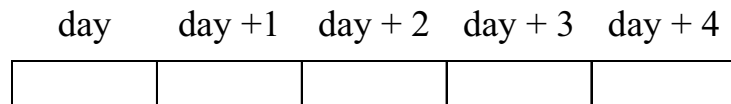
Both **a** and **b** can be used in the same way from the viewpoint that both `a[5][5]` and `b[5][5]` are the references to a specific value of **int** type. But **a** is a real array: 100 memory cells are taken for it and conventional calculations for finding of any indicated element are performed with indices, which need multiplication. For **b** the description singles out only 10 pointers. Each of them must be set in such a way that it points to the integer array.

Assuming each of them points to the array consisting of 10 elements, so that 100 memory cells plus 10 more cells for pointers will be assigned somewhere. Thus, a pointer array uses a rather greater volume of memory and may require the evident initialization step. At the same time, two advantages arise: the access to the element is performed indirectly by means of the pointer instead of multiplication or addition, and the rows of the array may have different lengths. It means that not every element of **b** must point necessarily at the vector consisting of 10 elements. This difference can be seen in the following examples.

Example 1:

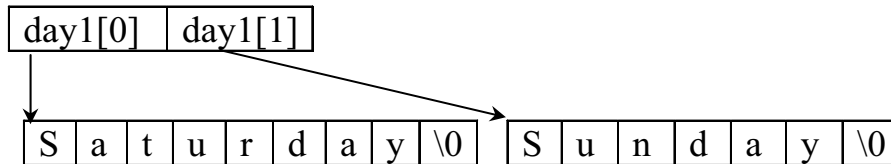
```
char day[5][12] ={
    "Monday",           // In each row there are 12 symbols.
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday"
};
```

Here the constant pointers `day[0]`, `day[1]`, ..., `day[4]` address memory areas of the equal length of 12 bytes each:



Example 2:

```
char *day1[2] = {"Saturday",           // 8 symbols + '\0'
                 "Sunday"};          // 6 symbols + '\0'
```



Here the variables-pointers **day1[0]** и **day1[1]** address the memory areas of 9 and 7 bytes accordingly.

15. DYNAMIC MEMORY MANAGEMENT OPERATORS

15.1. THE OPERATOR *NEW* FOR MEMORY ALLOCATION

The expression containing the operator **new** has the following representation:

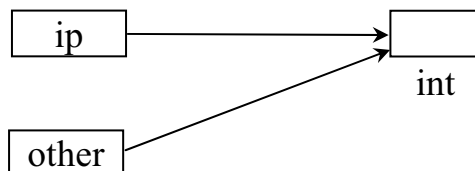
`pointer_to_a_type = new the_name_of_the_type(initializer)`

An initializer is an optional initializing expression, which may be used for all types, except arrays. When performing the statement

```
int *ip = new int;
```

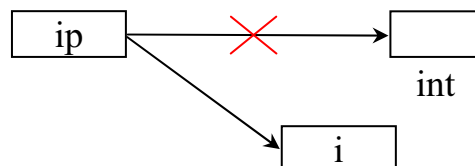
two objects are created – a dynamic nameless object and a pointer to it with the name **ip**, which value is the address of the dynamic object. It is possible to create another pointer to the same dynamic object:

```
int *other = ip;
```



If another value is assigned to the pointer **ip**, the access to the dynamic object becomes lost:

```
int *ip = new(int);  
int i = 0;  
ip = &i;
```



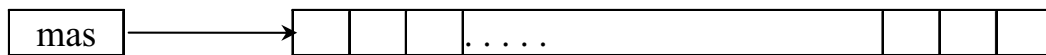
Now a dynamic object still exists, but it is impossible to address to it. We end up with a *memory leak*. A memory leak is a chunk of dynamically allocated memory that we no longer have a pointer to, and thus we cannot return it to the program and use it later.

When allocating, an object can be initialized:

```
int *ip = new int(3);
```

It is possible to allocate memory dynamically for the array:

```
double *mas = new double[50];
```



Now it is possible to work with this dynamically allocated memory as with a common array:

```
*(mas+5) = 3.27;
mas[6] = mas[5] + sin(mas[5]);
```

In the case of successful completion, the statement **new** returns the pointer with the value, different from zero.

The result of the statement, equal to zero, i.e. zero pointer NULL, indicates that a continuous free area of the memory of a required size is not found.

15.2. THE OPERATOR *DELETE* FOR MEMORY DEALLOCATION

The operator **delete** releases a memory area, allocated by the operator **new** earlier, for further use:

```
delete ip;           // Delete a dynamic object of int type,
                    // if it was created as "ip = new int;"
delete mas;         // Delete a dynamic array of the length of 50, if
                    // it was "double *mas = new double[50];"
```

It is absolutely safe to apply the operator to the pointer NULL. The result of the repeated application of the operator **delete** to the same pointer is not defined. A mistake usually takes place, and it leads to circularity.

To avoid such mistakes, it is possible to apply the following construction:

```
int *ip = new int[500];
. . .
if(ip){
delete ip; ip = NULL;
}
else{
cout <<" memory has already been released \n";
}
```

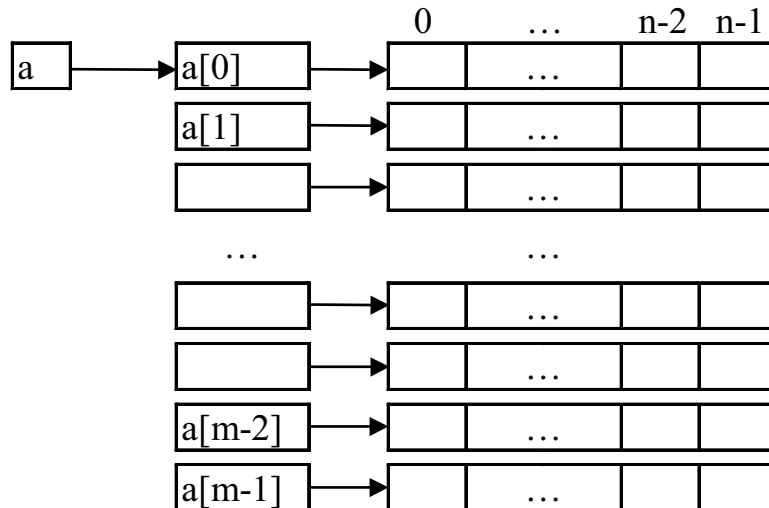
Example. Allocate memory for the matrix with **m** rows and **n** columns:

```
int m, n;
cout<<"Set the number of rows and columns for matrix: \n";
cin>>m>>n;
double **a = new double *[m];           // array of m pointers to double
for(int i = 0; i < m; i++)
if((a[i] = new double[n])!=NULL)       // a row of matrix is allocated
{ cout<<"no memory!\n"; exit(1); }
```

Now it is possible to address to the elements of this matrix in the ordinary way:

```
a[i][j] or *(a[i] + j) or *(* (a + i) + j)
```

It is possible to represent memory allocation, corresponding to the fragment mentioned above, in the following way:



It is possible to release memory here in the following way:

```
for(i = 0; i < m; i++) delete a[i];
delete a;
```

or in such a way:

```
for(i = 0; i < m; i++){
delete a[i]; a[i] = NULL;}
delete a;
```

ADVICE

1. Don't panic! All will become clear in time.
2. You don't have to know every detail of C++ to write good programs.
3. Focus on programming techniques, not on language features.
4. Don't reinvent the wheel; use libraries.
5. Don't believe in magic; understand what your libraries do, how they do it, and at what cost they do it.
6. When you have a choice, prefer the standard library to other libraries.
7. Do not think that the standard library is ideal for everything.
8. Remember to *#include* the headers for the facilities you use.
9. Remember that standard library facilities are defined in namespace *std*.
10. Use *string* rather than *char**.
11. Keep scopes small.
12. Don't use the same name in both a scope and an enclosing scope.
13. Declare one name (only) per declaration.
14. Keep common and local names short, and keep uncommon and nonlocal names longer.
15. Avoid similar looking names.
16. Maintain a consistent naming style.
17. Choose names carefully to reflect meaning rather than implementation.
18. Use a *typedef* to define a meaningful name for a built-in type in cases in which the built-in type used to represent a value might change.
19. Use *typedefs* to define synonyms for types; use enumerations and classes to define new types.
20. Remember that every declaration must specify a type (there is no "implicit *int*").
21. Avoid unnecessary assumptions about the numeric value of characters.
22. Avoid unnecessary assumptions about the size of integers.
23. Avoid unnecessary assumptions about the range of floating-point types.

24. Prefer a plain *int* over a *short int* or a *long int*.
25. Prefer a *double* over a *float* or a *long double*.
26. Prefer plain *char* over *signed char* and *unsigned char*.
27. Avoid making unnecessary assumptions about the sizes of objects.
28. Avoid unsigned arithmetic.
29. View *signed* to *unsigned* and *unsigned* to *signed* conversions with suspicion.
30. View floating-point to integer conversions with suspicion.
31. View conversions to a smaller type, such as *int* to *char*, with suspicion.
32. Avoid nontrivial pointer arithmetic.
33. Take care not to write beyond the bounds of an array.
34. Use 0 rather than NULL.
35. Use *vector* and *valarray* rather than built-in (C-style) arrays.
36. Use string rather than zero-terminated arrays of char.
37. Minimize use of plain reference arguments.
38. Avoid *void** except in low-level code.
39. Avoid nontrivial literals (“magic numbers”) in code. Instead, define and use symbolic constants.
40. Prefer the standard library to other libraries and to “handcrafted code”.
41. Avoid complicated expressions.
42. If in doubt about operator precedence, parenthesize.
43. Avoid explicit type conversion (casts).
44. When explicit type conversion is necessary, prefer the more specific cast operators to the C-style cast.
45. Avoid expressions with undefined order of evaluation.
46. Avoid *goto*.
47. Avoid *do* statements.
48. Don’t declare a variable until you have a value to initialize it with.
49. Keep comments crisp.
50. Maintain a consistent indentation style.
51. Prefer defining a member operator *new* () to replacing the global operator *new* ().
52. When reading input, always consider ill-formed input.

EXERCISES

1. Get the ‘‘Hello, world!’’ program to run.
2. For each declaration in Chapter 6, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.
3. Write a program that prints the sizes of the fundamental types, and a few pointer types. Use the *sizeof* operator.
4. Write a program that prints out the letters ‘a’...‘z’ and the digits ‘0’...‘9’ and their integer values. Do the same for other printable characters. Do the same again but use hexadecimal notation.
5. Develop the program for computing of arithmetic expression and the output of obtained result. Enter the source data from a keyboard.

$$a = \ln y^{-\sqrt{|x|}} (\sin x + e^{x+y})$$

Source data: x, y .

6. Develop the program for computing of the expression and for outputting of the obtained result. Input the corresponding source data from the keyboard.

$$a = \begin{cases} (x+y)^2 - \sqrt{xy}, & \text{for } xy > 0, \\ (x+y)^2 + \sqrt{|xy|}, & \text{for } xy < 0, \\ (x+y)^2 + 1, & \text{for } xy = 0. \end{cases}$$

Source data: x, y .

7. Calculate and print the table of three functions y, z, w . Argument x varies from x_0 to x_k with step h . Function y is defined by the convergent series, which sum has to be calculated until its next member modulo becomes less than specified minor positive e . Consider the task with several different e ($e = 0.01, 0,0001, 0,000001$).

The table should be represented as follows:

x	y	z	w
—	—	—	—
—	—	—	—
...
—	—	—	—

Adjust the data in the table with the help of functions `cout.width()`, `cout.precision()`.

$$y = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots;$$

$$z = \arctan x;$$

$$w = y - z;$$

$$x_0 = -0.5, \quad x_k = 0.5, \quad h = 0.1.$$

8. What, on your system, are the largest and the smallest values of the following types: *char*, *short*, *int*, *long*, *float*, *double*, *long double*, and *unsigned*.
9. What is the longest local name you can use in a C++ program on your system? What is the longest external name you can use in a C++ program on your system? Are there any restrictions on the characters you can use in a name?
10. Write declarations for the following: a pointer to a character, an array of 10 integers, a reference to an array of 10 integers, a pointer to an array of character strings, a pointer to a pointer to a character, a constant integer, a pointer to a constant integer, and a constant pointer to an integer. Initialize each one.
11. What, on your system, are the restrictions on the pointer types *char**, *int**, and *void**? For example, may an *int** have an odd value?
12. Use *typedef* to define the types *unsigned char*, *const unsigned char*, pointer to integer, pointer to pointer to *char*, pointer to arrays of *char*, array of 7 pointers to *int*, pointer to an array of 7 pointers to *int*, and array of 8 arrays of 7 pointers to *int*.
13. What is the size of the array *str* in the following example:

char str[] = "a short string"?

What is the length of the string "*a short string*" ?

14. Read a sequence of words from input. Use *Quit* as a word that terminates the input. Print the words in the order they were entered. Don't print a word twice. Modify the program to sort the words before printing them.

15. Run some tests to see if your compiler really generates equivalent code for iteration using pointers and iteration using indexing. If different degrees of optimization can be requested, see if and how that affects the quality of the generated code.

16. Rewrite the following *for* statement as an equivalent *while* statement:

```
for(i =0; i<max_length; i++) if (input_line [i] == '?') quest_count++.
```

Rewrite it to use a pointer as the controlled variable, that is, so that the test is of the form `*p == '?'`.

17. Fully parenthesize the following expressions:

```
a = b + c * d << 2 & 8
a & 0 7 7 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = -1 ++ b -- -5
a = b == c++
a = b = c = 0
a [4][2] *= * b ? c : * d * 2
a b,c =d
```

18. Read a sequence of possibly white space separated (name, value) pairs, where the name is a single white-space-separated word and the value is an integer or a floating-point value. Compute and print the sum and mean for each name and the sum and mean for all names.

19. Write a table of values for the bitwise logical operations for all possible combinations of 0 and 1 operands.

20. What happens if you divide by zero on your system? What happens in case of overflow and underflow?

21. Fully parenthesize the following expressions:

```
*p++
* -- p
++a --
(int*)p -> m
*p . m
*a[i]
```

22. See how your compiler reacts to these errors:

```
void f(int a, int b)
{
  if(a = 3) // ...
  if(a & 077 == 0) // ...
  a := b+1;
}
```

Devise more simple errors and see how the compiler reacts.

23. Modify the program from 15 to also compute the median.

24. What does the following example do?

```
void send(int* to, int* from, int count )
// Duff's device. Helpful comment deliberately deleted.
{
  int n = (count +7 )/8 ;
  switch (count%8 ) {
  case 0: do { *to++ = *from++;
  case 7: *to++ = *from++;
  case 6: *to++ = *from++;
  case 5: *to++ = *from++;
  case 4: *to++ = *from++;
  case 3: *to++ = *from++;
  case 2: *to++ = *from++;
  case 1: *to++ = *from++;
  }while (n>0) ;
  }
}
```

Why would anyone write something like that?

UNIT 3

16. FUNCTIONS

16.1. FUNCTION DEFINITION AND CALL

C++ program consists of one or several functions. The functions split large goals into small subgoals. The name of one of the functions, which must be present necessarily in any program, **main**, is reserved. The function **main** should not necessarily be the first, although the program performance starts with it.

A function cannot be defined in another function.

There are three notions connected with the use of a function – function definition, declaration and call.

Function definition has the following representation:

type name(the_list_of_argument_descriptions){statements}

Here **name** is the name of the function; **type** is a type of the value, returned by the function; **statements** in curly braces {} are also called the body of the function. The arguments in the list of descriptions are called **formal arguments**.

For example, the function that finds and returns the maximal values from two integer values **a** and **b** can be determined as:

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

This definition indicates that the function with the name **max** has two arguments and returns an integer value. If the function really must return the value of a certain type, the statement **return** followed **expression** must be necessarily present in its body; the function operation stops by this statement, the control is passed to the function, calling a given function, and the value of the called function is the value of the **expression**.

```
int max(int a, int b){ return(a >=b)? a:b; }
void main( ){
int i = 2, j = 3;
int c = max( i, j );
cout<<" max= "<<c<<"\n";
c = max( i*i, j )*max( 5, i - j );
cout<<" max= "<<c<<"\n";}
```

In this program, the **max** function definition and three calls to it are given. When calling, the function name and the list of **actual parameters** in the round brackets are indicated.

If a function doesn't have formal arguments, it is defined in the following way:

```
double f(void){function body};
```

or, the same,

```
double f( ){function body};
```

It is possible to call this function in the following way:

```
a = b*f( ) + c;
```

The function may not return any value. In this case its definition is following:

```
void name(the list of argument descriptions){ statements }
```

The call of such function looks like:

```
name(list of actual arguments);
```

The performance of this function, which doesn't return any value, comes to an end with the statement return without the expression, following it. The performance of such function and return from it into the calling function occurs also in case the transition to the last closing bracket of this function occurs when performing the function body.

As an example, let's take the function, copying one string into another:

```
void copy(char *to, char *from){
while(*to++ = *from++);}
void main(){
char str1[ ]="string1";
char str2[ ]="string2";
copy(str2, str1);
cout<<str2<<"\n";
}
```

Let's note that a library function **strcpy** has different definition and its headline can be represented as:

```
char *strcpy(char *to, const char *from);
```

Its action is copying of the string **from** to the string **to** and, besides, it returns the pointer to the string **to**, that is, it has a statement **return to**.

16.2. ARGUMENT PASSING

Argument passing **by value**. In the examples above, a so called argument passing **by value** occurs. Such argument passing means that a *local* object, which is initialized by the value of an actual argument, is created in a called function for each formal argument. Consequently, during such passing, the value change of formal arguments of the function does not cause the change of the values of actual arguments corresponding to them.

Let's discuss, for example, the function, raising the integer **x** to the **n**-th power, where this way is used.

```
int power(int x, int n){
for(int p = 1; n > 0; n--) p *= x;
return p;}
```

Here the argument **n** is used as a temporary variable. Whatever happens to **n** inside the function **power**, it in no way influences the actual argument, which was initially passed to this function by a calling function:

```
void main(){
...
int n = 6, x = 3;
x = power(x, n);           // n - is not changed.
```

In case the function must change its parameters, it is possible to use the pointers. The pointers are also passed according to their values; a local variable – a pointer is created inside the function. But as this pointer is initialized by the address of the variable from the called program, this variable can be changed using this address.

As an example, let us consider the function, swapping its parameters:

```
void swap(int* x, int* y){
int t = *x;
*x = *y;
*y = t;
}
```

It is possible to call this function in the following way:

```
int a = 3, b = 7;
swap(&a, &b);           //Now a = 7, and b = 3.
```

The use of the arrays as parameters has some peculiarity. This peculiarity is that the array name is converted to the pointer to its first element, that is, the passing of the pointer takes place during passing of the array. Because of this reason, a called function cannot recognize, whether the passed pointer refers to the beginning of the array or to one single object.

```
int summa(int array[ ], int size){
int res = 0;
for(int i = 0; i < size; i++) res += array[i];
return res;
}
```

In the headline **int array []** can be replaced by **int *array**, and the expression in the function body **array[i]** can be replaced by ***(array+i)** or even ***array++**, as **array** is not the name of the array, and, consequently, is not a constant pointer. It is possible to call the function **summa** in the following way:

```
int mas[100];
for(int i = 0; i < 100; i++) mas[i] = 2*i + 1;
int j = summa(mas, 100);
```

Example. Calculation of the polynomial by its coefficients.
Let it is required to calculate polynomials

$$P_3(x) = 4x^3 + 2x^2 + 1,$$

$$P_5(x) = x^5 + x^4 + x^3 + x^2 + x + 7,$$

$$P_9(x) = x^9 + 2x^7 + 3x^6 + x^5 + x^2 + 2$$

at the point $x = 0.6$.

Considering the importance of calculation of polynomials, let's set up the function, implementing the calculation of the polynomial of the **n**-th power:

$$P_n(x) = C_0 + C_1x + C_2x^2 + \dots + C_nx^n$$

by its coefficients C_i . For effective calculation of the polynomial it is necessary to use a so-called Horner's method (which in fact was described by Newton 100 years before Horner). This method consists in rewriting a polynomial in the following form:

$$P_n(x) = (\dots((0*x+C_n)*x + C_{n-1})*x + \dots + C_1)*x + C_0$$

In such order of calculation for obtaining the value $P_n(x)$ only the **n** number of multiplication and the **n** number of additions are required.

The coefficients of the polynomials will be stored in the array **c**.

For polynomial calculation let's write a program, where Horner's method is implemented in the function **pol()**.

```
#include <iostream.h>
const n = 10;
double pol(int n, double c[ ], double x){
double p=0;
for (int i = n; i >= 0; i - -) p = p*x + c[i];
return p;}
void main(){
```

```

double x=0.6, p, c[n] = {1, 0, 2, 4};
p=pol(3, c, x);
cout<<"x= " <<x<<" Polynom = "<<p<<"\n";
c[0]=7;
c[1]=c[2]=c[3]=c[4]=c[5]=1;
p=pol(5, c, x);
cout<<"x= " <<x<<" Polynom = "<<p<<"\n";
c[0]=2; c[2]=1; c[5]=1; c[6]=3; c[7]=2; c[9]=1;
c[1]=[3]=c[4]=c[8]=0;
cout<<"x= " <<x<<" Polynom = "<<pol(9, c, x)<<"\n";
}

```

16.3. MULTIDIMENSIONAL ARRAYS PASSING

If a two-dimensional array is passed to the function, the description of the corresponding function parameter must contain the number of columns; the number of rows is not essential, as in fact the pointer is passed.

Let's consider the example of the function multiplying the matrices **A** and **B**; the result is **C**. The dimension of the matrices is not greater than 10.

```

const nmax = 10;
void product(int a[ ][nmax], int b[ ][nmax], int c[ ][nmax],
int m, int n, int k){
    /* m – the number of rows in matrix a;
    n – the number of rows in matrix b (must be equal to the number
    of the columns in matrix a);
    k – the number of columns in matrix b.
    */
    for(int i = 0; i < m; i++)
        for(int j = 0; j < k; j++){
            c[i][j] = 0;
            for(int l = 0; l < n; l++) c[i][j] += a[i][l]*b[l][j];
        }
}

```

If, for example, square matrices **a** and **b** with the size 5x5 are set, their product **c** can be obtained in the following way:

```
product(a, b, c, 5, 5, 5);
```

In a given example there is one imperfection – here the maximal dimensionality of matrices is fixed beforehand. There are several ways to avoid it; one of them is the use of the accessory array-pointers to the arrays.

Let's write the function, transposing a square matrix of the arbitrary dimension **n**.

```

void trans(int n, double *p[ ]){
double x;
for(int i = 0; i < n-1; i++){
    for(int j = i+1; j < n; j++){
        x = p[i][j]; p[i][j] = p[j][i]; p[j][i] = x;
    }
}
void main(){
double A[4][4] = { 10, 12, 14, 17
                  15, 13, 11, 0
                  -3, 5.1, 6, 6
                  2, 8, 3, 1};
double ptr[ ] = {(double*)&A[0], (double*)&A[1],
                 (double*)&A[2], (double*)&A[3]};
int n = 4;
trans(n, ptr);
for(int i = 0; i < n; i++){cout<<"\n string"<<(i+1)<<":";
    for (int j; j<n; j++)
        cout<<"\t"<<A[i][j];
    cout<<"\n";
}
}

```

In the function **main()**, the matrix is represented as two-dimensional array **double A[4][4]**. Such array is impossible to use directly as an actual argument, corresponding to the formal **double *p[]**. In this case, an additional accessory array of the pointers **double *ptr[]** is introduced. The addresses of matrix rows, converted to the type **double*** are assigned to the elements of this array as initial values.

A multidimensional array with alternating dimensions can be dynamically formed inside the function. It is possible to pass it to the called function as a pointer to the multidimensional array of the pointers onto one-dimensional arrays with the elements of the known dimensionality and of a given type.

As an example, let's consider the function, forming a unitary matrix of the **n**-th order.

```

int** singl(int n){
int **p = new int *[n];
    /* Type int *[n] – the array pointers to integers.
    The operator new returns the pointer to the allocated memory for this
    array and the type of the variable p is int**. Thus, p is the array of the
    pointers to the rows of the integers of the future matrix.
    */
if(p == NULL){cout<<"A dynamic array is not created!\n";
exit(1);}
    // a loop for creation multidimensional arrays – matrix rows:
for(int i = 0; i < n; i++){p[i] = new int[n];
    if( !p[i] ){cout<<"A dynamic row is not created!\n";

```

```

exit(1);}
    for(int j = 0; j < n; j++)
        p[i][j] = (i == j )? 1: 0;}
return p;}

void main(){
int n;
cout<<"\n Specify the order of the matrix: ";
cin>>n;
int** matr;                //the pointer to the matrix
matr = singl(n);
for(int i = 0; i < n; i++){cout<<"\n row";
    cout.width(2);
    cout<<i+1<<" : ";
    for(int j = 0; j < n; j++){
        cout.width(4);
        cout<<matr[i][j];
    }
}
for(i=0; i<n; i++) delete matr[i];
delete matr;
}

```

In this program, the call of the function **cout.width(k)** specifies the width of the field of the next output into the **k**-th positions, that allows to line up the view of the obtained matrix.

16.4. THE POINTERS TO THE FUNCTIONS

Let's determine the pointer to the function in the following way:

function_type(*pointer_name)(list_of_the_parameters);

For example:

```
int (*fptr) (double);
```

Here **fptr** is determined as a pointer to the function with one argument of **double** type, which returns the value **int**. The name of the function without braces following it, is a pointer to the function, which contains the address of the beginning of this function code.

Example:

```

void f1(void){
cout<<"\n f1() is performed.";}
void f2(void){
cout<<"\n f2() is performed.";}
void main(){
void(*ptr) (void);

```

```

ptr = f2;
(*ptr) ();           //call of function f2();
ptr = f1;
(*ptr) ();           //call of f1();
ptr();               //another way!
}

```

The result:

<pre> f2() is performed. f1() is performed. f1() is performed. </pre>

Sometimes it is convenient to use formal parameters of the function, which are the pointers to the function.

Let's illustrate this when solving the following problem.

Calculate the integral of two different functions using a method of trapezoids.

```

// File TRAP.CPP
double trap(double a, double b, int n, double(*func)(double)) {
double x, h = (b-a)/n, i = ((*func)(a) + (*func)(b))/2;
for(x = a; n > 1; n - -) i += (*func)(( x+=h ));
return h*i;
}

//File INTEGRAL.CPP
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include "trap.cpp"
double f1(double x) {
return x*x + sin(3 + x);}
double f2(double x) {
return x/(x*x + 1) + exp(-2*x*x);}
void main( ) {
double a = 1, b = 3;
double i = trap(a, b, 50, f1);
cout <<"integral from the first function = " << i << "\n";
i = trap(a, b, 50, f2);
cout <<"integral from the second function = " << i << "\n";
}

```

It should be mentioned here that it is possible to use the calls **func(a)**, **func(b)**, etc. in the body of the function **trap**.

16.5. REFERENCES

The type “reference to the **type**” is defined as: **type&**, for example:

int& or **double&**

Reference types set the object aliases. A reference must be initialized. After initializing, the use of the reference yields the same result as a direct use of the renamed object.

Let’s consider the initializing of the reference:

```
int i = 0;
int& iref = i;
```

Here a new variable of a type reference to **int** with the name **iref** is created.

Physically **iref** is a constant **pointer** to **int** and, consequently, the meaning of the reference after initializing can be changed. In this case the initializing value is the address of the variable **i**, that is, during initializing the reference acts as the pointer.

When used, the reference doesn’t act as a pointer, but as a variable, which address it has been initialized by:

```
iref ++;           // the same as i++;
int *ip = &iref;  // the same as ip = &i.
```

So **iref** has become another name, an alias of the variable **i**.

The reference can be defined in such a way:

a reference is a constant pointer to the object, which the operator of pointer resolution * is implicitly applied to, when it is used.

If the type of the initialized reference doesn’t coincide with the object type, a new anonymous object, for which the reference is an alias, is created. The initializer is converted and its value is used for setting the anonymous object value.

```
double d = 0.0;
int& ir = d;           // an anonymous object of int-type is created;
ir = 3.0;             // d – is not changed!
```

Here the anonymous variable of **int** type, which is initialized by the value, obtained as a result of conversion of **double** type value to **int** type value, is created. Then the reference is initialized by the value of the address of this variable.

The anonymous object is also created, when the initializer is not an object, but, for example, a constant:

```
int& ir = 3;          // The anonymous object obtained the value 3.
```

Here the anonymous object of **int** type is created first, and it is initialized by the value 3. After it **ir** reference is created and is initialized by the address of the anonymous object. Now **ir** is its alias and the statement

```
ir = 8;
```

specifies a new value of this anonymous object.

16.6. REFERENCES AS FUNCTION PARAMETERS

References are often used as formal parameters of the function. A mechanism of parameter passing to the functions by means of references is called in programming as argument passing *by reference*. By means of references it is possible to achieve the value alteration of actual parameters at the calling program (without alteration of the pointers).

```
void swap(int &x, int &y){  
int t = x;  
x = y;  
y = t;  
}
```

Now the calling function has the following view:

```
int a = 3, b = 7;  
swap(a, b);
```

Thus, the local relatively the function **swap()** variables of the reference type are created. These local variables (**x** and **y**) are aliases of the variables **a** and **b** and are initialized by the variables **a**, **b**. After that, all the actions with **x** and **y** are equivalent to the actions with **a** and **b**, that causes the value alterations for **a** and **b**.

Let's note that in the last example it is possible to call the function **swap()** both with arguments of different types (not only **int**), and with arguments, which are not objects at all:

```
float a = 5, b = 2.7;  
swap(a, b);  
swap(3, a+b);
```

However, in these cases the function **swap()** in fact does not carry out any actions with its arguments. Temporary objects of **int** type, which are initialized by the values, obtained as a result of conversion of **a**, **b**, **a+b** to the **int** type, are created; then the references **x** and **y** are initialized by the values of the addresses of these anonymous objects; anonymous objects will be changed. At the same time actual parameters remain unchangeable.

A compiler yields a warning that it has to specify temporary variables and will work with them.

16.7. THE DEFAULT ARGUMENTS

A convenient property of C++ is presence of predefined initializers of the arguments. The values of the default arguments can be specified in the function declaration, at the same time they are automatically substituted to the function call, which contains less number of the arguments than it was declared. For example, the next function is declared with three arguments, two of which are initialized:

```
error(char *msg, int level = 0, int kill = 0);
```

This function can be called with one, two or three arguments:

```
error("Error!");           // Error("error", 0, 0) is called;
error("Error!", 1);        // error("error", 1, 0) is called;
error("Error!", 3, 1);     // default argument values
                           // are not used.
```

All the default arguments must be the last arguments in the list; not even a single argument can be to the right of it.

If the default argument has already been determined in one declaration, it cannot be overloaded in the other one. The default arguments must be declared during the first declaration of the function name and are not to be the constants:

```
int i = 8;
void func(int = i);
```

Let's note that if the argument initialization is performed in the function prototype, it is not necessary to set the initialization of the arguments in the function definition.

16.8. FUNCTION OVERLOADING

In C++, it is possible to overload the functions names and to use the same name for several functions with different type or number of the arguments. Let the following functions be declared:

```
int func(int, int);
int func(char, double);
int func(long, double);
int func(float, ...);           // The function with undefined
                                // number of the arguments.
int func(char*, int);
```

Let's consider what will happen to the name **func** with some list of the arguments during the function call. The first thing the compiler will perform is will try to find the function, which formal arguments correspond to the actual ones without any conversions except inevitable – for example, the array name to the pointer or the variable value to the constant or vice versa.

```
char string[ ] = "String - is a symbol array";
int i = func(string, 13);                // func(char*, int);
int j = func(1995L, 36.6);              // func(long, double);
```

If at the first stage a suitable function is not found, during the second stage an attempt to select such function is made, so that for adequacy of formal and actual arguments it will be enough to use only those standard conversions, which do not cause the conversions of the integer types to the floating and vice-versa. In this case a function, for which the number of such conversions would be minimal, is selected.

Let the function reference looks like this:

```
float a = 36.6;
j = func('a', a);
```

Applying indicated standard conversions, let's find that the function with prototype **func(char, double)** will be called and the argument **a** will be converted to the **double** type.

The third stage is selection of such function, for the call of which it is necessary to implement any standard conversions of the arguments (and again in such a way that these conversions would be as fewer as possible).

Thus, in the statement

```
int l = func("YEAR:", 2002.3);
```

the function **func(char*, int)** will be called, which actual argument of **double** type will be converted to **int** with truncation of the fractional part of the number.

At the fourth stage the functions, for which the arguments can be obtained by means of all conversions, considered before, and the type conversions, defined by the programmer himself, are selected.

If in this case the only necessary function is not found, at the last fifth stage a compiler tries to find a correspondence, taking into account the list of the undefined arguments.

Thus, when calling

```
func(1, 2, 3);
```

the only one function **func(float, ...)** is suitable here.

When calling

```
int i, j, n;  
...  
n = func(&i, &j);
```

a compiler will not find any suitable function and will yield the error message.

16.9. FUNCTION TEMPLATES

The aim of introduction of the function templates is automation of function creation, which can process heterogeneous data. In definition of the templates of the collection of functions an accessory word **template** is used, which is followed by the list of template parameters in angle brackets. Each formal template parameter is marked by the accessory word **class**, which is followed by the parameter name.

Example: definition of the function template, calculating the value of modules of different types.

```
template <class type>  
type abs(type x){return x > 0 ? x: -x;}
```

A function template consists of two parts – the template headline and ordinary function definition, where the type of the returned value and the types of any parameters and local variables can be defined by the names of template parameters, introduced in its headline.

Example (again function **swap**):

```
template <class T>  
void swap(T& x, T& y){T z = x; x = y; y = z;}
```

A template of the collection of functions is used for automatic formation of specific function definitions using the same calls, which a translator finds in the program text. For example, when addressing

```
abs(-10.3)
```

a compiler will form the following function definition:

```
double abs(double x){return x > 0? x: -x;}
```

The performance of exactly this function will be further organized, and the value 10.3 will return to the point of call as a result. Example: a function template for searching in the array.

```
#include <iostream.h>  
template <class type>  
type &r_max( int n, type d[ ] ){  
int im = 0;  
for(int i = 1; i < n; i++) im = d[im] > d[i] ? im : i;
```

```

return d[im]; }
void main(){
int n = 4, x[ ]={10, 20, 30, 5};
cout<<"\n r_max(n, x)="<< r_max(n, x); // Printing of the maximal
//element.

r_max(n, x) = 0; // Replacement the maximal
// element with zero.

for(int i = 0; i < n; i++)
cout<<"\t x["<<i<<"]="<< x[i];
float f[]={10.3, 50.7, 12.6};
cout<<"\n r_max(3, f)="<< r_max(3, f);
r_max(3, f) = 0;
for(i = 0; i < 3; i++)cout<<"\t f["<<i<<"]="<<f[i]; }

```

The result of the program performance

```

r_max(n, x)=30 x[0]=10 x[1]=20 x[2]=0 x[3]=5
r_max(3, f)=50.7 f[0]=10.3 f[1]=0 f[2]=12.6

```

When using templates, there is no necessity to prepare all the variants of functions with an overloaded name. A compiler, analyzing the function calls in the text of the program, forms necessary definitions automatically namely for such types of parameters, which are used in the call.

Let's enumerate the main properties of the template parameters.

- The names of template parameters must be unique in the whole template definition.
- The list of the parameters of the function template can not be empty.
- There may be several parameters in the list of parameters of function template. Each of them must be started with the accessory word **class**.
- It is not acceptably to use the parameters with the same names in headline of the template.
- The name of the template parameter has all rights of the type name in the function, defined by the template. The parameter name of the template is seen in the whole definition and hides other uses of the same identifier in the scope, exterior relatively a given template.
- All template parameters must be necessarily used in specifications of formal parameters of function definition.

Let's note that if necessary, it is possible to use prototypes of function template. For example, a function prototype swap():

```

template <class type>
void swap(type&, type&);

```

During specialization of template definition of the function it is necessary that when calling a function, the types of actual parameters, corresponding to formal parameters, which are parameterized in a similar way, are identical.

Thus, it is inadmissible that:

```
int n = 5;  
double d = 4.3;  
swap(n, d);
```

ADVICE

1. Be suspicious of non-const reference arguments; if you want the function to modify its arguments, use pointers and value return instead.
2. Use const reference arguments when you need to minimize copying of arguments.
3. Use const extensively and consistently.
4. Avoid macros.
5. Avoid unspecified numbers of arguments.
6. Don't return pointers or references to local variables.
7. Use overloading when functions perform conceptually the same task on different types.
8. When overloading on integers, provide functions to eliminate common ambiguities.
9. When considering the use of a pointer to function, consider whether a virtual function or a template would be a better alternative.
10. If you must use macros, use ugly names with lots of capital letters.

EXERCISES

1. Define functions `f(char)`, `g(char&)`, and `h(const char&)`. Call them with the arguments 'a', 49, 3300, c, uc, and sc, where c is a char, uc is an unsigned char, and sc is a signed char. Which calls are legal? Which calls cause the compiler to introduce a temporary variable?
2. Define an array of strings in which the strings contain the names of the months. Print those strings. Pass the array to a function that prints those strings.
3. Write a function that counts the number of occurrences of a pair of letters in a string and another that does the same in a zero-terminated array of char. For example, the pair "ab" appears twice in "xabaacbaxabb".
4. Write a function that swaps (exchanges the values of) two integers. Use `int*` as the argument type. Write another swap function using `int&` as the argument type.
5. Write a function `rev()` that takes a string argument and reverses the characters in it. That is, after `rev(p)` the last character of p will be the first, etc.
6. Write these functions: `strlen()`, which returns the length of a string; `strcpy()`, which copies a string into another; and `strcmp()`, which compares two strings. Consider what the argument types and return types ought to be. Then compare your functions with the standard library versions as declared in `<cstring>` (`<string.h>`).
7. Write a function `atoi(const char*)` that takes a string containing digits and returns the corresponding `int`. For example, `atoi("123")` is `123`. Modify `atoi()` to handle C++ octal and hexadecimal notation in addition to plain decimal numbers.
8. Write a function `itoa(int i, char b[])` that creates a string representation of `i` in `b` and returns `b`.
9. Write a program that strips comments out of a C++ program. That is, read from `c i n`, remove both `//` comments and `/* */` comments, and write the result to `c o u t`. Do not worry about making the layout of the output look nice (that would be another, and much harder, exercise). Do not worry about incorrect programs. Beware of `//`, `/*`, and `*/` in comments, strings, and character constants.
10. Look at some programs to get an idea of the variety of indentation, naming, and commenting styles actually used.

11. Write a program like “Hello, world!” that takes a *name* as a command-line argument and writes “Hello, *name*!”. Modify this program to take any number of names as arguments and to say hello to each.
12. Write a function to invert a two-dimensional array.
13. Write an encryption program that reads from *cin* and writes the encoded characters to *cout*. You might use this simple encryption scheme: the encrypted form of a character *c* is $c \wedge key[i]$, where *key* is a string passed as a command-line argument. The program uses the characters in *key* in a cyclic manner until all the input has been read. Reencrypting encoded text with the same key produces the original text. If no key (or a null string) is passed, then no encryption is done.
14. Look at some programs to get an idea of the diversity of styles of names actually used. How are uppercase letters used? How is the underscore used? When are short names such as *i* and *x* used?
15. Write a factorial function that does not use recursion.
16. Calculate integral

$$\int_1^{3.5} \frac{\ln x}{x\sqrt{1+\ln x}} dx$$

by the mid-rectangle method. Set up the function, implementing a computational method. Transfer a subintegral function as a parameter (a pointer to function). Provide the use of one parameter by default. Calculate the integral:

1. with the parameter by default
2. with different parameter
3. by Newton-Leibniz formula

17. Calculate integral

$$\int_1^4 \frac{\ln^2 x}{x} dx$$

by Simpson method. Set up the function, implementing a computational method. Transfer a subintegral function as a parameter (a pointer to function). Provide the use of one parameter by default. Calculate the integral:

1. with the parameter by default
2. with different parameter
3. by Newton-Leibniz formula

UNIT 4

17. CLASSES

17.1. DECLARATION OF CLASSES

A data type **class** can be defined by means of the following construction

a_key_of_a_class a_name_of_a_class{a_list_of_members};

Here the **a_key_of_a_class** is one of the keywords **struct**, **union**, **class**; **a_name_of_a_class** is an arbitrary identifier; **a_list_of_members** is definitions and descriptions of the members of the class, which are data and functions.

A class is a collection of one or more variables and functions, perhaps, of different types, grouped under one name.

The example of the structure is a registration card of the employee, where there is the surname, name, middle name, address, position, the year of arrival at work and so on. Some of these attributes can become structures themselves. Thus, S.N.M. has three components; the address has also several components.

A class may have a name, sometimes called a tag. A tag becomes a name of the new type in the program. Each member of the class is recognized by its name, which must be unique in a given class. The members of the class are sometimes called its elements and fields. Although a definite type is matched with each name of the class member, such member is not an independent object. The memory is allocated only for a definite object of the newly defined type as a whole.

Let's introduce new types **ID** and **staff**:

```
struct ID{char surname[39],  
  firstname[30],  
  midname[30]  
};  
struct staff{ID name;  
  char position[30];  
  int year;  
  float salary};
```

Here two new types of structural variables are set; and the names of such types are **ID**, **staff**. Let's note that the presence of ';' after braces is necessary here.

Now it is possible to declare structural variables of **ID** or **staff** type in an ordinary way:

```
ID name1, name2, name3;
staff s1, s2, s[50];
```

Now a compiler will allocate the memory for variables **name1**, **name2**, **name3**, **s1**, **s2** and for array **s** from fifty structures. Let's note that the number of bytes, allocated for a structural variable, is not always equals the sum of lengths of separate structure members because of the effect of alignment, made by a compiler. To define a specified number of bytes, it is necessary to use the operator **sizeof**, for example, like this:

```
int nf = sizeof(ID), ns = sizeof(staff);
```

Let's note that it is possible to declare structural variables simultaneously with the tag definition of the structure:

```
struct DATE{
int day;
int month;
int year;
char mon_name[4] } d1, d2, d3;
```

Here three variables **d1**, **d2**, **d3** are declared. They have the type of the **DATE** structure. It is possible to declare a structural variable without introduction of structure name (tag):

```
struct{int price;
float length[10] } a, b, c, d;
```

After definition of the structural variables, the access to its members is implemented by means of operator of extraction **'.'**:

```
a.price c.length, d1.day, d3.mon_name, s[25].salary,
s[0].name.surname.
```

The names similar to **c.length**, **d1.day**, **d3.mon_name**, by means of which the access to the class members occurs, are sometimes called as qualified names. When defining the pointer to the structure, **DATE* datep = &d1**, it is possible to refer to the structure member in the following way: **(*datep).year**, or by means of operator of extraction from the pointer to the structure **"->"** like this **datep->year**, which is equivalent.

Now let's introduce the simplest class "complex number":

```
struct compl{ double real, imag;
void define(double re = 0.0, double im = 0.0 ){
real = re; imag = im; } // the setting of a complex number.
void display( ){cout << "real = " << real <<
", imag = " << imag << '\n';}
};
```

Here **real**, **imag** are **data members** or components, or member variables and **define()**, **display()**, are **member functions** or component functions, which are often called as **methods** of the class.

Now it is possible to declare the objects of **compl** type:

```
compl a, b, c, *pc = &c;
```

After such definitions data members of structural variables are accessible in the domain of their visibility:

```
a.define(3, 7);           // A complex number 3+7i is defined,
                        // i.e. a.real == 3; a.imag == 7;
b.define(2);            // A complex number 2+0*i == 2 is defined;
c.define( );           // A complex number == 0;
                        // both parameters are selected by default.
```

Data members can be set and used directly, not through the functions **define()**, **display()**:

```
a.real = 3; a.imag = 7; (*pc).real = 1; pc->imag = -1;
a.real+ = b.real*3+7;
cout <<"pc->real : " <<pc->real<<"\n";
a.display( );
b.display( );
c.display( );
```

Here data members of the structure are accessible for the use in the program, passing member functions. It is possible to forbid arbitrary access to the data. In doing so, the word **class** is usually used instead of the word **struct** in the definition of the **class**:

```
class complex{ double real, imag;
public:
void display( ){cout <<" real =" <<real;
cout <<" , imag =" << imag <<"\n";
}
void define( double re = 0.0, double im = 0.0){
real = re; imag = im;
}
};
```

The label **public**, which can be present in the declaration of the class, in our example divides its body into two parts – **private** and **public**. The access to data members of the class, being in the private part, is possible only through functions-members of the class:

```
complex s1, s2, *ps = &s1;
s1.define( );           // s1.real=0; s1.imag=0;
```

```

s1.display( );           // real=0, imag=0 are typed;
ps->display( );         // the same.
s2.real = 3;           // Error! Private member s2.real is inaccessible!

```

The label **private** can also evidently be present in class definition.

The labels **private** and **public** generally divide the class body into parts, which are discriminated by the level of access. The access to the members of the class, being in the **private** part, is possible only by means of member functions and so-called **user-friendly** or **privileged** functions. Reference to public members of the class is possible from any function of the program.

The main difference of **struct** and **class** is in the level of access by default. If there is not evident indication of the access level, all the members of the **structure** are considered as public and all members of the **class** are private. An evident indication of the access levels makes the words **struct** and **class** interchangeable. Usually the use of word **struct** instead of the word **class** indicates that there is no need to limit the access level to data (it is supposed that all members of structure are public).

Let's note that types created by the programmer by means of the mechanism of classes, are often called an **abstract data types**.

17.2. CONSTRUCTORS

In a previous example initializing of the objects of **complex**-type was made by means of the member function **define()**. In that case the variable **s2** remained uninitialized. In C++ special member functions of class, which in most cases are called not by a programmer but a compiler and are intended for initializing of the objects of the abstract types, are provided. Such functions are called **constructors**. Let's consider the example:

```

class cl{
int num;
public:
void set(int i){ num = i; }
void show( ){ cout <<"Number: " << num <<"\n"; }
};

```

Before using the object of such type, it must be declared, initialized, and after these, it can be used:

```

void f( ){
cl obj;           // The object is created.
obj.set(10);     // The object is initialized.
obj.show( );    // The object can be used.
}

```

Now let's use a constructor for initializing. It is simply a special member function of **cl** class, which name is necessarily coincides with the name of the class:

```
class cl{int num;
public:
cl( int i ){ num = i ; }           // Constructor.
void show( ){ cout << "Number:" << num << '\n';}
};
```

Let's note that the type of the result is never indicated for a constructor! The function, using this class, have the view as below:

```
void f( ){
cl obj(10);           // The object is created and initialized!
obj.show( );        // Here the object obj is used!
}
```

Another full form of the declaration of the object of the abstract type, having a constructor, is possible:

cl obj = cl(10);

In this example, the constructor is a so-called inline function, as its definition is in the class body. However, it can be represented as an ordinary function, for which the constructor in the class is only declared, but it is defined outside the class body with the use of a qualified name:

```
class cl{ int num;
public:
cl( int i );
void show( ){cout <<"Number:" << num <<'\n';}
};
cl::cl( int i ){           // Full or qualified name.
num = i;}
}
```

It is often convenient to provide several ways of initializing, using a mechanism of function overloading.

Let's give an example of the program, where the display of the symbol string occurs.

```
# include< conio.h >
# include< stdlib.h >
# include< string.h >
class string{ char *str;
unsigned char attr;
int row, col;
public:
string( );
string(char *, unsigned char, int = 0, int = 0 );
}
```

```

void write( );
};
    // A constructor without arguments: all data of the object are defined –
    // string, video attribute of its symbols and a position for displaying.
string::string( ){
str = new char[ sizeof "Hello !" ];
strcpy( str, "Hello !" );
attr = BLUE << 4 +YELLOW;           // A yellow symbol is
                                     // against a blue background.

row=15;
col=36;
}
string::string( char *line, unsigned a, int y, int x){
str = new char[ strlen(line) +1];
strcpy(str, line);
attr = a;
row = y;
col = x;
}
void string::write(){
textattr( attr );           // A standard function of
                             // video attribute identification.

gotoxy( col, row );
cputs( str );
}
void main( ){
string string1;             // Is equivalent to string string1=string( );
                             // To write string string( ); is impossible, as
                             // this is a function prototype!
string string2("The second string!", BLACK<<4+WHITE);
string string3("The third string!", BROWN<<4+GRAY, 17, 19);
                             // A printing of the strings:

string1.write( );
string2.write( );
string3.write( );
}

```

In case of calling the first constructor without arguments, the initialization of any object will always occur in the absolutely same way, using the same values, which are rigidly defined in this constructor. (In a given case the object **string1** is initialized by the constructor without arguments and while calling the function **string.write()**, the print of the string “Hello!” of yellow color against the blue background in the 15th row, beginning with the 36th position, will happen).

The objects **string2** and **string3** are initialized by another constructor. The choice of the required constructor, as well as other overloaded functions, is performed according to the number and the type of the arguments.

Let's note, that in the class there can be only one constructor with the parameters by default.

17.3. DESTRUCTORS

A deletion of the objects of the abstract types has an important role along with the initialization of such objects, which is a reverse operator. In particular, constructors of many classes allocate memory for the objects dynamically, and after the necessity in such objects disappears, they should be deleted.

It is convenient to perform in a **destructor** – the function, which is called for the object of an abstract type, when it leaves the domain of existence. In the example, considered above, the place for string storage in memory is allocated dynamically, that is why it is useful to define a destructor. The name of the destructor, as well as of constructor's, cannot be arbitrary, it is formed by the symbol ~ and the class name (addition to the constructor):

```
class string{ . . .
public:
~ string( ){ delete str; }
. . .
};
```

Here the destructor is very simple. It can be more complex and is designed in the form of outline-function. A destructor can never have any arguments. Let's note that it is not possible to obtain the address either a constructor or a destructor. The call of the constructor occurs automatically during defining of the object of the abstract type, the call of the destructor occurs automatically when the object leaves the domain of its existence. The destructor can be called explicitly with a required indication of its full name.

Let's also note that for a class without explicitly defined constructor a compiler generates independently a so called default constructor, which does not have any arguments, with an access level **public**. The same can be referred to a destructor.

Note: the data of the class must not necessarily be defined or described before their first use in the functions, belonging to the class. The same is true for the functions, belonging to the class, i.e. to call a function from another one of the class is possible before its defining inside the class body. All components of the class are seen in the whole class.

17.4. STATIC MEMBERS OF THE CLASS

Member variable of the class can be declared with an accessory word **static**. Memory for such data is reserved during the program start, i.e. before a programmer creates the first object of a given abstract type evidently. In this case, all these objects use this single copy of their static member, which is created beforehand. A static member of the class must be initialized after class defining and up to the first description of the object of this class by means of so called full or qualified name of the static member, which has the following view:

name_of_the_class::name_of_a_static_member.

If a static member has an access level **public**, it can be used in the program by means of a qualified name, as usually.

Example: Let's write the class **object** in a static member of which there is a number of objects of **object** type, existing at each instant of time.

```
class object{
char *str;
public:
static int num_obj;
object( char *s){           // Constructor.
str = new char [strlen(s) + 1];
strcpy( str, s );
cout <<"is created " << str <<"\n"; num_obj ++ ;
}
~ object( ){ cout <<"is destructed " <<str << "\n";
delete str;
num_obj - -;
}
};

int object::num_obj = 0; // Initializing. A keyword int indicates it!
object s1("The first global object ",
s2("The second global object.");
void f( char *str ){
object s( str );
cout <<"There are objects in all - " <<
object::num_object<<"\n";
cout <<"The function f() has worked" <<"\n";}
void main( ){
cout <<"Meanwhile, the objects are - " <<object::num_obj <<
"\n";
object m("The object in main( )");
cout <<"And now the objects are - " << m.num_obj <<"\n";
f("A local object");
f("Another local object.");
```

```

cout <<"Before finishing main() the objects are - "
<<s1.num_obj<<".\n";
}

```

The results of program operation:

```

The first global object is created.
The second global object is created.
Meanwhile, the objects are - 2.
The object in main() is created.
And now the objects are - 3.
A local object is created.
There are objects at all - 4.
The function f() has worked.
A local object is destructed.
Another local object is created.
There are objects at all - 4.
The function f() has worked.
Another local object is destructed.
Before finishing main() the objects are - 3.
The object in main() is destructed.
The second global object is destructed.
The first global object is destructed.

```

We should pay attention to the fact that the constructors for global objects are called before the function **main()**, and the destructors – after **main()**.

Let's note that the classes, defined inside the function, cannot have static members.

17.5. THIS POINTER

Let's consider the example below:

```

class str{
char *string;
public:
void set( char *text){string = text;}
void write(){
cout<<"String: "<<string<<' \n' ;}
};
void main(){
str str1, str2;
str1.set("Hello!");
str2.set("Hello!");
str1.write();
str2.write();
}

```

As a result of performing of this program, the following will appear on the display:

```
String: Hello!  
String: Hello!
```

Let's ask ourselves the question: how does the member function **write()** recognize what object it is called for? The member function defines the object it is called for, because the address of this object is transferred to it as an implicit first argument. In a given case it is a pointer of **str*** type.

Inside the member function of the class, this pointer can be used explicitly. It always has the name **this** (a key word).

Before beginning of the performing of function code, the pointer **this** is initialized by the address of object, for which a given member function is called. Thus, the function definition **str::write()**, given above, represents the following reduced form of recording:

```
void write() {  
    cout <<"String:"<<this -> string<<' \n';  
}
```

Let's note that an explicit assignment of some value to the pointer **this** is forbidden.

17.6. STATIC MEMBER FUNCTIONS

Before declaration of member function of the class, it is possible to put an accessory word **static**. The peculiarity of such static member function is the following: as in the case of static member variable of the class, it is possible to call it before the first object of such class is created in the program. Static member functions (component functions) allow to get the access to the private static member variable of the class, without possessing an object of a given type in the program. For a static component function a pointer **this** is not defined. When it is necessary, the object address, for which a static member function is called, must be passed to it explicitly in the form of the argument.

Example:

```
class prim{  
    int numb;  
    static stat;  
public:  
    prim(int i){  
        numb = i;  
    }  
}
```

```
/*
```

Then a static function follows. The pointer **this** is not defined, the choice of the object is performed according to explicitly passed pointer. The member **stat** does not require a pointer to the object, as it is common for all the objects of the class **prim**.

```
*/
```

```
static void func(int i, prim *p = 0){  
if(p) p->numb = i;  
else stat = i;  
}  
static void show( ){
```

```
/* A static function calls only a static member of the class, no pointers are  
required: */
```

```
cout<<"stat="<<stat<<' \n' ;  
}
```

```
}; // The end of the class prim.  
int prim::stat = 8; // The initialization of the static  
// member of the class.
```

```
void main(){
```

```
/* Before the creation of object of prim type there is only one way to call  
the static member function: */
```

```
prim::show();
```

```
// It is possible to change the value of a static member of the class:
```

```
prim::func(10);
```

```
/* After creation of the object of prim type it is possible to call a static func-  
tion in an ordinary for abstract type way: */
```

```
prim obj(23); // obj.numb becomes equal to 23.  
obj.show();
```

```
// It is possible to change the value of the created object:  
prim::func(20, &obj); // obj.numb == 20.  
obj.func(27, &obj); // obj.numb == 27.  
}
```

17.7. THE POINTERS TO THE CLASS MEMBERS

For class members (except bit fields), the operation of address resolution is determined. The pointers to variable members of the class have no peculiarities. The peculiarity of the pointer to the member functions of the class is its explicit presence at the declaration of the class name, which is followed by `::`.

```
class cl{. . .
public:
int f(char*, int);
void g();
. . .
};
```

/* During declaration of the pointer to the component function, it is necessary to declare the types of the result and arguments of a function, for which a pointer is introduced, as with the pointer to the ordinary function. As usual, the pointer can be initialized during declaration: */

```
int(cl::*fp)(char *, int) = cl::f;
```

Example:

```
struct s{int mem;
s(int a){mem = a;}
void func(int a){cout<<a + mem<<' \n';}
};
void main(){
void(s::*fp)(int) = s::func;
s obj(5);
s *p = &obj; // Two variants of member function call using the pointer:
(obj.*fp)(6); // using the object obj of s type
(p->*fp)(9); // and the pointer p to it.
}
```

Here `.*` (and `->*`) are symbols of one single operator, but are not symbols, being side by side, of two earlier studied operators `.'` (`'->'`) and `*`. The right operand of operator `.*` and `->*` must be a **pointer to the class member**, but not any pointer.

17.8. INITIALIZING OF DATA MEMBERS OF THE CLASS

Initializing of the Members of the Abstract Types

Let the class contain the members of abstract types. The peculiarity of its initialization is that it is performed by means of the corresponding constructor. Let's consider the class

```

class coord{double x, y, z;
public:
coord(){x = y = z =0;}
coord(double xv, double yv, double zv=0){ x = xv; y = yv; z
= zv;}
coord(coord & c){x = c.x; y = c.y; z = c.z;}
};
class triang{
coord vert1, vert2, vert3; // The coordinates of vertices of triangle.
public:
triang();
triang(coord &v1, coord &v2, coord &v3);
};

```

During initialization of some object of **triang** class it will be necessary to call constructors three times for its vertices – objects of **coord** type. For it, in the constructor definition for the **triang** class, after colon it is necessary to put the list of the calls to the constructors of the **coord** class:

```

Traing::triang(coord &v1, coord &v2, coord &v3):
vert1(v1), vert2(v2), vert3(v3){. . .}

```

The call of the constructors of the **coord** class occurs before performing of the constructor body of the **triang** class. The order of their call is determined by the order of declaration appearance of the members of **coord** type during **triang** class definition.

The **coord** class contains the constructor without arguments. Instead of the recording

```

triang::triang(): vert1(), vert2(), vert3(){. . .}

```

during call to such constructor it is allowed to type simply the following:

```

triang::triang(){. . .}

```

Initializing of the Constants

If among data members of the class there are members, described with the modifier **const**, the same form of constructor is used during initializing, as in the case of data of the abstract types:

```

class cl{ int v;
const c;
public:
cl(int a, int b):c(b){v=a;}
};

```

The constant can be initialized only in the constructor; the attempt to do this by any other means (for example, by means of another component

function) will lead to the error message. The constant initializing in the constructor body is also inadmissible.

Let's note that the way of the constructor recording, compulsory for constants and the data of the abstract types, can be used also for ordinary members of the class:

```
class ro{ int var; const c;
public:
ro(int v, int u): c(u), var(v){}
};
```

17.9. THE COPY CONSTRUCTOR AND ASSIGNMENT OPERATOR

When working with the objects of the abstract types, a situation, when an object must be the copy of the other one, can arise. In this case, two variants are possible:

- 1) *a newly created* object must be the copy of the existing one;
- 2) *both objects were created beforehand* and it is necessary to copy one object into another.

In the first case a **copy constructor** is used, in the second one – **the assignment operator**.

A copy constructor is a constructor, which first argument is a reference to the object of that type, where this constructor is declared.

```
class cl{. . .
cl(cl&);           // A copy constructor.
. . .
};
cl ca;           // Here the constructor without arguments is used.
cl cb = ca;     // A copy constructor is used.
```

Initializing by copying occurs both during the arguments being passed to their functions and during the result return. If an argument or a returned value has an abstract type, the copy constructor is called indirectly, as it was in the example with classes **coord** and **triang**. A copy constructor is generated by a compiler independently, if it was not written by a programmer. In this case an exact copy of the initializing object is created, which is far from being required quite often.

Example 1:

```
class cl{int num; float val;
public:
cl(int i, float x){num=i; val=x;}
};
```



```

void main(){cl obj1(10, 20.3);
            // For creation of the objects obj2 and obj3
            // a default copy constructor is used:
cl obj2(obj1);
cl obj3 = obj2;
}

```

Example 2:

```

class prim{int n; float v;
public:
prim(int i, float x){n=i; v=x;}
prim(const prim &obj, int i = 0){
if(i) n=i;
else n=obj.n;
v=obj.v; }
};
void main(){
prim obj1(10, 23.5);

```

/ For creation of the objects obj2 and obj3 an explicitly described copy constructor is used: */*

```

prim obj2 = obj1;
prim obj3(obj1, 12);
}

```

Now let's make minimal changes, and a compiler will have to add its own constructor in addition to the available copy constructor:

```

class prim{int n; float v;
public:
prim(int i, float x){n=i; v=x;}
prim(const prim &obj, int i){ n = i; v = obj.v; }
};
void main(){
prim obj1(10, 23.5);

```

```

            // Now a default copy constructor will be used
prim obj2=obj1;
            // And now an explicitly defined constructor will be used,
            // so that only a part of the object is copied:
prim obj3(obj, 12);
}

```

Let's note that a modifier **const** is used for preventing of changing the copied object.

An object of one class can be initialized by an object of the other class. Here a constructor is not a copy constructor, as a reference to the object of different class appears as an argument:

```

struct s1{int i; float x;
s1(int j, float y){i = j; x = y;}
};
struct s2{int i; float x;
s2(const s1& a)          // This is not a copy constructor!
i =a.i; x=a.x;}
};
void main(){
s1 obj1(1, 3.7);
s2 obj2(obj1);
}

```

Unlike a copy constructor, the assignment operator is used when the objects, being the operands of this operator, do exist. Assignment operator, along with the operator of address resolving, is defined by default for the objects of abstract types, and it can be used without any additional actions of a programmer.

```

class cl{. . . };
void f(){
cl obj1; cl obj2 = obj1;      // A copy constructor is used.
cl obj3;
obj3 = obj1;                 // Assignment!
}

```

It is not always required just to create a copy when performing assignment. If something different is required, it is necessary to overload the assignment operator for a class.

17.10. FRIENDLY FUNCTIONS

There can be situations when it is advisable to have an access to the private data of the class, omitting member functions. The most widespread situation is when the member function of one class must have the access to the private members of different one.

Let's again consider the examples with classes **coord** and **triang**.

```

class coord{double x, y, z;
public:
coord();
coord(double, double, double = 0);
coord(coord & c);
};
class triang{coord vert1, vert2, vert3;
public:
triang();
triang(coord &v1, coord &v2, coord &v3);
};

```

Let it be necessary to add a member function to the class **triang**, calculating the coordinates of the triangle's centre. The language gives a chance for several functions, both usual and the member functions of some class **X**, to obtain the access to the private members of the class **Y**. Such functions are called a **privileged function in the class Y** and **friendly for the class X**. For declaration of the privileged function, an accessory word **friend** is used. To make this function privileged in the class **Y**, it must be declared in this class as a friendly function.

Let's write three member functions of the class **triang**, calculating the coordinate centre of the triangle in each axis direction:

```
double triang::midx(){ return(vert1.x+vert2.x+vert3.x)/3;}
```

and similarly

```
triang::midy(), triang::midz() .
```

In order to the compiler will not yield the error message, it is necessary to add the following declarations to the class **coord** declaration, in any of its part:

```
class coord{
...
friend triang::midx();
friend triang::midy();
friend triang::midz();
}
```

The case, when all the member function of one class are privileged in a different class, is widespread; even a simplified form of the recording is anticipated:

```
class coord{...
friend triang;
...
};
```

or

```
class coord{...
friend class triang;
...
};
```

In this case, it is said that the class **triang** is friendly for the class **coord**.

Let's note that for friendly functions, the pointer **this** is not defined, they do not have implicit arguments, the access levels for them are not defined. The same function can be declared privileged in several classes at once.

The difference in the way of the use of member functions and friendly functions is shown in the following example:

```
class cl{int numb;
        // f_func() is not private member of the class,
        // though it is declared in the private part.
friend void f_func(cl*, int);
public:
void m_func(int);
};
void f_func(cl* cpt, int i){
cpt->numb = i;    // An explicit pointer to the object is necessary,
                // as the pointer this is not defined!
}
void cl::m_func(int i){
numb = i;        // The same as if this->numb = i;
}
void main(){
cl obj;
f_func(&obj, 10);
obj.m_func(10);
                // Compare the methods of function calls and arguments!
...}
```

The next example demonstrates the possibility of access to the static private members of the class before creating of even one object of this class.

```
class cl{static int num;
public:
void set(int i){num = i;}
void m_show(){cout<<num<<' \n' ;}
friend void f_show(){cout << cl::num<<' \n' ;}
};
int cl::num = 8;
void main(){
cout <<"The objects of cl-type are absent.\n";
cout <<"Static member of the class = ";
        // Still it is possible to use only a friendly function:
f_show();
cl obj;
obj.set(200);
cout <<"An object of the type cl.\n is created";
cout <<"A static member of the class = ";
        // So now it is possible to use the member function.
obj.m_show();
}
```

17.11. CONSTRUCTOR AND OPERATOR *NEW*

When an abstract type has a constructor without arguments, the call of the operator **new** completely coincides with that what is used for memory allocation for ordinary data types without initializing expression:

```
class integer{int i;};
void main(){integer *ptr = new integer; . . .}
```

If a constructor of the class **integer** has the arguments, the list of the arguments is placed where the initializing expression is when working with standard data types:

```
class integer{int i;
public:
integer();
integer(int j): i(j){}
};
void main(){
int *ip = new int(10);
integer *iptr = new integer(30);
}
```

If in the operator **new** the calling for the constructor without arguments occurs, the following both recordings are possible:

```
integer *ip1 = new integer();
integer *ip2 = new integer;
```

When a constructor without arguments for **X** class is not defined, during the attempt to perform the operator

```
X *xp = new X;
```

a compiler yields the error message. In this case, it is required to determine the constructor without arguments explicitly.

18. INHERITANCE

18.1. CONSTRUCTION OF A DERIVED CLASS

Inheritance is used to write a specialized or enhanced version of another class.

Let's consider the class with a constructor and a destructor:

```
class Base{
int *bmember;
public:
Base(int arg = 0){bmrmbber = new int(arg);}
~Base(){delete bmember;}
};
```

Let's assume that it is necessary to change this class in such a way that the object of such type would contain not one but two pointers. Instead of the change of the class **Base** it is possible to construct a new class **Derived** on the basis **Base**:

```
class Derived: public Base{
int *dmember;
public:
Derived(int arg){
dmember = new int(arg); }
~Derived(){ delete dmember; }
};
```

The notation of the type **class Derived: public Base** indicates that the class **Derived** is a new-created class, which is constructed on the basis of the class **Base**. In this case, class **Derived inherits** all the properties of the class **Base**. **Derived** is said to be **derivative** from the class **Base**, and the class **Base** is a **basic** class for **Derived**.

If an object of **Derived** type is created in the program, it will contain two pointers to two domains of dynamic memory – **bmember**, as a sub-object of **Base** type and **dmember**. The creation process of the object of the type **Derived** will take two stages: first, a “sub-object” of the **Base** type will be created, and in this case the constructor of **Base** class will perform it. Then the constructor of the class **Derived** will be performed. The destructor call is performed in the reverse order. As the constructor of the class **Base** can

require the presence of one argument when calling it, this argument needs to be passed. To be passed to the constructor of the base class, the argument list must be located in the constructor definition of the derived class, just as with data initializing of the abstract type, being the members of some class:

```
Derived::Derived(int arg): Base(arg) {  
    dmember = new int(arg);  
}
```

If a constructor of the base class does not have arguments or use the arguments by default, it is not necessary to place an empty list into the constructor of the derived type.

18.2. PROTECTED CLASS MEMBERS

For controlling the access level to the members of the class, accessory words **public** and **private** are used. For this purpose, a key word **protected** is also introduced. If class **A** is not a base for any other class, its protected members do not differ from private ones – only member functions of the given class and the functions, which are friendly for this class, have the access to them. If class **B** is derived from class **A**, the users of the classes **A** and **B** still do not have the access to the protected members, but member functions of class **B** and the functions, which are privileged in **B** can have such access:

```
class Base{  
private:  
int privatem;  
protected:  
int protectedm;  
};  
class Derived: public Base{  
memberF() {  
    cout<<privatem;           // Error!  
    cout<<protectedm;       // Correct.  
}  
};  
void main() {  
Base b;  
cout<<.protectedm;         //Error!  
Derived d;  
cout<<.protectedm;       //Error.  
}
```

18.3. CONTROL OF THE ACCESS LEVEL TO THE MEMBERS OF THE CLASS

In the previous examples, the base class was public base class for the derived class:

```
class Derived: public Base{...};
```

It means that the level of the access to the members of the class **Base** from member functions of the class **Derived** and simply of the users of the class **Derived** remained unchanged: private members of the class **Base** are not available in the class **Derived** and public and protected members of the class **Base** remained public and protected in **Derived**. The base class will be private by default if it is not marked as public:

```
class Derived: Base{...};          // Base – is a private base class.
```

If a base class is a private base class, its private members are still inaccessible both in the derived class and for the user of the derived class, and protected and public members of the base class become private members of the derived class.

A base class cannot be protected base class. If a base class is private, it is possible to restore the access level of the base class for some of its members in the derived class. In this case their full name is given in the appropriate part of the class definition:

```
class Base{
private: int privm;
protected: int protm;
public: int pubm;
};
class Derived: Base{          // A private base class.
public:
Base::pubm;                  // Now pubm is a public member
                             // of the class Derived;
Base::protm;                 // error – access level is changed.
protected:
Base::protm;                 // Now protm is a protected member
                             // of the class Derived;
Base::pubm;                  // error – access level is changed.
```

Structures can be used like classes, but with one peculiarity. If a structure is a derived class, its base class is always public base class, i.e. the declaration of the type

```
struct B: A{...};
```

is equivalent to

```
class B: public A{...};
```


If a derived class is constructed on the basis of the structure, the same happens as in the case of using a typical class as a base one. Thus, when a base class and a derived class are structures, the following recording

```
struct B: A{...};
```

is equivalent to

```
class B: public A{public: ...};
```

18.4. A SEQUENCE OF CONSTRUCTOR AND DESTRUCTOR CALL DURING CONSTRUCTING OF THE DERIVED CLASS BASED ON ONE BASE CLASS

The object of the derived class may contain the objects of the abstract types as member data of the class:

```
class string{. . .
public:
string(char*);
~string();
...
};
class Base{...
public:
Base(int);
~Base();
. . .
};
class Derived: public Base{
Base b;
string s;
public:
Derived(char*, int);
~Derived();
. . .
};
```

Before call the constructor itself of the class **Derived**, it is necessary, first, to create a sub-object of the **Base** type, second – to create members **b** and **s**. Since, it is necessary to call the constructors of the appropriate classes for creation of the above mentioned objects and we have to pass necessary lists of the arguments to all of them:

```
Derived::Derived(char *st, int len): Base(len), b(len+1),
s(str){...}
```

In this case, during creation of the object of **Derived** type, at first, a sub-object of **Base** type will be created. At the same time the constructor **Base::Base()** with the argument **len** will be called. Then the objects **b** and **s**

will be created in the order, in which they are indicated in the definition of the class **Derived**. After it the constructor **Derived::Derived()** will be performed. The destructors will be called in the reverse order.

18.5. TYPE CONVERSION

The object of the derived type can be considered as an object of its base type. The opposite is not true (a cat is a mammal, but not any mammal is a cat). A compiler can perform conversion of the object of the derived type to the object of the base type implicitly:

```
class Base{...};
class Der: public Base{...};
Der derived;
Base b = derived;
```

A reverse conversion – **Base** to **Der** – must be defined by the programmer:

```
Der tmp = b; // the error if for Der
             // the constructor Der(Base) is not defined.
```

The conversion of the pointers to types is used much more often than the conversion of the types themselves. There are two types of pointers conversion – explicit and implicit. An explicit conversion will be always performed, an implicit one – only in certain cases. If a base class is a public one, i. e. we deal with the relation of the following type:

```
class Base{...};
class Der: public Base{...};
```

the principles of conversion are very simple: the conversion of the pointer of **Der*** to the pointer of **Base*** type can be performed implicitly. A reverse conversion must necessarily be explicit. In other words, when addressing through the pointers, the object of the derived type can be considered as an object of the base type. The reverse statement is not correct:

```
Der derived;
Base *bp = &derived; // Implicit conversion.
Der *dp1 = bp; // Error.
Der *dp2 = (Der*) bp; // Explicit conversion; now it is correct.
```

The fact that a derived class can be considered in a certain way as its base class, influences the choice of necessary version of the overloaded function. The complexity occurs if it is necessary to perform implicit conversion of the types for the choice of one of the variants.

In this case the rules are the following.

If there is not exact conformance between the lists of formal and actual parameters, the conversions of the derived type to the base type have the highest priority among performed conversions. It can be referred to the type itself, and to the pointer to it. In case it is not possible, the compiler tries to perform other conversions (for example, standard pointer conversions).

Example:

```
class Base{...};
class Der: public Base{...};
func(Base*);
func(void*);
...
Der *dp;
float *fp;
func(dp); // Calling of func(Base*).
func(fp); // Calling of func(void*).
```

If several levels of derived classes are used in the program, the class of “the nearest” level is searched during performing of the implicit conversions of the pointer type:

```
class Base{...};
class A: public Base{...};
class B: public A{...};
func(Base*);
func(A*);
...
B *db;
func(db); // The call of function func(A*).
```

19. POLYMORPHISM

One of the shortest and the most significant definitions of polymorphism is the following: Polymorphism is a functional ability, allowing an old code to call a new one. This property allows widening and improving a software system, without affecting the existing code. Such approach is implemented by means of the mechanism of the virtual functions.

19.1. EARLY AND LATE BINDING

The mechanism of the virtual functions is used in cases it is necessary to place into the base class the function that must be performed differently in the derived classes. To be precise, not the same function from the base class must be performed differently, but a proper variant of this function is required in each derived class.

Let's suppose that it is necessary to write the member function **CalculatePay()**, which calculates monthly payments for object of the class **Employee**. It is easy if the salary is calculated using one method: it is possible to insert the type of the required object into the function call at once. The problems arise with the advent of other forms of payment. Let's presume that there is the class **Employee**, calculating the salary according to fixed salary. How to calculate the payment for a contractor? It is quite different method of calculation. The function has to be rewritten in the procedure approach, having included a new type of processing into it as in the former code there is not such processing. An object-oriented approach allows to perform different processing, due to polymorphism.

In such approach it is necessary to describe the base class **Employee** and then the classes, derived from it, must be created for all forms of payment. Each derived class will have its own actualizing of the method **CalculatePay()**.

Another example: a base class **figure** can describe the figure on the screen without specialization of its view, and the derived classes **triangle**, **ellipse**, etc. define its shape and size uniquely. If the function **void show()** is introduced into the base class for displaying, the performance of the function will be possible only for the objects of each of the derived classes, determining particular images. It is necessary to put its function **void show()** into each of the derived class for the imaging on the screen. The access to

the function **show()** of the derived class is possible by means of an explicit indication of its full name, for example:

```
triangle::show();
```

or with the use of the particular object:

```
triangle t;  
t.show();
```

However, in both cases the choice of necessary function is performed during writing the initial text of the program and is not altered after compiling. Such mode is called **early** or **static binding**.

Great flexibility, especially when using class libraries, prepared earlier, is provided by a so-called **late** or delayed, or **dynamic binding**, which is granted by the mechanism of virtual functions.

19.2. VIRTUAL FUNCTIONS

From the beginning, let's consider the behavior of non-virtual functions-members with similar names, signatures, and types of the returned values during inheritance.

```
struct base{  
void fun(int i){  
cout <<"base::i = " << i << '\n';}  
};  
struct der: public base{  
void fun(int i){  
cout << " der::i = " << i << '\n';}  
};  
void main(){  
base B, *dp = &B;  
der D, *dp = &D;  
base* pbd = &D;           // Implicit conversion from der* to base*.  
bp->fun(1);  
dp->fun(5);  
pbd->fun(8);  
}
```

The result:

```
base::i = 1  
der::i = 5  
base::i = 8
```

Here the pointer **pbd** has **base*** type, but its value is an address of the object **D** of the class **der**. When calling the member function through the pointer to the object, the choice of the function **depends only on the type of the pointer**,

but not its value, what is illustrated by the output **base::i = 8**. Having adjusted the pointer of the base class to the object of the derived class, **we will not be able** to call the function from the derived class by means of this pointer. Thus, it is not possible to obtain **late** or **dynamic** binding.

The dynamic binding is provided by the mechanism of the virtual functions. Any non-static function of the base class can be made virtual, if a specifier **virtual** is used in its declaration:

```
class base{
public:
int i;
virtual void print(){
cout << i << " inside base\n";}
};
class D: public base{
public:
void print(){
cout << i << " inside D\n";}
};
void main(){
base b;
base *pb = &b;
D f;
f.i = 1 +(b.i = 1);
pb->print();
pb = &f;           // Implicit conversion from D* to Base*.
pb->print();
}
```

The result:

```
1 is inside base
2 is inside D
```

Here a different version of function **print()** is performed in each case. The choice dynamically depends on the object type, **which the pointer points to**. An accessory word **virtual** means that function **print()** can have its versions for different derived classes. The pointer to the base class can point both to the object of the base class and to the object of the derived class. A chosen member function depends on the class, which object is pointed to, but not on the type of the pointer. If the member of the derived type is absent, the virtual function of the base class is used by default. Let's note the differences between the choices of the corresponding overloaded virtual function and overloaded member function (not virtual): the overloaded member function is chosen during the compiling by algorithm, based on

the rule of the signatures. During overloading, the member function can have different types of the returned value. If the function is declared as virtual, all its overloading in the derived classes must have the same signature and the same type of the returned value. At the same time, it is possible not to indicate the word **virtual** in the derived classes.

In the derived class it is not allowed to define the function with the same name and the same signature, but with the different type of the returned value than that of the base class. Let's note that the constructor can not be virtual, but the destructor can be.

Let's consider the example – calculation of the areas of different figures. Different figures will be derived from the base class **figure**.

```
class figure{
protected:
double x, y;
virtual double area(){
return 0;} // The area by default.
};
class rectangle: public figure{
private:
double height, width;
. . .
public:
rectangle(double h, double w){height=h; width=w;}
double area(){return height * width;}
. . .
};
class circle: public figure{
private:
double radius;
. . .
public:
circle(double r){radius=r;}
double area(){
return M_PI*radius*radius;}
. . .
};
```

The user's code may be represented as follow:

```
const N = 30;
figure *p[N];
double tot_area = 0;
. . . // Here the pointers p[i] are specified, for example,
. . . // rectangle r(3, 5); p[0]=&r; circle c(8); p[1]=&c; etc.
for(i = 0; i < N; i++) tot_area += p[i]->area(); // User's code.
```

The main advantage is that the user's code does not need alteration, even if new figures are added to the system of the existing figures.

Let's consider one more example for calculation of salaries with the class **Employee**.

```
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
#include <string.h>
class Employee{
protected:
char * firstName, * lastName;      // Name, surname.
int age;                          // Age.
double payRate;                   // The amount of payment.
public:
Employee(char* FN, char* LN, int a, double pay){
firstName = new char[strlen(FN) + 1];
strcpy(firstName, FN);
lastName = new char[strlen(LN) + 1];
strcpy(lastName, LN);
age = a;
payRate = pay;
}
virtual double CalculatePay(){
return 0;
}
void print(){cout<<firstName<<" "<<lastName
<<" this month has received ";}
virtual ~Employee(){};
};

class HourlyEmployee: public Employee{ // Hourly pay.
int hours;                          // the number of worked off hours.
public:
HourlyEmployee(char* FN, char* LN, int a, double pay, int
h):
Employee(FN, LN, a, pay){ hours=h; }
virtual ~HourlyEmployee(){delete firstName; delete
firstName;}
virtual double CalculatePay(){
return hours*payRate;
}
};

class ContractorEmployee: public Employee{ // The work against
// the contract.
public:
ContractorEmployee(char* FN, char* LN, int a, double pay):
```



```

Employee(FN, LN, a, pay){}
virtual double CalculatePay(){
return payRate;
}
virtual ~ContractorEmployee(){delete firstName; delete
firstName;}
};

class DaypaymentEmployee: public Employee{ // Day-work payment.
int days; // the number of worked off days.
public:
DaypaymentEmployee(char* FN, char* LN, int a, double pay,
int d):
Employee(FN, LN, a, pay){days=d;}
virtual double CalculatePay(){
return days*payRate/24.0; // There are 24 workdays in a month.
}
virtual ~DaypaymentEmployee(){delete firstName; delete
firstName;}
};

void loademployee(Employee* a[], int &n){
char FN[20], LN[20]; int age, arg; double pay;
char sel; // A selector, setting the type of the payment.
ifstream file("emp.dat"); // Let's create an input stream for
// reading the file and link it with
// the external file emp.dat.

n = 0;
while(file.peek( ) != EOF){ // Till the end of the file ...
file >> sel;
file >> FN;
file >> LN;
file >> age;
file >> pay;
file >> arg;
switch(sel){
case 'h': a[n] = new HourlyEmployee (FN, LN, age, pay, arg);
break;
case 'c': a[n] = new ContractorEmployee(FN, LN, age, pay);
break;
case 'd': a[n] = new DaypaymentEmployee(FN, LN, age, pay,
arg);
break;
}
n++;
}
}
void main(){

```

```

int n;
Employee* a[20];
clrscr();
loademployee(a, n);
double s=0, r;
for(int i=0; i<n; i++){
s+=(r=a[i]->CalculatePay());
a[i]->print();
cout.width(16); cout << r << "$\n";
}
cout<<"This month it is payd: ";
cout.width(16); cout << s << "$\n";
}

```

Let the input file **emp.dat** contains the following information:

c Dudin	Ivan	32	4340	0
c Muhin	Sergey	26	1320	0
h Mazin	Petr	27	15.3	32
d Bobrov	Mikhail	40	110	21

Then as a result of the program operation the following will be displayed:

Dudin Ivan this month has received	4340\$
Muhin Sergey this month has received	1320\$
Mazin Petr this month has received	489.6\$
Bobrov Mikhail this month has received	96.25\$
This month it is paid:	6245.85\$

19.3. ABSTRACT CLASSES

Let's again consider the example with calculation of the figure areas. In this program a virtual function **area()** is used. This function must be defined for the first time in the class **figure**. Since the objects of the type **figure** must not exist during normal operation, except the objects of the derived types, the version **area()** was defined as following:

```
figure::area{return 0;}
```

If the type of the returned value of the function were **void** (for example, during drawing of the figure **void show()**), it would be possible to write:

```
void figure::show(){}
```

In both cases these functions are dummy. The virtual functions of such type could be used for control of the error, connected with the creation of the objects of **figure** type:

```

void figure::area(){
cout <<"Error: the attempt to calculate the area ";
cout <<"of non-existing object!\n";
exit(1); return 1;
}

```

In C++ there is more convenient and reliable method. The version of the virtual function, which, on the one hand, must be defined, and on the other hand, must not be used, can be declared as a **pure virtual function**:

```

class figure{. . .
virtual double area() = 0;
};

```

If in the classes, derived from **figure**, there is its own version of the virtual function **area()**, it should be either defined or, in its turn, declared as a purely virtual function. During program performance an error message is yielded when addressing a purely virtual function and the program is aborted. A class containing even one purely virtual function, is called an **abstract class**. The creation of the objects of such class is forbidden. It allows to set the control of compiler over the false creation of the objects of the dummy types, similar to **figure**. Let's note that it is possible to create the pointers to the abstract classes.

20. ENUMERATIONS

Enumeration is a type of data, which is convenient to use in case of application of variables and constants, possessing the value from a comparatively small quantity of integer numbers, exactly such numbers that it is more reasonable to address them by the name. Perhaps their value itself is not important. The example of such quantity of constants can be the names of the colors, days of week and months, chess pieces or symbolic names of arithmetic operators of C++.

The definition of the **enumeration** type starts with the key word **enum**, after which the **name** of the type (sometimes it is called as tag) follows, which is followed by **the list of members of enumeration** – enumerators, in curly braces:

```
enum chess{king, queen, rook, bishop, knight, p};
enum month{Jan, Feb, Mar, Apr, May, Jun,
Jul, Aug, Sep, Oct, Nov, Dec};
enum operator_CPP{plus = '+', minus = '-', mult = '*',
div = '/', rem = '%'};
```

The members of the enumerations are the constants of **unsigned char** or **int** types, depending on their values and the mode of compiling. When using enumerator in the expression, its type is always converted into **int**.

If no values are assigned to enumerators, as in **chess** and **month**, the first of them is 0, the second is 1, and so on. Generally, any enumerator by default has the value which is greater by 1 than the value of the preceding one, if default is not cancelled by the explicit initializing.

All members of enumeration **operator_CPP** receive values, which are evidently specified, equal to ASCII codes of symbols '+', '-', '*', '/', '%'.

The values of enumerators are calculated at the stage of compiling that is why when setting their values, it is possible to use the values of all earlier defined constants. In this case it is possible that several enumerators have the same value:

```
const TWO = 2;
enum{first, second = first, next = first + TWO,
last = next * next + 1}dummy;
```

Let's note that in the last case enumerator tag is not introduced, and a variable **dummy** is described at once, which can possess one of four values, according to the enumeration template:

```
first, second, next, last.
```

Consequently, the following assignments are possible:

```
dummy = first;
dummy = last;
```

In a general case it is possible to assign the values, set by the enumerators, to a variable-enumerator. Thus, if the descriptions

```
month m1, m2;
operator_CPP op1, op2, op3;
enum colour{ red, green} c[10];
```

are given, the values **plus, minus, ...** can be assigned to the variables **op1, op2, op3; Jan, Feb** – to **m1, m2**, etc., and the values **red** and **green** can be assigned to the array elements **c[0] ... c[9]**.

An enumerator can appear at every place where the appearance of the value of **int** type is allowed. The reverse is not right without explicit type conversion.

```
int i = dummy;                // i == 5
month current_month = Jan;
current_month = 0;           // Error! A warning will be yielded.
current_month = ( month )0;  // Now it's correct!
Feb = 1;                     // Error: Feb – is a constant!
```

Example:

```
# include < iostream.h >
const NUMDAYS = 7;
enum DAYS{mon, tue, wen, thur, fri, sat, sun} day1, day2,
        day3;
DAYS day_before(DAYS), day_after(DAYS);
void print_day(DAYS);

void main( ) {
day1 = fri; day2 = day_after(day1); day3 = day_before(day1);
cout << "If today is";
print_day(day1);
cout << ", tomorrow will be ";
print_day(day2);
cout << ",\n and yesterday it was ";
print_day(day3);
cout << ".\n";
}
```

```

DAYS day_after(DAYS day) {
return((DAYS)((day + 1) % NUMDAYS));
}

DAYS day_before(DAYS day) {
int prev = (day - 1) % NUMDAYS ;
return( prev < 0 ) ? ( NUMDAYS - 1) : prev ;
}

void print_day(DAYS day) {
int day_i = day;
static char* days[ ] = {"Monday",
"Tuesday" ,
"Wednesday",
"Thursday",
"Friday",
"Saturday",
"Sunday"
};
if(day_i < 0 || day_i > NUMDAYS)
cout << " Error! \n" ; else
cout << days[day_i] ;
}

```

The result of the program performance:

If today is Friday, tomorrow will be Saturday,
and yesterday it was Thursday.

ADVICE

1. Use public data (structs) only when it really is just data and no invariant is meaningful for the data members.
2. A concrete type is the simplest kind of class. Where applicable, prefer a concrete type over more complicated classes and over plain data structures.
3. Make a function a member only if it needs direct access to the representation of a class.
4. Use a namespace to make the association between a class and its helper functions explicit.
5. Make a member function that doesn't modify the value of its object a `const` member function.
6. Make a function that needs access to the representation of a class but needn't be called for a specific object a `static` member function.
7. Use a constructor to establish an invariant for a class.
8. If a constructor acquires a resource, its class needs a destructor to release the resource.
9. If a class has a pointer member, it needs copy operations (copy constructor and copy assignment).
10. If a class has a reference member, it probably needs copy operations (copy constructor and copy assignment).
11. If a class needs a copy operation or a destructor, it probably needs a constructor, a destructor, a copy assignment, and a copy constructor.
12. Check for self-assignment in copy assignments.
13. When writing a copy constructor, be careful to copy every element that needs to be copied (beware of default initializers).
14. When adding a new member to a class, always check to see if there are user-defined constructors that need to be updated to initialize the member.

15. Use enumerators when you need to define integer constants in class declarations.
16. Avoid order dependencies when constructing global and namespace objects.
17. Use first-time switches to minimize order dependencies.
18. Remember that temporary objects are destroyed at the end of the full expression in which they are created.
19. Define operators primarily to mimic conventional usage.
20. For large results, consider optimizing the return.
21. Prefer the default copy operations if appropriate for a class.
22. Overload or prohibit copying if the default is not appropriate for a type.
23. Prefer member functions over nonmembers for operations that need access to the representation.
24. Prefer nonmember functions over members for operations that do not need access to the representation.
25. Use nonmember functions for symmetric operators.
26. Use () for subscripting multidimensional arrays.
27. Make constructors that take a single “size argument” explicit.
28. For nonspecialized uses, prefer the standard string to the result of your own exercises.
29. Be cautious about introducing implicit conversions.
30. Use member functions to express operators that require an l-value as its lefthand operand

EXERCISES

1. Define a table of the names of months of the year and the number of days in each month. Write out that table. Do this twice; once using an array of char for the names and an array for the number of days and once using an array of structures, with each structure holding the name of a month and the number of days in it.
2. Define a *struct* Date to keep track of dates. Provide functions that read Dates from input, write Dates to output, and initialize a Date with a date.
3. Write functions to add one day, one month, and one year to a date in the *struct* Date from exercise 2. Write a function that gives the day of the week for a given date. Write a function that gives the date of the first Monday following a given date.
4. Solve the following problem, reading the data from the file and using array of structures. The entrant list is given, containing those who passed entrance exams for the university. There is an entrant surname, his/her permanent residence and the exam marks in separate subjects (for example, physics, mathematics, literature). It is necessary to determine the quantity of the entrants, living in a selected town and having passed the exams with the average mark equaling or over 4, print their surnames in alphabetical order.
5. Solve the following problem, reading the data from the file and using array of structures. The file contains the data about books, available in the reading hall of the library: author's name, the name of the publishing house, publication date, the number of pages. Develop the updating program for separate elements of the file and the information output on demand.
6. Solve the problem, reading the file data and using the array of structures. The manager of the railway booking office stores the information about vacant seats in the trains in all directions for the next week. This information is represented as follow: the departure date, destination, departure time, the number of the vacant compartment seats, and the number of vacant numbered reserved seats. An organizational committee of international conference makes a request to the manager for reservation of 50 compartment seats to Berlin on Saturday. At the same time, the departure time of the train should be at least 10 o'clock in the evening. Print the departure time or the information about impossibility to fill the order in corpore.

7. Solve the problem, reading the file data and using the array of structures. The file contains the following information:

Name	Account number	Value in account	Date of the last change
------	----------------	------------------	-------------------------

Calculate the number of clients, having a value in the account, which is greater than that input on demand, and having visited the bank this month (on the basis of the input date), print their surnames in alphabetical order.

8. Solve the problem, reading the file data and using the array of structures. Receipts about deadlined radio equipment are stored in the radio atelier. Each receipt contains the following information: description of the item in the group (TV set, radio set and etc.), the brand mark of the item, the date of reception, the ready state of the item (completed or unfilled orders). Develop the program of history data analysis and information delivery about the number and the order nature for the current day and about the amount of completed orders for the current quarter in item groups.
9. Solve the problem, reading the file data and using the array of structures. The file contains the information about vacant seats in the trains in all directions for the next week: the departure date, train number, destination, departure time, the number of the vacant compartment seats, the number of vacant numbered reserved seats. Develop the program of separate file element correction. Prepare the information issue about the available seats in all trips on demand.
10. Solve the problem, reading the file data and using the array of structures. The timetable of the aircraft departures for the next day is stored in the airport directory inquiries. The number, aircraft type, destination, departure time are specified for each flight. Determine all the flight numbers, aircraft types, their departure time for a given destination.

Flight №	Aircraft type	Destination	Departure time	Arrival time	Beginning of check-in	Beginning of boarding
----------	---------------	-------------	----------------	--------------	-----------------------	-----------------------

Prepare the information issue on demand in the first five columns. Provide the possibility of the correction of separate file elements on demand.

11. Solve the problem, reading the file data and using the array of structures. There is the list of registration of every needy in housing improvement. Every recording of this list contains a surname, first name, middle name and the date of registration. The list is organized according to the registration date. The number of the flat, allotted according to the given list during the year, is known. Calculate how many years are necessary in average to obtain a flat. Display the whole list with indication of the expected year for obtaining a flat.

12. Solve the problem, reading the file data and using the array of structures. The file contains the following information:

Number of a workshop	Number of a sector	Number of a shift	Number of a brigade	Name of a worker	Quantity of manufactured articles per day
----------------------	--------------------	-------------------	---------------------	------------------	---

Provide the data about the results of shift-work by one worker, at one sector, in one workshop on request. Provide the probability of correction of the separate elements on request.

13. Solve the problem, reading the file data and using the array of structures. The file contains the following information:

Telephone number	Date of telephone call	Duration of telephone call	Prefix
------------------	------------------------	----------------------------	--------

Design a program of separate file data correction and information input on request.

UNIT 5

21. OVERLOADING OF STANDARD OPERATORS

21.1. THE BASIC DEFINITIONS AND PROPERTIES

In C++ there is a possibility to expand a standard operator action to the operands of abstract data types.

In order to overload one of standard operators for working with the operands of the abstract types, a programmer must write the function with the name

operator a,

where **a** is a symbol of this operator (for example + - / += etc.)

In this case there are several limitations in the language:

- one must not create new symbols of the operators;
- one must not overload the operators

:: * (dereferencing, not binary multiplication) ? :

sizeof . .* # ##;

- a symbol of the unary operator can not be used for overloading of binary operator and vice versa. For example, a symbol << can be used only for binary operators, ! is used only for unary, and & – for unary and binary one;
- overloading of the operators does not change neither their priorities nor the order of their performance (left-to-right or right-to-left);
- during operator overloading a computer does not make any assumptions about its properties. It means that if a standard operator += can be expressed through the operators + and =, i.e. a += b is equivalent to a = a + b, in the general case there are not such relations for overloaded operator, although, of course, a programmer can provide them. Besides, no assumptions are made, for example, about the operator + commutativity: compiler has no reasons to consider that a + b, where a and b of abstract types is the same as b + a;
- no operator can be overloaded for the operands of the standard types.

The function **operator a()** is an ordinary function, which can contain from 0 to 2 explicit arguments. It may be and may be not a member function of the class.

```
class cl{ int i;
public:
int get(){return i;}
int operator +(int);           // A binary plus.
};
int operator +(cl&, float);    // A binary plus.
```

In the first form of the binary plus there is not one, but two arguments. The first one is implicit; any non-static member function of the class has it. This argument is a pointer to the object, for which a function is called. The implementation of both functions can be represented as follow:

```
int cl::operator +(int op2){
return i + op2;}
int operator +(cl &op, float op2){
return op.get() + op2;}
```

What will happen if the second argument in the global function **::operator +()** has **int** type, not **float** type? In this case a compiler will yield an error message, as it will not manage to make a choice between the functions **cl::operator +()** and **::operator +()** as these functions are both equally suitable.

For performing of overloaded unary operator **ax** (or **xa**), where **x** is an object of some abstract type **Class**, the compiler attempts to find either function **Class::operator a(void)**, or **::operator a(Class)**. If both variants are found simultaneously, the error is fixed. Interpretation of the expression is performed either as **x.operator a(void)**, or as **operator a(x)**.

For performing a overloaded binary operator **x a y**, where **x** is necessarily an object of the abstract type **Class**, the compiler searches for either the function

```
Class::operator a(type y),
```

or the function

```
::operator a(Class, type y),
```

where **type** can be both of standard and abstract type.

The expression **x a y** is interpreted either as

```
x.operator a(y),
```

or as

```
operator a(x, y).
```

The number of the arguments of the function **operator a()** must exactly correspond the number of the operands of this operator both for unary and for binary operator. Let's note that it is often convenient to pass the parameter values to the function **operator a()** by reference, not by the value.

Let's consider, for example, the operator of addition, defined for the class "a complex number":

```
class complex{
double re, im;
public:
double & real(){ return re; }
double & imag(){ return im; }
//. . .
};
complex operator +(complex a, complex b){
complex result;
result.real() = a.real() + b.real();
result.imag() = a.imag() + b.imag();
return result;
}
```

Here both arguments of the function **operator +()** are transferred by the value, that is the copying of four numbers of **double** type is performed. Such costs can be found too high, especially if the operator is overloaded for such, for example, class as "matrix". It would be attractive to avoid burden, passing the pointers to the objects, not the objects themselves:

```
complex operator +(complex* a, complex *b){. . .}
```

but here it is impossible to deal this way, as both arguments are now the objects of a standard type – pointers, and operator overloading for standard types is forbidden.

In this situation, it is necessary to use the references – they do not change the type of the operands, and only influence the mechanism of parameter transfer:

```
complex operator +(complex &a, complex &b){
complex result;
result.real() = a.real() + b.real();
result.imag() = a.imag() + b.imag();
return result;
}
```

The function body **operator +()** is not changed here.

Example: the operator + definition for the class **stroka**:

```
class stroka{
char *c; // The pointer to the string.
int len; // The length of the string.
```

```

public:
stroka(int N = 80): len(0)      // A string, which does not contain
                               // information;
{
    i = new char[N + 1];      // memory allocation for the array.
    c[0] = '\\0';
} // The constructor allocates the memory for the string and make it empty.
stroka(const char * arg){
    len = strlen(arg);
    c = new char[len + 1];
    strcpy(c, arg);
}
int & len_str()      // Returns the reference to the length of the string.
{return len;}
char * string( ) {    // Returns the pointer to string.
return c;}
void display(){      // The typing of the information about the string.
cout << "String length: " << len << ".\n";
cout << "The content of the string: " << c << ".\n";
}
~ stroka(){delete c;}
};
stroka & operator +(stroka &a, stroka &b){
int ii = a.len_str() + b.len_str();      // The length of the
                                           // resulting string.
stroka * ps = new stroka(ii);
strcpy(ps->string(), a.string()); // Copies the string from a;
strcat( ps->string(), b.string()); // Adds the string from b;
ps->len_str() = ii;      // records the value of the string length;
return *ps;      // returns a new object stroka.
}
void main(){
stroka X("Vasya");
stroka Y(" goes");
stroka Z;
Z = X + Y + " by bicycle";
Z.display();
}

```

The result of program performance:

```

String length: 22.
The content of the string: Vasya goes by bicycle.

```

Let's note that another form of calling for **operator +()** instead of **Z = X + Y + " by bicycle"** is possible:

```
Z = operator +(X, Y);  
Z = operator +(Z, " by bicycle");
```

21.2. THE OPERATORS *NEW* AND *DELETE* FOR THE ABSTRACT TYPES

The operators **new** and **delete** are realized through the functions, and independently of that, whether **operator new()** and **operator delete** are described or not as **static**, they are always static functions. The operator **new** is predetermined for any type, including abstract type defined by means of the mechanism of classes. It is possible to overload both the global function **operator new()** and the function **class x::operator new()**. Global **new** and **delete** are overloaded in the ordinary way by means of the mechanism of signature adequacy.

As in the case of overloading of the global function **operator new()**, a overloaded function **class x::operator new()** must return the result of **void** type, and its first argument must have **size_t** type (that is **unsigned**), where the size of the allocated memory is stored. Let's note that when using the operator **new**, this argument is not specified and the size of the necessary memory area is calculated automatically based on the specified type.

The Use of *new* when Creating a Dynamic Object of the Abstract Type

Let's consider the fragment:

```
class C{ . . .  
public:  
C(int arg ){ . . .}  
};  
. . .  
C * cp = new C(3);
```

The creation of the dynamic object of **C** type can be divided into two stages:

- 1) the object creation itself – is performed by the constructor;
- 2) the location of this object in a definite memory area – performed by the operator **new**.

Here, first, the function **operator new()** is performed, and then the constructor places a created object in the allocated memory.

The operator **new** can be overloaded in the following way:

```
class cl{ . . .
public:
cl(){cout << "Class constructor cl.\n";}
void* operator new(unsigned);
};
void* cl::operator new(unsigned size){
cout <<"The function operator new() of the class cl;\n";
void* p = new char[size ];          // Global new!
if(p) return p; else{
cout<<"There is not any memory for the object of the type
cl!;\n"; exit(1);}
}
void main(){
cl * cp = new cl;}
```

The result is:

```
The function operator new() of the class cl;
Class constructor cl.
```

The Operator *delete*

The performance of the operator **delete**, applied to the pointer to the object of the abstract type, leads to the destructor call for this object.

```
cl * clp = new cl(5);    // Constructor call cl(5);
. . .
delete clp;              // Destructor call ~cl() before release of
                        // the dynamic memory.
```

The function **x::operator delete()** can be overloaded in the class **x**, while it can have only two forms:

```
void operator delete(void * );
void operator delete(void *, size_t );
```

If the second form of the given operator is present, the compiler uses just it.

21.3. TYPE CONVERSION

The types conversions can be divided into 4 groups:

- 1) standard-to-standard;
- 2) standard-to-abstract;
- 3) abstract-to-standard;
- 4) abstract-to-abstract.

The first conversions have already been considered earlier. The conversions of the second group are based on implicit and explicit use of the constructors.

Again, let's consider the class **complex**:

```
class complex{
double re, im;
public: complex(double r=0, double i=0){ re = r; im = i ; }
. . .
};
```

The declarations of the type

```
complex c1;
complex c2(1.8);
complex c3(1.2, 3.7);
```

provide the creation of complex numbers.

But the constructor may be called implicitly in case when the operand of **complex** type must be present in the expression, where in fact the operand of **double** type is present:

```
complex operator +(complex & op, complex & op2 );
complex operator -(complex & op, complex & op2 );
complex operator *(complex & op, complex & op2 );
complex operator /(complex & op, complex & op2 );
complex operator -(complex & op ); // Unary minus.
complex res;
res = -(c1 + 2) * c2 / 3 + .5 * c3;
```

The interpretation, for example, of the expression $-(c1 + 2)$ will be the following:

```
operator -(operator +(c1, complex(double(2))))).
```

When performing this expression, implicit constructor calls will create temporary constants of **complex** type: (2.0, 0.0), (3.0, 0.0), (4.5, 0.0), which will be destroyed immediately after they become unnecessary. Let's note that here not only an implicit constructor **complex** call takes place, but also an implicit standard conversion of the value of **int** type to the **double** type occurs.

The number of the levels of implicit conversions is limited. In this case, the rules are the following: the compiler can perform not more than one implicit standard conversion, and not more than one implicit conversion defined by a programmer.

Example:

```
class A{ public:
A(double d){. . .}
};
```

```

class B{ public:
B(A va){. . .}
};
class C{ public:
C(B vb){ . . .}
};

A var1(1.2);           // A(double)
B var2(3.4);           // B(A(double))
B var3(var1);          // B(A)
C var4(var3);          // C(B)
C var5(var1);          // C(B(A))
C var6(5.6);           // Error! C(B(A(double))) is implicitly called
C var7(A(5.6));        // C(B(A))

```

The error during creation of the variable **var6** is caused by the necessity of two levels of implicit non-standard conversions, performed by means of the constructor call: **double** to **A**, and then **A** to **B**.

When creating the variable **var7**, one of these conversions – **double** to **A** – is made explicitly, and now everything will be all right.

Thus, the constructor with one argument **Class::Class(type)** always defines the conversion of **type** type to **Class** type, not only the method of object creation in case of explicit call to it.

For an abstract type conversion to a standard one or abstract to abstract one, there is a mean – function, performing the conversion of types, or the **type conversion operator**.

It has the following view:

```
Class::operator type(void);
```

This function performs a conversion of **Class** type to **type** type, defined by a programmer. This function must be a member of the class **Class** and should not have arguments. Besides, the type of the returned value is not indicated in its declaration. This function call can be implicit and explicit. For performing of the explicit conversion, it is possible to use both traditional and “functional” forms.

Example 1:

```

class X{ int a, b;
public:
X(X & vx){ a = vx.a; b = vx.b; }
X(int i, int j){ a = 2*i, b = 3*j; }
operator double(){ return(a + b)/2.0; }

// Conversion of the abstract type to standard one.
};

```

```

int i = 5;
double d1 = double(i); // An explicit conversion
                        // of int-type to double;
double d2 = i ;       // implicit conversion of int-type to double;
X xv(5, -8);
double d3 = double(xv); // explicit conversion of X-type to double;
double d4 = xv;        // implicit conversion of X-type to double.

```

Example 2. The conversion of the abstract type to the abstract one:

```

class Y{
char * str1; // Strings str1 и str2 store symbolic
char * str2; // representation of integer numbers.
public:
Y(char *s1, char *s2): str1(s1), str2(s2){}
operator X(){ return X(atoi(str1), atoi(str2));}
};
. . .
Y yvar("12", "-25");
X xvar = yvar;

```

When creating the variable **xvar**, an implicit conversion of the value of the variable **yvar** to the type **X** will be performed before the call of the copy constructor **X::X(X&)**. The same conversion in the explicit form can be represented as follow:

```

X xvar = X(yvar);
X xvar = (X) yvar;

```

For the expression

```

X xvar = X("12", "-25");

```

a compiler will yield the error message “a constructor with specified arguments is not found”. The thing is that in contrast to the constructor, the operator-function of type conversion is not able to create the object of an abstract type. It can perform only value conversion of the object of one type, having been created earlier, to the value of another type. In the last example the object of **Y** type still does not exist.

22. SOME PECULIARITIES OF OVERLOADED OPERATORS

The restrictions during overloading of the operators =, [], (), -> consist in that the functions **operator =()** etc., realizing them, must be the members of the class and can not be static functions. Speaking about the inheritance mechanism it should be noted that usually a derived class inherits all the properties of the base class. There are two exceptions to this rule:

- 1) a derived class can not inherit the constructors of its base class;
- 2) the assignment operator, overloaded for the base class, is not considered as overloaded for its derived classes.

All the other operators are inherited in the ordinary way, that is, if a necessary operator is not overloaded for the derived class, but it is overloaded in its base class, the operator of the base class will be called.

22.1. OPERATOR =

Assignment operator = is predefined for any abstract type of data. In this case such overloaded operator of assignment is interpreted not as obtaining of the bitwise copy of the object, but as a sequential assignment of its members (of standard and abstract types). Bitwise copying occurs in case the operator = is not defined. Overloaded operator = can be overloaded.

```
struct memberone{
int i;
memberone & operator =(memberone & a){
cout<<"The operator of copying of the class memberone\n\n";
return a;
}
};
struct membertwo{ int j;
membertwo & operator =(membertwo & a){
cout<<"The operator of copying of the class membertwo\n\n";
return a; }
};
struct contain{ int k;
memberone mo;
membertwo mt;
};
```

```

void main(){
contain from;
from.mo.i = 1;
from.mt.j = 2;
from.k = 3;
contain to;
to.mo.i = 0;
to.mt.j = 0;
to.k = 0;
to = from;
    cout << "to.mo.i = " << to.mo.i << "\n\n"
    << "to.mt.j = " << to.mt.j << "\n\n"
    << "to.k = " << to.k << "\n\n";
}

```

The results of the program operation:

```

The operation of copying the class memberone
The operation of copying the class membertwo
to.mo.i = 0
to.mt.j = 0
to.k = 3

```

Example 2. Let's consider the class **stroka** again:

```

class stroka{
char *c;
int len;
public:
. . .
stroka & operator =( stroka & str );
. . .
};
stroka & stroka::operator =(stroka & str){
if( str.len > len ){
cout << "The string length is minor! Copying is impossi-
ble!\n";}
else{ strcpy( c, str.c ); len = str.len;}
return *this;
}
void main( ){
stroka A("String A"), B("String "), C("Str");
A = B; A.display( );
B = C; B.display( );
C = A; C.display( );
}

```

As a result of this program performance, the following will be displayed on the screen:

```
String length: 6
String content: String
String length: 3
String content: Str
String length is minor! Copying is impossible!
String length: 3
String content: Str
```

21.2. OPERATOR []

The expression `x[y]`, where `x` is an object of the abstract type **Class**, is interpreted as

```
x.operator[ ](y).
```

Let's note that the array of objects of the abstract type **Class**, as well as any standard type, has a standard type of a pointer. Even if **array** is an array of the elements of the abstract type **Class**, the expression **array [i]** still means ***(array + i)**, independently of whether the operator `[]` for the type **Class** is overloaded or not.

Example:

```
class A{
int a[10];
public:
A(){ for( int i = 0; i < 10; i ++ ) a[i] = i + 1; }
int operator[ ]( int j ){
return a[j]; }
};
void main( ){
A array[20];
cout << "array[3][5] = " << array[3][5] << ".\n";
}
```

The result of the program operation will be: **array[3][5] = 6**.

It is evident that the operator `[]`, used in the constructor of the class **A**, is standard, as it is performed over the array name. Now let's consider the expression **array[3][5]**. The result of its calculation is the same as it was expected, due to the following reason: the operator `[]` is performed from left to right. Consequently the expression **array[3][5]** is interpreted as **(array[3]).operator[](5)**. The first of two operators `[]` is standard, as it is performed over the array name. In this case the type of array's elements is not important. The second operator `[]` is overloaded, as the result of the first operator `[]` is the object of **A** type.

A question arises when does it make sense to overload the operator `[]`? Let's try and create **abstract data type**, which can be used in the program

similar to the array. To make the creation of such type sensible, it is necessary to overcome the main drawbacks, peculiar to ordinary arrays C++, i.e.:

- the necessity of setting the array size at the stage of compiling;
- the absence of control of the overrunning the array;
- impossibility of setting the arbitrary limits of index alteration;
- the absence of predefined operators of array assignment, performance of arithmetic operators with them, etc.

Let's create the class **Array**, which is a formalization of the concept **“one-dimension array of the integers”**. For simplicity let's suppose that the array of **Array** type has the same range of index alteration as an ordinary array C++ has, and the alteration of its dimension after creation is not possible. Let's define the assignment, addition, and output operators for the **Array** type. Let's overload the operator [] for referring to the elements of such array:

```
// File Array.cpp
#include < iostream.h >
#include < stdlib.h >
class Array{
int *pa;                // Integer array;
int sz;                // the size of the array.
public:
Array( const Array &v );
Array( const int a[ ], int s );
Array( int s );
virtual ~Array( ){ delete pa; }
int size( ){ return sz; }
int & operator[ ]( int );
Array & operator =( Array& );
                // The result returns by the reference for the opportunity
                // of multiple assignment of a=b=c-type;
Array & operator + ( Array& );
ostream & print( ostream& );
};
Array::Array(const int a[], int s ){
                // Initialization of the array of Array-type by an ordinary array.
if( s <= 0 ){ cout << "Incorrect array size: "<< s << "\n";
exit(1);}
if(!( pa = new int[ sz = s ] )){
cout<<"Failure at memory allocation \n"; exit(1); }
for( int i = 0; i<sz; i++) pa[ i ] = a[ i ];
}
Array::Array( const Array &v ) {                // Copy constructor.
if(!( pa = new int[ sz = v.sz ] )){
cout << "Failure at memory allocation \n"; exit(1);}
for( int i = 0; i < sz; i++) pa[ i ] = v.pa[ i ];
}
}
```



```

Array::Array( int s ){
    // The creation of non-initialized array with the size s.
    if( s<= 0){ cout <<" Incorrect array size \n"; exit(1); }
    if(!(pa = new int[ sz = s ] )){
    cout <<" Failure at memory allocation \n"; exit(1); }
    }
    int & Array::operator[ ]( int index ){
    if( index < 0 || index >= sz){
    cout <<" Array overrun!\n"; exit(1); }
    return pa[index];
    }

```

/* Since the result returns by the reference, the element returns itself, but not its value, therefore the expression of the form `c[i]`, where `c` of the Array type can be in the left part of the assignment operator. */

```

ostream & Array::print( ostream& out){
    out << '\n';
    for( int i = 0; i < sz; i++) out << pa[ i ]<<" ";
    out <<"\n";
    return out; }
ostream & operator <<( ostream & out, Array & v ){
    v.print( out);
    return out; }
Array & Array::operator +( Array & op2 ){
    if( sz != op.sz ){
    cout<<"The attempt to sum up the arrays of different dimen-
    sions!\n";
    exit(1);}
    Array & tmp = *( new Array(sz));
    for( int i = 0 ; i < sz ; i ++ ) tmp[ i ] = pa[ i ] +
    op2.pa[ i ];
    return tmp;
    }
Array & Array::operator =( Array &v ){
    if( sz != .sz ){
    cout <<"Different dimensions of the arrays at assign-
    ment!\n"; exit(1);}
    for( int i = 0; i < sz ; i ++ ) pa[ i ] = v[ i ];
    return( *this );
    }

```

// The end of the file Array.cpp.

Now it is possible to write the next program:

```

# include "Array.cpp"
void main( ){
    int a[ ] ={ 1, 7, 3, 15, 6, 20, 7 };
    Array mas(a, sizeof a / sizeof(int));

```

```

Array b(7); // Undefined array.
Array c = mas; // Copy constructor.
b = mas + c ;
mas = b +( c = mas );
for( int i=0; i < 7; i ++ ) cout << a[i] << " "; cout << "\n";
cout << mas << b << c; // Compare these two outputs!
}

```

Now let's create the class **Matrix**, which is formalization of the concept of two-dimensional array.

```

// File Matrix.cpp
class Matrix{ Array **pm; // Pointers array to Array.
int r, c; // Matrix dimensionality.
public:
Matrix( int, int );
virtual ~Matrix( );
int row( ){ return r;}
int col( ){ return c;}
Array & operator[ ](int);
Matrix & operator =(Matrix&);
Matrix & operator +(Matrix&);
Matrix & operator *(Matrix&);
ostream & print(ostream & s);
};

// The result of the operator [ ], applied to the object of Matrix-type,
// must be the object of Array-type:
Array & Matrix::operator[ ]( int index ){
if( index < 0 || index >= r ){
cout << " Array overrun ! \ n"; exit(1);}
return * pm[index];
}

// Constructor:
Matrix::Matrix(int row, int col ){
pm = new Array *[ row ];
for(int i = 0; i < row; i ++ ) pm[ i ] = new Array(col);
r = row; c = col;
}
Matrix::~~Matrix(){for( int i = 0; i<r; i ++ )delete pm[i];
delete pm; }
ostream & Matrix::print( ostream & s ){
s<<"\n";
for( int i =0; i < r; i++ ){
Array &v = *pm[ i ];
for( int j = 0; j < c; j++ ) s << v[ j ] <<" ";
s << "\n"; }
return s ;
}

```

```

Matrix & Matrix::operator =( Matrix & tmp ){
if( r != tmp.r ){ cout <<
"Different dimensions of the arrays!\n"; exit(1); }
for( int i = 0; i < r; i++ ) *pm[ i ] = tmp[ i ];
return *this;
}
Matrix & Matrix::operator +( Matrix & op2 ){
if( r != op2.r ){
cout << " Different dimensions of the arrays!\n"; exit(1);
}
Matrix & tmp = *( new Matrix( r, c ) );
for( int i = 0; i < r ; i++ )tmp[ i ] = *pm[i] + op2[i];
return tmp;
}
Matrix & Matrix::operator *( Matrix & op2 ){
if( c != op2.r ){ cout <<
"It is impossible to multiply the matrices!\n"; exit(1);}
Matrix & tmp = *( new Matrix( r, op2.c));
for( int i = 0; i < r ; i++ )
for( int j = 0; j<op2.c; j++){ tmp[ i ][ j ] =0;
for( int k = 0; k < c ; k++ ) tmp[ i ][ j ]+=
(*this)[ i ][ k ]*op2[k ][ j ];
}
return tmp;
}
ostream & operator <<( ostream &s, Matrix & m ){
m.print( s );
return s;
}

```

// The end of the file Matrix.cpp

Now it is possible to make a program, where new types of data **Array** and **Matrix** are used.

```

#include < iostream.h >
#include "Array.cpp"
#include "Matrix.cpp"

void main( ){
Matrix tbl( 3, 5), tbl2( 3, 5 );
for( int i = 0; i < 3; i++ )
for( int j = 0; j < 5; j++ ){
tbl[ i ][ j ] = i + j;
tbl2[ i ][ j ] =( i + j )*10;
}
Matrix tbl3 = tbl + tbl2;
cout << tbl3;
Array arr(10), arr2(10);
for( i = 0; i < 10; i++ ){
arr[i] = i; arr2[i] = i*10;
}
}

```

```
Array arr3 = arr + arr2;
cout << arr3;
Matrix mas( 5, arr.size());
for( i = 0; i < 5; i++ ) mas[i] = arr;
Matrix mas2( 3, arr.size());
mas2 = tbl*mas;
cout << mas << mas2;
}
```

ADVICE

1. Avoid type fields.
2. Use pointers and references to avoid slicing.
3. Use abstract classes to focus design on the provision of clean interfaces.
4. Use abstract classes to minimize interfaces.
5. Use abstract classes to keep implementation details out of interfaces.
6. Use virtual functions to allow new implementations to be added without affecting user code.
7. Use abstract classes to minimize recompilation of user code.
8. Use abstract classes to allow alternative implementations to coexist.
9. A class with a virtual function should have a virtual destructor.
10. An abstract class typically doesn't need a constructor.
11. Use ordinary multiple inheritance to express a union of features.
12. Use multiple inheritance to separate implementation details from an interface.
13. Use a *virtual* base to represent something common to some, but not all, classes in a hierarchy.
14. Avoid explicit type conversion (casts).
15. Use *dynamic_cast* where class hierarchy navigation is unavoidable; § 15.4.1.
16. Prefer *dynamic_cast* over *typeid*.
17. Prefer *private* to *protected*.
18. Don't declare data members *protected*.
19. If a class defines *operator delete()*, it should have a virtual destructor.
20. Don't call virtual functions during construction or destruction.
21. Use explicit qualification for resolution of member names sparingly and preferably use it in overriding functions.

EXERCISES

1. Define a table of the names of months of the year and the number of days in each month. Write out that table. Do this twice; once using an array of char for the names and an array for the number of days and once using an array of structures, with each structure holding the name of a month and the number of days in it.
2. Without looking in the book, write down as many C++ keywords you can.
3. Write a standards conforming C++ program containing a sequence of at least ten consecutive keywords not separated by identifiers, operators, punctuation characters, etc.
4. Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player “smart” enough to be worth playing against.
5. Define a graphical object class with a plausible set of operations to serve as a common base class for a library of graphical objects; look at a graphics library to see what operations were supplied there. Define a database object class with a plausible set of operations to serve as a common base class for objects stored as sequences of fields in a database; look at a database library to see what operations were supplied there. Define a graphical database object with and without the use of multiple inheritance and discuss the relative merits of the two solutions.
6. Define

```
class base {  
public:  
    virtual void iam() {cout << "base\n"; }  
};
```

Derive two classes from *base*, and for each define *iam()* to write out the name of the class. Create objects of these classes and call *iam()* for them. Assign pointers to objects of the derived classes to *base** pointers and call *iam()* through those pointers.

7. Implement a simple graphics system using whatever graphics facilities are available on your system (if you don't have a good graphics system or have no experience with one, you might consider a simple "huge bit ASCII implementation" where a point is a character position and you write by placing a suitable character, such as * in a position): *Window(n,m)* creates an area of size *n* times *m* on the screen. Points on the screen are addressed using (*x,y*) coordinates (Cartesian). A *Window w* has a current position *w.current()*. Initially, *current* is *Point(0,0)*. The current position can be set by *w.current(p)* where *p* is a *Point*. A *Point* is specified by a coordinate pair: *Point(x,y)*. A *Line* is specified by a pair of *Points*: *Line(w.current(), p2)*; class *Shape* is the common interface to *Dots*, *Lines*, *Rectangles*, *Circles*, etc. A *Point* is not a *Shape*. A *Dot*, *Dot(p)* can be used to represent a *Point p* on the screen. A *Shape* is invisible unless *draw()*. For example: *w.draw(Circle(w.current(), 10))*. Every *Shape* has 9 contact points: *e* (east), *w* (west), *n* (north), *s* (south), *ne*, *nw*, *se*, *sw*, and *c* (center). For example, *Line(x.c(), y.nw())* creates a line from *x*'s center to *y*'s top left corner. After *draw()*ing a *Shape* the current position is the *Shape*'s *se()*. A *Rectangle* is specified by its bottom left and top right corner: *Rectangle(w.current(), Point(10,10))*. As a simple test, display a simple "child's drawing of a house" with a roof, two windows, and a door.
8. Important aspects of a *Shape* appear on the screen as a set of line segments. Implement operations to vary the appearance of these segments: *s.thickness(n)* sets the line thickness to 0, 1, 2, or 3, where 2 is the default and 0 means invisible. In addition, a line segment can be *solid*, *dashed*, or *dotted*. This is set by the function *Shape::outline()*.
9. Provide a function *Line::arrowhead()* that adds arrow heads to an end of a line. A line has two ends and an arrowhead can point in two directions relative to the line, so the argument or arguments to *arrowhead()* must be able to express at least four alternatives.
10. Make sure that points and line segments that fall outside the *Window* do not appear on the screen. This is often called "clipping." As an exercise only, do not rely on the implementation graphics system for this.
11. Add a *Text* type to the graphics system. A *Text* is a rectangular *Shape* displaying characters. By default, a character takes up one coordinate unit along each coordinate axis.
12. Define a function that draws a line connecting two shapes by finding the two closest "contact points" and connecting them.

13. Add a notion of color to the simple graphics system. Three things can be colored: the background, the inside of a closed shape, and the outlines of shapes.

14. Consider:

```
class Char_vec {
    int sz;
    char element[1];
public:
    static Char_vec* new_char_vec(int s);
    char& operator [](int i) {return element[i]; }
    // ...
};
```

Define *new_char_vec()* to allocate contiguous memory for a *Char_vec* object so that the elements can be indexed through *element* as shown. Under what circumstances does this trick cause serious problems?

15. Given classes *Circle*, *Square*, and *Triangle* derived from a class *Shape*, define a function *intersect()* that takes two *Shape** arguments and calls suitable functions to determine if the two shapes overlap. It will be necessary to add suitable (virtual) functions to the classes to achieve this. Don't bother to write the code that checks for overlap; just make sure the right functions are called. This is commonly referred to as *double dispatch* or a *multi-method*.
16. Design and implement a library for writing event-driven simulations. Hint: *<task.h>*. However, that is an old program, and you can do better. There should be a class *task*. An object of class *task* should be able to save its state and to have that state restored (you might define *task::save()* and *task::restore()*) so that it can operate as a coroutine. Specific tasks can be defined as objects of classes derived from class *task*. The program to be executed by a task might be specified as a virtual function. It should be possible to pass arguments to a new task as arguments to its constructor(s). There should be a scheduler implementing a concept of virtual time. Provide a function *task::delay(long)* that "consumes" virtual time. Whether the scheduler is part of class *task* or separate will be one of the major design decisions. The tasks will need to communicate. Design a class *queue* for that. Devise a way for a task to wait for input from several queues. Handle run-time errors in a uniform way. How would you debug programs written using such a library?

UNIT 6

23. CLASSES AND TEMPLATES

A template of the class family determines the method of separate class construction similar to the method, in accordance with which the class determines the rules of construction and the format of separate objects. The template of the class family can be defined as:

template < the_list_of_template_parameters > class definition

In class definition, included into the template, the name of the class is of special importance. It is not the name of the separate class, but a parametrized name of the class family.

The template definition can be the only global.

Let's consider the class **vector**, among the data of which a one-dimension array is included. Irrespective of the element type of this array, the same base operations, for example, the access to the element by the index and something of this kind must be defined in the class. If the element type of the **vector** is specified as a parameter of the class template, the system will form the **vector** of a necessary type and a corresponding class during each definition of a specific object.

```

// File vec.cpp
// T – template parameter;
template < class T >
class vector{
T *pv; // one-dimension array;
int size; // the array dimension.
public:
vector( int );
~vector( ){ delete[]pv; }
T & operator[ ]( int i ){ return pv[ i ]; }
. . . };
template < class T >
vector < T >::vector( int n ){
pv = new T[n];
size = n; }
// The end of the file vec.cpp
```

When the template is introduced, the possibility appears to define specific objects of the particular classes, each of which is parametrically generated from the template. The format of object definition for the class, generated by the template, is the following:

**the_name_of_the_generic_class <actual_template_parameters>
the_object_name (constructor_parameters)**

In our case it is possible to define the vector of the 100 components of **double** type in the following way:

```
vector < double > x( 100 );
```

The program can be presented as:

```
#include < iostream.h >
#include "vec.cpp"
void main( ){
vector < int > x( 5 );
vector < char > c( 5 );
for( int i = 0; i < 5; i++ ){
x[ i ] = i; c[ i ] = 'A' + i;}
for( i = 0; i < 5; i++ ) cout << " " << x[ i ] << " " << c[
i ];
cout << "\n";
}
```

The result:

0 A 1 B 2 C 3 D 4 E

In the list of the template parameters there can be formal variables, which do not define the type. To put it more precisely, these are the parameters, for which the type is fixed:

```
template < class T, int size = 64 >
class row{
T * data;
int length;
public: row( ){ length = size; data = new T[ size]; }
~row( ){ delete T[] data; }
T & operator[ ]( int i ){ return data[i]; }
};
void main( ){
row < float, 7 > rf;
```

```
row < int, 7 > ri;  
for( int i = 0; i < 7; i++ ){ rf[i] = i ; ri[i] = i * i; }  
for( i = 0; i < 8; i++ )  
cout << " " << rf[i] << " " << ri[i];  
cout << "\n";  
}
```

The result:

0	0	1	1	2	4	3	9	4	16	5	25	6	36
---	---	---	---	---	---	---	---	---	----	---	----	---	----

A constant is taken as an actual argument for the parameter **size**. In the general case a **constant** expression can be used, however, it is forbidden to use the expressions, containing variables.

24. DYNAMIC DATA STRUCTURES

24.1. LISTS

Let's consider the following structure

```
typedef int ETYPE;  
struct elem{ ETYPE data;  
            elem *next;  
};
```

Let's call **data** as a data entry. Here it is of **int** type, but it can be of any complex type **ETYPE**, which is necessary for us.

The pointer **next** points to the object of **elem** type. The objects of **elem** type can be ordered by means of the pointer **next** in the following way (*Fig. 3*):



Fig. 3. The structure of unidirectional list

Such data structure is called unidirectional, or one-way list, sometimes a chain.

The objects of **elem** type from this list are called the elements or units of the list. Each element of the chain, except the last one, contains the pointer to the element, following it. The marker of the last element in the list is that the member of the **elem*** type of this unit is equal to **NULL**. A variable, which value is the pointer to the first element of the list, is considered together with each list. If the list does not have any elements, i.e. it's empty, the value of this variable must be **NULL**.

Let's consider the methods of working with such lists. Let the variables **p, q** have **elem*** type:

```
elem *p, *q;
```

Let's make up the list of two elements, containing numbers 12 and -8. The value of the variable **p** will always be the pointer to the first element of the list part, which has been already made. The variable **q** will be used for allocation of the memory space for location of new elements of the list by means of operator **new**.

Operator performance

```
p = NULL;
```

leads to the creation of the empty list. After performing of the operator

```
q = new elem; q->data = -8; q->next = p; p = q;
```

there is the list, consisting of one element, containing number -8 in the data part. The variables **p**, **q** point to this element (Fig. 4a).

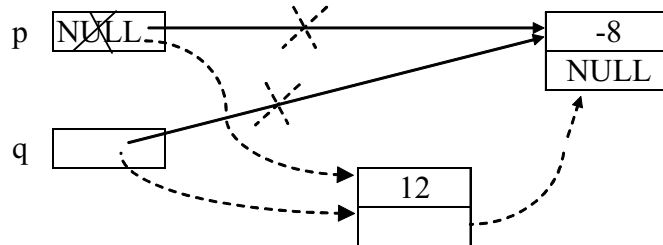


Fig. 4. Creation of the list of one (a – solid lines) and two (b – dotted line) elements

Later, the operator performance (Fig. 4b)

```
q = new elem; q->data = 12; q->next = p; p = q;
```

leads to adding of a new element, containing number 12, to the beginning of the chain. The result is the list presented in Fig. 5. The value of the variables **p** and **q** is a pointer to the first element of the list.

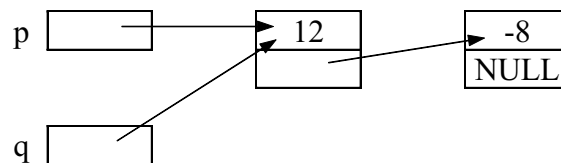


Fig. 5. The list of two elements

In fact, the operation of including of a new element into the beginning, or head of the list, was considered, and the list formation consists in starting from the empty list and successive addition of the elements to the beginning.

Example. Let's make up a list, which elements contain integer numbers 1, 2, 3, ..., n.

```
p = NULL;
while( n > 0 ){
q = new elem;
q->data = n;
q->next = p; p = q;
n --;}

```

Let's note that during including the element into the head of the list the order of the element arrangement in the list is inverse to the order of their including.

The main operation during working with the list is a passage through the list. Let's suppose that it is necessary to perform some operation, which is realized by the function **void P(ETYPE)**, with each data element of the unit. Let **p** again point to the beginning of the list. Then the passage through the list is performed in the following way:

```
q = p;
while(q) {
P( q->data );
q = q->next; }
```

Example. There is a sequence, containing odd number of integer numbers, in the input file **num.dat**. Let's make a program performing the output of the number, standing centrally in this sequence.

```
#include < fstream.h > // For working with input file.
#include < stdlib.h >
struct elem{ int data;
elem *next; };
void main( ){
ifstream infile( "num.dat" );

/* An input stream with the name infile is created for data reading, the file
with the name "num.dat" is being searched, if such file does not exist,
the constructor aborts the work and returns a zero value for infile.*/

if( !infile ){
cout << "Error while opening the file num.dat!\n"; exit(1);
}
elem *p = NULL, *q;
int j = 0;
while( infile.peek() != EOF ){

/* The member function peek() of ifstream class returns the next symbol from
the input stream infile, without its extracting out of it. If the end of the file is
met, the value EOF will be returned, i.e. -1. */

q = new elem;
infile >> q->data;
q->next = p; p = q;
j++;
}
for( int i = 1; i <= j/2; i++ )
q = q->next;
cout << q->data << "\n";
}
```

24.2. THE OPERATIONS OVER UNIDIRECTIONAL LISTS

There are three main operations over the lists.

1) The passage **through the list**, or transition from one element to the next one.

As it has been considered above, it is implemented by means of assignment $q = q \rightarrow \text{next}$;

2) Including an item into the list.

Let q , r are the variables of elem^* type. Let's suppose that it is necessary to include a new element to the list after some element, to which q points. Let's create this new element by means of the pointer r and put number 19 into its data part. Such inclusion is implemented by the following operators:

```
r = new elem;           // stage (1) – a new unit creation;
r->data = 19;           // the entry of the number 19 into the data part;
r->next = q->next;     // stage (2) – the pointer r->next of a new unit
                        // is adjusted in such a way that it points to
                        // the element, containing the number 12;
q->next = r;           // stage (3) – the pointer q->next of the unit,
                        // containing 5, now points to a new
                        // element, the old link is broken.
```

Let's illustrate this in the next figure (Fig. 6).

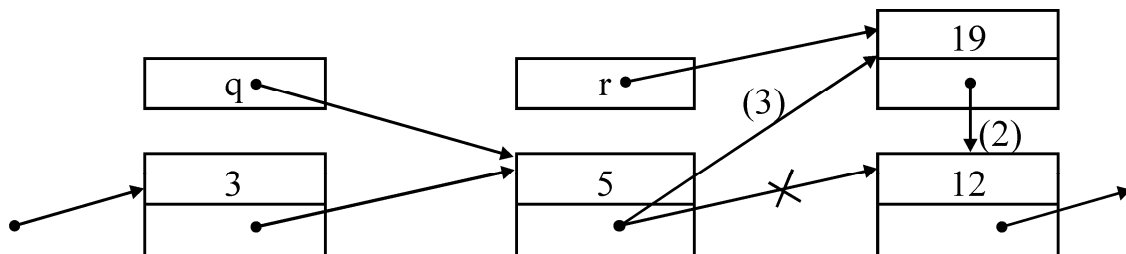


Fig. 6. Element inclusion into the list

3) Elimination an item from the list.

Let the value of the variable q will be a pointer to some, not last, element of the list and it is required to eliminate the element, following it, from the list. It can be done the following way:

```
r = q->next;
q->next = q->next->next;
r->next = NULL;
```

The second of the given assignments is the elimination from the list itself, and the first is performed for storing the pointer to the eliminated element, i. e. for being accessible and for making it possible to carry out actions with it, after elimination from the list. For example, to insert it into different place or

to release the memory, occupied by it, by means of the operator **delete r**. The third assignment is performed to make the elimination final, i. e. in such way, that it would be impossible from the eliminated element to get into the list, from which it was eliminated.

Implementation of the list

Let's implement the list notion through the mechanism of classes.

```
// File "list.cpp"
#include < iostream.h >
#include < stdlib.h >
typedef int ETYPE;
struct elem{
ETYPE data;
elem * next;
elem( ETYPE e, elem * n ){ data = e; next = n;}
};
class list{
    elem *h;           // The addresses of the beginning of the list.
public:
    list( ){ h = NULL; }
    ~list( ){release( ); }
    void create( ETYPE );           // Adds the element to the
                                   // beginning of the list.
    void release( );               // Eliminates the list.
    void insert(elem* q, ETYPE c); // Inserts c after q in the list.
    void del0( ){                   // Eliminates the first element.
        elem *t = h; h = h->next; delete t;}
    void del( elem * q );           // Eliminates the element after q.
    void print( );                 // Prints the list.
    friend class iter;
    elem *first( ){ return h; }
};
class iter{
elem * current;
public:
iter( list & l ) { current = l.h; }
elem * operator ++( );           // Movement through the list.
};
void list::create( ETYPE c ){ h = new elem( c, h ); }
void list::insert( elem *q, ETYPE c ){
q->next = new elem( c, q->next );
}
void list::del( elem *q ){ if( q->next == NULL ){
cout << "The end of the list! "

```



```

<<"Elimination of the next element is impossible!\n"; exit(1); }
elem * r = q->next; q-> next = q->next->next;
r->next = NULL;
delete r;
}
elem* iter::operator ++( ){
/* Returns the pointer to the current element of the list. Implements
the movement along the list. Memorizes the new current element of the list.
*/

if( current ){ elem * tmp = current;
current = current->next;
return tmp; }
return NULL;
}
void list::release( ){
iter t( *this );
elem *p;
while(( p = ++t )!= NULL ){h = h->next; delete p;}
}
void list::print( ){
iter t( *this );
elem *p;
while(( p = ++t )!= NULL )
cout << p->data << " ";
cout << '\n';
}
// The end of the file list.cpp

```

Here the unidirectional list is realized. This is one of the simple patterns of structures of data control. The class **list**, realizing this pattern, is a representative of so called **container** types. The class **iter** is created specially for searching of the elements of the arbitrary list of **list** type. The objects, assigned for searching of the elements inside of some set of data, are usually called **iterators**.

Let's give an example of the use of unidirectional list.

A nonempty sequence of the integer numbers $A(1), A(2), \dots, A(n)$ is in the file **int.dat**. Determine the quantity of these numbers **n** and type them in the ascending order. For solution this problem, it will be necessary to make up a list, which elements are arranged in the ascending order of the contained integer numbers. The first step is a list creation, consisting of one element containing $A(1)$. It is evident that this list is arranged. At the *i*-th stage ($i = 2, 3, \dots, n$) let's pass from the ordered list, which elements contain the numbers $A(1), \dots, A(i-1)$, to the ordered list, which elements contain $A(1), \dots, A(i-1), A(i)$. For performing such transition it is enough to include a new

element, containing $A(i)$, into the list. It should be inserted immediately after the last element containing the number which is less than $A(i)$.

If all the elements of the initial list contain the numbers, not less than $A(i)$, the new element should be inserted into the beginning of the list.

```
#include "list.cpp"
#include < fstream.h >
void main( ){
ifstream file( "int.dat" );
list lst; int i, n;
file >> i; n = 1;
lst.create( i );
while( file.peek( )!= EOF ){
file >> i; n ++;
iter tmp( lst );           // Let's create the object-iterator for
                           // searching of the elements of the list lst.

elem *p, *q;
while(( p = ++tmp)!= NULL )
if( p->data < i ) q = p;
else break;
if( p == lst.first( ) ) lst.create( i );
else lst.insert( q, i ); }
cout <<" There are " << n << " numbers in the file\n";
cout <<" An ordered list:\n";
lst.print( );
}
```

In the last operator **if-else** the verification **p == lst.first()** is carried out. This is necessary because the mechanism of unit insertion into the beginning of the list and into the list after the pointer **p** is different. The difference appears as the first element does not have the previous one. Sometimes a so called heading element, which is never deleted and no element is inserted before it, is placed at the beginning of the list for uniformity. Usually its data part is not used.

24.3. DOUBLE-LINKED AND CIRCULAR LISTS

To make an access to the previous elements convenient, let's add one more pointer that points to the preceding unit of the list, to each element of the list:

```
struct elem{
ETYPE data;
elem * next;
elem * prev;
elem( ETYPE c, elem * n, elem * p ){ data = c; next = n;
prev = p; }
};
```

By means of the elements of such type (Fig. 7) it is possible to make up a so called double-linked or bidirectional list (with the heading element):

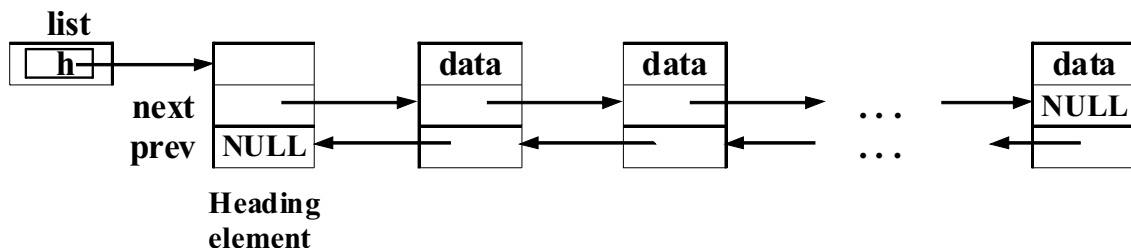


Fig. 7. Double-linked list

Here in the field **prev** of the heading unit is an empty pointer NULL, denoting that the heading element does not have a preceding one. The double-linked lists are generalized in the following way (Fig. 8 and Fig. 9): the pointer to the heading (or the first) unit is taken as a value of **next** of the last unit, and the pointer to the last unit is taken as a value of the field **prev** of the heading (correspondingly the first) unit.

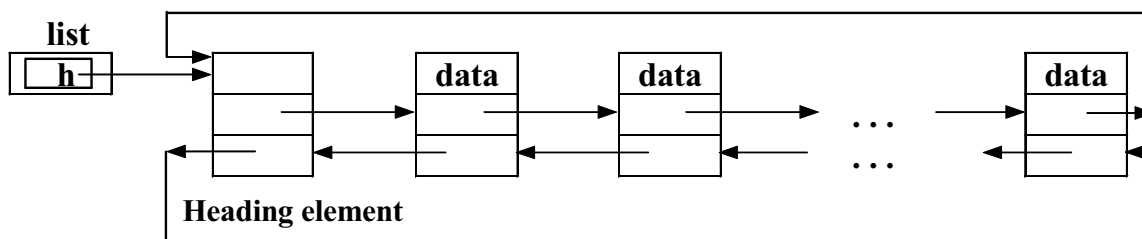


Fig. 8. The first variant of the double-linked circular list

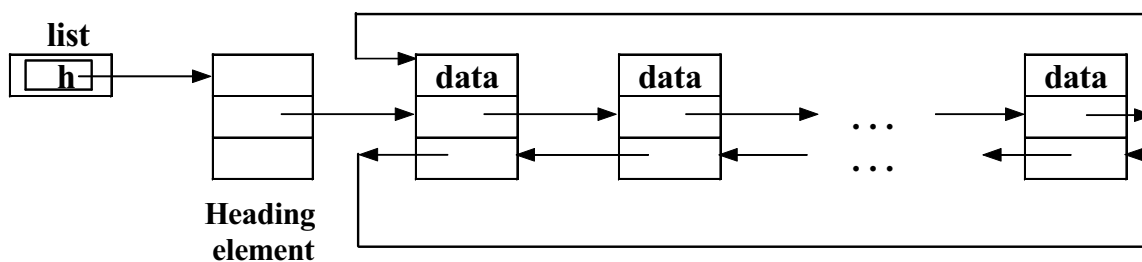


Fig. 9. The second variant of the double-linked circular list

Here the list turns into a circle, that is why the lists of such type are called double-linked circular.

In the first variant (Fig. 8) the insertion of a new unit into both the beginning of the list after the heading element and into its end is realized very simply, as the insertion of the unit into the end of the list is equivalent to its insertion before the heading element. But here during cyclic processing of the elements it should be checked whether the next unit is heading. The second variant is free of this imperfection (Fig. 9), but in this case it is more difficult to realize the addition to the end of the list.

Let's consider in the next topic the basic operations with circular double-linked lists of the first variant (Fig. 8).

24.4. THE CIRCULAR LIST OPERATIONS

Element insertion

Let **h**, **p**, **q** be the variables of **elem*** type, and **k** be a variable of **int** type. The **k** value must be set in the data part of the element, which must be inserted after the unit, which the pointer **p** points to.

This insertion can be implemented like this:

```
q = new elem(k, p->next, p);
p->next->prev = q; p->next = q;           // In such an order!
```

For insertion of a new element into the beginning of the list it is enough for the pointer **p** to possess the value of the address of the heading element of the list: **p=h**;

Element deletion

The possibility to move by the pointers in any direction allows to set a deleted unit by the pointer **p** directly to this unit:

```
p->next->prev = p->prev;
p->prev->next = p->next;
delete p;
```

Element search

Let **h** be a pointer to the heading element of the list, **r** be a pointer, which will point to the obtained unit, containing **k**. Also let **p**, **q** be variables of **elem*** type, and **b** – of **int** type. The search of the element, containing the number **k**, is performed like this:

```
b = 1; h->data = k + 1; // In the data part of the heading unit
                          // the number, different from k, is set.
p = h->next;           // First, p points to the first unit.
r = NULL;
q = p;                // now q points to the first unit.
do{
  if(p->data == k){
    b = 0;
    r = p;
  }
  p = p->next;
}while((p != q) && b);
```

Let's note that if there is not a unit, containing **k**, in the list at all, the value **b** will remain equal to one, the pointer **r** will be NULL, **p** will possess the value **q**, i.e. it will point to the first unit (after the heading one) again, after searching.

24.5. STACKS

In the programming a data structure, which is called a **queue**, is often used. Two operations on the queue are defined: they are the enqueue of the element and the selection of the element from the queue. In this case, a selected element is excluded from the queue. In the queue, two positions are accessible – its beginning (from this position an element is selected from the queue) and the end (an element for enqueue is placed into this position). There are two main types of the queues, which are different in service procedure. During the first of the procedures, the element, having entered the queue first, is selected first and deleted from the queue. This queue service procedure is called as FIFO (First In – First Out).

Let's consider in more detail the queue with such service procedure, when that element of the queue is selected for service first, which is entered the queue as the last. This service procedure is usually called as LIFO (Last In – First Out). In programming the queue of such type is called a **stack**. There is only one position in a stack available, called a top of a stack. This is a position, where is the element, which entered the queue last in the course of time. Let's map a stack onto a suitable structure of data of C++.

Stack Realization through the Array

```
// File stack0.cpp
typedef char ETYPE;
class stack{
enum{EMPTY = -1};
char *s;
int max_len;
int top;
public:
stack( ){s = new ETYPE[100];
max_len = 100;
top = EMPTY; // Stack is empty.
}
stack(int size){s = new ETYPE[size]; max_len = size;
top = EMPTY; }
stack(const ETYPE a[ ], int len){ // Initialization by the array.
max_len = len;
s = new ETYPE[max_len];
```

```

for(int i = 0 ; i < max_len; i ++ ) s[i] = a[i];
top = max_len - 1; }
stack(const stack & a){ // Initialization by the stack.
s = new ETYPE[a.max_len];
max_len = a.max_len; top = a.top;
for(int i = 0 ; i < max_len; i ++ ) s[i] = a.s[i];
}
~ stack(){ delete s ;}
void reset(){ top = EMPTY; } // Reset of the stack
// into the state EMPTY.
void push(ETYPE c ){ s[ ++ top ] = c ; } // Putting into stack.
ETYPE pop(){ return(s[ top -- ]); } // Extraction from the stack.
ETYPE top_show() const{ return(s[top]); }

```

/ Returns the element from stack, without its actual extracting. A modifier “const” guarantees that this function will not change data members of the objects of a stack-type */*

```

int empty() const{ return(top == EMPTY); }
// Checks whether the stack is empty. Returns 1,
// if the stack is empty, 0 – if it isn't empty.
int full() const{ return(top == max_len - 1); }
// Checks whether there is empty space in the stack.
};

```

// The end of the file stack0.cpp

Now the following operators may appear in the program:

```

stack data(1000); // Stack creation for the length of 1000.
stack d[5] // A default constructor creates an array
// of 5 stacks of 100 elements each.
stack w("ABCD", 4); // w.s[0] = 'A' ... w.s[3] = 'D'.
stack cop(w); // cop is the copy of stack w.

```

As an example, let's consider the problem of string output in the reverse order.

```

# include <iostream.h>
# include "stack0.cpp"
void main(){
char str[ ] = "Uncle Vasya!";
stack s;
int i = 0;
cout << str << '\n';
while( str[ i ] )

```

```

if( !s.full()) s.push(str[ i++]);
else{cout << "Stack is filled!" <<'\n'; break;}
while(!s.empty()) cout<<s.pop(); // Print in the reverse order.
cout <<'\n'; }

```

The result of the program performance

<pre> Uncle Vasya! !aysaV elcnU </pre>
--

It is possible to solve this problem like this:

```

char str[ ] = "Uncle Vasya!";
stack s(str, 12);
while(!s.empty()) cout<<s.pop;cout<<'\n';

```

Stack Realization through the Dynamic Chain of Links

Let the value of a pointer representing the stack as a whole, be the address of the top of the stack. As in a case of one-way list, each link will contain the pointer to the next element, and the “bottom” of the stack (i. e. the element, placed into the stack as the earliest) contains the pointer NULL.

// File stack.cpp

```

typedef char ETYPE;
struct elem{
    ETYPE data;
    elem* next;
    elem(ETYPE d, elem* n){ data = d; next = n; }
};
class stack{
    elem*h; // The address of the top of the stack.
public:
    stack(){h = NULL;} // A creation of the empty stack.
    stack(ETYPE a[ ], int len){ // Stack initialization
                                // by the array.
        h = NULL;
        for(int i = 0; i< len; i++) h = new elem(a[i], h);
    }
    stack(stack &a){ // Stack initialization by the other stack.
        elem *p,*q;
        p = a.h; q = NULL;
        while(p){ q = new elem(p->data, q);
            if(q->next == NULL) h = q;
            else q->next->next = q;
            p = p->next;
        }
    }
};

```

```

q->next = NULL;
}
~stack(){reset();}
void push(ETYPE c){h = new elem(c, h);} // To place into the stack.
ETYPE pop(){
elem *q = h; ETYPE a = h->data; // To extract from the stack.
h = h->next; delete q;
return a;
}
ETYPE pop_show(){return h->data;} // To indicate the top.
void reset(){ while(h){ elem *q = h; h = h->next; delete q;
}
}
int empty(){ return h ? 0:1;}
};
// The end of the file stack.cpp

```

Let's cite a problem, in the solution of which it is convenient to use a stack.

The string of letters is given in the file. It is required to check the balance of the brackets in this string.

The balance is maintained if each of the following conditions is met:

- 1) for each opening bracket there is a corresponding closing bracket, which is to the right of it; vice versa, for each closing bracket there is a corresponding opening bracket to the left of it;
- 2) corresponding pairs of brackets of different types are correctly put into each other.

Thus, in the string

```
{ [(a*b) + (n-4)] / [7-sin(x)] + exp(d) } * s
```

the balance of brackets is maintained, and in the string

```
{ a+b[i] ((x-sin(x)) ] d )
```

it is not maintained.

It is necessary to produce the message about the balance maintenance, and also the beginning of the string up to the first in order imbalance of brackets as a result.

For solution of the problem let's form, first, an empty stack. Then the letters of the string will be considered one after another. If a regular symbol is an opening brackets, let's place it to the stack. Then, if a regular symbol will become a closing bracket, let's select the last of the opening brackets, placed into stack, and compare these brackets for their correspondence to each other. If there is a correspondence, the effect must be the same as if these pairs of brackets were absent from the string at all. If these brackets do not correspond to each other, there is no meeting of the second condition of brackets

balance. If at the moment of selecting the regular closing bracket from the string the stack remained empty or if after completion of the look-up the string, the stack turned out to be empty, the first condition of brackets balance is not met.

Let's label the stack as **s**, a processed symbol as **sym**, and an integer variable, fixing the fact of the correspondence of the closing bracket from the string with the opening bracket from the top of the stack – as **b**. A special function **accord()**, which returns an integer value and deletes an opening bracket from the stack, must be described for checking the correspondence of the closing bracket, being the value of the variable **sym**, and the opening bracket at the top of the stack.

```
# include <fstream.h>
# include <stdlib.h>
# include "stack.cpp"
int accord(stack & s, char sym){
    char r = s.pop( );
    switch(sym){
        case `)` : return r == `(`;
        case `]` : return r == `[`;
        case `}` : return r == `{`;
        default : break;
    }
}
void main( ){
    ifstream file( "a.dat" );
    if( !file ){
        cout << "An error during opening the file a.dat!\n"; exit(1); }
    stack s;
    char sym;
    int i, n, b = 1;
    while( file.peek() != EOF && b){
        file >> sym; cout <<sym;
        switch(sym){
            case `(` : case `[` : case `{` : s.push(sym);
            break;
            case `)` : case `]` : case `}` :
                if( s.empty() || !accord(s, sym)) b = 0; break;
        }
    }
    if(!b || !s.empty())
        cout << "\nThere is not a bracket balance!\n";
    else
        cout << "\nThe brackets are balanced\n";
}
```

24.6. BINARY TREES

Definition and Construction

A coupled graph, without any cycles, is called a tree graph. The tree is called directed if the directions (input and output or in and out) are indicated at each of its edges.

A binary tree is a directed tree, which:

1) has just one node, which contains no input edge; this node is called a root of the binary tree;

2) there is one input edge in each of the nodes, except for the root;

3) there are no more than two output edges in each of the nodes.

Let's represent binary trees in such a way that the root is higher than other nodes.

For edges, coming out from any node, there are two possibilities – to be directed to the left-downward and to the right-downward. In this case, if single edge comes out of the node, the direction indication for the edge (left-downward or right-downward) is not essential. At such agreements, there is no use to indicate the directions on the edges – all the edges are oriented downward.

When solving some problems, it is convenient to represent the set of the objects as binary trees. Let's consider the problem of coding of the non-empty sequence of integer numbers.

Let the sequence of the integer numbers $\mathbf{a}(1), \dots, \mathbf{a}(n)$ and the function $\mathbf{F}(x)$ of the integer argument, which possesses integer values, are given. The value $\mathbf{F}(x)$ will be called a code of the number x . It is required to encode the number data, that is to calculate the values $\mathbf{F}(\mathbf{a}(1)), \dots, \mathbf{F}(\mathbf{a}(n))$, and there are frequent repetitions of the element values in the sequence $\mathbf{a}(1), \dots, \mathbf{a}(n)$.

To avoid repeated calculations of the same values, the table of the codes, found earlier, must be constructed during coding. The structure of the table must be of the sort that it is possible, firstly, to find the elements with a given value quite quickly (or to determine its absence from the table), and secondly, to add new elements into the table without any difficulties.

Presentation of the table in the form of a binary tree, in which nodes are different elements from the given sequence and their codes, satisfies these requirements.

The process of binary tree construction is realized in the following way: the first number and its code form the root. The next number, which should be coded, is compared to the number in the root, and if they are equal, the binary tree is not growing and the code can be taken from the root. If a new number is less than the first one, the edge is growing from the root to

the left-downward, otherwise to the right-downward. This new number and its code are placed into the formed node.

Let some binary tree has already been built and there is some number x , which must be coded. First, let's compare the number at the root with the x . In case of equality, the search is completed and the code for x is extracted from the root. Otherwise, it is necessary to turn to the left-downward node if x is less than the number at the root, or to the right-downward node if x is greater than considered one. Here, it needs to compare x with the number at this node, etc. The process finishes in one in two cases:

- 1) the node, containing the number x , is obtained;
- 2) the node, which is necessary to turn to for performing the next step, is absent from the tree.

In the first case, the code of the number x is extracted from the obtained node. In the second one, it is necessary to calculate $F(x)$ and to connect the node, where x and $F(x)$ are, to the tree. Let the sequence starts with the numbers 8, 4, 13, 10, 14, 10. Then at first the tree will grow in the following way (see Fig. 10a-e).

When number 10 appears as a sixth element of the sequence, a new node is not added to the tree, and the value $F(10)$ is extracted from the available node (Fig. 10e).

Let's define the following structure in the program of the sequence code construction:

```
struct node{
int num, code;
node* left, * right;
node(int n, int c, node *l, node *r){
num = n; code = c;
left = l; right = r;}
};
```

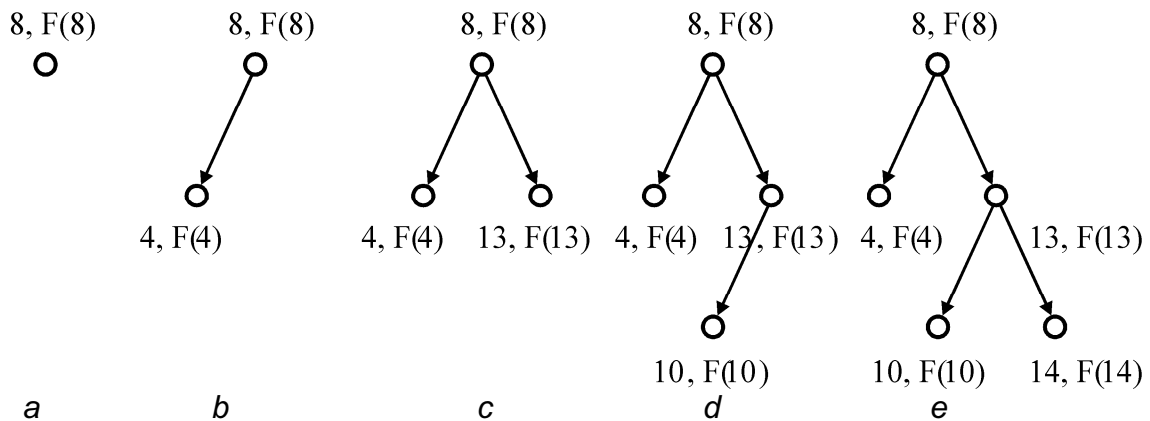


Fig. 10. The growing of the binary tree in the coding problem

The objects of **node** type are the structures, where in each of the fields **left** and **right** there is either NULL or the pointer to the memory space, allocated by means of **new** for the object of **node** type. The tree can be represented as an ensemble of the objects of **node** type, connected by the pointers. These objects themselves will be the nodes of the tree, and the pointers to the memory space, allocated for the objects of **node** type, will be the edges of the tree. If in this case the field **left** (**right**) is NULL, it means that in the tree there is no edge forwarded from the given node to the left-downward (to the right-downward). Let's depict a tree presentation in the computer memory corresponding to Fig. 10e (Fig. 11).

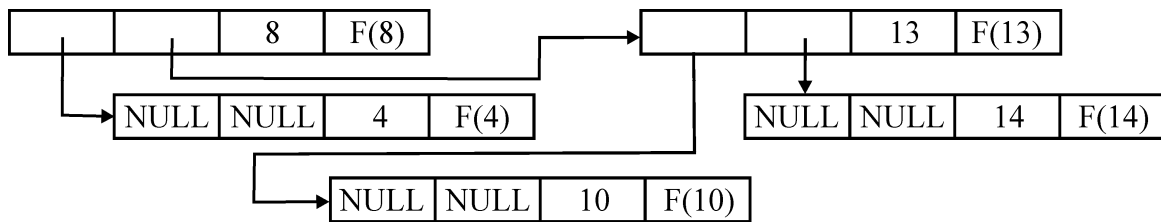


Fig. 11. Presentation of the tree in the computer memory

Assignment $v = v \rightarrow \text{left}$ ($v = v \rightarrow \text{right}$) means transition to the node, located immediately to the left-downward (to the right-downward) from the given one, if, of course, a corresponding field of the given top is not NULL. Thus, it is possible to move from one node to another one top-down. Introduction of the new node into the tree means the value alteration of the fields of **node*** type of some nodes of the given tree.

Along with each tree a variable, which value is a pointer to the tree root, is considered. If the tree does not have any nodes, this variable value must be equal to a zero pointer NULL.

Let's make the program, during which the sequence of natural numbers, placed in the input file NUM.DAT, is coded. For coding the file COD.DAT is used, which components are integer numbers. The code **F(k)** of the number **k** is considered to be a component of the file COD.DAT, which is the **k-th** in the order.

```
struct node{
int num, code;
node* left, *right;
node(int n, int c, node* l, node* r){
num = n; code = c; left = l; right = r;}
};
int f(int);
void insert(int n, node* root){
node*temp = root;
node* old;
while(temp !=0 ){
```

```

old = temp;
if(temp->num == n){ cout << temp->code <<" "; return; }
if(temp->num > n) temp = temp->left;
else temp = temp->right;
}
int k = f(n); cout << k << " ";
if(old->num > n) old->left = new node(n, k, 0, 0);
else old->right = new node(n, k, 0, 0);
}
ifstream num( "num.dat" ), cod( "cod.dat" );
int f(int k){ int i, j;
cod.seekg(0); // Determines the position of
// reading from the file into 0.

for( i = 1; i <=k; i++) cod >> j;
return j;
}
void main(){
int n;
num >> n;
node* root = new node(n, f(n), 0, 0);
cout << root->code << " ";
while(num.peek() != EOF){ num >> n;
insert(n, root);
}}

```

24.7. TABLES

The tree, built in the last example, is often called a searching tree. The searching tree is sometimes used for table construction, where different data are stored, usually in the form of structures. In this case they are usually named for structure ordering, and for the effective search of the structure by its name it is necessary to compare any two names and to determine which of them is “greater”, and which is “less”. The structure name in the table is often called a key of the structure. Integer numbers or the strings of the equal length are often used as a key.

The following operations are defined over the table as a structure:

- searching in the table for the structure with a given key;
- including into the table a structure with a given key;
- excluding from the table a structure with a given key.

We will consider the table organization in the form of the binary tree. In the example of coding, the numbers from the file NUM.DAT serve as a key. In fact, the operations of searching in the tree and including an element into the tree according to the given key have already been considered. Now let’s design the operation of element with a given key excluding from the tree in

the form of the function. The direct deletion of the structure is realized simply, if a deleted node of the tree is final or only one edge comes out of it (Fig. 12).

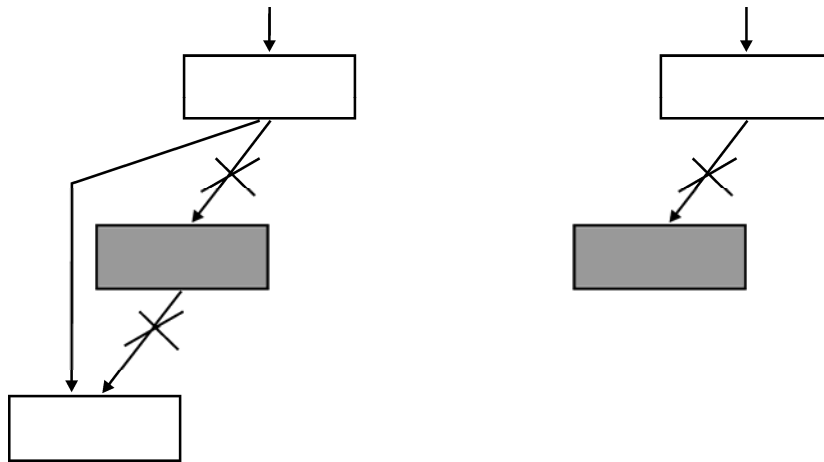


Fig. 12. Element deletion from the tree, when a deleted node is final or only one edge comes out of it

The difficulty is in node deletion, of which two edges come out. In this case, it is necessary to find an appropriate edge of the tree, which can be inserted into the place of the deleted one, so as this appropriate edge must be only moved. Such edge always exists: it is either the very right element of the left subtree, or the very left element of the right subtree. In the first case, it is necessary to pass into the next node along the left edge, and then to pass into the regular nodes only using the right edges until the regular pointer becomes equal to NULL. In the second case, vice versa, it is necessary to pass into the next node along the right edge, and then to pass into the regular nodes only using the left edges until the regular pointer becomes equal to NULL. Such appropriate edges cannot have more than one branch. Below (Fig. 13) the excluding of the node with the key 50 from the tree is sketched.

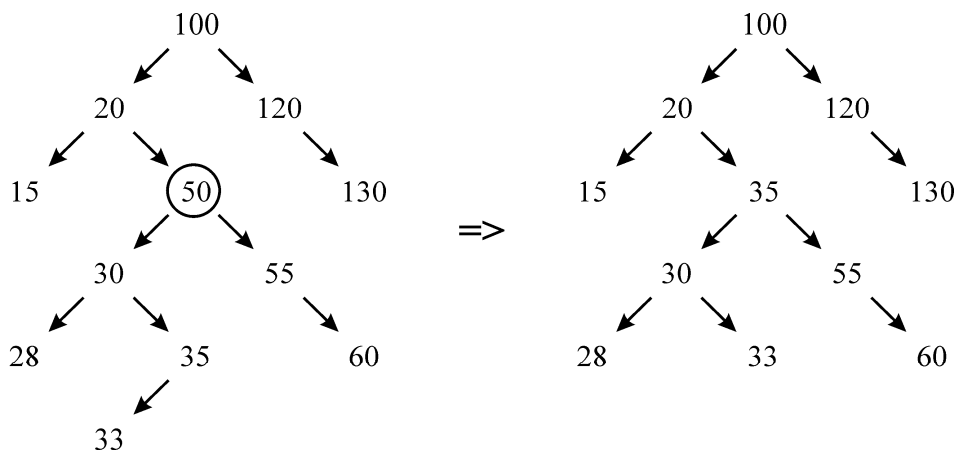


Fig. 13. Excluding the node with the key 50 from the tree

Let's write a program, realizing the searching tree with all the operations by means of the class **tree** and the function **insert** that is slightly changed.

```

#include <fstream.h>
struct node{
int key, code;
node *left, *right;
node(int k, int c, node *l, node *r){ key = k; code = c;
left = l; right = r; }
};
int f(int);
class tree{
node *root;
void print(node *r);           // Print starting with the pointer r.
public:
int insert(int);
void del(int);
tree(){ root = 0; }           // Creation of the empty tree.
node *&find(node *&r, int key); // Searching the element
// in the subtree
// with the top r.
node *&find(int key){        // Searching throughout
// the whole tree.

return( find(root, key) );
}
void print(){ print(root); } // Print of the whole tree.
};
int tree::insert(int key){    // Returns the code.
node* t = root;
node* old;
int k;
if(root == 0 ){ k = f( key );
root = new node( key, k, 0, 0 );
return k;
}
while(t !=0){
old = t;
if(t->key == key ){
return t->code; }
if( t->key > key ) t = t->left;
else t = t->right; }
k = f(key);
if( old->key > key) old->left = new node( key, k, 0, 0 );
else old->right = new node( key, k, 0, 0 );
return k; }
node *&tree::find( node *&r, int key){ // Recursive function

```

```

if( r == 0) return r; // of the search in the subtree r.
if( r->key == key ) return r;
if( r->key > key ) return( find( r->left, key ));
else
return( find( r->right, key ));
}
void del_el( node *&r, node *q){ // Recursive function
// of the element deletion,
// to which q points to.
if( r->right == 0 ){
q->key = r->key;
q->code = r->code;
q = r; r = r-> left;
} else
del_el( r->right, q );
}
void tree::del(int key){ // Deletion of the element with the key key.
node *q = find(key); // Adjustment of the pointer q to the deleted
// element.
if( q == 0){ cout << "than the element with the key " <<
key <<" no\n";
return; }
if( q->right == 0) q = q->left;
else
if( q->left == 0) q = q->right;
else
del_el( q->left, q);
}
void print( node *r){ // Global function of printing
// of the element of the tree r.
cout << r->key << " " << r->code << " ";
}
void tree::print( node *r){ // Recursive function of printing
// of the subtree, starting with r.
if(r != 0){
::print( r);
print( r->left );
print( r->right );
}
}
ifstream numb("num.dat"), cod("cod.dat");
int f( int k ){ int i, j;
cod.seekg( 0 );
for( i = 1; i <= k; i++) cod >> j;
return j;
}

void main(){

```



```

cout << "-----\n";
int key;
tree t;
while( num.peek() != EOF) {
num >> key;
cout << t.insert( key ) << " ";    // Insertion and printing
                                   // of the element,
                                   // or simply printing, if
}                                   // the element already exists.
cout << '\n';
t.print();
cout << "\n\n";
key = 50;
t.del( key );
t.print();
cout << '\n';
}

```

Let's note that the recursive function **del_el()** is called, if two edges of the tree are directed from the extracted top. It descends up to the very right node of the left subtree of the extracted element ***q**, and then it replaces the values of members **key** and **code** of ***q** with corresponding values **key** and **code** of the node ***r**. After it the node, which **r** points to, can be excluded by means of the operator **r = r->left**. It is possible to modify this function, releasing the memory, occupied by the deleted node by means of the operator **delete**.

ADVICE

1. Use templates to express algorithms that apply to many argument types.
2. Use templates to express containers.
3. Provide specializations for containers of pointers to minimize code size.
4. Always declare the general form of a template before specializations.
5. Declare a specialization before its use.
6. Minimize a template definition's dependence on its instantiation contexts.
7. Define every specialization you declare.
8. Consider if a template needs specializations for Cstyle strings and arrays.
9. Parameterize with a policy object.
10. Use specialization and overloading to provide a single interface to implementations of the same concept for different types.
11. Provide a simple interface for simple cases and use overloading and default arguments to express less common cases.
12. Debug concrete examples before generalizing to a template.
13. Remember to *export* template definitions that need to be accessible from other translation units.
14. Separately compile large templates and templates with nontrivial context dependencies.
15. Use templates to express conversions but define those conversions very carefully.
16. Use explicit instantiation to minimize compile time and link time.
17. Prefer a template over derived classes when runtime efficiency is at a premium.
18. Prefer derived classes over a template if adding new variants without recompilation is important.
19. Prefer a template over derived classes when no common base can be defined.
20. Prefer a template over derived classes when built-in types and structures with compatibility constraints are important.

EXERCISES

1. Fix the errors in the definition of *List* (below) and write out C++ code equivalent to what the compiler must generate for the definition of *List* and the function *f()*. Run a small test case using your handgenerated code and the code generated by the compiler from the template version. If possible, on your system given your knowledge, compare the generated code.

```
template<classT> class List {
struct Link {
Link* pre;
Link* suc;
T val;
Link(Link* p, Link* s, const T&v):pre(p), suc(s), val(v) { }
} // syntax error: missing semicolon
Link* head;
public:
List() : head(7) { } // error: pointer initialized with int
List(const T&t) : head(new Link(0,o,t)) { } // error: undefined identifier 'o'

// ...
void print_all() {for(Link* p=head; p; p=p->suc)
cout << p->val << '\n'; }
};
```

2. Write a singly linked list class template that accepts elements of any type derived from a class *Link* that holds the information necessary to link elements. This is called an intrusive list. Using this list, write a singly linked list that accepts elements of any type (a nonintrusive list). Compare the performance of the two list classes and discuss the tradeoffs between them.
3. Write intrusive and nonintrusive doubly linked lists. What operations should be provided in addition to the ones you found necessary to supply for a singly linked list?
4. Define a *sort()* that takes its comparison criterion as a template argument. Define a class *Record* with two data members *count* and *price*. Sort a *vector<Record>* on each data member.
5. Write a program that reads (*key, value*) pairs and prints out the sum of the *values* corresponding to each distinct *key*. Specify what is required for a type to be a *key* and a *value*.

6. Make the sum program from previous exercise work correctly for names containing spaces; for example, “thumb tack.”
7. Construct an example that demonstrates at least three differences between a function template and a macro (not counting the differences in definition syntax).
8. Write *readline()* templates for different kinds of lines. For example (item, count,price).
9. Rewrite the following class definition to make it a class template:

```
class example1 {
public:
    example1( double min, double max );
    example1( const double *array, int size );
    double& operator[]( int index );
    bool operator==( const example1& ) const;
    bool insert( const double*, int );
    bool insert( double );
    double min() const { return _min; };
    double max() const { return _max; };
    void min( double );
    void max( double );
    int count( double value ) const;
private:
    int size;
    double *parray;
    double _min;
    double _max;
};
```

10. Given the following class template

```
template <class elemType>
class Example2 {
public:
    explicit Example2( elemType val = 0 )
        : _val( val ){}
    bool min( elemType value ) { return _val < value; }
    void value( elemType new_val ) { _val = new_val; }
    void print( ostream &os ) { os <<_val; }
private:
    elemType _val;
};
template<class elemType>
ostream& operator<<( ostream &os, const Example2<elemType> &ex )
    { ex.print( os ); return os; }
```

what happens when we write the following?

- (a) `Example2< Array<int>* > ex1;`
- (b) `ex1.min(&ex1);`
- (c) `Example2< int > sa(1024), sb;`
- (d) `sa = sb;`
- (e) `Example2< string > exs("Walden");`
- (f) `cout <<"exs:" <<exs <<endl;`

11. Identify which, if any, of the following template class declarations (or declaration pairs) are illegal.

- (a) `template <class Type>`
`class Container1;`
`template <class Type, int size>`
`class Container1;`
- (b) `template <class T, U, class V>`
`class Container2;`
- (c) `template <class C1, typename C2>`
`class Container3 {};`
- (d) `template <typename myT, class myT>`
`class Container4 {};`
- (e) `template <class Type, int *ptr>`
`class Container5;`
`template <class T, int *pi>`
`class Container5;`
- (f) `template <class Type, int val = 0>`
`class Container6;`
`template <class T = complex<double>, int v>`
`class Container6;`

C++ GLOSSARY

A

abstract class – a *class* that can only be used as a *base class* for some other class. A class is abstract if it has at least one *pure virtual function*.

access control – a C++ mechanism for prohibiting or granting access to individual members of a *class*. See *public*, *private*, *protected*, and *visibility*.

access declaration – a way of controlling access to a specified *member* of a *base class* when it is used in a *derived class*.

access specifier – a way of labelling members of a *class* to specify what access is permitted. See *public*, *private*, and *protected*.

aggregate – an *array* or *object* of a *class* with no *constructors*, no *private* or *protected* members, no *base classes*, and no *virtual functions*. See *initializer* and *initialization*.

allocation – the process of giving memory space to an *object*. See *dynamic storage*, *static storage*, and *deallocation*.

ANSI – acronym for American National Standards Institute, a standards body currently standardizing C++.

argument – when calling a *function*, refers to the actual values passed to the function. See *parameter*.

argument matching – the process of determining which of a set of functions of a specified name matches given *arguments* in a function call.

ARM – acronym for the book *The C++ Annotated Reference Manual*, a C++ reference book by Ellis and Stroustrup.

array – an ordered and indexable sequence of values. C++ supports arrays of a single dimension (a *vector*) or of multiple dimensions.

asm – C++ *keyword* used to specify assembly language in the middle of C++ code.

assignment – the process of giving a value to a pre-existing *object*. See *copy constructor* and *initialization*.

assignment operator – an *operator* for doing *assignment*. See also *copy constructor*.

auto – a C++ *keyword* used to declare a stack-based *local variable* in a *function*. This is the default and is normally not needed. See *storage class*.

B

base class – a *class* that serves as a base for a *derived class* to inherit *members* from. See *inheritance*.

bit field – a member of a *class* that represents small integral values.

bitwise copy – to copy an *object* without regard to its structure or *members*. See *memberwise copy*.

bool – C++ *keyword* used to declare a Boolean data *type*.

break – C++ *keyword* used to specify a *statement* that is used to break out of a *for* or *while* loop or out of a *switch* statement.

browser – a software development tool used for viewing *class* declarations and the *class hierarchy*. See *programming environment*.

built-in type – see *fundamental type*.

C

C – a programming language in widespread use. C++ is based on C.

C-style string – refers to a *char** and to the contents of any *dynamic storage* it may point at. C++ does not have true strings as part of the language proper, though a standard string *class library* is envisioned as part of the *ANSI* standardization effort.

call by reference – passing a *pointer* to an *argument* to a *function*. The function can then change the argument value. See *call by value*.

call by value – passing a copy of an *argument* to a *function*. The function cannot then change the argument value. C and C++ use call by value argument passing. But also see *pointer* and *reference*, also *call by reference*.

calling conventions – refers to the system-specific details of just how the *arguments* to a *function* are passed. For example, the order in which they are passed on the stack or placed in machine registers.

case – a C++ *keyword* used to denote an individual element of a *switch* statement.

cast – a way of doing explicit *type conversion* via a cast operator. See *new-style cast*, *old-style cast*.

catch – a C++ *keyword* used to declare an *exception handler*.

cerr – in C++ *stream I/O*, the standard error *stream*.

cfront – a C++ *front end* that translates C++ source code to C code, which is then compiled via a C compiler. Originally developed by AT&T Bell Labs in the mid-1980s.

char – a C++ *keyword* used to declare an *object* of character *type*. Often considered the same as a byte, though it is possible to have multi-byte characters.

cin – in C++ *stream I/O*, the standard input *stream*.

class – a C++ *keyword* used to declare the fundamental building block of C++ programs. A class has a *tag*, *members*, *access control* mechanisms, and so on.

class hierarchy – see *base class*, *derived class*.

class layout – the way in which data *class members* are arranged in a *class object*.

class library – a set of related *classes* declared in *header files* and defined in *object files*

class member – a constituent member of a *class*, such as a *data declaration*, a *function*, or a *nested class*.

class template – a *template* used for generating *class types*.

comments – C++ has C-style comments delimited with */** and **/*, and new C++-style line-oriented comments starting with *//*.

compilation unit – see *translation unit*.

compiler – a software tool that converts a language such as C++ into a different form, typically assembly language. See *front end*.

const – a C++ *keyword* used to declare an *object* as constant or used to declare a constant *parameter*.

constant – a *literal* or *variable* declared as *const*.

constant expression – a C++ expression that can be evaluated by the *compiler*. Used to declare bounds for an *array* among other things.

constructor – a *function* called when a *class object* comes into *scope*. The constructor is used to *initialize* the object. See *allocation*, *copy constructor*, and *destructor*.

const_cast – a C++ *keyword* used as a style of *cast* for explicitly casting away *const*.

container class – a type of *class* or *template* that is used to hold *objects* of other *types*. Lists and stacks would be examples of container classes.

continue – C++ *keyword* used with *for* and *while* statements to continue the iteration at the top of the loop.

conversion – to convert from one data *type* to another.

copy constructor – a special type of *constructor* that is called when an *object* is copied. See *memberwise copy*.

cout – in C++ *stream I/O*, the standard output *stream*.

D

data abstraction – the idea of defining a data representation (for example, to represent a calendar date), and a set of operations to manipulate that representation, with no public access to the representation except via the operations. See *class*.

deallocation – the processing of freeing memory space previously used by an *object*. See *allocation*.

debugger – a tool for stepping through the execution of a program, examining variables, setting breakpoints, and so on.

declaration – a C++ entity that introduces one or more *names* into a program.

declaration statement – a *declaration* in the form of a *statement* that may be used in C++ where statements would normally be used.

declarator – a part of a *declaration* that actually declares an identifier *name*. A declarator appears after a sequence of *type* and *storage class* specifiers.

default argument – an optional *argument* to a *function*. A value specified in the function declaration is used if the argument is not given.

delete operator – C++ *keyword* and *operator* used to delete *dynamic storage*.

delete[] operator – See *delete operator*. Used to delete *array* objects.

demotion – converting a *fundamental type* to another fundamental type, with possible loss of precision. For example, a demotion would occur in converting a *long* to a *char*.

deprecate – to make obsolete (a language feature).

derived class – a *class* that inherits *members* from a *base class*. See *inheritance*.

destructor – a *function* called when a *class object* goes out of *scope*. It cleans up the object, freeing resources like *dynamic storage*. See *constructor* and *deallocation*.

dialect – refers to a variant of a programming language, used by a subset of the software community. Can also refer to a particular style of programming.

do – see *while*.

dominance – refers to the case where one *name* is used in preference to another. See *multiple inheritance*.

double – C++ *keyword* used to declare a *floating point type*.

dynamic storage – refers to memory allocated and deallocated during program execution using the *new operator* and *delete operator*.

dynamic_cast – a C++ *keyword* that specifies a style of *cast* used with *run-time type information*. Using `dynamic_cast` one can obtain a pointer to an *object* of a *derived class* given a *pointer* of a *base class* type. If the object pointed to is not of the specified derived class, `dynamic_cast` will return 0.

E

else – C++ *keyword*, part of the *if statement*.

embedded system – a low-level software program that executes without much in the way of *run-time* services, such as those provided by an operating system.

encapsulation – a term meaning to wrap up or contain within. Used in relation to the *members* of a *class*. See *access control*.

enum – C++ *keyword* used to declare an *enumeration*.

enumeration – a set of discrete named integral values. See *enum*.

enumerator – a member of an *enumeration*.

exception – a value of some *type* that is *thrown*. See *exception handling*.

exception handler – a piece of code that *catches* an *exception*. See *catch* and *try block*.

exception handling – the process of signalling that an exceptional condition (such as divide by zero) has occurred. An *exception* is *thrown* and then caught by an *exception handler*, after *stack unwinding* has occurred.

explicit – a C++ *keyword* used in the declaration of *constructors* to indicate that *conversion* of an *initializer* should not take place.

expression – a combination of *constants*, *variables*, and *operators* used to produce a value of some *type*.

expression statement – a *statement* that is an *expression*, such as a *function call* or *assignment*.

extern – a C++ *keyword* used to declare an *external name*.

external name – a *name* available to other *translation units* in a program. See *linker* and *global variable*.

F

false – C++ *keyword* used to specify a value for the *bool type*.

finalization – to declare that an *object* or resource is no longer needed, and initiate cleanup of that object. See *initialization*.

float – a C++ *keyword* used to declare a *floating point type*.

floating point – non-integral arithmetic. A floating-point number is typically represented as a base-two fraction part and an exponent.

for – a C++ *keyword* used to specify an iteration or looping *statement*.

forward class – a *class* for which only the *tag* has been declared. Such a class can be used where the size of the class is not needed, for example in *pointer declarations*.

free store – see *dynamic storage*.

friend – a type of *declaration* used within a *class* to grant other classes or functions access to that class. See *access control*.

front end – often refers to the early stages of C++ compilation, such as *parsing* and *semantic analysis*.

function – a C++ entity that is a sequence of *statements*. It has its own *scope*, accepts a set of *argument* values, and returns a value on completion.

function template – a *template* used for generating *function types*.

fundamental type – a *type* built in to the C++ language. Examples would be integral types like *int* and *pointer* types such as *void**.

G

garbage collection – a way of automatically managing *dynamic storage* such that explicit cleanup of storage is not required. C++ does not have garbage collection. See *new operator* and *delete operator*.

generic programming – see *template*.

global name – a name declared at *global scope*.

global namespace – the implicit *namespace* where *global variables* reside.

global scope – see *global namespace*.

global variable – a *variable* that is accessible throughout the whole program, whose *lifetime* is that of the program.

goto – C++ *keyword*, used to transfer control within a C++ *function*. See *label*.

grammar – a way of expressing the *syntax* of a programming language, to describe exactly what usage is valid and invalid.

H

header – see *header file*.

header file – a file containing *class* declarations, *preprocessor* directives, and so on, and included in a *translation unit*. It is expanded by the *preprocessor*.

heap storage – see *dynamic storage*.

helper class – a *class* defined as part of implementing the details of another class.

hiding – see *encapsulation*.

I

if – C++ *keyword* used in conditional *statements*.

implementation–dependent behavior – not every aspect of a programming language like C++ is specified in a language standard. This term refers to behavior that may vary from implementation to implementation.

implicit conversion – a *conversion* done as part of another operation, for example converting a *pointer type* to *bool* in an *if* statement.

inheritance – the process whereby a *derived class* inherits *members* from a *base class*. A derived class will also add its own members to those of the base class.

initialization – to give an initial value to an *object*. See *constructor* and *assignment*.

initialize – the process of *initialization*.

initializer – a value or *expression* used to initialize an *object* during *initialization*.

inline – C++ *keyword* used to declare an *inline function*.

inline function – a *function* that can be expanded by a *compiler* at the point of call, thereby saving the overhead time required to call the function.

instantiation – see *template instantiation*.

int – a C++ *keyword* and *fundamental type*, used to declare an *integral type*.

integral conversion – the process by which an integer is converted to *signed* or *unsigned*.

integral promotion – the process by which a *bool*, *char*, *short*, *enumerator*, or *bit field* are converted to *int* for use in expressions, *argument* passing, and so on.

K

keyword – a reserved identifier in C++, used to denote data *types*, *statements* of the language, and so on.

L

label – a *name* that is the target of a *goto statement*.

layout – refers to the way that *objects* are arranged in memory.

library – a set of *object files* grouped together. A *linker* will search them repeatedly and use whatever object files are needed. See *class library*.

lifetime – refers to the duration of the existence of an *object*. Some objects last for the whole execution of a program, while other objects have a shorter lifetime.

linkage – refers to whether a *name* is visible only inside or also outside its *translation unit*.

linker – a program that combines *object files* and *library* code to produce an executable program.

literal – a *constant* like 1234.

local – typically refers to the *scope* and *lifetime* of *names* used in a *function*.

local class – a *class* declared *local* to a *function*.

local variable – a *variable* declared *local* to a *function*.

long – C++ *keyword* used to declare a long integer data *type*.

long double – a *floating point type* in C++.

lvalue – an *expression* referring to an *object*. See *rvalue*.

M

macro – a *preprocessor* feature that supports parameter substitution and expansion of commonly-used code sequences. See *inline function*.

mangling – see *name mangling*.

member – see *class member* and *namespace member*.

member function – a *function* that is an element of a *class* and that operates on *objects* of that class via the *this* pointer to the object.

memberwise copy – to copy an *object* a *member* at a time, taking into account a *copy constructor* for the member. See *bitwise copy*.

method – see *member function*.

mixed-mode arithmetic – mixing of integral and *floating point* arithmetic.

module – see *translation unit*.

multiple inheritance – a *derived class* with multiple *base classes*. See *inheritance*.

mutable – C++ *keyword* declaring a *member* non-constant even if it is a member of a *const object*.

N

name – an identifier that denotes an *object*, *function*, a set of *overloaded functions*, a *type*, an *enumerator*, a *member*, a *template*, a *namespace*, or a *label*.

name lookup – refers to taking a *name* and determining what it refers to, or its value, based on the *scope* and other rules of C++.

name mangling – a way of encoding an *external name* representing a *function* so as to be able to distinguish the *types* of its *parameters*. See *overload*.

name space – a grouping of *names*.

namespace – a C++ *keyword* used to declare a namespace, which is a collection of *names* such as *function* declarations, *classes*, and so on.

namespace alias – an alias for a *namespace*, that can be used to refer to the namespace.

namespace member – an element of a *namespace*, such as a *function*, *typedef*, or *class declaration*.

nested class – a *class declaration* nested within another *class*.

new handler – a *function* established by calling *set_new_handler*. It is called when the *new operator* cannot obtain *dynamic storage*.

new operator – C++ *keyword* and *operator* used to allocate *dynamic storage*.

new-style cast – a *cast* written in functional notation.

new[] operator – see *new operator*. Used to allocate *dynamic storage* for *array* objects.

NULL – a special constant value that represents a *null pointer*.

null pointer – a *pointer* value that evaluates to zero.

O

object – has several meanings. In C++, often refers to an instance of a *class*. Also more loosely refers to any named *declaration* of a *variable* or other entity that involves storage.

object file – in C or C++, typically the output of a *compiler*. An object file consists of machine language plus an *external name* list that is resolved by a *linker*.

object layout – refers to the ordering of data members within a *class*.

object-oriented – this term has various definitions, usually including the notions of *derived classes* and *virtual functions*. See *data abstraction*.

old-style cast – a *cast* written in C style, with the *type* in parentheses before the value being casted.

OOA / OOD – acronym for *object-oriented* analysis and *object-oriented* design, processes of analyzing and designing *object-oriented* software.

OOP – acronym for *object-oriented* programming.

operator – a builtin operation of the C++ language, like addition, or an *overloaded* operator corresponding to a *member function* of a *class*. See *function* and *operator overloading*.

operator overloading – to treat a C++ operator like `<<` as a *function* and *overload* it for particular *parameter types*.

overload – to specify more than one *function* of the same *name*, but with varying numbers and *types of parameters*. See *argument matching*.

overload resolution – see *argument matching*.

P

parameter – refers to the *variables* passed into a *function*. See also *argument*.

parameterized type – see *template*.

parser – see *parsing*.

parsing – the process by which a program written in some programming language is broken down into its syntactic elements.

placement – the ability to define a variant of the *new operator* to take an additional *argument* that specifies what storage is to be used.

pointer – an address of an *object*.

pointer to data member – a *pointer* that points at a data member of a *class*.

pointer to function – an address of a *function* or a *member function*.

pointer to member – see *pointer to data member*, *pointer to function*.

polymorphism – the ability to call a variety of *member functions* for a given *class object* using an identical interface in each case. See *virtual function*.

postfix – refers to *operators* that appear after their operand. See *prefix*.

pragma – a *preprocessor* directive used to affect *compiler* behavior in an implementation-defined way.

prefix – refers to *operators* that appear before their operand. See *postfix*.

preprocessing – a stage of compilation processing that occurs before the *compiler* proper is invoked. Preprocessing handles *macro* expansion among other things. In C++ use of *const* and *inline functions* makes preprocessing less important.

preprocessor – see *preprocessing*.

private – a C++ *keyword* used to specify that a *class member* can only be accessed from *member functions* and *friends* of the class. See *access control*, *protected*, and *public*.

programming environment – a set of integrated tools used in developing software, including a *compiler*, *linker*, *debugger*, and *browser*.

promotion – see *integral promotion*.

protected – a C++ *keyword* used to specify that a *class member* can only be accessed by *member functions* and *friends* of its own *class* and by *member functions* and *friends* of classes derived from this class. See *private*, *public*, and *access control*.

PT – see *parameterized type*.

public – a C++ *keyword* used to specify that *class members* are accessible from any (non–member) *function*. See *access control*, *protected*, and *private*.

pure virtual function – a *virtual function* with a "*= 0*" initializer. See *abstract class*.

Q

qualification – to prefix a *name* with the name of a *class* or *namespace*.

R

recursive descent parser – see *parsing*. This is a type of parsing used in C++ compilers. It is more flexible than the older Yacc approach often used in C compilers.

reference – another name for an *object*. Access to an object via a reference is like manipulating the object itself. References are typically implemented as *pointers* in the underlying generated code.

register – C++ *keyword* used as a hint to the *compiler* that a particular *local variable* should be placed in a machine register.

reinterpret_cast – a C++ *keyword* used as a style of *cast* for performing unsafe and implementation dependent casts.

repository – a location where an instantiated *template class* can be stored. See *template instantiation*.

resolution – see *overload resolution*.

resumption – a style of *exception handling* where program execution continues from the point where an *exception* is *thrown*. C++ uses the *termination* style.

return – C++ *keyword* used for returning values from a *function*.

return value – the value returned from a *function*.

RTTI – acronym for *run-time type information*.

run-time – refers to actions that occur during program execution.

run-time efficiency – refers to the issue of whether basic C++ operations will cause a performance penalty when the program is run.

run-time type information – a system for determining at *run-time* what the *type* of an *object* is.

rvalue – a value that may appear on the right-hand side of an *assignment*.

S

scope – the region of a program where a *name* has *visibility*.

semantic analysis – a stage that a *compiler* goes through after *parsing*. In this stage the meaning of the program is analyzed.

semantics – the meaning of a program, as opposed to its *syntax*.

separate compilation – refers to the process by which each *translation unit* of a program is compiled separately to produce an *object file*. The object files are then combined by a *linker*.

set_new_handler – a *function* used to establish a *new handler*.

short – a C++ *fundamental type* used to declare small integers.

signed – C++ *keyword* used to indicate a signed data type.

sizeof – C++ *keyword* for taking the size of an *object* or *type*.

smart pointer – an *object* that acts like a *pointer* but also does some processing whenever an object is accessed through them. The C++ *operator* `->` can be *overloaded* to achieve this effect.

specialization – a special case of a *template* defined for particular *template argument* types.

stack frame – refers to a region of storage on the hardware stack, used to store information such as *local variables* for each invocation of a *function*.

stack unwinding – see *exception handling*. When an exception is thrown, each active *stack frame* must be removed from the stack until an *exception handler* is found. This process involves calling a *destructor* as appropriate for each local *object* in the stack frame, and so on.

standard conversion – refers to standardized conversions between *types*, such as *integral conversion*.

standard library – see *library*. The C++ standard library includes much of the C standard library along with new features such as strings and *container class* support.

statement – the parts of a program that actually do the work.

static – see *static member*, *static object*, and *static storage*.

static member – a *class member* that is part of a *class* for purposes of *access control* but does not operate on particular *object* instances of the class.

static object – an *object* that is *local* to a *function* or to a *translation unit* and whose *lifetime* is the life of the program.

static storage – storage that persists throughout the life of the program. See *static object* and *dynamic storage*.

static type checking – refers to *type checking* that occurs during compilation of a program rather than at *run-time*.

static_cast – a C++ *keyword* specifying a style of *cast* meant to replace old-style C casts.

storage class – see *auto* and *static*.

stream – an *object* used to represent an input or output channel. See *stream I/O*.

stream I/O – a C++ *I/O library* using overloaded operators << and >>. It has more *type safety* than C-style I/O.

string – see *C-style string*.

struct – a C++ *class* in which all the class *members* are by default *public*.

switch – C++ *keyword* denoting a *statement* type, used to dispatch to one of several sequences of statements based on the value of an *expression*.

symbol table – a *compiler* structure used to record *type* information about program *names*. The symbol table is used to generate compiler output.

syntax – the rules that govern how C++ *expressions*, *statements*, *declarations*, and programs are constructed. See *grammar* and *semantics*.

systems programming – refers to low-level programming, for example writing I/O drivers or operating systems. C and C++ are suitable languages for this type of programming.

T

tag – a name given to a *class*, *struct*, or *union*.

template – a parameterized *type*. A template can accept *type parameters* that are used to customize the resulting type.

template argument – an actual value or *type* given to a *template* to form a *template class*. See *argument*.

template class – a combination of a *template* with a *template argument* list via the process of *template instantiation*.

template declaration – a *declaration* of a *template* with its associated *template parameter* list.

template definition – an actual definition of a *template* or one of its *members*.

template instantiation – the process of combining *template arguments* with a *template* to form a *template class*.

template parameter – a value or *type* declared to be passed in to a *template*. See *parameter*.

temporary – an unnamed *object* used during the evaluation of an *expression* to store intermediate values.

termination – a style of *exception handling* where control does not return to the point where an *exception* is *thrown*. C++ uses this style of exception handling.

this – C++ *keyword* used in a *member function* to point at the *object* currently being operated on.

throw – C++ *keyword* used to throw (initiate) an *exception*. See *exception handling*.

translation limit – a limit on the size of a source program that a *compiler* will accept.

translation unit – a source file presented to a *compiler* with an *object file* produced as a result.

trigraph – a sequence of characters used to represent another character, for example to represent a character not normally found in the character set.

true – C++ *keyword* used to specify a value for the *bool* type.

try – C++ *keyword* used to delimit a *try block*.

try block – a *statement* that sets up a context for *exception handling*. A subsequent *throw* from a *function* called from within the try block will be caught by the *exception handler* associated with the try block or by a handler further out in the chain of handlers.

type – a property of a *name* that determines how it can be used. For example, an *object* of a *class* type cannot be assigned to an integer *variable*.

type checking – see *type system*.

type conversion – converting a value from one *type* to another, for example via a *constructor*.

type safety – see *type system*.

type system – a system of *types* and operations on *objects* of those types. Type checking is done to ensure that the operations for given types are appropriate, for example that a *function* is called with *arguments* of the appropriate types.

type-safe linkage – refers to the process of encoding *parameter type* information in *external names* so that the *linker* will reject mismatches between the use and definition of *functions*. See *name mangling*.

typedef – a C++ *keyword* used to declare an alias for a *type*.

typeid – an *operator* that returns an object describing the *type* of the operand. See *run-time type information*.

U

union – a structure somewhat like a *class* or *struct*, except that individual union members share the same memory. See *class layout*.

unsigned – a C++ *keyword* used to declare an integral unsigned *fundamental type*.

unwinding – see *stack unwinding*.

user-defined conversion – a *member function* that supports *conversion* from an *object* of *class type* to any target type.

user-defined type – a *class* or *typedef*.

using declaration – a *declaration* making a *class* or *namespace name* available in another *scope*.

using directive – a way of making available to a program the members of a *namespace*.

using namespace – see *using directive*.

V

variable – an *object* that can be assigned to.

vector – a one-dimensional *array*.

virtual base class – a *base class* where a single subobject of the base class is shared by every *derived class* that declared the base class as virtual.

virtual function – a *member function* whose interpretation when called depends on the *type* of the *object* for which it is called; a function for an object of a *derived class* will override a function of its *base class*.

virtual table – a lookup table used for dispatching *virtual function* calls. A *class object* for a *class* containing virtual functions will contain a pointer to a virtual table.

visibility – refers to the processing of doing *name lookup* without regard to whether a *name* is accessible. Once a name is found, then *type checking* and *access control* are applied.

void – a C++ *keyword* used to declare no *type*. It has special uses in C++, for example to declare that a *function* has no *parameter* list. See also *void**.

void* – a pointer to a *void* type. Often used as the lowest common denominator type of pointer in C and C++.

volatile – a *type* qualifier used to indicate that an *object* may unpredictably change value (for example if it is mapped to a machine register) and thus should not have accesses to it optimized.

W

wchar_t – C++ *keyword* to declare a *fundamental type* used for handling wide characters.

while – C++ *keyword* used to declare an iteration *statement*.

APPENDIX

PUNCTUATION MARKS AND SPECIAL SYMBOLS

Symbol	Name	Symbol	Name
,	comma	{	left curly bracket
.	point	}	right curly bracket
;	semicolon	<	greater
:	colon	>	less
?	question mark	[left square bracket
'	apostrophe]	right square bracket
!	exclamation mark	#	number or grid
	vertical bar	%	percent
/	slash	&	ampersand
\	backslash	^	logical NOT
~	tilde	-	minus
*	asterisk	=	equals sign
(left round bracket	“	quotation mark
)	right round bracket	+	plus

CONTROL SEQUENCES

Control sequence	Name
\a	Ring
\b	Step back
\t	Horizontal tabulation
\n	Line feed
\v	Vertical Tab
\r	Carriage return
\f	Form feed
\"	Quotation marks
\'	Apostrophe
\\	Backslash

DATA TYPES

Type	Length, bytes	Range	Decimal digits
signed char	1	-128 ... 127	–
unsigned char	1	0 ... 255	–
int	2	-32 768 ... 32 767	–
unsigned int	2	0 ... 65 535	–
long	4	0 ... 4 298 876 555	–
float	4	3.4e-38 ... 3.4e38	7
double	8	1.7e-308 ... 1.7e308	15
long double	10	3.4e-4932 ... 1.1e4932	19

OPERATORS PRECEDENCE AND THE EXECUTION ORDER

Priority	Operator	Note	Execution order
1	:: -> .	context resolution, extraction	left-to-right
	[]	array indexing	left-to-right
	()	function call	left-to-right
	()	type conversion	left-to-right
2	++ -- ~ !	increment, decrement, complement, not	right-to-left
	- +	unary - unary +	right-to-left
	&	address of	right-to-left
	*	pointer resolving	right-to-left
	new, delete	create and destroy	right-to-left
	sizeof	size of object	right-to-left
3	*	multiplication	left-to-right
	/	division	left-to-right
	%	remainder	left-to-right
4	->* .*	extraction	left-to-right
5	+	binary addition	left-to-right
	-	binary subtraction	left-to-right
6	<< >>	shifts	left-to-right
7	< <= > =>	comparison	left-to-right
8	== !=	equal not equal	left-to-right
9	&	bitwise AND	left-to-right
10	^	XOR (excluding OR)	left-to-right
11		bitwise OR	left-to-right
12	&&	AND-logical	left-to-right
13		OR-logical	left-to-right
14	? :	ternary operator	right-to-left
15	= *= /= %= += so on	assignment operators	right-to-left
16	,	sequencing	left-to-right

C++ KEYWORDS

<i>and</i>	<i>and_eq</i>	<i>asm</i>	<i>auto</i>	<i>bitand</i>	<i>bitor</i>
<i>bool</i>	<i>break</i>	<i>case</i>	<i>catch</i>	<i>char</i>	<i>class</i>
<i>compl</i>	<i>const</i>	<i>const_cast</i>	<i>continue</i>	<i>default</i>	<i>delete</i>
<i>do</i>	<i>double</i>	<i>dynamic_cast</i>	<i>else</i>	<i>enum</i>	<i>explicit</i>
<i>export</i>	<i>extern</i>	<i>false</i>	<i>float</i>	<i>for</i>	<i>friend</i>
<i>goto</i>	<i>if</i>	<i>inline</i>	<i>int</i>	<i>long</i>	<i>mutable</i>
<i>namespace</i>	<i>new</i>	<i>not</i>	<i>not_eq</i>	<i>operator</i>	<i>or</i>
<i>or_eq</i>	<i>private</i>	<i>protected</i>	<i>public</i>	<i>register</i>	<i>reinterpret_cast</i>
<i>return</i>	<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>	<i>static_cast</i>
<i>struct</i>	<i>switch</i>	<i>template</i>	<i>this</i>	<i>throw</i>	<i>true</i>
<i>try</i>	<i>typedef</i>	<i>typeid</i>	<i>typename</i>	<i>union</i>	<i>unsigned</i>
<i>using</i>	<i>virtual</i>	<i>void</i>	<i>volatile</i>	<i>wchar_t</i>	<i>while</i>
<i>xor</i>	<i>xor_eq</i>				

STANDARD FUNCTIONS

<i>abort</i>	<i>fmod</i>	<i>isupper</i>	<i>mktime</i>	<i>strftime</i>	<i>wctomb</i>
<i>abs</i>	<i>fopen</i>	<i>iswalnum</i>	<i>modf</i>	<i>strlen</i>	<i>wcscat</i>
<i>acos</i>	<i>fprintf</i>	<i>iswalpha</i>	<i>perror</i>	<i>strncat</i>	<i>wcschr</i>
<i>asctime</i>	<i>fputc</i>	<i>iswcntrl</i>	<i>pow</i>	<i>strncmp</i>	<i>wcscmp</i>
<i>asin</i>	<i>fputs</i>	<i>iswctype</i>	<i>printf</i>	<i>strncpy</i>	<i>wcscoll</i>
<i>atan</i>	<i>fputwc</i>	<i>iswdigit</i>	<i>putc</i>	<i>strpbrk</i>	<i>wcscpy</i>
<i>atan2</i>	<i>fputws</i>	<i>iswgraph</i>	<i>putchar</i>	<i>strrchr</i>	<i>wcscspn</i>
<i>atexit</i>	<i>fread</i>	<i>iswlower</i>	<i>puts</i>	<i>strspn</i>	<i>wcsftime</i>
<i>atof</i>	<i>free</i>	<i>iswprint</i>	<i>putwc</i>	<i>strstr</i>	<i>wcslen</i>
<i>atoi</i>	<i>freopen</i>	<i>iswpunct</i>	<i>putwchar</i>	<i>strtod</i>	<i>wcsncat</i>
<i>atol</i>	<i>frexp</i>	<i>iswspace</i>	<i>qsort</i>	<i>strtok</i>	<i>wcsncmp</i>
<i>bsearch</i>	<i>fscanf</i>	<i>iswupper</i>	<i>raise</i>	<i>strtol</i>	<i>wcsncpy</i>
<i>btowc</i>	<i>fseek</i>	<i>iswxdigit</i>	<i>rand</i>	<i>strtoul</i>	<i>wcspbrk</i>
<i>calloc</i>	<i>fsetpos</i>	<i>isxdigit</i>	<i>realloc</i>	<i>strxfrm</i>	<i>wcsrchr</i>
<i>ceil</i>	<i>ftell</i>	<i>labs</i>	<i>remove</i>	<i>swprintf</i>	<i>wcsrtombs</i>
<i>clearerr</i>	<i>fwide</i>	<i>ldexp</i>	<i>rename</i>	<i>swscanf</i>	<i>wcsspn</i>
<i>clock</i>	<i>fwprintf</i>	<i>ldiv</i>	<i>rewind</i>	<i>system</i>	<i>wcsstr</i>
<i>cos</i>	<i>fwrite</i>	<i>localeconv</i>	<i>scanf</i>	<i>tan</i>	<i>wcstod</i>
<i>cosh</i>	<i>fwscanf</i>	<i>localtime</i>	<i>setbuf</i>	<i>tanh</i>	<i>wcstok</i>
<i>ctime</i>	<i>getc</i>	<i>log</i>	<i>setlocale</i>	<i>time</i>	<i>wcstol</i>
<i>difftime</i>	<i>getchar</i>	<i>log10</i>	<i>setvbuf</i>	<i>tmpfile</i>	<i>wcstombs</i>
<i>div</i>	<i>getenv</i>	<i>longjmp</i>	<i>signal</i>	<i>tmpnam</i>	<i>wcstoul</i>
<i>exit</i>	<i>gets</i>	<i>malloc</i>	<i>sin</i>	<i>tolower</i>	<i>wcsxfrm</i>
<i>exp</i>	<i>getwc</i>	<i>mblen</i>	<i>sinh</i>	<i>toupper</i>	<i>wctob</i>
<i>fabs</i>	<i>getwchar</i>	<i>mbrlen</i>	<i>sprintf</i>	<i>towctrans</i>	<i>wctomb</i>
<i>fclose</i>	<i>gmtime</i>	<i>mbrtowc</i>	<i>sqrt</i>	<i>towlower</i>	<i>wctrans</i>
<i>feof</i>	<i>isalnum</i>	<i>mbsinit</i>	<i>srand</i>	<i>towupper</i>	<i>wctype</i>
<i>ferror</i>	<i>isalpha</i>	<i>mbsrtowcs</i>	<i>sscanf</i>	<i>ungetc</i>	<i>wmemchr</i>
<i>fflush</i>	<i>iscntrl</i>	<i>mbstowcs</i>	<i>strcat</i>	<i>ungetwc</i>	<i>wmemcmp</i>
<i>fgetc</i>	<i>isdigit</i>	<i>mbtowc</i>	<i>strchr</i>	<i>vfprintf</i>	<i>wmemcpy</i>
<i>fgetpos</i>	<i>isgraph</i>	<i>memchr</i>	<i>strcmp</i>	<i>vfwprintf</i>	<i>wmemmove</i>
<i>fgets</i>	<i>islower</i>	<i>memcmp</i>	<i>strcoll</i>	<i>vprintf</i>	<i>wmemset</i>
<i>fgetwc</i>	<i>isprint</i>	<i>memcpy</i>	<i>strcpy</i>	<i>vsprintf</i>	<i>wprintf</i>
<i>fgetws</i>	<i>ispunct</i>	<i>memmove</i>	<i>strcspn</i>	<i>vswprintf</i>	<i>wscanf</i>
<i>floor</i>	<i>isspace</i>	<i>memset</i>	<i>strerror</i>	<i>vwprintf</i>	

REFERENCES

1. Krogstie J., Halpin T., and Siau F. Information Modeling Methods and Methodologies. – Hershey, PA: Idea Group Publishing, 2005. – 356 p.
2. Bergeron, B.P. Essentials of knowledge management. – Hoboken, NJ: John Wiley & Sons, 2003. – 225 p.
3. Knowledge Management: Current Issues and Challenges / E. Coakes (ed). – Idea Group Publishing, 2003. – 303 p.
4. Oualline S. Practical C++ Programming, Second Edition. – O'Reilly, 2003.– 574 p.
5. Stroustrup B. The C++ Programming Language, Third Edition. – Addison-Wesley, 1997. – 1022 p.
6. Lippman S. B., Lajoie J. C++ Primer, 4rd Edition. – Addison-Wesley, 2005. – 912 p.
7. Press W. H., Teukolsky S. A., Vetterling W. T. Numerical recipes in C++: the art of scientific computing. – New York: Cambridge University Press, 2002. – 318 p.
8. Programming languages – C++. International Standard ISO/IEC 14882:1998(E). – New York: American National Standards Institute, 1998. – 776 p.
9. Cline M. P. C++ FAQs, 2nd Edition. – Addison-Wesley, 1998. – 624 p.
10. Davis S. R. C++ for Dummies. – For Dummies, 2004. – 432 p.
11. Sutter. H. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions.– Addison Wesley, 1999. – 240 p.
12. Dale N.B. C++ plus Data Structure. Sudbury: Jones and Bartlett Publishers, 2003. – 820 p.
13. Josuttis N.M. C++ Standard Library: A Tutorial and Reference, The. – Addison Wesley, 1999. – 832 p.
14. Vandevoorde D., Josuttis N.M. C++ Templates: The Complete Guide. – Addison Wesley, 2002. – 552 p.
15. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied. – Addison Wesley, 2001. – 352 p.

16. Gromov G.R. The essays of information technology [in Russian]. – M.: InfoArt, 1990. – 254 p.
17. Introduction into the information business: Textbook [in Russian] / Edited by V.P. Tikhomirov, A.V. Khoroshilov. – M.: Finansy i Statistika, 1996. – 320 p.
18. Tsimbal A.A., Maiorov A.G., Kozodoev M.A. Turbo C++: Language and Application [in Russian]. – M.: “Jan I Ltd”, 1993. – 512 p.
19. Klimova L.M. Fundamentals of Hands-on Programming in C++ [in Russian]. – M.: Prior, 1999. – 464 p.
20. Karpov B., Baranova. T. C++: Special Reference Manual [in Russian]. SPb: Piter, 2001. – 480 p.
21. <http://ru.wikipedia.org/wiki/C++>; <http://en.wikipedia.org/wiki/C++>
22. <http://www.cplusplus.com>
23. <http://www.cprogramming.com>
24. <http://www.intap.net/~drw/cpp>
25. <http://cyberdiem.com/vin/learn.html>
26. <http://www.oonumerics.org/blitz>
27. <http://www.boost.org>
28. <http://www.awprofessional.com/meyerscddemo/demo/magazine/index.htm>
29. http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_toc.html
30. http://www.josuttis.com/libbook/auto_ptr.html
31. <http://www.osl.iu.edu/research/mtl>
32. <http://www.cantrip.org/traits.html>
33. http://www.robertnz.com/nm_intro.htm
34. <http://www.pooma.com>
35. <http://www.research.att.com/~bs/glossary.html>
36. <http://www.erwin-unruh.de/primorig.html>
37. <http://osl.iu.edu/~tveldhui/papers>
38. <http://reality.sgi.com/austern/std-c++/faq.html>
39. <http://www.dinkumware.com/refxcpp.html>
40. <http://www.cyberport.com/~tangent/programming/stl/resources.html>

Educational Edition

Томский политехнический университет

Лопаткин Сергей Анатольевич

Рейзлин Валерий Израилевич

КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ В НАУКЕ И ОБРАЗОВАНИИ

Учебное пособие

Издательство Томского политехнического университета, 2008

На английском языке

Science Editor

Doctor of Physics and Mathematics,
Professor

V. K. Pogrebnoy

Typesetting

V. P. Arshinova

Cover design


*O. Yu. Arshinova
O. A. Dmitriev*

Signed for the press 03.09.2008. Format 60x84/16. Paper "Snegurochka".
Print XEROX. Arbitrary printer's sheet 13.67. Publisher's signature 12.36.
Order 79•. Size of print run •00.



Tomsk Polytechnic University
Quality management system
of Tomsk Polytechnic University was certified by
NATIONAL QUALITY ASSURANCE on ISO 9001:2000



TPU  **PUBLISHING HOUSE.** 30, Lenina Ave, Tomsk, 634050, Russia