Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра информационных технологий автоматизированных систем

А. А. Навроцкий, А. Б. Гуринович

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ALGORITHMS AND DATA STRUCTURES

*Допущено Министерством образования Республики Беларусь
в качестве учебного пособия для иностранных студентов
учреждений высшего образования по специальности
«Автоматизированные системы обработки информации»*

Минск БГУИР 2022

УДК [004.42:004.4’6](075.8)
ББК 32.972.13я73
Н15

Р е ц е н з е н т ы:
кафедра технологий программирования
Белорусского государственного университета
(протокол № 11 от 21.04.2022);

кафедра межкультурных коммуникаций и технического перевода учреждения
образования «Белорусский государственный технологический университет»
(протокол № 9 от 28.04.2022);

доцент кафедры высшей математики учреждения образования
«Военная академия Республики Беларусь»
кандидат физико-математических наук, доцент Т. А. Макаревич

Навроцкий, А. А.
Н15 Основы алгоритмизации и программирования = Algorithms and Data Structures : учеб. пособие / А. А. Навроцкий, А. Б. Гуринович. – Минск : БГУИР, 2022. – 152 с.
ISBN 978-985-543-640-0.

Содержит теоретические сведения о языке С++. Рассмотрены примеры написания программ в среде Microsoft Visual Studio C++. Представлены задания для лабораторных работ по дисциплине «Основы алгоритмизации и программирования».

Может быть полезно ИТ-инженерам, научным работникам, преподавателям, специалистам, самостоятельно изучающим основы алгоритмизации и программирования.

УДК [004.42:004.4’6(075.8)
ББК 32.972.13я73

CONTENT

THEORY PART

1. C++ language Basic Elements

The C++ alphabet contains uppercase (capital) and lowercase letters of the Latin alphabet, Arabic numbers, special characters, spaces, and separators.

The lexemes (elementary language constructions) are formed by alphabet characters. The lexemes include: identifiers, reserved words, operation signs, constants, separators.

1.1. Identifiers

Identifier is a sequence of digits and letters of the Latin alphabet, and special symbols, but the first one is a letter or an underline. Two identifiers with matching uppercase and lowercase letters are considered different.

▨	Note: aaa, Aaa, aAa, AaA are four different identifiers

Any number of characters can be used in an identifier, but only the first 32 are considered significant ones.

When choosing the identifier it is necessary:

– to make sure that the identifier does not coincide with key reserved words and names of library functions;

– to use with caution the underscore as the first character of an identifier and the combination "_t" at the end of an identifier, since such identifiers were reserved by the ANSI C standard for use by compiler developers.

For identifiers it is desirable to adhere to the following commonly accepted agreements:

– names of variables and functions are written with lowercase letters;
– names of types begin with uppercase letter;
– names of constants are written with uppercase letters.

The identifiers names usually conform with the internal essence of the object.

1.2. Keywords

The keywords list is defined by the ANSI C standard: **auto, double, int, struct, break, long, switch, register, typedef, char, extern, return, void, case, float, unsigned, default, for, signed, union, do, if, sizeof, else, while, volatile, continue, enum, short**.

1.3. Comments

A comment is the textual or symbolic information. A comment is used to explain sections of the program. The comments do not affect the execution of the program because they are not tokens (lexemes) and are not included in the contents of the executable file.

A comment is text. The compiler ignores this text but it is useful for programmers. Comments are normally used to annotate code for future reference.

A C++ comment is written in one of the following ways:

– the **/*** (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the ***/** characters;

– the // (two slashes) characters, followed by any sequence of characters and end with the end of the line. Therefore, it is commonly called a "single-line comment".

1.4. Operation Symbols

The operation symbols is one or more symbols used to define the action on the operands. The spaces are not allowed inside the operation character.

1.5. C++ Program Structure

The C ++ program consists of one or more functions. The presence of the **main()** function is mandatory to transfer the control while the program starts.

The simplified program structure is as follows:

< *The preprocessor directives* >
< *The user types description* >
< *The function prototypes* >
< *The global variables description* >
< *The function bodies* >

1.6. Preprocessor Directives

A preprocessor is a special part of the compiler to process directives before compiling a program. A preprocessor directive starts with a **#**. It must be the first character in a string. It is followed by the name of the directive. There is no semicolon at the end of the directive. To transfer the directive to the next string it is used the symbol '\'.

The **include** directive is used to connect header files to the program.

The file will be searched in the standard directory if the file identifier is enclosed in angle brackets. If the file identifier is enclosed in the quotation mark, then the search is performed in the following order:

- the directory is containing the file containing the directive;
- the files directories have already been included by the directive;
- the program current directory;
- the directories are specified by the compiler option '**\I**';
- the directories are specified by the **include** environment variable.

Processing the **include** directive by the preprocessor is reduced to the fact that the specified in the directive copy file is placed in the directive place.

The define directive is used to define symbolic constants. For example, if it is defined at the beginning of the program:

```
#define PI  3.14159265359
```

then, in the whole text, during compilation the **PI** identifier will be replaced by the text 3.14159265359. The constant identifier replacement is not performed in comments and in lines. If the alternative text is not specified in the directive, then the corresponding identifier is erased in the entire text.

> The constants description using preprocessor directives is characteristic of the C language. It is recommended to use the keyword **const** in C ++.
> For example:
>
> **const double** pi = 3.14159265359;

The **define** directive is also used for writing macros

```
#define name(parameters)implementation
```

The macros name is replaced with the string of its implementation in the program. For example, there is the following macro definition:

```
#define MAX(A,B) ((A)>(B)?(A):(B))
```

If the program contains the line

```
s = MAX(a,b),
```

Then each macro is replaced by a macro definition before compilation:

```
s = ((a)>(b)?(a):(b));
```

It is better to place each parameter in brackets since their absence can provoke the error. For example, a macro is created:

```
#define SQR(A) (A*A)
```

It is used in a program as

```
s = SQR(a + b);
```

The comprising error line will be formed:

```
s = a + b * a + b;
```

It should be written as follows

```
#define SQR(A) ((A) * (A))
```

then, the line will look like this:

```
s = (a + b) * (a + b);
```

The **#undef** directive is used to cancel the effect of the **#define** directive. This directive syntax is:

```
#undef идентификатор
```

For example: #undef MAX

Directives can also be used for conditional compilation and to change line numbers and file ID.

1.7. C++ Standard Libraries

The files from the library are connected to the source program files when creating an executable file. Usually these files contain already compiled functions and have the ***lib*** extension. During the compilation the linker extracts the used in the program functions from the library files. To connect with the library file a header file is connected to the program.The header file contains information about the names and the types of functions from the library. Header files are included using pre-processor directives.

	The standard C ++ header files do not have an extension. For the inherited from C files extension should be specified. For example, #include <cmath>

1.8. The cmath Library

All arguments in trigonometric functions are in radians. The parameters of the rest of the functions are of the **double** type. Some mathematical functions are listed in tab. 1.1.

Table 1.1

Math function	Library function math.h	Calculation content
1	2	3

|x|	abs(x)	Calculating the absolute value of a number.
		For example:
		s = abs(−3) → The result is $s = 3$
		s = abs(3) → The result is $s = 3$
		s = abs(−3.9) → The result is $s = 3$
		s = abs(3.2) → The result is $s = 3$

arccos(x)	acos(x)	Calculating the value of the arc cosine of the number x. The x value can only be specified in the range −1...1. In The result is the execution of the function, a value from the range −π/2...π/2 is returned.
		For example:
		s = acos (−1) → The result is $s = 3.14159$
		s = acos (0.4) → The result is $s = 1.15928$
		s = acos (1.5) → The result is s = −1.#IND

arcsin(x)	asin(x)	Calculating the value of the inverse sine of x. The x value can only be specified from the range –1...1. The result is the execution of the function, a value from the range 0…π is returned.
		For example:
		s = asin (−1) → The result is $s = -1.5708$
		s = asin (0.9) → The result is $s = 1.11977$

1	2	3
arctg(x)	atan(x)	Calculating the value of the arctangent. The result is the execution of the function returns a value from the range π/2... π/2. For example: x = atan (3.5) → The result is s = 1.2925

arctg(x/y)	atan2(x, y)	Calculating the value of the arc tangent of two arguments. The result is of executing the function, a value from the range −π…π is returned. If x is 0, then the function returns π/2 if x > 0; 0 if x = 0; −π/2 if x < 0.
		For example:
		s = atan2(4.5, 9.2) → The result is s = 0.454914
		s = atan2(−7.3, 0) → The result is s = −1.5708

Rounding to greater	ceil(x)	The function returns the smallest integer value greater than or equal to x (return ceiling value of number).
		For example:
		s = ceil(−3.4) → The result is s = −3
		s = ceil(3.4) → The result is s = 4

$\sqrt[3]{x}$	cbrt(x)	Returns the cube root value of x
cos(x)	cos(x)	Calculation cos(x)
ch(x)	cosh(x)	Calculating the hyperbolic cosine
e^x	exp(x)	Calculating the exponent of x
$\mid x \mid$	fabs(x)	Calculating the absolute value of x

Rounding to smaller	floor(x)	The function returns the larger integer value less than or equal to x (returns floor value of decimal number).
		For example:
		s = floor (−3.4) → The result is s = −4
		s = floor (3.4) → The result is s = 3

At least	fmin (x, y)	Calculation of the minimum value from x and y
At most	fmax (x, y)	Calculation of the maximum value from x and y

Remaining from dividing x by y	fmod(x,y)	Function returns the valid value corresponding to the remainder of division x on y. Computes floating point remainder of division.
		For example:
		s = fmod (3, 4) → The result is s = 3
		s = fmod (6.4, 3.1) → The result is s = 0.2

ln(x)	log(x)	Returns natural logarithm of x
$lg_{10}(x)$	log10(x)	Returns base 10 logarithm of x
x^y	pow(x, y)	Computes power y of a number x
Rounding	round(x)	Returns the integral value nearest to the argument x
sin(x)	sin(x)	Returns sine of the argument x
sh(x)	sinh(x)	Returns hyperbolic sine of an angle x

1	2	3
$\sqrt{x}$	sqrt(x)	Computes square root of a number *x*
tg(*x*)	tan(x)	Returns tangent of the argument *x*
tgh(*x*)	tanh(x)	Returns hyperbolic tangent of an angle *x*

1.9. Formated Input/Otput of Data

The formatted input and output functions are located in the library *stdio.lib*.

int **scanf**(const char **format* [, *arguments*]) reads formatted data from the keyboard and writes it to the location specified by the argument. Each argument is a pointer to a variable of the same type as the corresponding formatting character. In case of an error, the function returns the value 1 (**EOF**).

The formatting string consists of three kinds of characters:

– format specifiers;

– characters are not delimiters (except for the '%' character);

– separator characters (space ' ', tabulation '**\t**', jump to the next line '**\n**').

The format specifier begins with the '**%**' character and specifies the type of arguments to read. The format characters for the **scanf** function are shown in tab. 1.2.

Table 1.2

Format character	Outcome	Argument type
c	Reads one character	char
d	Reads an integer decimal number	int
i	Reads an integer in decimal, octal or hexadecimal	int
e, f, g	Reads a real number	float
le, lf, lg	Reads a real number	double
o	Reads an octal number	int
s	Reads a string of characters	char *
x	Reads a hexadecimal number	int
u	Reads unsigned decimal integer	unsigned int
p	Reads pointer value	void *
n	Gets the number of the read characters	int

An integer between the '**%**' character and the format character allows you to specify the maximum number of characters to be read and passed to the argument.

If the next character in the formatting string *is not a separator character* or format specifier, then the **scanf** function compares it with the current character in the input stream: if it matches then it skips, otherwise it stops working.

int **printf** (const char * *format [, arguments]*) displays formatted data on the screen.

The formatting string consists of:

– characters directly displayed on the screen;

– control characters;

– format specifiers.

Control characters are shown in tab. 1.3.

Table 1.3

Symbol	Operation
\a	Signal
\b	Step back
\f	formfeed
\n	Transfer to the beginning of the next line
\t	Tab
\r	Carriage return
\v	Vertical tab
\\	Backslash backslat
\’	Single quote
\”	Double quote
\?	Interrogatory mark
\0	Zero byte (each character is 0)

This format specifier has the general view:

% [*flag*] [*width*] [.*precision*] <*format_character*>

The flag parameter determines the alignment of the number in the output. Possible flag values are given in tab. 1.4.

Table 1.4

Flag	Appointment
−	Aligns the displayed number to the left of the field
+	The sign of a number will always be displayed
Space	Sets a space in front of a positive number and a minus in front of a negative
#	Outputs 0 before an octal number, 0*x* before a hexadecimal number

The **width** parameter defines the minimum number of output characters.

The **precision** parameter has different purposes for different types of output data. For real numbers printed they use the "**%f**" and "**%e**" specifiers. The precision determines the number of decades. The "**%g**" specifier determines the number of significant digits. The precision determines the maximum length of the output field for the outputting strings and for the outputting integers it determines the maximum number of digits.

The format characters for the **printf** function are shown in tab. 1.5.

Table 1.5

Character format	Value
1	2
c	Output of one character
d	Output of the integer decimal number
i	Output of the integer number in decimal, octal or hexadecimal format

1	2
e	Number output with the fixed point ($\pm x.xx\, e \pm xx$)
f	Output of the floating-point number ($\pm xx.xxx$)
g	Selects shorter output from %e and %f
o	Output of the octal number
s	Output of the line of characters
x	Output of the hex number
u	Output of the decimal integer number without sign
p	Output of value of the pointer ($XXXX : XXXX$)
n	Displays number of the read characters

	Scanf and **printf** operators were defined in the C language, and can be used in C++. However language C ++ has more convenient functions for input-output of data

1.10. Stream Input/Output

In language C++ the input/output of data is made with use of flows (**iostream.h** library). ***The flow*** is this logic device which performs data transmission from the source to the receiver. In **iostream** library four strandartd flows are defined:

cout – the standard flow of input, is moved from random access memory to the external device (by default – on the computer screen);

cin – the standard output stream, is sent from the external device (by default – from the keyboard) to random access memory;

cerr – flow of the standard error;

clog – the buffered flow of standard errors.

Paste operations in the flow (<<) and extraction from the flow are applied to work with flows (>>).

Let us enter, for example, from the keyboard variable x and we will display:

```
cout << "Enter x" << endl;
  cin >> x;
cout << "x =" << x << endl;
```

Here the **endl** manipulator transfers the cursor to the beginning of the next line.

	In the C language, the '**\n**' control character was used to switch to a new line. However the **endl** manipulator except transfer of the line makes also reset of buffers of the output stream that increases reliability of the program, but reduces the speed of its execution a little

For management of input-output flags of the formatted input-output or formatting manipulators are used.

Flags set input-output parameters which will work on all subsequent operators until are cancelled.

For flag activation of the output construction is used

cout.setf(ios:: *flag*)

For removal of the flag construction is used

cout.unsetf(ios:: *flag*)

If it is necessary to set several flags, then it is possible to use operation "or" (|), for example:

cout.setf(ios:: *flag1* | ios:: *flag2* | ios:: *flag3*)

Some flags for the formatted input are given in tab. 1.6.

Table 1.6

Flag	Description
right	Flushing right
left	Flushing left (by default)
boolalpha	Output of logical values in text form
dec	Output of values in the decimal numeral system (by default)
showpos	Displays the character '+' for positive numbers
scientific	Output of real numbers in the exponential form
fixed	The fixed form of the output of real numbers (by default)

Manipulators (iomanip.lib library) are located in operators of input/output just before the formatted value. Some manipulators of formatting are given in tab. 1.7.

Table 1.7

Manipulator	Description
setw(n)	Sets output field width in *n* of characters
setprecision(n)	Sets the number of digits (*t – 1*) in the fractional part of number
left	Alignment of number on the left border (by default)
right	Alignment of number on the right border
boolalpha	Output of logical values in text form
noboolalpha	Output of logical values in the numerical type
showpos	Displays the character '+' for positive numbers
noshowpos	Does not display the character '+' for positive numbers
scientific	Exponential form of the output of real numbers
fixed	The fixed form of the output of real numbers (by default)
setfill(ch)	Set *ch* character as filler

It is possible to set output field width also as follows:

cout.width(n) – sets output field width, equal *n* of positions;

cout.presicion(m) – defines *m* of digits in the fractional part of number.

2. Fundamental Data Types

2.1. Data Types

The data type determines the values of variables, their structure, operations at them and the cells quantity to place them. Data can be devided into two groups: scalar (simple) and structured (composite).

The scalar type is data represented by a single value (number, symbol) and located in one cell of several bytes.

Structured types are user-defined as a combination of scalar and the previously described structured types.

The basic data types are integer, real and symbolic types.

The data can be ***constants*** and ***variables***. Constants (unlike variables) cannot change their value during program execution.

2.2. Variables and Constants Declare

Variables can be declared anywhere in the program before the first use of them. To increase the program readability, it is better to do this at the program beginning.

To declare a variable, first you need to specify the data type, then a list of variables of this type separated by commas. For example

```
int x, y, k;
double s, z;
```

It's possible to use two forms of the variables:

– declarating without the memory allotment;

– declarating with the memory allotment in accordance with the specified type.

When declaring with the memory allotment, it can immediately initialize (assign a certain initial value) to a variable. For example:

```
int k=3, m=34;
```

The const keyword is used to declare constants:

```
const double pi = 3.14159265359;
```

2.3. Integer Data Type

An integer data type is characterized by the absence of a fractional part in the value. The C++ language uses the following integer data types:

1) **int(__int32, signed)** is the system-dependent variable (has different lengths in different systems). In 32-bit systems it has a length from –2 147 483 648 to 2 147 483 647 and occupies 4 bytes of memory;

2) **long (long int)** is the system-independent variable (has permanent length in different systems). It occupies 4 bytes of memory and has length from –2 147 483 648 to 2 147 483 647. In 32-bit systems matches the int type;

3) **short (__ int16, short int)** – the system-independent variable (has permanent length in different systems). It occupies 2 bytes of memory and has length from-32 768 to 32 767. A variable of this type is the same as int on 16-bit systems. It is

undesirable to use this type since having smaller length, it is processed more slowly than the int type;

4) **long long (__ int64)** is the system-independent variable for 64 bit systems. Occupies 8 bytes of memory and has length from –9 223 372 036 854 775 808 to 9 223 372 036 854 775 807.

The unsigned attribute is used to shift the range borders only to the positive area. For example, unsigned int has length from 0 to 4 294 967 295.

The integer type constants are the sequence of digits. They begin with negative sign for negative constants and from plus sign or without it for positive constants. For designation of **long** constants after number put letter **L** or **l**.

Constants can be provided in different numeral systems.

Decimal constants: the sequence of numbers from 0 to 9 starting not from scratch, for example, **134**.

Octal constants: the sequence of numbers from 0 to 7 starting from scratch, for example, **045**.

Hexadecimal constants: the sequence of numbers from 0 to 9 and letters from **A** to **F**, beginning with characters **0x**, for example **0xF5C3**.

2.4. Character Data Type

The character type is meant for one character storage. Consequently, it is enough to select 1 byte of memory. Data of this kind are considered by the compiler as whole therefore it is possible to store integer numbers from range in signed char variables –128…127. For storage of the **unsigned char** characters which allows to store 256 characters of the code chart ASCII (*American Standard Code for Information Interchange*) is used. The standard symbol set of ASCII uses only 7 bits for each character (range 0…127). The standard ASCII character set uses only 7 bits for each character (range 0...127). The addition of the 8th digit made it possible to increase the number of ASCII table codes to 255. The codes from 128 to 255 are an extension of the ASCII table for storing of the national alphabets characters and the pseudographic characters.

Values of the code chart ASCII with numbers 0…32 and 127 contain the hidden characters. They have no graphical representation, but ones influence display of the text. Characters with codes 32…127 are presented in tab. 2.1. Characters with codes 128…255 (code chart 866 – *MS-DOS*) are presented in tab. 2.2.

Table 2.1

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
32	space	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	“	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	del

Table 2.2

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
128	А	160	а	192	└	224	р
129	Б	161	б	193	┴	225	с
130	В	162	в	194	┬	226	т
131	Г	163	г	195	├	227	у
132	Д	164	д	196	─	228	ф
133	Е	165	е	197	┼	229	х
134	Ж	166	ж	198	╞	230	ц
135	З	167	з	199	╟	231	ч
136	И	168	и	200	╚	232	ш
137	Й	169	й	201	╔	233	щ
138	К	170	к	202	╩	234	ъ
139	Л	171	л	203	╦	235	ы
140	М	172	м	204	╠	236	ь
141	Н	173	н	205	═	237	э
142	О	174	о	206	╬	238	ю
143	П	175	п	207	╧	239	я
144	Р	176	░	208	╨	240	Е
145	С	177	▒	209	╤	241	е

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
146	Т	178	▓	210	╥	242	Є
147	У	179	│	211	╙	243	є
148	Ф	180	┤	212	╘	244	Ї
149	Х	181	╡	213	╒	245	ї
150	Ц	182	╢	214	╓	246	Ў
151	Ч	183	╖	215	╫	247	ў
152	Ш	184	╕	216	╪	248	°
153	Щ	185	╣	217	┘	249	∙
154	Ъ	186	║	218	┌	250	·
155	Ы	187	╗	219	█	251	√
156	Ь	188	╝	220	▄	252	№
157	Э	189	╜	221	▌	253	¤
158	Ю	190	╛	222	▐	254	■
159	Я	191	┐	223	▀	255	

The character type variables are registered in single.

2.5. Real Data Type

The real data type is characterized by the fractional part. The number is represented in the exponential form: ***±n.mE±p***, where ***n.m*** is the mantissa (***n*** is the integer part, ***m*** is the fractional part), ***p*** is the power.

The C++ language uses the following types of real data:

– **float** type stores the numbers occupying 4 bytes of memory and being in the range from $3.4 \cdot 10^{-38}$ up to $3.4 \cdot 10^{+38}$. This type allows to store numbers to within 7 signs after the comma;

– **double** type stores the numbers occupying 8 bytes of memory and being in the range from $1.7 \cdot 10^{-308}$ up to $1.7 \cdot 10^{+308}$. This type allows to store numbers to within 15 signs after the comma.

The real number is stored in the computer memory in normalized form (more than one and less than two). If the normalization is violated, the mantissa is shifted to the left until the most significant digit of the mantissa becomes one. Since the first digit of the normalized mantissa is always equal to one, it may not be stored in memory. The saved bit is used to improve the precision of the number representation. The unit is implicitly present in the number and is called the implicit unit. The order of the number is stored in shifted form so that the entire range of values is in the positive range. This saves one more bit.

When defining real constants, the letter ***F*** is added at the end for the **float** type, ***D*** for the **double** type (optional) and ***L*** for the **long double**.

2.6. Boolean Data Type

The logical **bool** type can accept two values: *true* (1) or *false* (0). 1 byte is allocated to store this type of data.

As the **bool** type occupies 1 byte, it can accept values from 0 to 255. Values from 1 to 255 are treated as *true* (1), and value 0 – as *false* (0).

```
bool b;
      b = true;     cout << b << endl;        // Display: 1
      b = 1;        cout << b << endl;        // Display: 1
      b = 225;      cout << b << endl;        // Display: 1
      b = false;    cout << b << endl;        // Display: 0
      b = 0;        cout << b << endl;        // Display: 0
```

2.7. Void Data Type

This type describes the empty set of values. As a rule this type is used for the description of the functions. The type is usually used to describe functions that do not return a value or to declare untyped pointers (to use them, they must be cast to a specific type).Variable declaration like **void** is forbidden.

2.8. Declaration of auto

Since MS VS 2010, the key word of **auto** is used for determination of variable type proceeding from the initializating expression type.

Format:

```
auto initializer = initializating expression;
```

For example:

```
auto x = 5;           // x is the int variable
auto y = 7.8;         // y is double variable
auto m1 = {1, 2, 3};  // m1 is the array like int
auto m2 = {1.5, 2.4}; // m2 is the array like double
```

2.9. Mathematical Constants

Mathematical constants are defined:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

Mathematical constants are presented in tab. 2.3.

Table 2.3

Constant	Mathematical formula	Value
1	2	3
M_E	Число e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417

1	2	3
M_LN10	ln(10)	2.30258509299404568402
M_PI	π	3.14159265358979323846
M_PI_2	π/2	1.57079632679489661923
M_PI_4	π/4	0.785398163397448309616
M_1_PI	1/π	0.318309886183790671538
M_2_PI	2/π	0.636619772367581343076
M_2_SQRTPI	2/sqrt(π)	1.12837916709551257390
M_SQRT2	sqrt(2)	1.41421356237309504880
M_SQRT1_2	1/sqrt(2)	0.707106781186547524401

2.10. Implicit Type Conversion

In most cases the conversion type is automatic with the priority of types using. Types have the following priority sequence:

char → short →**int**→ unsigned int → long → unsigned long → float →**double**

The priority increases from left to right (towards increase in the memory occupied by type). The arithmetic operations rule is: the operand with lower priority will be transformed to the operand with higher priority, and the **char** and **short** types values will always be transformed to the **int** type.

The main objective of implicit type conversion is saving of value that is reached by increase in level of the priority (increase in the size of data type).

The type conversion does not execute by the assignment operation. By assignment to the variables with the lower priority of type the variables values with the higher priority type it is possible loss of information.

For example, it is necessary to calculate: s = a + b, where (s is **double** variable, a is **char** variable, b is **int** variable). Let a = 'd', b = 45. The expression will be evaluated in the following order.There are two variables of different types, the variable with the smaller priority (a) will be brought to the **int** type at this arithmetic expression. For this purpose in memory of the computer the temporary variable like **int** which will store number of the character 'd' equal 100 is created (see table 2.1). The sum operation result will be equal to 145 (100 + 45). The result (145) is appropriated to variable s. When an assignment operation is performed, no type conversion occurs. But the size of the **double** variable is larger than the size of the **int** variable.So information is not lost.

2.11. Explicit Type Conversion

If implicit type conversion does not result in required result, the programmer can set type conversion explicitly:

```
static_cast <type> (variable)
```

The operator returns the error if types used for reduction are completely incompatible.

From the C language the form of reduction of types remained outdate (not recommended by the developer)

```
(type) variable
```

or

```
type (variable)
```

Example:

```
int a, b, s, f;
        a = b = 2147483647;
        s = (a * b) / a;
        f = (static_cast <double> (a) * b) / a;
```

Result: *s* = 0, *f* = 2147483647.

When calculating variable s the calculated value of work **a** on **b** oversteps the bounds of range of values which can be stored in the **int** variable. The temporary variable created for storage of intermediate result like **int** obtains wrong information, therefore, the result of calculation will be incorrect.

In the next line variable **a** is explicitly given to the **double** type. Therefore, the result of calculation of the work will be kept in temporary variable of the greatest longwise like (from **int** and **double**) the **double** type. The received result does not overstep the bounds of range of **double** values therefore the error does not arise.

3. C++ Language Operations

3.1. Arithmetic Operations

The simplest arithmetic operations are +, –, *, /. These operations are applicable both to integer and to real data types and rules of their use are similar to their use in mathematical calculations. The sequence of operations can be changed by means of brackets.

For work only with integer numbers there is the operation of receipt of the remainder of division – **%**.

For example: 10 % 6 = 4, 7 % 10 =7, 10 % 5 =0.

3.2. Assignment Operation

Operation format:

operator_1 = *operator_2*;

The *operator_2* value is brought in the *operator_1* variable. As *operator_1* it is possible to use only the variable. As *operator_2* it is possible to use the constant, the variable, expression or function.

It is acceptably to use the following writing:

a = b = c = d; that is equivalent a =d; b = d; c = d;

Often in programming operations of this kind are used:

operator_1 **=** *operator_1* ***symbol_operation*** *operator_2*;

For the records reduction of these operators it is possible to use abbreviated form of record:

operator_1 ***symbol _operation=*** *operator_2*;

For example, operator

s = s + 2;

it is possible to replace with the operator

s += 2;

If *operator_2* for operations of sum and subtraction it is equal to unit, then it is better to use operations of the increment:

operator_1 ++;

or decrement

operator_1––;

For example, instead of writing of i = i + 1 it is possible to use i++, and instead of writing of i = i – 1 it is possible to use i––.

The sign of the increment or decrement can be written in two forms: in prefix (for example ++i) or in postfix (for example i++). The method of writing influences much execution of operations in expression. At the prefix form at first the increment or decrement, and then arithmetic operations is executed. In the postfix form, the arithmetic operations are performed first, followed by the increment or decrement.

3.3. Relational and Comparison Operators

Operations of comparison are applied during the work with two operands and return *true* (1) if result of comparison – the truth, and *false* (0) – if result of comparison – the lie. In the C language the following operations of comparison are defined:

< (it is less), **<=** (it is less or equally), **>** (more),

>= (it is more or equally), **!=** (not equally), **==** (equally).

It is better that the operands have the same type (it is acceptable to compare integer and real types).

3.4. Logical Operations

Logical operations work with operands of scalar types and return the result of logical type. Three logic operations "!", "&&" and "|" are defined.

Unary logical operation **NOT** (**!**) does return *true* (1) if the operand has null value, and *false* (0) if the operand is other than zero.

For example:

```
k = 5;
a =!(k > 0);
```

Result: 0 (*false*) since the operand matters 1 (*true*) which changes operation "!" on the return – 0 (*false*).

Logical operation **AND** (**&&**) returns *true* (1) if operands have nonzero values, and *false* (0) – if at least one operand has null value.

For example:

```
k = 5;
a = (k > 0 && k <= 10 && k! = 5);
```

Result: 0 (*false*) since the two first the operand matter 1 (*true*), and the last operand matters *0* (*false*).

If to enter:

```
k = 5;
a = (k > 0 && k <= 10 && k == 5);
```

Result: 1 (*true*) since all operands matter 1 (*true*).

Logical operation **OR** (|) returns *true* (1) if at least one operand has nonzero value, and *false* (0) if all operands have null value.

For example:

```
k = 5;
a = (k > 0 || k <= 10 || k != 5);
```

Result: 1 (*true*) since the two first the operand matter 1 (*true*).

If to enter:

```
k = 5;
a = (k <= 0 || k > 10 || k != 5);
```

Result: 0 (*false*) since all operands matter 0 (*false*).

Mixing of different logic operations in one expression is allowed:

```
k = 5;
a = !(k >= 0 && k < 10 || k != 5);
```

Result: 0 (*false*).

3.5. Bitwise Operators

Bitwise operators modify variables considering the bit patterns that represent the values they store.

The following operations are defined:

"~" – digit-by-digit denial (0 changes on 1, and 1 on 0);

"&" – digit-by-digit AND;

"|" – digit-by-digit OR;

"^" – digit-by-digit excluding OR;

"<<" – the digit-by-digit left shift;

">>" – the digit-by-digit right shift.

The unary digit-by-digit operation "~" inverts each bit of the operand.

The table of truth for operations "&", "|", "^" in tab. 3.1.

Table 3.1

Value of bits	b & b_{12}	b_1|b_2	b ^ b_{12}
$b_1 = 0, b_2 = 0$	0	0	0
$b_1 = 0, b_2 = 1$	0	1	1
$b_1 = 1, b_2 = 0$	0	1	1
$b_1 = 1, b_2 = 1$	1	1	0

The >> (right shift) takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift. The right bits are lost. If the left operand is unsigned number, then the left free bits are filled with zero. If there is the sign of the character, then cells are filled with this character. The shift of the integer number is equivalent to integer division on 2^n.

The << (left shift) takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift. The right bits are lost. The left bits are lost, and right are filled with zero. If there is the sign of the character, then cells are filled with this character. The shift of the integer number is equivalent to multiplication on 2^n.

3.6. Priority of Operations in C++

The priority of operations in language C++ is presented in tab. 3.2. The priority decreases from top to down.

Table 3.2

Level priority	Operation type	Operators
1	Permission of the scope	::
2	The choice of the element for the pointer (the object or the pointer)	. (point), -> (arrow)
	Index of the array	[ ]
	Function call, brackets	( )
	Prefix increment and decrement	++, --
	Object type name	typeid
	Explicit reduction of type	const_cast, dy-namic_cast, reinter-pret_cast, static_cast
3	Size of the object or type	sizeof
	Digit-by-digit denial	~
	Logical denial	!
	Unary plus and minus	+, –
	Taking of the address and razadresation	&, *
	Creation and destruction of the object	new, delete
	Explicit reduction of type	()
4	Pointer on the element (the object or the pointer on the object)	.*, ->*
5	Arithmetic operations	*, /, %
6	Arithmetic operations	+, –
7	Shift	<<, >>
8	Comparison operations	<, >, >=, <=
9	Comparison operations	==, !=
10	Bit-by-bit And	&
11	Bit-by-bit excluding OR	^
12	Bit-by-bit OR	|
13	Logical And	&&
14	Logical OR	||
15	Conditional operation	? :
16	Postfix increment and decrement	++, ––
17	Assignment	=, *=, /=, %=, +=, –=, <<=, >>=, &=, |=, ^=
18	Sequence	, (comma)

3.7. Blocks

The group of operators in the curly brackets is called ***the block***. The compiler considers such group of operators as one compound statement. In any construction of language C++ the simple operator it is possible to replace with the block.

For example, instead of

```
operator;
```

it is possible to put

```
{
operator_1;
…
operator_n;
 }
```

4. Branching Algorithms

The algorithm is called ***branching*** if it contains several branches differed from each other in the content of calculations. The computation process output is defined on this or that branch of the algorithm by continuous data.

4.1. Conditional Transfer Control Operator if

Format of case statement:

```
    if (logical_expression) operator_1;
                        else operator_2;
```

If *the logical_expression* is true, then is executed *operator_1*, differently – *operator_2*.

For example:

```
 if (f > 10) x = 3; else x = 24;
```

True logical expression is considered if it is equal:

– *true*;

– to nonzero arithmetic value;

– to the pointer value other than *nullptr*;

– to nonzero value of the class type defining unique transformation to arithmetic, logical type or pointer type.

The operator has abbreviated form:

```
    if (logical_expression) operator_1;
```

For example: **if** (f == 0) x = 4;

*Logical_expression*is always located in parentheses. If *operator_1* or *operator_2* support more than one operator, then, the block is used.

The **if** operators can be used as *operator_1* and *operator_2*. Such operators call *enclosed*. In the enclosed if operators the key word of **else** belongs to the next **if** preceding it.

For example:

```
 if (logical_expression _1) operator_1;
 if (logical_expression _2) operator_2;
                   else operator_3;
```

Operator_3 it will be executed if *logical_expression_2* it is false. The value *logical_expression_1* has no impact on the *operator_3* execution.

It is possible to change the procedure for test by using curly brackets:

```
 if (logical_expression_1) {
      operator_1;
 if (logical_expression_2)   operator_2;
                          }
 else operator_3;
```

Operator_3 it will be executed if *logical_expression_1* it is false. The value *logical_expression_2* has no impact on the *operator_3* execution.

4.2. Conditional Operation

Format of the conditional operations:

condition **?** *operator_1* **:** *operator_2*;

If the value the *condition* is true, then the operation result is *operator_1*, else it is *operator_2*.

For example, to find the greater of two numbers:

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

The condition can be any scalar expression, and operators can have practically any type.

Application of conditional operation reduces the code, however, it has no impact on the speed of program execution.

4.3. Multiple Selection Operator switch

Use the **switch** statement to select one of many code blocks to be executed.

Format of the operator following:

switch (*variable*) {
case *const1*: *operators_1*; **break**;
…
case *constN*: *operators_n*; **break**;
default: *operatosr_n+1*;
}

The operator is executed as follows. At first variable value of the choice is analyzed and checked whether it matches value of one of constants. At coincidence operators of this **case** are executed. Construction of **default (**can be absent**)** is executed if the result of expression did not match one of constants. The variable type of the choice of can be integer, character or listed. The type of constants of comparison shall match choice of variable type.

Example 4.1. Decrypt assessment on the five-point system:

```
int otc;
cin >> otc;
switch (otc)  {
case 2: cout << "unsatisfactory" << endl; break;
case 3: cout << "satisfactory" << endl; break;
case 4: cout << "good" << endl; break;
case 5: cout << "excellent" << endl; break;
default: cout << " no such assessment" << endl;
}
```

At the end of each set of operators the **break** operator which completes execution of **switch** operator is put. If one does not to put **break**, then after execution of the corresponding section the control will be transferred to the operators belonging to other branches of **switch**. The lack of **break**, as a rule, leads to the error in calculations. However, in certain cases use of the sections **case** without **break** is justified, for example, if it is necessary for different values of constants of comparison to execute the identical sequence of operators.

Example 4.2. Identify time of the year by the number of a month.

```
int month;
cin >>month;
switch (month)
{
case 12:
case 1:
case 2: cout << "Winter"; break;
case 3:
case 4:
case 5: cout << "Spring"; break;
case 6:
case 7:
case 8: cout << "Summer"; break;
case 9:
case 10:
case 11: cout << "Autumn"; break;
default: cout << "Input Error";
}
```

Example 4.3. Calculate value of expression

$$s = \begin{cases} f(x)\cdot\sin(x), \text{if } x+y > 12, \\ e^{2x}+e^{-x}, \ \text{ if } x+y \ \le 5, \\ \sqrt[3]{|y\cdot f(x)|}\,, \ \text{ else.} \end{cases}$$

To provide the choice of the type of function $f(x)$ is: tg(x) or x^2.

```
double x, y, f, a, res;
int k;
cout << "Enter x ";        cin >> x;
cout << "Enter y ";        cin >> y;
cout << "Initial data: x = " << x << " y = " << y << endl;
cout << "Select f : 1 - tg(x), 2 - x^2 ";
cin >> k;
switch (k)
```

```
{
case 1: f = tan(x);    break;
case 2: f = pow(x, 2); break;
default: cout << "No function selected";  return 1;
}
f = x + y;
if (a > 12) res = f * sin(x);
else
    if (a <= 5) res = exp(2 * x) + exp(-x);
    else
        res = pow(fabs(y * f), 1. / 3);
cout << "Result = " << res << endl;
```

5. Cyclic Algorithms

5.1. Loop Operator for

General view of the operator:

```
for(init-expression; cond-expression; loop-expression)
                         {
loop-body
                         }
```

Most often all three expressions contain one variable which is called the loop counter.

Init-expression is executed only once at the beginning of the cycle execution. As a rule it is used for initialization of the loop counter.It may contain declarations and operators.

Cond-expression is checked at the beginning of each cycle. If the result has integer value (**true**), other than zero, then the cycle is repeated, otherwise the operator following the loop body is executed. If logical expression is absent, then it is considered that it is **true**.

Loop-expression is used for the value change of the loop counter. Changing of the counter happens after each execution of the loop body.

Loop-body is the sequence of operators that is executed repeatedly until the condition of the loop termination is satisfied. The loop body may contain in itself any constructions of language C++ and any quantity of nested loops.

The scheme of work of the cycle for is submitted in fig. 5.1.

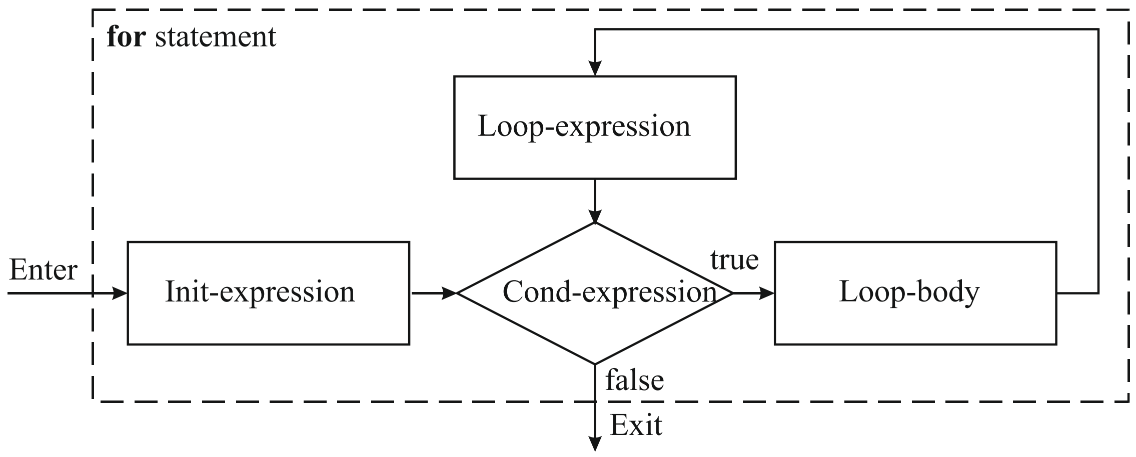


Fig. 5.1

Let's consider work of the following operator:

```
for (i = 1; i < 10; i++) cout << i << endl;
```

At the beginning of the cycle in variable **i** number **1** will be brought. Then the value of logical expression increases and since it is **true** (1 < 10), the loop body will be executed will be checked (the value **i** will be displayed). After that the incrementing expression of **i++** is executed and again the value of logical expression will be checked. The loop body will be executed until logical expression does not accept **false** value (10 < 10). A result will be displayed as digits from 1 to 9.

> If it is necessary to display digit from 1 to 10, then it is possible to use construction:
>
> `for (i = 1; i <= 10; i ++)`
>
> However in C++ usually use constructions with strict inequality:
>
> `for (i = 1; i < 11; i++)`

It is convenient to combine execution of the incrementing expression with the description of the loop counter.

```
for (int I = 1; i < 11; i ++)
```

Such writing is convenient that is why the announced variable according to standard C++ will exist only in the cycle, and further the name of this variable can be used for other purposes.

Any of sections in for operator is not obligatory therefore there can be no one or several expressions. This writing of the infinite loop is possible:

```
for ( ; ; )
```

> To place several operators in one section of the **for** operator, the "comma" operation is used, which allows you to place several operators in those places where only one operator can be used. Operation format
>
> `Operator_1, Operator_2, ..., Operator_n`

The calculation program of the number ***n*** factorial can look as follows:

```
for (f = 1,  i = 1; i <= n; f* = i, i++);
```

The semicolon at the end of for operator means that the loop body is absent.

In language standard of C++ 2011 the new form of for operator (range-based for) which allows to address consistently each element of the collection is entered.

```
for (element: collection)
{
      // Loop-body
}
```

5.2. Loop Operator while

Loop operator with the precondition

```
while (condition)
{
      // Loop-body
}
```

will organize repetition of operators of the loop body until the value of logical expression is true. As soon as the value of logical expression becomes equal 0 (*false*), cyclic process stops and the operator, the first after the cycle, is executed. If the cycle condition is equal to 0 (*false*) at once, then the loop body is never executed.

5.3. Loop Operator do-while

Iteration statement with postcondition

```
do {
      // Loop-body
} while (condition);
```

will organize repetition of operators of the loop body until the value of logical expression is true. As soon as the value of logical expression becomes equal 0 (*false*), cyclic process stops and the operator, the first after the cycle, is executed. Regardless of value of logical expression the loop body will be executed not less once.

	The ***do-while*** loop operator is dangerous in that the body of the loop must be executed at least once. That's why it is necessary to check the condition for its completion before entering the loop. Therefore you should avoid using this operator if possible

5.4. Operators and Functions of the Control Transfer

Operators and functions of the control transmission allow to change the standard execution order of operators.

5.4.1. Continue Operator

It is used to organize the cyclic processes. The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between. The **continue** operator is usually used together with if operator in order at certain values of data to complete the current cycle and to transfer control to the following cycle.

5.4.2. Break Operator

Allows to pass to the operator the following block. For example, in cycles it provides early loop termination, and in **switch** operator is the output from the choice block. It is necessary to pay attention that the **break** operator goes out only of the current block, i.e. in case of nested loops the output comes only from one cycle.

5.4.3. Return Operator

Completes execution of function and transfers control of the call function (or OS for the **main()** function). The control is transferred in the call function in call point.

Format of the operator:

```
return expression;
```

If the expression value is set, then the result is returned in the call function as value of the caused function.

5.4.4. Exit Function

It is in *stdlib.lib* library.It correctly interrupts the execution program, writes all buffers, closes all flows. Function format:

```
void exit(int)
```

Parameter is the office message to the system. As a rule, 0 speaks about the successful completion of the program, nonzero values speaks about the error.

5.4.5. Abort function

It is in *stdlib.lib* library. It generates the exception and interrupts program execution. The abort function does not close open and temporary files, does not clean buffers of flows. Function format:

```
void abort(void)
```

5.4.6. Unconditional Satement goto

It transfers control to the operator marked with the tag. Use of **goto** operator significantly reduces readability of the program and increases error probability. Therefore use of **goto** in programs is undesirable.

For example, it is necessary to use the unconditional jump operator to organize an exit from several nested loops at once:

```
for (i = 0; i < n; i++)
   for (j = 0; j <m; j++)   {
     if (logic_expression) goto met;
                            }
met: …
```

5.5. Loop Algorithms

Example 5.1. Output the table of function values $y(x) = \sin(x)$ on the interval from *a* to *b* with *h* step.

Option 1 (with use of iteration statement of **for**).

```
for (double x = a; x < b+h/2; x += h)
cout << "x =" << x << "y =" << sin(x) << endl;
```

Logical expression is equal in the operator $x < b+h/2$, not $x <= b$. It is caused by the following. The loop counter on *x* each step increases the value by *h*. If *h* is fractional number, then in variable x can collect rounding errors of which lead to the result that the value *x*, for example, will be equal 2.00000000001 when value $b = 2.0$. The result of operation of comparison $x <= b$ will matter in this case *false*, and, therefore, the last value of the table will not be displayed. To guarantee execution of the last iteration of cyclic process, the value of the right border of the interval increases by the value which is not exceeding *h* (for example on *h/2*).

Option 2 (with use of iteration statement of **while**).

```
x = a;
while (x < b + h/2)
{
   cout << "x =" << x << "y =" << sin(x) <<endl;
x += h;
}
```

Example 5.2. Calculate integral $s = \int_a^b \sin x \, dx$ by method of averages.

```
      h = (b-a)/100;
for (x = a + h/2, s = 0; x < b; s += sin(x)*h, x += h);
```

Example 5.3. Calculate the sum $s(x) = \sum_{k=1}^{100} (-1)^k \frac{x^k}{k!}$.

In the beginning it is necessary to receive the recurrent formula. For receiving of the formula values composed at different values k are calculated: at $k = 1;\ a_1 = -1\frac{x}{1};$ at $k = 2;\ a_2 = 1\frac{x \cdot x}{1 \cdot 2};$ at $k = 3;\ a_3 = -1\frac{x \cdot x \cdot x}{1 \cdot 2 \cdot 3}$ etc. It is visible that on each step composed in addition is multiplied on $-1\frac{x}{k}$. Proceeding from it the formula of the recurrent sequence will be $a_k = -a_{k-1}\frac{x}{k}$. The received formula allows to get rid of repeated calculation of the factorial and exponentiation.

```
s = 0;          // Starting value of the sum
a = 1;          // Starting value for calculation of next
                // member of the recurrent sequence
for (int k=1; k <= 100; k++)
{
a *= -x/k;      // Calculation of the next member
                // recurrent sequence
  s += a;       // Summation of the all summands
}
```

Example 5.4. Calculate the sum $s(x) = \sum_{k=0}^{100} (-1)^k \frac{x^{2k}}{(2k)!} \sin(x)$.

In this formula it is difficult to receive recurrent dependence for **sin(*x*)** therefore the **sin(*x*)** function will be separately calculated (as the non-recurrent part). For the formula rest $\sum_{k=0}^{100} (-1)^k \frac{x^{2k}}{(2k)!}$ values are calculated at different values *k*: at $k = 0$; $a_1 = 1\frac{1}{1}$; at $k = 1$; $a_1 = -1\frac{\boldsymbol{x^2}}{\mathbf{1 \cdot 2}}$; at $k = 2$; $a_2 = +1\frac{x^2 \cdot \boldsymbol{x^2}}{1 \cdot 2 \cdot \mathbf{3 \cdot 4}}$; at $k = 3$; $a_3 = -1\frac{x^2 \cdot x^2 \cdot \boldsymbol{x^2}}{1 \cdot 2 \cdot 3 \cdot 4 \cdot \mathbf{5 \cdot 6}}$ etc. The formula of the recurrent sequence will be $a_k = -a_{k-1}\frac{x^2}{(2k-1)\cdot(2k)}$. It is convenient to begin calculation not with the zero element, and with the first. Therefore the value of the zero element is calculated manually and is substituted in starting value of the sum.

Text of the program:

```
s = sin(x);                    //  Value of the sum for the zero element
  a = 1;
 for (int k = 1; k <= 100; k++)
{
  a *= -sqr(x) / (2 * k * (2 * k - 1));
  s += a*sin(x);               //  Here the non-recurrent part is added
 }
```

6. Arrays

Array is a homogeneous data structure. Each element is stored in the separate cell, access is provided by its number. The array is characterized by the array name, type of the stored data, the size (quantity of elements) and dimension (the form of representation of array cells). The array cell number is called *the index*. The array indexes have the integer type. The array elements can have any type.

6.1. One-dimensional Arrays

Declaration of the one-dimensional array:

```
array_type array_name[size];
```

The array declaration example:

```
int c[4];
```

The size of the static array is set by the constant or constant expression of the whole type.

Indexes in language of C/C++ start from 0. For example, the above-announced array consists of four elements: c[0], c[1], c[2] and c[3]. The location of the array elements in memory is shown in fig. 6.1.

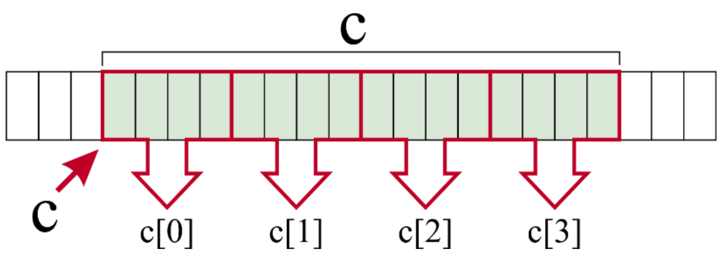


Fig. 6.1

Along with the declaration it is possible to initialize array cells:

```
double b[4] = { 1.5, 2.5, 3.75, 3.04 };
        int a[4] = {1, 4};
```

If there are not enough starting values n the initialization group, then the remainder elements are filled with zero, for example, *a* array: a[0] = 1, a[1] = 4, a[2] = 0, a[3] = 0.

The number of elements can be omitted from the list of the declarations initialization. In this case the size of the array will be equal to the starting values amount. Declaration

```
char mc [] = { ‘3’, ‘f’, ‘w’}
```

will create the array from three elements.

The appeal to the array cell happens through the indication of the name of the array and in square brackets of the item number of the array. For example:

```
x = a[3];   a[4] = b[0] + a [2];
```

6.2. One-dimensional Arrays Operation Algorithms

Example 6.1. Input and output of the one-dimensional array.

```
int s[10], i, j, n;
// Input of the one-dimensional array
cout <<"Enter size:";
cin >> n;
for (i = 0; i < n; i++)
{
cout << "Enter s [" << i << "] =";
cin >> s[i];
}
// Output of the one-dimensional array
for (i = 0; i < n; i++)
cout << s[i] <<" ";
```

Example 6.2. Finding of the sum and work of elements of the one-dimensional array.

```
   s = 0; p = 1;
for (i = 0; i < n; i++)
{
    s += a[i];
    p *= a[i];
}
```

Example 6.3. Finding of the minimum and maximum elements of the one-dimensional array.

Option 1:

```
min = max = a [0];
for (i = 1; i < n; i++)
{
        if (a[i] < min) min = a[i];
        if (a[i] > max) max = a[i];
}
```

Option 2:

```
min = max = a[0];
for (int x: a)
{
```

```
                min = fmin(min, x);
                max = fmax(max, x);
        }
```

Example 6.4. To remove from the one-dimensional array of all negative elements.

```
for (i = 0; i < n; i++)
if (a[i] < 0)
  {
for (j = i + 1; j < n; j++) a[j - 1] = a[j];
 n--; i--;
}
```

6.3. Multidimensional Arrays

Declaration of the one-dimensional array:

```
array_type array_name [size_1] [size_2] … [size_n];
```

Example of the declaration of the two-dimensional array:

```
int m[4][5];
```

Here the two-dimensional array from 4 · 5 = 20 elements is announced.

It is possible to initialize array cells along with the declaration:

```
 int s[2][3] = { {3, 4, 2}, {6, 3, 4} };
```

In the one-dimensional array the first index is row number, and the second – column number. Therefore, for example, the value of the s[1][0] element is equal to 6. Mathematically the array **s** represents the matrix

```
3  4  2
6  8  5
```

In the computer memory this array is located consistently on lines (fig. 6.2).

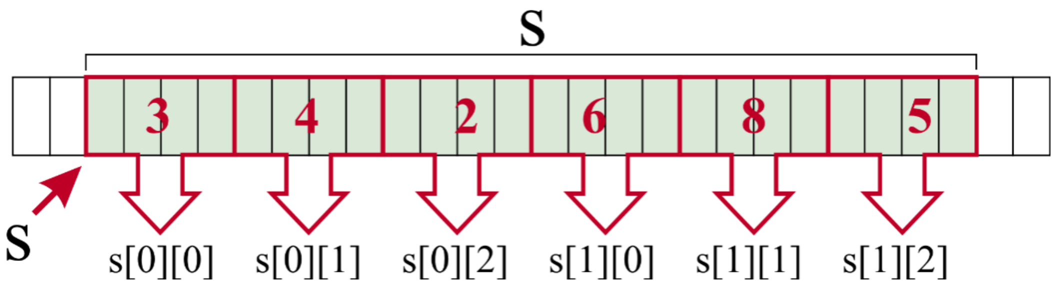


Fig. 6.2

The appeal to the element of the two-dimensional array happens through the indication of the name of the array and in square brackets of row numbers and columns of the array. For example:

```
x = s[0][2];
s[1][2] = m[3][2] + s[0][1];
```

6.4. Two-dimensional Arrays Operation Algorithms

Example 6.5. Input and output of the two-dimensional array of integer numbers.

```
int n, m, i, j;
  double s[10][10];

    // Input
    cout << "Enter n m:" << endl;
    cin >> n >> m;
    for (i = 0; i < n; i++)
            for (j = 0; j < m; j++)
            {
            cout << "Enter s [ " << i << "][" << j << "]:";
            cin >> s[i][j];
            }
    // Output
    for (i = 0; i < n; i++)
    {
            for (j = 0; j < m; j++)
                    cout << setw(8) << s[i][j] << " ";
            cout << endl;
    }
```

Example 6.6. The output of the two-dimensional array from real numbers.

```
for (i = 0; i < n; i++)
{
for (j = 0; j < m; j++)
  cout << setiosflags(ios :: fixed) <<
    setw(10) << setprecision(3) << s[i][j] << " ";
  cout << endl;
}
```

Example 6.7. Fill a two-dimensional array (3×3) randomly with real numbers in the range from 30 to 70.

The following function is used to generate random numbers:

```
errno_t rand_s(unsigned int * randomValue);
```

Function generates pseudorandom number from range 0 … UINT_MAX (4294967295). For use of the rand_s function it is required that to the operator of inclusion the constant _CRT_RAND_S was defined.

```
#define _CRT_RAND_S
#include <iostream>
using namespace std;
int main ()
{
	const int n = 3, m = 3;
	double nmin = 30, nmax = 70, s[n][m];
	unsigned int r;

	for (int i = 0; i < n; i++)
		for (int j = 0; j < m; j++) {
			rand_s(&r);
	s[i][j] = nmin + r / (static_cast <double>(UINT_MAX) + 1)
	                                         * (nmax - nmin);
	cout << s[i][j] <<endl;
}
}
```

Example 6.8. To find the sum of the elements on the side diagonal.

```
s = 0;
for (i = 0; i < n; i++) s += a[i][n-i-1];
```

Example 6.9. Shift of lines with numbers k_1 and k_2.

```
for (j = 0; j < m; j++)
 {
     t = a[k1][j];
  a[k1][j] = a[k2][j];
  a[k2][j] = t;
 }
```

Example 6.10. To find the sum of the elements above the main diagonal.

```
s = 0;
for (i = 0; i < n - 1; i++)
for (j = i + 1; j < m; j++)
 s += a[i] [j];
```

Example 6.11. Sorting matrix columns in non-decreasing order of their maximum elements.

```
for (i = 0; i < n; i++)
   {
b[i] = a[i][0];
for (j = 1; j < m; j++)
if (a[i][j] > b[i]) b[i] = a[i][j];
   }

for (i = 0; i < n - 1; i++)
for (j = i + 1; j < m; j++)
if (b[i] > b[j])
{
 t = b[i];
 b[i] = b[j];
 b[j] = t;
for (k = 0; k < m; k++)
  {
 t = a[i][k];
 a[i][k] = a[j][k];
 a[j][k] = t;
  }
   }
```

Example 6.12. Obtaining a matrix of order *n* – 1 from an *n*-th order matrix by removing from the original matrix the rows and columns at the intersection where the element with the smallest value is located.

```
imin = jmin = 0;
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
if (a[i][j] < a[imin][jmin]) { imin = i; jmin = j; }

for (i = 0; i < n; i++)
for (j = jmin; j < m - 1; j++) a[i][j] = a[i][j+1];
        m--;
for (j = 0; j < m; j++)
for (i = imin; i < n - 1; i++) a[i][j] = a[i+1][j];
        n--;
```

7. Pointers

7.1. Pointer Declaration

Memory of the computer represents the array of consistently numbered cells. At data declaration in memory the continuous area for their storage is selected. For example, for the type **int** variable is allocated in 4-byte memory area (8 bytes on 64-bit systems). The first byte number selected under the memory section variable is called the address of this variable.

The pointer is the variable of the memory address value.

Pointers are used for:

– dynamic memory allocation;

– parameter passings in functions;

– appeal to the data structures elements.

Format of the declaration of the pointer:

```
data_type *pointer_name;
```

For example:

```
int *a; double *b, *d; char *c;
```

Any number of pointers, including different types, can point to the same memory location. Pointer variables can be described by a pointer (a pointer to a memory cell, which in turn contains the address of another memory cell).For example:

```
int *um1, **um2, ***um3;
```

In the C language there are three types of pointers:

1) the pointer on the object of the known type;

2) **void** pointer. It is applied in cases when the object type is not defined in advance;

3) The pointer on function. Allows to handle with functions, as variables.

7.2. Operations over Pointers

7.2.1. Unary Operations

Two unary operations can be performed on pointers:

1. "**&**" **(address-of operator address-of operator)**. Operation allows to receive the variable address.

2. "*****" **(indirection operator)**. Allows to get access to the value located at the specified address.

7.2.2. Arithmetic and Comparison Operations

Pointer arithmetic automatically takes into account the size of the data to point to.

Increment and decrement. Moves the pointer to the next or previous array cell. For example:

```
int *um, a[5] = {1, 2, 3, 4, 5};
um = a;
```

```
cout << *um << endl; // Displays: 1
    um++;
cout << *um << endl; // Displays: 2
```

Adding or subtraction. Moving a pointer by a number of bytes equal to the product of the size of the given type pointed to by the pointer times the value of the constant being added or subtracted. For example:

```
int *um, a[5] = {1, 2, 3, 4, 5};
um = a;
cout << * m << endl; // Displays: 1
    um += 3;
cout << *um << endl; // Displays: 4
```

Difference of pointers. The difference of two pointers is equal to the number of the objects of the corresponding type placed with this address range. For example:

```
int * um, a[5] = {1, 2, 3, 4, 5};
 um = &a[0];
 un = &a[4];
   k = un - um;
cout << k << endl; // Displays: 4
```

Comparison operations. Compare the addresses of objects.

7.3. Pointers Initialization

Initialization by empty value. For example:

```
//C style
        int *a = NULL;
        int *b = 0;
// Style C++ (since v.11)
        int *c = nullptr;
```

Assignment to the pointer **of the address of already existing object**. For example:

```
int k = 23;
        int *uk = &k;  // or  int *uk(&k);
        int *us = uk;
```

Assignment to the pointer of the address of the selected section of the dynamic memory:

```
int *s = new int;
        int *k = (int*)malloc(sizeof(int));
```

Here the operation **sizeof** which determines the size of the specified parameter in bytes is used.

7.4. Dynamic Memory

Dynamic memory (*heap*) is the memory special area for the program runtime. It is possible to select and make place according to the current requirements. Access to the memory selected sections is provided through pointers. For work with the dynamic memory in the C language (*malloc.lib* library) the following functions are defined:

1) void ***malloc**(size) selects area of memory with the size **size** of bytes. Returns the address of the selected memory unit. If the function failed to allocate the requested block of memory, a **NULL** pointer is returned;

2) void ***calloc** (n, size) selects area of memory with size **n** of blocks on **size** of bytes. Returns the address of the selected memory unit. If the function failed to allocate the requested block of memory, a **NULL** pointer is returned. All selected memory is filled with zero;

3) void ***realloc** (*u, size) the extent of earlier selected memory connected with pointer **u** on new number of bytes. On success, returns the pointer to the beginning of newly allocated memory. On failure, returns a **NULL** pointer;

4) void **free**(*u) releases the memory section connected with pointer **u**.

In language C++ for selection and release of memory the operations **new** and **delete** are defined.

Two forms of operations are had:

1) *type *pointer* = **new** *type*(*value*) selection of a memory cell by a given type for a specified value.

delete pointer is release of the selected memory;

2) *type *pointer* = **new** *type[n]* is selection of memory size of the **n** cells of the specified type.

delete []*pointer* is release of the selected memory.

The **delete** operation does not destroy the values connected with the pointer, and permits the compiler to use this section of memory.

7.5. One-dimensional Dynamic Array

For creation of the one-dimensional dynamic array it is necessary to know type of array cells and their quantity. For example, the following functions can be used to create a one-dimensional dynamic array of n real numbers:

```
umas1 = static_cast <double*> (malloc(n*sizeof(double)));
```

(deallocat memory – **free**(umas1))

or

```
umas1 = static_cast <double*> (calloc(n, sizeof(double)));
```

(deallocat memory – **free**(umas1))

or

```
umas1 = new double(n*sizeof(double));
```

(deallocat memory – **delete** umas1)

or

```
    umas1 = new double[n];
```

(deallocat memory – **delete** []umas1)

7.6. Two-dimensional Dynamic Array

The two-dimensional dynamic array is considered by the compiler as pointer array on one-dimensional arrays (fig. 7.1).

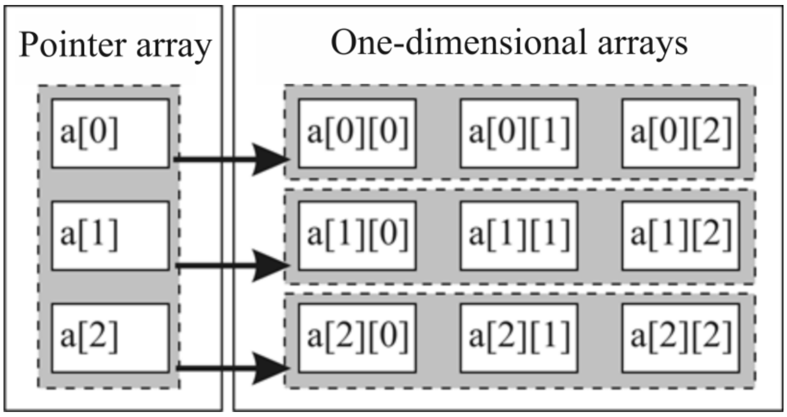


Fig. 7.1

In the beginning memory under one-dimensional pointer array is selected, then each pointer receives the address of the created one-dimensional dynamic array. Release of memory is performed upside-down.

```
    double **umas2;            // The declaration of the pointer on the array
    // Memory allocation for placement of pointer array
       umas2 = new double* [n];
    // Memory allocation for placement of one-dimensional arrays
       for (i = 0; i < n; i++) umas2[i] = new double[m];

       …             // Work with the array

    // Release of the memory selected for one-dimensional arrays
          for (i = 0; i < n; i++) delete []umas2[i];
    // Release of the memory selected for pointer array
          delete []umas2;
       umas2 = nullptr;        // Cleaning of the pointer
```

The following method of memory allocation is resolved:

```
    const int m = 3;
           int n = 4;

          int(*mas)[m] = new int[n][m];
```

```
        …   // Work with the array

     delete []mas;
  mas = nullptr;  // Cleaning of the pointer
```

When selecting a multidimensional array, all dimensions except the first must be positive constants or constant expressions of an integer type. The first dimension can be a positive integer variable.

8. Functions

8.1. Function Concept

Function is the operators sequence issued in such a way that it can be caused by name from any place of the program.

Function is described as follows:

```
 type_returned_value name_function(parametr_list)
          {
     Function_body
           }
```

The first line of this description is called *function heading*. The returned value type can be any, except for the array or function. If function does not return value, then the **void** type is specified. In C++ by default type of the returned result − **int**.

The parameter list of function (formal parameters) represents the set of constructions of the following form:

```
parametr_type  parametr_name
```

For example:

```
int sum(int a, double b, char c)
```

If function does not obtain any data, then brackets remain empty:

```
int fun()
```

Prototypes of functions (their preliminary declaration) are widely used. The prototype is similar to the function heading except for that names of formal parameters are not entered (there are only types), and the semicolon is put at the end:

```
int sum(int, double, char);
```

Wide use of prototypes is caused by the following:

– functions with prototypes can be called from other modules;

– use of prototypes allows to place functions in any order (but not before their first use);

– the prototypes placement in one place makes the program more readable.

Rules of the design of the function body are the same, as well as for any other section of the program. All declarations have local character, i. e. the announced variables are available only in function.

Attachment of functions to each other is not allowed.

The output comes from function at achievement of the bracket closing function or after execution of **return** operator.

Some specifiers are allowed when declaring a function:

1) **constexpr** – the function returns a constant;

2) **inline** – instructs the compiler to replace each function call with the code of the function itself. Increases the speed of the program. If a specifier is ineffective, the compiler can ignore it;

3) **noexcept** – The function throws an exception.

8.2. Parameter Passing

During the program executing the following rule should apply: when declaring and calling a function, the arguments (actual parameters) and formal parameters must match in number, sequence order and types.There are three main methods of parameter passing: transfer on value, according to the link and according to the pointer.

8.2.1. Parameter Passing on Value

In function temporary variables to which values from defiant function are transmitted are created. For example:

```
     int fun1(double, int, char);             // Function prototype
   …
      int fun1(double a, int b, char c)       // Function heading
           {
                  //   Function body
           }
   …
    int s = fun1(d, 8, chr);                  // Function call
```

At the time of the appeal to function in memory temporary variables with names *a*, *b*, *c* are created. In the created variables values are copied: *d*, *5*, *chr*. After that communication between the transferred and temporary variables is broken off.

Parameter passing advantages on value:

1. As arguments it is possible to use variables, constants, expressions, structures, classes, enumeration.

2. Data in the main program are protected from change in function.

Parameter passing shortcomings on value:

1. Costs of time and memory for copying of values. Copying structures and classes can lead to considerable decline in production therefore it is not recommended.

2. There is no possibility of data transmission (through parameters) in defiant function.

8.2.2. Parameter Passing According to Link

The addresses of arguments from the main program are transferred to function. Reference parameter ("alias") is the alias of the corresponding argument. For receipt of the address the operation "take the address" is used. For example:

```
   void fun2(double&, int&);       // Function prototype
                 …
   void fun2(double &a, int &b)  // Function heading
       {
                 //   Function body
    }
                 …
```

```
fun2(d, r);      // Function call
```

By such challenge not the variable, but its address received with use of the operation "take the address" is transferred. Therefore, at the appeal to formal parameter in fact there is the appeal to the argument in defiant function.

Parameter passing advantages according to the link:

1. Economy of resources, connected with the fact that to transfer there is no copying of arguments.
2. Possibility of transfer to defiant function of any amount of values.
3. The possibility of change of arguments at change of parameters.

Parameter passing shortcomings according to the link:

1. On function call it is impossible to determine the method of the parameter transfer (by value or by the link).
2. Function can change value of the agrument that can lead to program errors. For the solution of this problem it is possible to use the **const** key word before the corresponding agrument.

8.2.3. Parameter Passing to Address

At function call as the argument is not the variable, but its address is transferred. For example:

```
void fun3(double *, int *);       // Function prototype
                …
void fun3(double *a, int *b)    // Function heading
  {
      //   Function body
  }
                …
      fun3(&f, &k); // Function call
```

Parameter is the pointer receiving the argument address. Application the redirection operation allows to change argument value. If a parameter is assigned a different address, then it will lose its connection with the argument and will not be able to use its value.

	During the performance with the parameters transferred according to the pointer it is necessary to use the redirection operation, for example: `s = (*a + *b)/2;`

The pros and cons of passing parameters to an address are similar to those of passing parameters by reference.

8.2.4. Parameters with Values by Default

At function declaration for some arguments it is possible to set value by default which is transmitted to function if by the challenge the corresponding argument is not set. As the compiler appropriates the available values consistently from left to right, the arguments important set by default should be located more to the right of the arguments which do not have such value. For example:

```
void fun4(double, int b = 3, double h = 0.1); // Function prototype
                 …
void fun4(double a, int b, double h)         // Function heading
     {
       // Function body
         }
```

By the challenge

```
        fun4(d);      // Function call
```

the value **d** specified by the challenge, and the rest since they are absent in the list is transferred to variable **a**, values by default are appropriated (*b* = 3, *h* = 0.1).

By the challenge

```
        un4 (d, r)
```

variable **a** will transfer value **d**, to variable **b** – value **r**, and variable **h** value by default *of h* = 0.1.

By the challenge

```
        un4 (d, r, f)
```

the values specified by the challenge will be transferred to all variables. Values by default are not used.

The admission of arguments at function call is prohibited. Default arguments should be specified at the first mentioning of function.

8.2.5. Arrays Transfer to Functions

By transfer of the array to function the corresponding parameter should contain type, the name of the array and square brackets. By the challenge only the array name is entered. For example:

```
void funm1(int[]); // Function prototype
                 …
void funm1(int b[]) // Function heading
{
//   Function body
}
                 …
funm1(a);   // Function call
```

C++ transfers the array name according to the link, i.e. at change of array cells in function elements of the corresponding array in the defiant procedure change.

As a rule, not only the array itself is passed to the function, but also its size is passed too.

By transfer of the multidimensional array of the bracket for the first dimension remain empty, and for other dimensions the size should be specified by the constant. For example, transfer of the two-dimensional array of 3×3 in size will be organized as follows:

```
void funm2(int [][3]);    // Function prototype
              …
void funm2(int b [][3])  // Function heading
{
      //   Function body
}
              …
      funm2(a);              //  Function call
```

8.2.6. Transfer Parameters of Variable Number

Function declaration format with variable number:

```
name_function type_returned_value(list_parametr, …)
```

The parameter list contains at least one required parameter. The dots (the ellipsis, English "ellipsis") indicate the possibility of adding of any number of parameters.

For work with parameters the type of the va_list list and three macros is defined:

```
void va_start(va_list pointer, name_last_required_argument)
```

begins work with the list. Sets the pointer on the first optional argument.

```
void va_arg(va_list pointer, type_argument)
```

returns the next argument value from the list. Each start of the macro transfers the pointer to the following argument. Achievement of the last argument of the list is not controlled.

```
void va_end(va_list pointer)
```

completes the work with the list and releases memory.

Example: Count the sum of the entered arguments. The input termination condition is the argument value equal to –1.

```
int fun5(int...);             // Function prototype
              …
int fun5(int a...)            // Function heading
{
      int ar, s;
 va_list argm;
   s = a;
 va_start(argm, a);
    ar = va_arg(argm, int);
while (ar != -1)
    {
    s += ar;
     ar = va_arg(argm, int);
    }
 va_end(argm);
return s;
}
```

Function call: **int** r = fun5(1, 2, 3, 4, 5, 6, -1);

8.3. Functions Overload

The functions overload is understood as using different functions with the identical name. Overloaded functions differ with the compiler on types and number of parameters. For example, if it is necessary to calculate the area of the circle or rectangle, it is possible to write the following functions:

```
double Ploch(double a, double b) { return a*b; }
double Ploch(double r) { return 3.14*pow(r, 2); }
```

8.4. Function Pointer

The function name is a constant pointer to the beginning of the function in random access memory. It is allowed to use pointers on function in the program.

For example, there is the function

```
double y(double x, int n)
{
      // Function body
}
```

The pointer on such function has the appearance:

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

If we assign the address of the function y to the pointer **fun**:

```
fun = y;
```

that function can be caused

```
x = fun(t, m);
```

Example. Display the table of function values $y(x) = \sin x$ and its decomposition in a row $s(x) = x - \frac{x^3}{3!} + ... + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$ with ε accuracy = 0.001. Display the number of iterations necessary for achievement of given accuracy.

```
#include <iostream>
#include <cmath>
```

```
#include <iomanip>
using namespace std;
typedef double (*uf)(double, double, int&);
void tabl(double, double, double, double, uf);
double y(double, double, int&);
double s(double, double, int&);
int main()
{
cout << setw(8) << "x" << setw(15) << "y(x)"
                                                        << setw(10) << "k" << endl;
       tabl(0.1, 0.8, 0.1, 0.001, y);
cout << endl;
cout << setw(8) << "x" << setw(15) << "s(x)"
                                                       << setw(10) << "k" << endl;
       tabl(0.1, 0.8, 0.1, 0.001, s);
       return 0;
}
void tabl(double a, double b, double h, double eps, uf fun)
{
       int k = 0;
       double sum;
       for (double x = a; x < b + h / 2; x += h)
       {
       sum = fun(x, eps, k);
cout << setw(8) << x << setw(15) << sum << setw(10) << k << endl;
       }
}
double y(double x, double eps, int& k)
{
       return sin(x);
}
double s(double x, double eps, int& k)
{
       double a, c, sum;
       sum = a = c = x;
       k = 1;
       while (fabs(c) > eps)
       {
            c = pow(x, 2) / (2 * k * (2 * k + 1));
            a *= -c;
            sum += a;
            k++;
        }
        return sum;
}
```

9. String Variables

There are two modes of work with string data in language C++: use of the array of characters like **char** and use of the class **string**. In this benefit the first method of the organization of work with lines is considered.

9.1. Rows Declaration

The declaration of the string is similar to array declaration:

char name_of_the_string[size]

Unlike the array the string will come to an end with null character '\0' – (zero terminator). Length of the string is equal to quantity of characters plus null character. At set the null character is located in the end terms automatically. For example, in string

char st1[10]="123456789";

characters are located as follows:

‘1’	‘2’	‘3’	‘4’	‘5’	‘6’	‘7’	‘8’	‘9’	‘\0’

If the size of the string is not announced obviously, then it will be set automatically and will be equal to quantity of the entered characters +1.

char st2[] = "1234";

It is authorized to use pointers for the string:

char *st3 = st2;

Access to separate characters of the string is provided by their indexes. For example: st[2] = 'e';

In the C language single quotes are used for designation of characters, and double is used for designation of lines

The array of lines appears as the two-dimensional array of characters:

char name [quantityof lines][quantity of characters in string];

For example,

char str[10][5].

Appeal to the zero string of the array of lines: str[0].

9.2. Rows Functions

The ***stdio.lib*** library functions are used to *input/output* strings and characters, the ***string.lib*** library functions are used to handle with null-terminated strings, the ***stdlib.lib*** library functions are used for type conversion, and the ***ctype.lib*** library functions are used for character recognition.

The following functions are most often applied:

1. int **puts** (const char *str) displays the string **str**. Transfers the pointer to following term.

2. char ***gets_s**(char *str, int n) places to the string **str n** of the characters entered from the keyboard. Returns **NULL** in case of the error.

3. errno_t **strcpy_s**(char *str1, int d, const char *str2) copies contents of string **str2** at string **str1**. Parameter d sets the size of the buffer which is used for transfer of the string. For receipt of the necessary size of the buffer it is possible to use **_countof** macro. Function returns zero in case of the error and the error code in case of failure (**errno_t** is the whole data type).

For example:

```
strcpy_s(str, _countof(str), "abc");
```

Result: **str = "abc"**.

4. errno_t **strcat_s**(char *str1, int d, const char *str2) – adds contents of string **str2** to **str1** string end.

For example:

```
char str[10] = "xyz";
        strcat_s(str, _countof(str), "abc");
```

Result: **str = "xyzabc"**.

5. int **strcmp**(const char * str1, const char * str2) – compares contents of strings **str1** and **str2** If **str1** < **str2**, result is equal –1 if **str1** = **str2**– result is equal to zero, if **str1** > **str2** – result is equal to 1.

```
char st1[40] = "ABCD", st2[40] = "xyz";
k = strcmp(st1, st2);
```

Result: k = –1.

6. char ***strchr**(char *str, int c) the pointer on the first emergence of the character ***c*** in the string **str**.

For example, it is required to define the position of the first emergence of the character BC in the string ABCD.

```
char st1[40] = "ABCD";
char *s = strchr(st1, 'C');
int k = static_cast <int> (s - st1);
```

Result: k = 2.

7. char ***strstr**(char *str1, const char *str2) the pointer on the first emergence of string **str2** in string **str1**.

For example, it is required to define the position of the first occurrence of the string BC at string ABCD.

```
char st1[40] = "ABCD";
char * s = strstr (st1, "BC");
k = static_cast <int> (s - st1);
Result: k = 1.
```

8. char* **strtok_s**(char *str, const char *dlm, char **context) the pointer on the lexeme which is in the string **str** (the symbol set separated from other lexemes by the delimiting character which is in the string **dlm** is considered the lexeme). The **context** parameter is used for storage of data on the unchecked part of the string.

At the first function calls of **strtok_s** function passes the leading separator and returns the pointer on the first lexeme in **str**. The lexeme comes to the end with null character. At the subsequent function calls with **NULL** value as the first argument the pointer the same way passes to the following lexemes. After finding of all lexemes the pointer receives **NULL** value.

Example 9.1. Display the lexemes separated by characters '-' and ':'.

```
char str[50], *wrd, *cn = NULL;
 gets_s(str,40);
 char dlm[] = "-:";
     wrd = strtok_s(str, dlm, &cn);
while (wrd != NULL)
{
        puts(wrd);
        wrd = strtok_s(NULL, dlm, &cn);
}
```

If st1 = "AAAA:BBBB B-C CC:-:DDDD D", it will be displayed:

```
AAAA
BBBB B
C CC
DDDD  D
```

If for the same string to put the divider equal to the space (char st2 [] = " ";), then the result will be following:

```
AAAA:BBBB
B-C
CC:-:DDDD
D
```

9. size_t **strlen**(const char *str) – returns length of the string **str** (zero terminator '\0’ it is not considered).

```
char str[40] = "ABCD";
      int k = strlen(str);
```

Result: k = 4.

10. char* **_strrev** (char *str) the sequence of characters in the string **str** to opposite.

```
char str[40] = "ABCD";
```

```
_strrev(str);
```

Result: str = "DCBA".

11. char* **_strdup** (const char *str) – duplicates the string **str**. For memory allocation under new the string function causes **malloc** (it is necessary to use **free()** for cleaning of memory at the end of work).

```
          char str1[40] = "ABCD";
          char *str2;
          str2 = _strdup(str1);
…
                     free(str2);
```

Result: st2 = "ABCD".

12. errno_t **_strlwr_s**(char *str, size_t n)– will transform n of capital characters (upper case) of the string **str** to lower case characters (lower case).

```
char str[40] = "aBcD";
         _strlwr_s (str, strlen(str) + 1);
```

Result: st = "abcd".

13. errno_t **_strupr_s**(char *str, size_t n) – will transform **n** of lower case characters (lower case) of the string **str** to capital characters (upper case).

```
char str[40] = "aBcD";
         _strupr_s(str);
```

Result: st = "ABCD".

14. int **atoi**(const char *str) – will transform to the string **str** to number of the whole type.

```
char st1[40] = "354553";
         int k = atoi(st1);
```

Result: k = 354553.

15. double **atof**(const char *str) – will transform the string **str** to number of the valid type.

```
char str[40] = "354.553";
double b = atof(str);
```

Result: b = 354.553

16. errno_t **_itoa_s**(int k, char *str, size_t n, int d) – will transform **n** of characters of the decimal integer number of **k** to the string **str** according to the set numeral system (from 2 to 36).

```
_itoa_s(25, str, _countof(str), 10);
```

Result: str = 25 in the decimal numeral system.

```
_itoa_s(25, str, _countof(str), 2);
```

Result: str = 11001 in the binary numeral system.

17. errno_t **_gcvt_s**(char *str, size_t n, double val, int dgt) will transform number of the valid val type to the string **str** size **n** of characters. The size of buffer **n** is recommended to be set the constant _CVTBUFSIZE (309 + 40). The number of decimal places DHT will be no more than 18.

```
double a = -254.2965;
char str[_CVTBUFSIZE];
        _gcvt_s(str, _CVTBUFSIZE, a, 7);  // st = "-254.2965"
        _gcvt_s(str, _CVTBUFSIZE, a, 5);  // st = "-254.3"
        _gcvt_s(str, _CVTBUFSIZE, a, 3);  // st = "-254"
        _gcvt_s(str, _CVTBUFSIZE, a, 1);  // st = "-3e+003"
```

Character recognition functions:

– int **isalnum**(*character*) returns nonzero value (*true*) if the **character** is the letter or digit;

– int **isalpha**(*character*) returns nonzero value (*true*) if the **character** is the letter;

– int **isdigit**(*character*) returns nonzero value (*true*) if the **character** is digit;

– int **ispunct** (*character*) returns nonzero value (*true*) if the **character** the punctuation symbol;

– int **islower**(*character*) returns nonzero value (*true*) if the **character** is the letter of the lower case;

– int **isupper**(*character*) returns nonzero value (*true*) if the **character** is the upper case letter;

– int **isspace**(*character*) returns nonzero value (*true*) if the **character** the space, the sign of tabulation, carriage return, the newline character, vertical tabulation, transfer of the page.

9.3. Operation Algorithms with Strings

Example 9.2. Check is the word "*visual*" at the set string.

```
char st[30];
        char* ch = nullptr;
        puts("Enter the string ");
        gets_s(st,20);
        ch = strstr(st, "visual");
        if (ch != nullptr) puts("Present");
        else puts("Not Present");
```

Example 9.3. In the string st to delete all characters of 'z'.

```
for (int i = 0; i < strlen(st); i++)
        if (st[i] == 'z')
        {
      for (int j = i; j < strlen(st); j++)  st[j] = st[j + 1];
```

```
            i--;
        }
```

Example 9.4. Select and print all words of any string. Words separate from each other one or several spaces.

```
char str[100], sl[100];     int k = 0;
 gets_s(str, 100);
  strcat_s(str, _countof(str), " ");
    int n = strlen(str);
for (int i = 0; i < n; i++)
   if (str[i] != ' ') sl[k++] = str[i];
    else
      if (k > 0) {
                   sl[k] = '\0';
                     puts(sl);
          sl[0] = '\0';
          k = 0;
                 }
```

Example 9.5. Define whether the string is the palindrome, i.e. whether it is read from left to right as well as from right to left (for example, "Was it a cat I saw").

```
char str[80] = "Was it a cat I saw";
         _strlwr_s(str, strlen(str) + 1);
        int i = 0,  j = strlen(str) - 1;
                   bool bl = true;
        while (i <= j) {
                   while (str[i] == ' ') i++;
                   while (str[j] == ' ') j--;
                             if (str[i++] != str[j--])
                   {
                             bl = false;
                             break;
                   }
                             }
        if (bl)  cout << "Palindrome" << endl;
           else cout << "Not a palindrome" << endl;
```

Example 9.6. Find the shortest and longest word in a given sentence.

```
char str[100];
char *wrd, *cmin, *cmax, *cn = nullptr;
gets_s(str, 100);
char sl[] = " ";
wrd = strtok_s(str, sl, &cn);
cmin = cmax = wrd;
while (wrd != nullptr)
{
        if (strlen(wrd) > strlen(cmax)) cmax = wrd;
        else
                if (strlen(wrd) < strlen(cmin)) cmin = wrd;
        wrd = strtok_s(nullptr, sl, &cn);
}
```

10. Users Data Types

10.1. Structures Declaration and Implementation

Was it a cat I saw is the composite data type in which under one name functions and data of different types are joint. Declaration of structure:

```
struct name_structure
 {
member-list
    } ;
```

Data is named by *fields* and functions are called by *methods*. They can be structure members.

The fields and methods description rules are similar to the description of data and functions.

Example of structure declaration with several fields:

```
struct struc1
{
        int m1;
        double m2, m3;
};
```

Fields of structure can be any type, including arrays and structures.

After the curly bracket it is admissible to specify variables of the corresponding structural type:

```
struct struc1
{
        int m1;
        double m2, m3;
} a, b, c;
```

Variable declaration of structural type:

```
struc1 x;
```

It is possible *to address* separate parts of structure through the compound name. Address format:

```
name_structure.name_field_or_method
```

or

```
pointer_to_structure->name_field_or_method
```

For example, if the structure is announced as follows:

```
struct struc1
```

```
{
    int m1;
    double m2, m3;
} x, *y;
```

that it is possible to address *m*1 field (after memory allocation for *y*):

```
x.m1 = 35;
```

or

```
(&x)->m1 = 35;
```

or

```
y->m1 = 35;
```

or

```
(*y).m1 = 35;
```

The rules for processing with structure fields are identical to working with variables of the corresponding types. It is possible to initialize variables structures by the room behind the declaration of the list of starting values.

```
struct struc1
{
    int m1;
    double m2, m3;
} a = {5, 2.6, 34.2};
```

As fields other structures can be used.

```
struct struc1
{
    int m1;
    double m2, m3;
      struct
       {
             int mm1;
       } m4;
} s;
```

The appeal to **mm1** field in this case will be the following:

```
s.m4.mm1 = 3;
```

If the name of structure is not entered, then such determination is called *anonymous*.

It is admissible to use operation of assignment for structures of one type. For example:

```
struc1 x, y;
      …
  x = y;
```

In this case all field values of structure of **y** are copied in the corresponding fields of structure **x**.

As a rule, arrays are organized from structures:

```
struct struc1
{      int m1;
       double m2, m3;
};
…
struc1 ms[100];          //  Array declaration of structures
…
ms[99].m1 = 56;  //  Appeal to the field of the array of structures
```

Example. There is a list of residents of the apartment house. Each element of the list contains the following information: the surname, number of the apartment, the number of rooms in the apartment. Display in alphabetical order surnames of the residents living in two-room apartments. Select memory for storage of the list dynamically.

```
int main()
{
   struct tzhilec
   {
      char fio[50];
      int nomer;
      int nrooms;
   } *spisok;

   int n, i, j;
   cout << "Enter the number of residents: " << endl;
   cin >> n;
   spisok = new tzhilec[n];
   for (i = 0; i < n; i++)
   {
      cout << "Enter your last name: ";
      cin >> spisok[i].fio;
      cout << "Enter apartment number: ";
      cin >> spisok[i].nomer;
      cout << "Enter the number of rooms: ";
```

```
        cin >> spisok[i].nrooms;
        cout << endl;
    }
    tzhilec tmp;
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (spisok[i].nrooms == 2 && spisok[j].nrooms == 2
                && strcmp(spisok[i].fio, spisok[j].fio) == 1)
            {
                tmp = spisok[i];
                spisok[i] = spisok[j];
                spisok[j] = tmp;
            }
    for (i = 0; i < n; i++)
        if (spisok[i].nrooms == 2)
            cout << spisok[i].fio << ", apartment number - "
            << spisok[i].nomer << endl;
    delete[]spisok;
    return 0;
}
```

10.2. Unions

*Union (*union) is placement under one name of the definite data set so that the size of the allocated memory is sufficient to accommodate the data with the largest size. These structures are used when individual fields exist at different times.

Union Declaration:

```
union union_name
    {
member-list
    };
```

For example:

```
    union per {
        int a;
        double b;
        char  c;
    } un;
    un.a = 567;
cout << un.a << endl;              // The un.a value is equal to 567
```

```
un.b = 8.2;
cout << un.a << endl;          // 1717986918 is Error!
cout << un.b << endl;          // The un.b value is equal to 8.2
```

10.3. Enumerations

The enumeration (*enum*) sets is a set of values for the uservariable set. Declaration of enumeration:

```
enum name {enum-list};
```

For example:

```
enum otc {unsatisf, satisf, good, exc};
```

Number is assigned to each value in enumeration. By default the first value has number 0, the second is 1, etc. It is possible to set numbering, other than set, by default:

```
enum otc {unsatisf = 2, satisf, good, exc};
```

The declaration can be combined with initialization of variables:

```
enum otc {unsatisf = 2, satisf, good, exc} a, b = satisf;
```

Transfers can implicitly be transformed to integral types, but not on the contrary:

```
otc a = unsatisf;
int k = a;    // Admissible operation
    a = 2;    // Unsupported operation
```

11. Files

11.1. File Concept

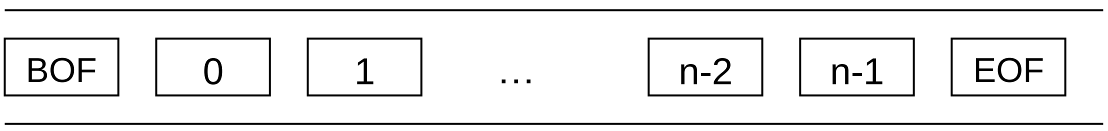
A file is a named set of data located on an external storage device. At the start of the process, the file must be open for data access. After the file is opened, the current position pointer is placed at the beginning of the file. After any data operation, the pointer moves forward one position. At the end of processing, the file is closed, i.e. access to the data placed in the file will be denied. Information about the file is stored in a control structure of the **FILE** type.

There are two types of files: *text* and *binary*.

Text files store information in the form of string. The output is performed similar to the output to the screen. Text files can be edited in any text editor.

Binary files are intended for storage of the sequence of bytes. The structure of such file is defined programmatically.

The files placed on information mediums have the following structure:



At the beginning of the file, information about the BOF file (Begin of File), its name, type, length, etc. is written, at the end of the file there is the final character of the EOF file (End of File). For the empty **BOF** and **EOF** are combined.

During the work with files the following macroes are used:

– NULL defines the empty pointer;
– EOF is the value returned in attempt of reading after the end of the file;
– FOPEN_MAX returns the maximum number of at the same time open files.

11.2. Files Functions

Functions for file operations are located in the *stdio.lib* and *io.lib* libraries. **FILE** pointers are used for file operations. The file pointer declaration format is:

```
FILE *file_pointer;
```

For example:

```
FILE *fl1, *fl2;
```

The pointer contains the address of the structure including different data on the file, for example, its name, the status and the pointer for the beginning of the file.

Function

```
errno_t fopen_s(FILE **pFile, const char *filename, const char *mode);
```

opens the file and connects it with the flow. Returns the pointer to the open file.

Parameters:

– *pFile* is the file pointer which receives the pointer on the open file;

– *filename* is the pointer on the string of characters in which the file name and the way to it is stored. For example: "d:\\work\\lab2.dat";

– *mode* is the pointer on the string of characters in which the mode of opening of the file is specified. The admissible modes are given in tab. 11.1.

Table 11.1

Mode opening	Action
r (or rt)	Opens the text file for reading. In case of lack of the file with the entered name there is the error
w (or wt)	Creates the text file for record. If the file with the entered name already exists, then former information is destroyed
a (or at)	Opens the text file for record. The pointer is established in the end of the file
rb	Opens the binary file for reading. In case of lack of the file with the entered name there is the error
wb	Creates the binary file for record. If the file with the entered name already exists, then former information is destroyed
ab	Opens the binary file for record. The pointer is established in the end of the file
r + (or rt+)	Opens a text file for the reading and writing data
w + (or wt+)	Creates the text file for the reading and writing data
a + (or wt+)	Opens the text file for the reading and writing data. The pointer is established in the end of the file. If the file with the entered name is absent, then it will be created
rb + (or r+b)	Opens the binary file for the reading and writing data
wb + (or w+b)	Creates the binary file for the reading and writing data
ab + (or a+b)	Opens the binary file for the reading and writing data. The pointer is established in the end of the file. If the file with the entered name is absent, then it will be created

By default the file opens in the alpha mode.

If the file was successfully opened, the function returns 0, otherwise it returns an error code.

The file can be created in the following way:

```
FILE* fl;
 errno_t err;
 err = fopen_s(&fl, "lab.dat", "w+b");
 if (err == 0)  cout << "The file was opened";
 else {
     cout << "The file was not opened";
```

```
            return 1;
          }
```

For the error exception arising when the nonexistent file opening it is possible to use construction

```
      err = fopen_s(&fl, "lab.dat", "r");
       if (err) err = fopen_s(&fl, "lab.dat", "w");
```

Records are exchanged not directly to a file, but to some kind of buffer. The information from the buffer is overwritten to the file only when the buffer overflows or the file is closed.

For closing of the file function is used

```
   int fclose(FILE *file_pointer);
```

The function closes the stream that was opened by calling **fopen()** and writes any data still left in the disk buffer to a file. The file access after execution of function will be prohibited. If the file was closed without errors, then function returns zero, differently is to **EOF**.

For closing of all open files function is used

```
   int _fcloseall(void);
```

Function

```
   int fputc(int character, FILE *file_pointer);
```

writes the *character* in current position of the specified open file. If function was executed successfully, then it returns the written character, differently – **EOF**.

Function

```
   int fgetc(FILE *file_pointer);
```

reads one character from current position of the specified open file. After reading the pointer moves on one position forward. The function returns **EOF** at the end of the file.

Function

```
   int feof(FILE *file_pointer);
```

returns other than 0 value (*true*) in attempt of data reading after the end of the file, and 0 (*false*) if the end of the file is not reached. Function operates with files of all types.

Function

```
   int fputs(const char *string, FILE file_pointer);
```

writes the *string* of characters in current position of the specified open file. In case of the error this function returns **EOF**. The null character in the file does not register.

Function

```
   char* fgets(char *string, int length, FILE *file_pointer);
```

reads the *string* of characters from current position of the specified open file until the character of word wrapping or quantity of the read characters is read there will be no **length-1** equal. In case of the error function returns **EOF**.

Function

```
int *fprintf(FILE * file_pointer,
                    const char * string_of_formatting[arguments]);
```

writes the formatted data in the file. The string of formatting is similar to the string of formatting of the printf function.

Function

```
int fscanf_s(FILE * file_pointer,
const char* string_of_formatting[arguments]);
```

reads the formatted data from the file. The *string* of formatting is similar to the *string* of formatting of the scanf function.

Function

```
void rewind(FILE *file_pointer);
```

sets the pointer of current position in the beginning of the file.

Function

```
int ferror(FILE *file_pointer);
```

defines whether there was the error in operating time to the file. It returns nonzero value if at the last operation with the file there was the error, differently returns 0 (*false*).

Function

```
size_t fwrite(const void *writing_data, size_t size_element,
                              size_t number_element, FILE *file_pointer);
```

writes the set number of data of given size in the file. Data size is set in bytes. The **size_t** type is defined as whole without sign. Function returns number of recorded spots.

Function

```
size_t fread(void *variable, size_t size_element,
                              size_t number_element, FILE *file_pointer);
```

reads out the set number of data of the specified size in the specified variable. Data size is set in bytes. Function returns number of the read elements.

Function

```
int _fileno(FILE *file_pointer);
```

returns value of the descriptor of the specified file (the descriptor – the logical file number for the set flow).

Function

```
long _filelength(int descriptor);
```

returns file length with the corresponding descriptor in bytes.

Function

```
int _chsize(int descriptor, long size);
```

sets new file size with the corresponding descriptor. If file size increases, null characters are added to the end if file size decreases, all excess data are removed. In case of successful change, function returns 0 (differently – 1).

Function

```
long ftell(FILE * file_pointer);
```

returns value of the pointer on current position of the file.

Function

```
int fseek (FILE * file_pointer, long int number _bytes, int starting_point);
```

sets the pointer in the set position. The set number of bytes is counted from the reference mark which is set by the following macroes: the beginning of the file is **SEEK_SET**, current position is **SEEK_CUR**, the end of the file is **SEEK_END**. At the successful completion of work, function returns zero, and in case of the error nonzero value.

Example. Create a program code for work with the binary file containing the list of residents of the apartment house. Each element of the list contains the following information: the surname, number of the apartment, the number of rooms in the apartment. Find and display the text file containing information on the residents living in three-room apartments.

```
#include <iostream>
#include <stdio.h>
#include <io.h>
using namespace std;

FILE* fl;
struct TOwners  {
   char fio[50];
   int num;
   int nrooms;
}  *list, owner;
char fname[20] = "";
int n = 0;
void fadd();                  // Enter the list
void frd();                   // Read the list
void rezc();                  // Display result
void rezf();           // Display result in the file
int menu();            // Menu
```

```
bool flopen(const char*); // Work with the file

int main()  {
   while (true)
   {
      switch (menu())
      {
      case 1: fadd();    break;
      case 2: frd();   break;
      case 3: rezc(); break;
      case 4: rezf();    break;
      case 5: return 0;
      default: "Error!";
      }
      system("pause");   // Key press waiting
      system("cls");        // Cleaning of the screen
   }
   return 0;
}

int menu()  {
   cout << "Select:" << endl;
   cout << "1. Enter data and write to file" << endl;
   cout << "2. Read data from file" << endl;
   cout << "3. Display the result on the screen" << endl;
   cout << "4. Output the result to a file" << endl;
   cout << "5. Exit" << endl;
   int i;     cin >> i;
   return i;
}
bool flopen(const char* r)  {
   if (!strlen(fname)) {
      cout << "Enter file name" << endl;
      cin >> fname;
   }
   if (fopen_s(&fl, fname, r)) {
      cout << "Error" << endl;
      return false;
```

```
    }
    else  return true;
}

void fadd()   {
    if (!flopen("ab+")) return;
    int i, n;
    cout << "Enter the number of owners";
    cin >> n;
    for (i = 0; i < n; i++) {
        cout << "Enter name: ";                      cin >> owner.fio;
        cout << "Enter apartment number: ";    cin >> owner.num;
        cout << "Enter the number of rooms: ";  cin >> owner.nrooms;
        fwrite(&owner, sizeof(TOwners), 1, fl);
    }
    fclose(fl);
}

void frd()  {
    if (!flopen("rb")) return;
    n = _filelength(_fileno(fl)) / sizeof(TOwners);
    list = new TOwners[n];
    fread(list, sizeof(TOwners), n, fl);
    for (int i = 0; i < n; i++)
        cout << endl << list[i].fio << "Number kvartiry - "
                        << list[i].num << "Chislo komnat - " << list[i].nrooms;
    cout << endl;
    delete[]list;
    fclose(fl);
}
void rezc()  {
    if (!flopen("rb")) return;
    n = _filelength(_fileno(fl)) / sizeof(TOwners);
    for (int i = 0; i < n; i++)
    {
        fread(&owner, sizeof(TOwners), 1, fl);
        if (owner.nrooms == 3)
   cout << owner.fio << " ,apartment number - " << owner.num << endl;
```

```
    }
    fclose(fl);
}

void rezf()   {
    char fnamet[20];
    cout << "Enter the name of the text file" << endl;
    cin >> fnamet;
    FILE* ft;
    if (fopen_s(&ft, fnamet, "w")) {
        cout << "File not created";
        return;
    }
    if (!flopen("rb")) return;
    n = _filelength(_fileno(fl)) / sizeof(TOwners);
    for (int i = 0; i < n; i++)
    {
        fread(&owner, sizeof(TOwners), 1, fl);
        if (owner.nrooms == 3)
         fprintf(ft, "%s, apartment number - %d\n", owner.fio, owner.num);
    }

    fclose(fl);
    fclose(ft);
}
```

12. Visibility Area and Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ program. The *storage class* is time during what the variable exists in memory of the computer and within a C++ Program. The time frame between creation and destruction of the variable is called *lifetime* of the variable.

In language C++ 4 storage classes are defined:

Automatic, local (**auto**) storage class. The area of visibility of local variables is limited to function or the block in which it is announced. Lifetime of local variable is the period between its declaration and completion of work of function or the block in which it is announced. Time constraint on life of the variable allows to save random access memory. This storage class is used by default.

Static, local (**static**) storage class. The variable has the same area of visibility, as well as automatic. Lifetime of static local variable is the period between the first appeal to the function supporting it and completion of work of the program. Initialization of the variable happens only at the first appeal to function. The compiler keeps variable value from one function call to another. If the static variable is not initialized obviously, it by default matters 0.

External, global (**extern**) storage class. Global variables appear out of functions and are available in all functions which are below the description of global variable. At the time of creation the global variable is initialized by zero. Inclusion of the key word *of extern* allows function to use external variable even if it is defined later in this or other file. Memory for global variables is selected at the beginning of the program and released at completion of its work.

Register, local (**register**) storage class. There is only a "wish" for the compiler to place frequently used variables in processor registers to speed up the program execution. If the compiler refuses to place the variable in the processor registers, then the variable becomes "automatic".

If at variable declaration the storage class is not specified obviously, then it is set automatically depending on location of the variable in the text of the program. The variables announced in function by default have **auto** storage class, and the others are **extern**.

13. Recursive Algorithms

13.1. Recursion Concept

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function The problem recursively solving means to spread out it to subtasks which then the same way (i. e. recursively) break into smaller subtasks, and so until at the certain level of the subtask do not become so simple that can be solved is trivial. By the consecutive solution of all elementary subtasks it is possible to receive the solution of all the task. Function is called *recursive* if its body contains the link of the similar function.

For example, it is necessary to calculate number n factorial ($n!$). It is known that $n! = n \cdot (n - 1)!$. Therefore, for calculation of $n!$ it is necessary to calculate $n \cdot (n - 1)!$, in turn for calculation $(n - 1)!$ we calculate $(n - 1) \cdot (n - 2)!$, for calculation $(n - 2)!$ we calculate $(n - 2) \cdot (n - 3)!$ etc. On each step the value of the calculated factorial decreases per unit. The task breaks to until the value n becomes equal 0, i. e. trivial solution will be received 0!= 1. The program code of the factorial calculation:

```
int fact(int n)
{
        if (n <= 0) return 1;
        else return n*fact(n-1);
}
```

Let's consider work of function for calculation 4!. Process of recursive calls and return of values is shown in fig. 13.1.

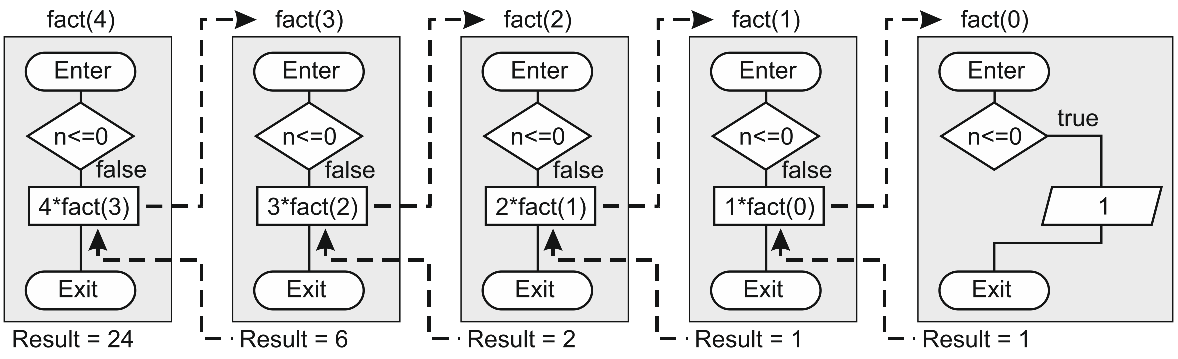


Fig. 13.1

During program execution data are stored in special area of the memory called by the *stack*. Values of variables for which memory is automatically selected are kept in stack area of memory. The stack structure: the data is entered sequentially and then retrieved in the reverse order (first in - last out). By each challenge of recursive function local data remain in the stack. After achievement of the bottom of the recursion there is consecutive data sampling from the stack.

13.2. Recursive Algorithm Termination Condition

If the condition of the recursive calls of function termination is not provided in the recursive algorithm, then such algorithm will cause it infinitely (until the stack is crowded). Therefore, the program must have an operator to stop the recursive function call for certain values of the current data. To prevent a program stack from overflowing, it is necessary to estimate the maximum recursion depth.

Infinite recursion can occur not only in the absence of a termination condition for a recursive function call. This is also possible with incomplete consideration of all possible recursion conditions. For example, calculating the factorial:

```
int fact(int n)
{
      if (n == 0) return 1;
      else return n*fact(n-1);
}
```

Then if a number less than zero is entered, the function will be called indefinitely.

13.3. Examples of Recursive Algorithms

Example 13.1. Find the sum $S_n = \sum_{i=1}^{n} a_i$.

```
int sumr(int i)
{
  if (i < 0) return 0;
    else return a[i] + sumr(i-1);
}
```

Example 13.2. Find the greatest common divisor of two numbers. Euler found the following ratio: if B shares on A totally, then GCD(A, B) = A else GCD(A, B) = GCD(B % A, A).

```
int GCD(int a, int b)
{       if( b%a == 0) return a;
                  else return GCD(b%a,a);     }
```

Example 13.3. Find max ($a_1...a_n$). This task can be broken into the following **elementary subtasks**: max(max ($a_1...a_{n-1}$), a_n), and further max max (max (max ($a_1...a_{n-2}$), a_{n-1}) a_n),…, each of which is solved the choice: if ($x > y$), then $mx = x$, else $mx = y$. At the last level there will be **the trivial task** max (a_1) $\rightarrow$ $mx = a_1$, there is max(mx, a_2), etc.

```
int maxr2(int i)
{
      if (i == 0) return a[0];
```

```
            else   {
               int mx = maxr2(i-1);
                 if (a[i] > mx)  return a[i];
                              else return mx;
            }
    }
```

Example 13.4. Find the sum of elements of the one-dimensional array. At recursive splitting the array should be divided into two half.

```
    int sumr(int a[], int i, int j)
    {
            if (i == j) return a[i];
            else
             return sumr(a,i,(j+i)/2) + sumr(a,(j+i)/2+1,j);
    }
```

Example 13.5. Calculation of Fibonacci's numbers are defined by the following recursive ratio: $b_0 = 0$; $b_1 = 1$; $b_n = b_{n-1} + b_{n-2}$.

```
    int fibr (int n)
    {   if (n <= 1) return n;
        else return fibr(n-1)+fibr(n-2);    }
```

The result of the function call will be the call of two more functions. As n grows, the number of accesses increases as $2n - 1$. For example, for $n = 5$, the program call tree will look like (fig. 13.2)

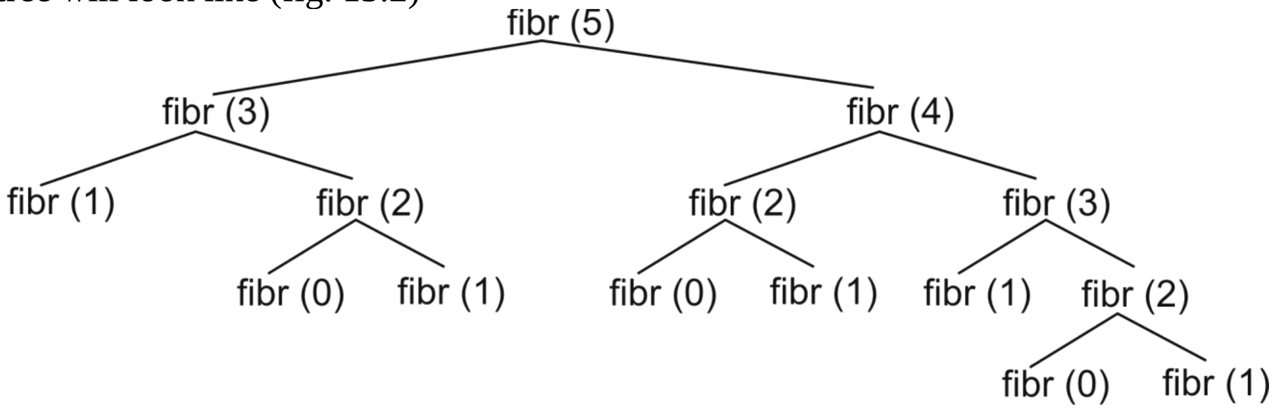


Fig. 13.2

When executing a program, stack memory is required to store data for 16 (2^4) functions. The big algorithm disadvantage is the function multiple call with the same parameters.

As Fibonacci's function grows quickly enough, for great values *of* **n** the recursive algorithm will work slowly or absolutely will cease to work because of stack overflow. Therefore such recursive program is not of practical interest.

The example reviewed above shows that the compact and beautiful program is not always effective. For calculation of numbers of Fibonacci it is convenient to use the normal iterative algorithm or function with one recursive call:

```
int fibri (int x1, int x2, int n) {
if (n == 1) return x2;
    else   if (n == 0) return x1;
                else    {
  x2 += x1;
  x1 = x2-x1;
       return fibri(x1, x2, n-1);
       }
}
```

Appeal to function: fibri (0.1, n);

Example 13.6. The Task about the Hanoi Tower. Three rods on one of which *n* of rings are strung are given. The rings differ in size and are located smaller on larger ones. It is necessary to transfer the tower to other rod. For once it is allowed to transfer only one ring, and it is impossible to put the bigger ring on smaller. For intermediate storage of disks it is possible to use the third rod.

Task solution for one disk to shift the disk from *the first* rod to *the second* rod.

Task solution for two disks:

– to shift the disk from *the first* rod to *the third* rod;
– to shift the disk from *the first* rod to *the second* rod;
– to shift the disk from *the third* rod to *the second* rod.

The tower consisting of *n* of disks is considered as the tower from two disks: the first disk is upper disk of the tower, and the second disk is all disks which are located under the upper disk. After shift of these two compound disks the problem is solved for *n* – 1 disks.

```
void hanoy(int n, int sterg1, int sterg2, int sterg3)
{
      if (n > 0) {
      hanoy(n-1,sterg1,sterg3,sterg2);
       cout << "transfer disk from " << sterg1 << " to " << sterg2 << endl;
      hanoy(n-1,sterg3,sterg2,sterg1);
                }
}
```

Appeal to function: hanoy (n, 1,2,3);

Example 13.7. Calculate $y = x^n$ on the following algorithm: $y = (x^{n/2})^2$, if *n* even; $y = x \cdot x^{n-1}$, if *n* odd.

```
double st(int n)
{
      if (n == 0) return 1;
      if (n % 2 == 0) {
                  double p = st(n/2);
                  return p * p;
            }
            else return x * st(n-1);
}
```

13.4. Reasonability of Use Recursion

Recursive algorithms are well suitable for the tasks allowing recursive splitting into elementary subtasks. However it does not mean that for the solution of such tasks use of recursive programs is indisputable. In most cases use of the recursion is absolutely inefficient.

Recursion summary.

1. Big expense of memory and resources. It is caused by the fact that by each challenge of the subprogramme the system leaves all local data in memory. Processing of the difficult chain of recursive calls requires selection of big resources of the system.

2. Often when using the recursive program some calculations are executed repeatedly that significantly reduces high-speed performance of the program. The classical example is calculation of numbers of Fibonacci.

3. Often, despite the seeming simplicity, programs are difficult for understanding and for debugging.

The recursive algorithms are used quite often despite their shortcomings. A large number of problems are solved quite difficult without the recursion using.

14. Sorting Techniques

Sorting is the process of the regrouping of array cells resulting in their ordered arrangement concerning the set key. Depending on the solvable task any field of structure can be considered as the key. The purpose of sorting is simplification of search of elements.

It is accepted to call sorting of arrays *internal* unlike sorting of files (lists) which is called *external*.

The following criteria are used for assessment of the sorting efficiency:

1. Sorting *speed*. It is defined by a number of comparisons and exchanges. Also sorting speed *in the best and worst cases* is evaluated. Since there are algorithms that have a good average speed and are slow in the worst case.

2. *Naturalness* of sorting. A sorting is named *natural* if the sorting time is minimal for an already ordered array and enlarge as the disorder level of the array increases.

3. *Stability* of sorting. The algorithm of sorting is steady if in the sorted array elements with identical keys are located in the same order in which they were located in the initial array. The algorithms which are not rearranging elements with identical keys are considered as the best.

The complexity of an algorithm defines the performance of the algorithm in terms of the input data size.

There are three main methods of sorting:

1. **Exchange**. At such method the elements located not one after another interchange the position. Exchange continues until all elements are not arranged.

2. **Choice**. In the beginning the smallest element is looked for and put on the first place, then the element following on the importance is looked for and is established on the second place, etc. As a result all elements are located in the necessary positions.

3. **Insert**. Consistently all elements get over. Each element moves to that position where it shall stand.

14.1. Simple Sorting Methods

14.1.1. Bubble Sort method

It is the most known, the most popular and one of the worst algorithms of sorting. This sorting belongs to the class of exchange sortings. Its algorithm contains repeated comparisons of the next elements and in need of their exchange. Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration.

```
void s_bub(tmas a[], int n)
{
   tmas t;
   for (int i = 1; i < n; i++)
      for (int j = n - 1; j >= i; j - - )
         if (a[j - 1].key > a[j].key)
         {
                t = a[j - 1];
                a[j - 1] = a[j];
                a[j] = t;
             }
```

Features of bubble sort:

– the complexity of sorting is $O(N^2)$;

– as the number of sorted array elements increases, the number of exchanges decreases, but the number of comparisons always remains the same;

– sorting is steady;

– it is simple in understanding and implementation.

14.1.2. Sorting by Choice

In the array the element with the smallest value is selected and interchanged the position with the first element. Then from the remained elements there is the smallest and it interchanges the position with the second element, etc.

```
void s_vb(tmas a[], int n)
{
   int imin, i, j;
    tmas t;
 for(i = 0; i<n-1; i++)
 {
     imin = i;
  for(j= i+1; j < n; j++)
  if (a[imin].key > a[j].key) imin = j;
    if (imin != i)
    {
            t = a[imin];
            a[imin] = a[i];
            a[i] = t;
    }
    }
}
```

Features of sorting by the choice:

– complexity of sorting is $O(N^2)$;

– with an increase in the degree of sorted array, the number of exchanges decreases and the number of comparisons always remains the same;

– sorting is unstable;

– number of exchanges are much less than in bubble sort.

14.1.3. Sort by Insertion

Firstly the two first array cells are sorted. Then the algorithm inserts the third element into the necessary position in relation to the first two elements. After that the fourth element is located in the corresponding position of the list from three elements. Process repeats until all elements are inserted.

```
void s_vst(tmas a[], int n)
{
 int i, j;
 tmas t;
 for(i =1 ; i < n; i++)
 {
        t = a[i];
  for(j = i-1;  j >= 0 && t.key < a[j].key;  j--) a[j+1] = a[j];
        a[j+1] = t;
 }  }
```

Features of sorting by the insert:

– complexity of sorting at best $O(N)$, and in the worst is $O(N^2)$;

– with an increase in the degree of sorted array, the number of exchanges and the number of comparisons decreases;

– sorting is stable.

14.2. Improved Sorting Methods

All algorithms considered above have one fatal shortcoming.They work very slowly. The applied methods of code optimization do not give significant gain of performance of the algorithm. There is the rule: if the algorithm used in the program is too slow in itself, any volume of manual optimization does not make the program rather fast. The solution consists in application of the best algorithm of sorting.

14.2.1. Shell's Method

The general idea is borrowed from sorting by the insert. At first the elements located at distance of three positions from each other are sorted. Then the elements located at distance of two positions are sorted. At last, all next elements are sorted. The sequence of steps can be also another, however the last step shall be plains 1. Complexity of the algorithm is about $O(N^{\log 2N})$.

```
void s_shell(tmas a[], int n)
{
   int i, j;
   tmas t;
for(int d = 3; d > 0; d--)
   for(i = d; i < n; i++)
```

```
 {
    t = a[i];
  for(j = i-d;  j >= 0 && t.key < a[j].key;  j -= d)  a[j+d] = a[j];
       a[j+d] = t;
 }
}
```

Features of sorting by the insert:

– the complexity of sorting at best $O(N)$, on average is $O(N^{7/6})$ and in the worst is $O(N^{4/3})$ (depends on the selected sequence of steps);

– shifts number of elements is significantly reduced in comparison with simple methods of sorting.

– average rate of sorting is much higher, than at sorting by the insert;

– sorting is natural;

– sorting is unstable.

14.2.2. Merge Sort

The merge sort algorithm is as follows:

1. The sorted array recursively breaks into adjacent sections of approximately identical size until in each section about one element does not have.

2. Adjacent ordered sections of the array connect in one ordered section for the smallest elements are consistently retrieved and are located in the resulting array. When in one of adjacent sections elements come to an end, all remained elements from other section of the array move to the resulting array. The result array registers in the place of the considered adjacent sections.

3. The algorithm stops working at the moment when all adjacent sections are connected.

Merge function:

```
void slip(int left, int m, int right)
{
   int i = left, j = m + 1, k = 0;
   while ((i <= m) && (j <= right))
     if (a[i].key < a[j].key) { c[k++] = a[i++];}
                          else { c[k++] = a[j++];}
   while (i <= m)    c[k++] = a[i++];
   while (j <= right) c[k++] = a[j++];
   // Writing a sorted section to an array
   k = 0; i = left;
   while (i <= right) a[i++] = c[k++];
}
```

Sorting function:

```
void s_sl(int left, int right)
{
    if (left < right)
    {
        int m = (left + right) / 2;
        s_sl(left, m);
        s_sl(m + 1, right);
        slip(left, m, right);
    }

}
```

Challenge:

```
s_sl(0, n-1);
```

Features of merge sort:
– complexity is $O(N^{\log N})$;
– average rate of sorting is much higher, than at sorting by the insert;
– sorting is not natural (the speed does not depend on the source data order);
– sorting is steady;
– the algorithm requires an additional array, so it is usually used for external sorting.

14.2.3. Quick Sort Method (*Hoare's Quick Sort*)

Bubble sort is the cornerstone of this sorting. At first the pivot element is selected (average or a random value). Then elements greater than or equal to the main ones are moved to one subarray and smaller ones are moved to the other subarray. After that similar actions repeat separately for each subarray. Process repeats until the array is not sorted. The algorithm can be implemented in the form of recursive function.

```
void s_qsr(int left, int right)
{
   int i = left, j = right;
   tmas t, x;
x = a[(i+j)/2];
do {
      while (a[i].key < x.key && i < right) i++;
      while (a[j].key > x.key && j > left)   j--;
  if (I <= j) {
        t = a[i];
        a[i] = a[j];
```

```
          a[j] = t;
     i++;  j--;
      }
     } while( i <= j );
     if(left < j)  s_qsr(left, j);
     if(i < right) s_qsr(i, right);
   }
```

For fast work of the algorithm QuiskSort it is necessary to select pivot element correctly. If the value of pivot element at each division is equal to the greatest element, then sorting according to speed will become equal the bubble sort algorithm. The technique of the choice of pivot element makes a start by nature sorted array. For example, if data are fairly uniform, then it is better to select the average element. In other cases it is possible to use the random choice of pivot element.

The recursive implementation of sorting described above has the beautiful and clear algorithm. But knowledge of features of the recursive programs allows to assume that nonrecursive implementation will be more effective:

```
struct St {
 int l;
 int r;
} stack[10];
void push(int l, int r, int &s) {
 stack[s].l = l;
 stack[s].r = r;
 s++;
}
void pop(int &l, int &r, int& s) {
 s--;
 l = stack[s].l;
 r = stack[s].r;
}
void s_qs(tmas a[], int n) {
 int i, j, left, right, s = 0;
 tmas x;
 push(0, n-1, s);
 while (s != -1) {
          pop(left, right, s);
        while (left < right) {
                i = left; j = right; x = a[(left + right) / 2];
                while (i <= j) {
                        while (a[i].key < x.key) i++;
                        while (a[j].key > x.key) j--;
                        if (i <= j) { swap(a[i], a[j]); i++;  j--; }
```

```
                  }
        if ((j - left) < (right - i)) {   // The choice of shorter part
                           if (i < right) push(i, right, s);
                    right = j;
                  }
                  else {
                           if (left < j) push(left, j, s);
                    left = i;
                  }
        }
 }
}
```

The pivot element is selected to be the average element in this function. An array is divided into subarrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

Features of quick sort:

– complexity of sorting is $O(N^{\log N})$, at worst (is improbable) is $O(N^2)$;

– sorting is natural;

– sorting is unstable;

– algorithm execution does not require the use of an additional array;

– the number of comparisons is significantly less than any previously considered method.

15. Search Algorithms

The aim of search consists in finding of the element having the preset value of the key field.

15.1. Linear Search

Linear search is used in case the array is not sorted by the set key. Search represents consecutive search of array cells before detection of the required key or until the end of the array if the key is not found:

```
int p_lin1(tmas a[], int  n, int x)
{
for(int i = 0 ; i < n; i++)
        if (a[i].key == x) return i;
   return -1;
}
```

In this algorithm, two checks are performed at each step: checking for the equality of the key field and the desired key and checking the condition for continuing the cyclic algorithm. An auxiliary element (*barrier*) has been introduced that excludes checking the continuation condition of the cyclic algorithm to protect against array overflow:

```
int p_lin2(tmas a[], int n, int x)
{
 a[n].key = x;
   int i = 0;
 while (a[i].key != x) i++;
  return i;
}
```

If the function returns a value equal to n, then this indicates that the required element was not found. This algorithm is almost twice as efficient as the previous one.

15.2. Binary Search

Binary search is used when data are arranged, for example, on nondecrease of the key field. The algorithm consists in the consecutive exception of that part of the array in which the required element cannot be. For this purpose the average element undertakes and if the value of the key field of this element is more than value of the required key, then it is possible to exclude the right half of the array from consideration, differently the left half of the array is excluded. The process continues until no element remains in the considered part of the array.

```
int p_dv(tmas a[], int n, int x)
{
 int i = 0, j = n - 1, m;
while(i < j)  {
m = (i + j)/2;
if (x > a[m].key) i = m + 1; else j = m;
  }
if (a[i].key == x) return i;
return -1;
}
```

15.3. Interpolation Search

For arrays with uniform distribution of elements it is possible to use the formula allowing to define approximate location of the element:

$$m = i + \frac{(i-j)(x-a[i].key)}{a[i].key - a[j].key},$$

where, *i*, *j* is beginning and end of the interval; *x* required value of the key field.

```
int p_dv(tmas a[], int n, int x)
{
 int i = 0, j = n-1, m;
while(i < j)
{
 if (a[i].key == a[j].key) // Prevention of division into zero
    if (a[i].key == x) return i;
                   else return -1;
      m = i + (j-i) * (x-a[i]) / (a[j]-a[i]);
if (a[m].key == x) return m;
  else
   if (x > a[m].key) i = m+1; else j = m-1;
}
 return -1;
}
```

This search is 3-4 times faster than binary search. However, it can behave unstable near the key field. Therefore, it is common to do the first few steps using interpolation search, and then use binary search.

16. Dynamic Data Structures

16.1. List, Stack and Queue Concept

A data object is a dynamic structure if its size, relative position and relationships of its elements change during the program execution.

The list is the sequence of the same data, work with which is conducted in random access memory. In the course of work the list can change the size. The two most common forms of working with a list are the queue and the stack.

Stack is the list with one point of entry. Data are added to the list and the sequences (stack top) are removed from it only on the one hand. Thus, the “last in, first out” rule is implemented.

Queue is the list with one or two points of entry. Data are added to the end of queue, and retrieved from the beginning of queue. Thus, the principle "first in first out".

A special *recursive data type* is provided to implement the list. Its description contains a pointer to a structure similar to this type

The most commonly used recursive data type construct is:

```
struct TInf
 {
// Set of fields of structure
};

struct TNode
 {
   TInf inf;          // Information part of structure
   TNode *a;          // Address part of structure
 };
```

The following structure is used to simplify:

```
struct TNode
 {
   int inf;           //  Information part of structure
   TNode *a;          //  Address part of structure
 } ;
```

The unidirectional linked lists will be organized as follows: memory for each element is selected separately (as required). The information part contains the necessary data, and the address part contains the address of the previous or next structure.

16.2. Stack Implementation

For execution with the stack it is enough to know the pointer on stack top. For movement on the stack it is necessary to pass from one cell to another consistently:

```
spt = top; // Installation of the current pointer in the beginning of the stack
spt = spt->a; // Movement to the following element
spt = spt->a->a; // Movement of two elements
```

A method of *indirect addressing* in which the contents of the address specified in the instruction (which may itself be an effective address) are themselves an address to be used to provide the desired memory reference.. Unlike addressing by the index the indirect addressing mode is less evident, however has bigger flexibility.

Example 16.1. Execution with the stack.

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

struct TNode       // Description of the element of the stack
{
  int inf;         // Information part of structure
  TNode *a;        // Address part of structure
};

struct stack       // Structure for work with the stack
{
  TNode *top = nullptr;   // The pointer on stack top
  int size = 0;    // Quantity of elements of the stack
    // Verification of presence of elements in the stack
    bool empty() {
        if (top) return false;
            else return true;
    }
    // Adding of the element in the stack
    void push(int inf) {
        TNode *spt = new TNode;
        spt->inf = inf;
        spt->a = top;
        top = spt;
        size ++;
    }

    // Removal of the element from the stack
    void pop() {
         TNode *spt = top;
         top = top->a;
```

```
     delete spt;
     size--;
}
 // Stack output to the screen
 void print() {
   TNode *spt = top;
   while (spt != nullptr)
   {
       cout << spt->inf << " ";
       spt = spt->a;
   }
   }
 // Search of the element with the set key
  TNode* search(int x) {
   if (!top) return nullptr;
      TNode *spt = top;
   while (spt->inf != x && spt->a != nullptr) spt = spt->a;
    if (spt->inf == x) return spt;
    else return nullptr;
  }
 // Search of the previous element (excepting the first)
  TNode* searchp(int x) {
    if (!top || !top-> a) return nullptr;
      TNode *spt = top;
      while (spt->a->inf != x && spt->a->a != nullptr) spt = spt->a;
      if (spt->a->inf == x) return spt;
      return nullptr;
  }
 // Removal of the element, with the set key
    void del(int x) {
      if (!top) return;
      if (top-> nf == x) pop();
      TNode *spt, *spp;
       spp = searchp(x);
       spt = spp->a;
      spp->a = spp->a->a;
    delete spt;
    }
```

```
        // Exchange of the following for specified elements
        void exchange(TNode *sp) {
            TNode *spt;
            spt = sp->a->a;
            sp->a->a = spt->a;
            spt->a = sp->a;
            sp->a = spt;
        }
} ;

int main () {
  stack s;
   s.push(4);  s.push(2);  s.push(1);  s.push(6);  s.push(9);
   s.print ();                                          // Displays: 9 6 1 2 4
   TNode *d1 = s.search(1);  cout <<d1> inf;            // Displays: 1
   TNode *d2 = s.searchp(1); cout <<d2> inf;            // Displays: 2
    s.exchange (d2);
    s.print ();                                         // Displays: 9 6 2 1 4
    s.del(6);
    s.print ();                                         // Displays: 9 2 1 4
 while (!s.empty()) s.pop ();
 if (s.empty) cout << "Stack is empty";
    return 0;
}
```

16.3. Unidirectional Queue Implementation

Working with a unidirectional queue is similar to working with a stack. But the data is placed at the end of the list, and retrieved from the beginning.

Example 16.2. The Unidirectional Queue Implementation.

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

struct TNode {
  int inf;
  TNode* a;
};

struct queue {
    TNode* front = nullptr;
    TNode* back = nullptr;

    bool empty() {
        if (front) return false;
        else return true;
    }

    void push(int inf) {
        TNode* spt = new TNode;
        spt->inf = inf;
        spt->a = nullptr;
         if (!front) front = back = spt;
        else {
            back->a = spt;
            back = spt;
        }
    }

    void pop() {
        TNode* spt = front;
        front = front->a;
        delete spt;
        if (!front) back = nullptr;
    }

    void print() {
        TNode* spt = front;
        while (spt != nullptr) {
            cout << spt->inf << " ";
            spt = spt->a;
        }
    }
};
int main() {
    queue s;
   s.push(4);
   s.push(2);
   s.push(1);
   s.push(6);
   s.push(9);
   s.print(); // Displays: 4 2 1 6 9
 while (!s.empty()) s.pop();
 if (s.empty()) cout << "Queue is empty";
    return 0;
}
```

16.4. Doubly Linked Lists Implementation

A doubly linked list consists of structures with fields for storing the addresses of the previous and next elements. This organization allows to move through the list in any direction.

Example 16.3. Exercise with the two-linked List.

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

struct TNode {
  int inf;
  TNode* left;
  TNode* right;
};

struct list {
    TNode* front = nullptr;
    TNode* back = nullptr;

    bool empty() {
        if (front) return false;
        else return true;
    }

    void push(int inf) {
      TNode* spt = new TNode;
             spt->inf = inf;
             spt->right = nullptr;
         if (!front) {
           spt->left = nullptr;
           front = back = spt;
      return;
      }
    back->right = spt;
    spt->left = back;
    back = spt;
 }

void pop() {
    TNode* spt = front;
    front = front->right;
    delete spt;
    if (!front) back = nullptr;
    else
    front->left = nullptr;
}

void print() {
    TNode* spt = front;
    while (spt != nullptr) {
        cout << spt->inf << " ";
        spt = spt->right;
    }
}

TNode* search(int x) {
    if (!front) return nullptr;
    TNode* spt = front;
    while (spt->inf != x && spt->right != nullptr) spt = spt->right;
    if (spt->inf == x) return spt;
    else return nullptr;
}

void del(int x) {
    TNode* spt = search(x);
    if (!spt) return;
    if (front == back)
    {
      front = nullptr;
```

```
          back = nullptr;
        }
        else
          if (!spt->left) {
            front = spt->right;
            front->left = nullptr;
        }
          else
            if (!spt->right) {
              back = spt->left;
              back->right = nullptr;
            }
            else {
              spt->right->left = spt->left;
              spt->left->right = spt->right;
            }
        delete spt;
    }

    void pushleft(TNode* spp, int inf) {
        TNode* spt = new TNode;
        spt->inf = inf;
        spt->right = spp->right;
        spt->left = spp;
        spp->right = spt;
        if (spt->right) spt->right->left = spt;
    }
};

void main() {
    list s;
   s.push(4);
   s.push(2);
   s.push(1);
   s.push(6);
   s.push(9);
   s.print();                          // Displays: 4 2 1 6 9
   s.del(6);
```

```
    s.print();                          // Displays: 4 2 1 9
    s.pushleft(s.search(2), 7);
    s.print();                          // Displays: 4 2 7 1 9
    while (!s.empty()) s.pop();
}
```

16.5. Doubly Linked Circular Lists Exercise

Circular lists are one or bidirectional queues in which the last element indicates the beginning of queue (fig. 16.1). Concepts of the beginning and the end of queue do not make sense here, it is enough to know the address of any element of queue.

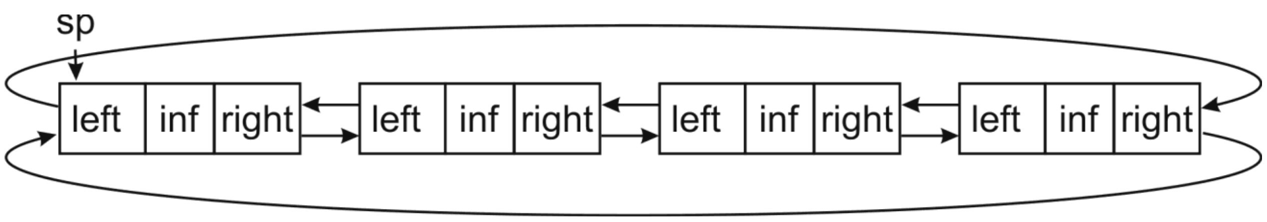


Fig. 16.1

17. Nonlinear Lists

17.1. Tree Data Structures

The tree data structure considered (fig. 17.1).

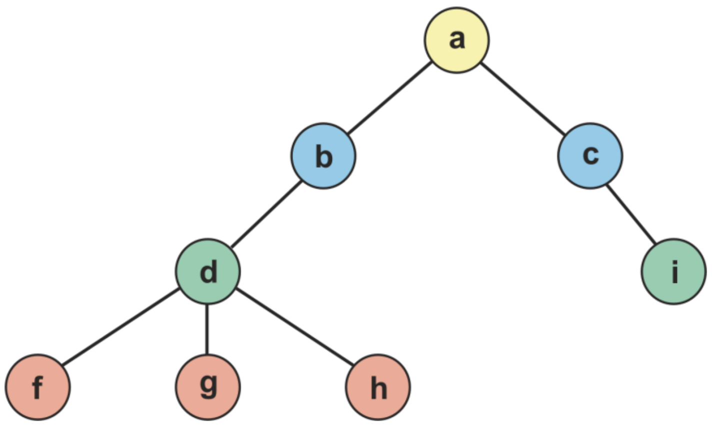


Fig. 17.1

All data are called ***nodes***.

The links between nodes are called ***branches***.

The topmost node is called ***root*** of the tree (a).

Nodes from which connections do not go out are ***leaves*** of the tree (*f, g, h, i*).

The node which is directly over another is called ***the parent node*** (*for node d node b is parent*). The node which is directly below is called ***child*** (*for node b node d is child*).

All nodes which are above considered are his ***ancestors*** (*for node d ancestors of b and a*), and all nodes which are below are ***descendants*** (*for node b descendants are d, f, g, h*).

The nodes having the same parent are called ***sisterly*** (*f, g, h*).

The node which is not the leaf is called ***internal*** (*b or d or with or a*).

Node order (or node degree) is quantity of child sites (*for node b the order 1, for node d the order 3*).

Degree of the tree is the maximum order of its nodes (*the considered tree has the third order*). The tree of the second degree is called ***binary***. The degree tree three is called the ***ternary*** tree.

Node depth is number of ancestors plus unit (*for example, for node d depth is equal to 3*).

Tree depth is the largest depth of all nodes (*for this tree – 4*).

17.2. Tree Structures Implementation

For the tree structures implementation the following construction of recursive type is used:

```
struct ttree  {
  tinf inf;
  ttree *a1;
  ttree *a2;
  …
    ttree *an;
   } *proot, *p;
```

Placement in memory of the fig. 17.1 structure is implemented in the following way.

```
ttree *proot, *p;
proot = new ttree;  proot->inf = 'a';  proot->a2 = nullptr;
p = proot;
p->a1 = new ttree;
        p = p->a1;  p->inf = 'b';  p->a2 = nullptr;  p->a3 = nullptr;
p->a1 = new ttree;   p = p->a1;   p->inf = 'd';
p->a1 = new ttree;
        p->a1->inf = 'f';  p->a1->a1 = nullptr;  p->a1->a2 = nullptr;
        p->a1->a3 = nullptr;
p->a2 = new ttree;
         p->a2->inf = 'g';  p->a2->a1 = nullptr;  p->a2->a2 = nullptr;
        p->a2->a3 = nullptr;
p->a3 = new ttree;
         p->a3->inf = 'h';  p->a3->a1 = nullptr;  p->a3->a2 = nullptr;
        p->a3->a3 = nullptr;
         p = proot;
p->a3 = new ttree;
         p = p->a3;  p->inf = 'c';   p->a1 = nullptr;   p->a2 = nullptr;
p->a3 = new ttree;
         p = p->a3;   p->inf = 'i';  p->a1 = nullptr;   p->a2 = nullptr;
        p->a3 = nullptr;
```

Apparently from the stated above program fragment, direct filling even of the small tree demands quite bulky sequence of commands. Therefore for work with trees use the set of specific algorithms.

Bypass of the tree is called the consecutive appeal to all its nodes. The following recursive procedure performs such bypass with printout of each node:

```
void obh(ttree *p) {    //  Bypass of all tree
if (p == nullptr) return;
 // The output at the direct bypass
obh (p-> a1);
  obh (p-> a2);
…
  obh tree(p-> an);
// The output at the return bypass
}
```

Direct bypass: ***a b d f g h c i***.
Return bypass: ***f g h d b i c a***.

17.3. Binary Search Tree

If key fields in the tree are located in such a way that for any node of value of the key the left successor has less, than at right, then such tree is called *the binary tree of search* (*Binary Search Tree*). Let's assume that there is the data set, arranged on the key: 1, 5, 6, 9, 14, 21, 28, 32, 41. For such data the binary tree of search looks as follows (fig. 17.2).

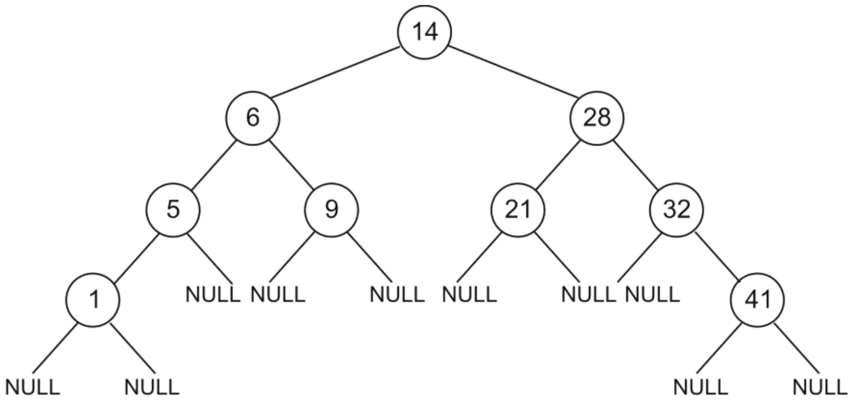


Fig. 17.2

The efficiency of information search in such dynamic data structure is comparable with efficiency of binary search in the array.

Tree which has nodes having only one daughter not higher than two last levels are located, is called **the balanced tree**.

For work with the binary tree of search the construction of recursive type similar to the description of the two-linked list is used.

During removal of the node of the tree, three options of placement of the deleted node are possible:

1. If the node which does not have descendants (fig. 17.3) is removed.

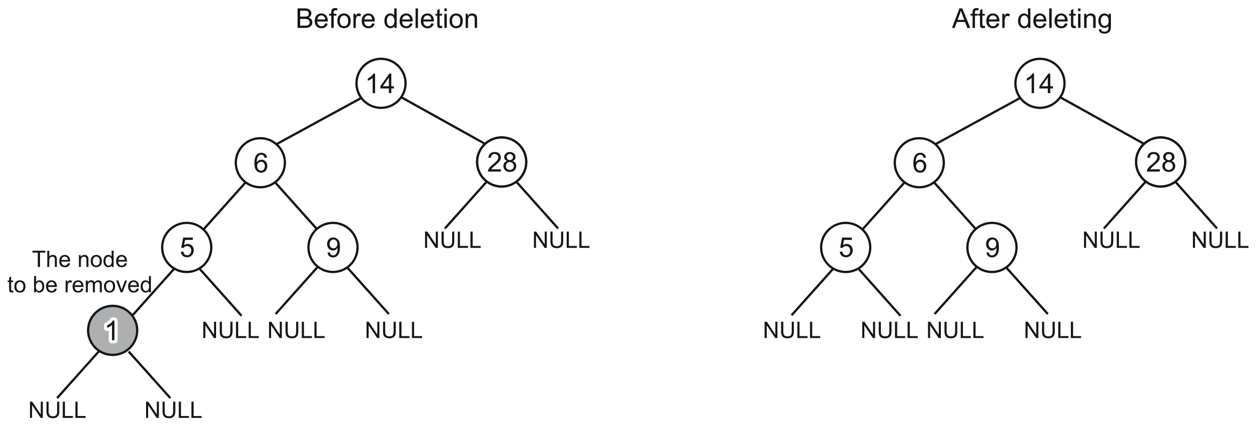


Fig. 17.3

2. If the node having one daughter (fig. 17.4) is removed.

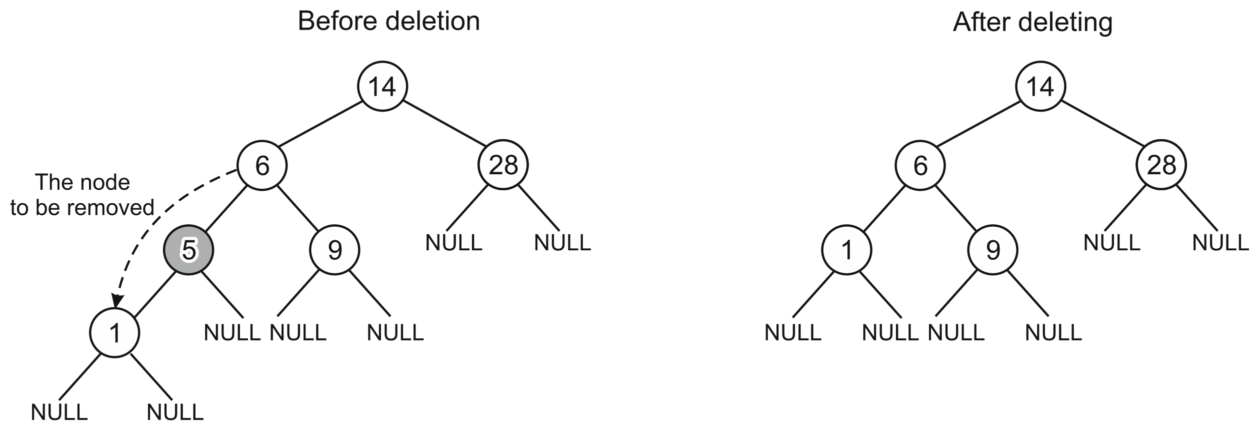


Fig. 17.4

3. If the node having two daughters is removed, then the deleted node is replaced with the node having the greatest key in the left subtree or the smallest key in the right subtree (fig. 17.5).

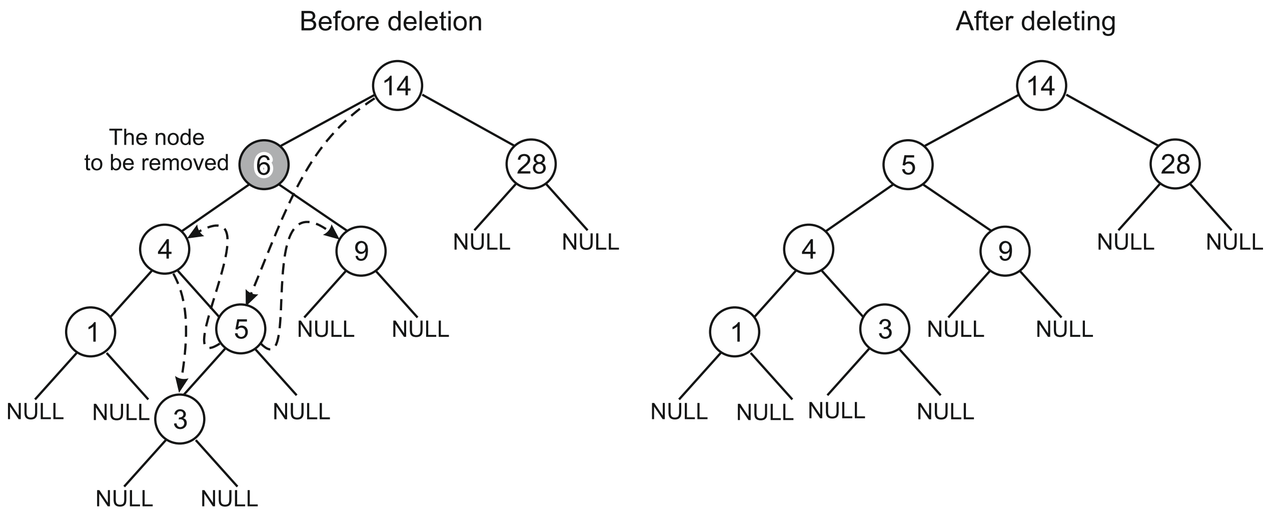


Fig. 17.5

Example 17.1. Exercise with the Binary Search Tree.

```
struct TNode {
  int inf;
  TNode* left;
  TNode* right;
};

struct TTree {
   TNode* root = nullptr;

   bool empty() {
    if (root) return false;
    else return true;
}

void push(int inf) {
    TNode* nu = new TNode;
    nu->inf = inf;
    nu->left = nullptr;
    nu->right = nullptr;
    if (!root) { root = nu; return; }
    bool b;
    TNode* pdel = root;  TNode* del = root;
    while (del) {
        pdel = del;
        b = inf < del->inf;
        if (b) del = del->left;
        else del = del->right;
    }
    if (b) pdel->left = nu;
    else pdel->right = nu;
  }

void print(TNode* p)
{
  if (!p) return;
  print(p->left);
    cout << p->inf << " ";
  print(p->right);
}

void pop(int x) {
        TNode* del = root, * pdel = root, * rep, * prep;
        while (del && del->inf != x) {
            pdel = del;
            if (x < del->inf )  del = del->left;
                           else del = del->right;
        }
        if (!del) return;
```

```
          prep = del;
        if (!del->left) rep = del->right;
        else {
                rep = del->left;
                while (rep->right != nullptr) {
                    prep = rep;
                    rep = rep->right;
                }
            }
        if (rep){
            if (prep->left == rep) prep->left = rep->left;
                    else   prep->right = rep->left;
                rep->right = del->right;
                rep->left = del->left;
                  }
      if (del == root) root = rep;
            else
                if (pdel->left == del) pdel->left = rep;
                else  pdel->right = rep;
            delete del;
      }

int search_max()
{
    TNode* p = root;
    while (p->right != nullptr) p = p->right;
    return p->inf;
}

int search_min()
{
    TNode* p = root;
        while (p->left != nullptr) p = p->left;
    return p->inf;
}

TNode* search(int key)
{
        TNode* p = root;
        while (p)
        {
            if (p->inf == key) return p;
            if (key < p->inf) p = p->left;
                         else p = p->right;
        }

    }

    TNode* pop_all(TNode* p)
    {
        if (!p) return nullptr;
        pop_all(p->left);
        pop_all(p->right);
        delete(p);
    }
};

void main() {
    TTree s;
    s.push(4); s.push(1); s.push(2);
    s.push(9); s.push(6);
    s.print(s.root); // Displays: 1 2 4 6 9
    cout << endl;
    s.pop(4);
    s.print(s.root); // Displays: 1 2 6 9
    cout << "min = " << s.search_min() << endl; // Displays:  min = 1
    cout << "max = " << s.search_max() << endl; // Displays: max = 9
    TNode* m = s.search(6);
    if (m) cout << m->inf << endl; // Displays:  6
    s.root = s.pop_all(s.root);
    if (s.empty()) cout << "Tree removed"; // Displays:  Tree removed
}
```

18. Parsing of Arithmetic Expressions (Syntactic Analysis)

The main reason for the development of the high-level programming languages is the computational tasks with a large amount of routine calculations. The maximum approximation of the arithmetic expressions form to the mathematics natural language is the main requirement for these languages. Expressions in mathematics are usually written in infix form, such as $(a + b) * (k - d)$. The main inconvenience for computer processing of such expressions is that the presence of parentheses changes the standard order of operations. This is the main inconvenience for computer processing of such expressions. Therefore, the study of parsing arithmetic expressions is the main task of system programming.Among the received results the most successful is to use *postfix* (the sign of operation is put after operands) the form of representation of arithmetic expressions offered by the Polish mathematician Jan Lukasiewicz. Such form of record of arithmetic expressions received the name of the reverse Polish notation (RPN). The advantage of the RPN is that parentheses are not needed to write expressions. The resulting sequence of operands and operations is convenient for decoding.

18.1. Conversion Expression Algorithm to the RPN Form

Edsger Dijkstra invented the algorithm for *conversion of expressions from the infix form in the* RPN *form*. Because of similarity of the sequence of operations with the events in railway switchyard the algorithm received the name "switchyard".

The essence of the algorithm consists in the following. The string is consistently browsed from left to right. Operands are added to the output string at once. Other characters are processed as follows:

1. If the current character is operation, and the stack is empty, then operation registers in the stack.

2. If the current character is the open parenthesis, then it registers in the stack.

3. If the current character is the closing parenthesis, then elements from the stack are retrieved in the output string until the open parenthesis does not become the upper element of the stack. The open parenthesis is removed from the stack, but is not added to the output string.

4. If the current character is operation, and the stack is not empty, then from the stack in the output string all operations with the big or equal priority are undergone. The current operation is moved onto the stack after this

5. After viewing all the characters in the string, the operations are moved from the stack to the output string.

The algorithm *of the expression evaluation written in the form of RPN* is based on use of the stack. When viewing expression from left to right values of operands are brought in the stack. If operation is found, then from the stack two operands to which the found operation is applied are retrieved. The result is brought in the stack. After execution of all operations in the stack there is one value (result of calculation of arithmetic expression).

Example 18.1. Calculations of Arithmetic Expressions.

```
struct tstk {
    double inf;
    tstk* a;
};

tstk* push(tstk* sp, double inf)
{
    tstk* spt = new tstk;
    spt->inf = inf;
    spt->a = sp;
    return spt;
}

tstk* pop(tstk* sp, double& inf)
{
    tstk* spt = sp;
    inf = sp->inf;
    sp = sp->a;
    delete spt;
    return sp;
}

double masz[122];
char str[100], strp[100];

int priority(char ch)  //  Calculation of the priority of operations
{
    switch (ch)
    {
    case '(': case ')': return 0;
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    default: return -1;
    }
}
```

```
void AddPostFix(char* strin, char* strout)
{
    tstk* sp = nullptr;
    int n = 0;
    char ch;
    double inf;
    for (int i = 0; i < strlen(strin); i++)
    {
        ch = strin[i];
        // If it is the operand
        if (ch >= 'A') { strout[n++] = ch; continue; }
        // If the stack is empty or the open parenthesis is found
        if (!sp || ch == '(') { sp = push(sp, ch); continue; }
        // If the open parenthesis is found
        if (ch == ')') {
            while (sp->inf != '(') {
                sp = pop(sp, inf);
                strout[n++] = (char)inf;
            }
            sp = pop(sp, inf);  // Removal of the open parenthesis
            continue;
        }
        // If operation
        int pr = priority(ch);
        while (sp && priority((char)sp->inf) >= pr)
        {
            sp = pop(sp, inf);
            strout[n++] = (char)inf;
        }
        sp = push(sp, ch);
    }  // end for
    while (sp != nullptr)
    {
        sp = pop(sp, inf);
        strout[n++] = (char)inf;
    }
    strout[n++] = '\0';
}
```

```
double rasAV(char* str, double* mz)
{
    tstk* sp = nullptr;
    char ch;
    double inf, inf1, inf2;
    for (unsigned int i = 0; i < strlen(str); i++)
    {
        ch = str[i];
        // If the operand is found
        if (ch >= 'A') { sp = push(sp, mz[int(ch)]); continue; }
        // If the sign of operation is found
                sp = pop(sp, inf2);
        sp = pop(sp, inf1);
        switch (ch)
        {
        case '+': sp = push(sp, inf1 + inf2); break;
        case '-': sp = push(sp, inf1 - inf2); break;
        case '*': sp = push(sp, inf1 * inf2); break;
        case '/': sp = push(sp, inf1 / inf2); break;
        }
    }
    sp = pop(sp, inf);
    return inf;
}
int main()  {
    cout << "Vvedite a" << endl;  cin >> masz[int('a')];
    cout << "Vvedite b" << endl;  cin >> masz[int('b')];
    cout << "Vvedite c" << endl;  cin >> masz[int('c')];
    cout << "Vvedite d" << endl;  cin >> masz[int('d')];
    cout << "Vvedite f" << endl;  cin >> masz[int('f')];
    cout << "Vvedite viragenie (a ,b, c, d, f)" << endl;
    cin >> str;
    AddPostFix(str, strp);
    cout << endl << strp;
    double s = rasAV(strp, masz);
    cout << endl << "Res =" << s << endl;
    return 0;
}
```

19. Hashing

19.1. Hashing Concept

The *hashing* algorithm was invented to solve the fast lookup problem. The data keys are stored in a special hash table. Then by means of certain simple function *i* = *h*(*key*) the algorithm of hashing determines provision of the required element in the table by value of its key.

The example. There is the array of structures from 7 elements which values of keys are in range of 0…15.

```
mas[0].key = 5;
mas[1].key = 15;
mas[2].key = 1;
mas[3].key = 10;
mas[4].key = 8;
mas[5].key = 3;
mas[6].key = 11;
```

It is necessary to find the element with the key 3. For this purpose the method of linear search will take 6 steps, and use of binary search will require preliminary sorting. The quantity of steps depends on the sorting method, but costs in this case will be higher, than by linear search.

For acceleration of search we will create the new array (hash table) in which the item number will be equal to value of the key:

```
H [Mas[i].key] = Mas[i];
```

All not used array cells of **H** matter –1:

```
H[0].key =-1;
H[1] = mas [2];          // H[1].key = 1
H[2].key =-1;
H[3] = mas [5];          // H[3].key = 3
H[4].key =-1;
H[5] = mas [0];          // H[5].key = 5
H[6].key =-1;
H[7].key =-1;
H[8] = mas [4];          // H[8].key = 8
H[9].key =-1;
H[10] = mas [3];         // H[10].key = 10
H[11] = mas [6];         // H[11].key = 11
H[12].key =-1;
H[13].key =-1;
H[14].key =-1;
```

```
H[15] = mas[1];            // H[15].key = 15
```

With such an organization, to find any element, it is enough to make only one step x = H [key] (the complexity of the algorithm is *O*(1)). For removal of the element it is enough to put value 1 in the corresponding field.

Such approach is unacceptable for the solution of real tasks since the size of the array shall be sufficient for placement of the element with the maximum key. This significantly increases the size of the hash table. For example, the array from 9,999,999 elements is necessary for storage of telephone base with seven-digit numbers. Various hashing schemes are used to reduce the size of the hash table.

19.2. Hashing Schemes

The various key compression algorithms are used to reduce the number of elements in a hash table. At compression of the table several different elements can get the identical number in the hash table therefore the scheme of hashing shall contain ***the conflict resolution algorithm*** defining behavior of the program if the new key gets on already taken position.

The scheme of work ***of the placement algorithm*** of elements in the hash table:

1. The key value is used to calculate the position number in the hash table *i* = *key* % *m* (*m* is the number of elements in the hash table).

2. If the received position is already taken, then the algorithm of the conflict resolution finds the new position.

3. If the new position is taken too, item 2 repeats until the free position is found.

The search algorithm on value of the key finds the position of the required element in the hash table. If the value of the key of the element does not match the required key, then further search, according to the selected conflict resolution algorithm is performed.

19.3. Hash Table with Linear Addressing

Conflict resolution algorithm: if position **i** found for the element is already taken, then the first unoccupied position is looked for (since *i* + 1).

For example, there is the following array:

```
Mas[0].key = 5;
Mas[1].key = 15;
Mas[2].key = 3;
Mas[3].key = 10;
Mas[4].key = 125;
Mas[5].key = 333;
Mas[6].key = 11;
Mas[7].key = 437;
```

Data are placed in the hash table from 10 elements. Placement function: *i* = *key of* % 10.

Received the hash table:

```
H[0] = Mas [3];     // H[0].key = 10;
H[1] = Mas [6];     // H[1].key = 11;
H[2].key = -1;
H[3] = Mas[2]];     // H[3].key = 3;
H[4] = Mas[5]];     // H[4].key = 333;
H[5] = Mas[0]];     // H[5].key = 5;
H[6] = Mas[1]];     // H[6].key = 15;
H[7] = Mas[4]];     // H[7].key = 125;
H[8] = Mas[7]];     // H[8].key = 437;
H[9].key = -1;
```

Example 19.1. The hash table of the placement algorithm with linear addressing:

```
void sv_add(int key, int m, int* H)
{
    int i = abs(key % m);
      while (H[i] != -1) {
            i++;
            if (i == m) i = 0;
        }
    H[i] = key;
}
int sv_seach(int key, int m, int *H)
{
    int i = abs(key % m);
    while (H[i] != -1)
    {
        if (H[i] == key) return i;
           i++;
           if (i == m) i = 0;
    }
    return -1;
}
void main()
{
    const int n = 8;  // The number of elements in the array
    int mas[n] = {5, 15, 3, 10, 125, 333, 11, 437};
    const int m = 10;  // The number of elements in the hash table
```

```
        int H[m];
        int i;
        for (i = 0; i <m; i++) H[i] =-1;  // All positions are not taken
        for (i = 0; i <n; i++) sv_add (mas[i], m, H);
        //  Search of the element with the key 333
        int key = 333;
        int k = sv_seach(key, m, H);
          if (k == -1) cout << "Item not found" << endl;
            else cout << H[k] << endl;
    }
```

Advantage: simple algorithm of placement and search of elements.

Shortcomings:

1. Fixed size hash table.

2. The difficult algorithm of removal of the element since removal of the element often results in need of reorganization of all the table. For overcoming this shortcoming it is possible to use several statuses of the cell: "is busy", "it is not busy", "is deleted". If the algorithm gets to search time on the cell with the status "is deleted", then search continues further. When adding data, the cell with the status "is deleted" is considered free.

3. If data in the table are located unevenly, then the speed of search can be very bad. For overcoming this shortcoming it is possible to use the following hash function: *i* = (*key* + *r*) % 10. The prime number *r* is generated by the random number generator. For the correct work of search algorithms and placement the sensor shall be set to identical initial position always.

19.4. Hash Table with Square and any Addressing

Unlike the linear addressing method, the quadratic addressing method does not search for a free cell sequentially (*i*++), but according to the formula $i = i + p^2$ (*p* is the attempt number).

In *the method with any addressing* the unoccupied position is looked for on the formula: *i* += *i* + *r*[*p*] (*r* – in advance generated array of random numbers; *p* – number of attempt).

In comparison with linear addressing these methods give more uniform distribution of data in the table, however work slightly more slowly.

19.5. Hash Table with Double Hashing

Method algorithm:

1. Find the element position in the hash table on formula *i* = *m* % *key*.

2. If the cell with number *i* is free, then item 6 is executed.

3. Calculate *c* = 1 + (*key of %* (*m* – 2)).

4. Find the element position in the hash table on the formula $i = i - c$ (if $i < 0$, that $i = i + m$).

5. If the cell with the found number i is occupied, then item 4 is executed.

6. Insert the element into the found position.

In comparison with the previous ones, this method (due to independent search chains for a free cell) gives a more uniform distribution of data in the hash table. The complication of the algorithm leads to a decrease in the speed of its work..

19.6. Hash Table on the Linked Lists Basis

One of the most effective methods of the conflict resolution consists that the elements getting on the same position are placed in linked lists (see subject 18). For example, there is the following array:

```
Mas[0].key = 5;
Mas[1].key = 15;
Mas[2].key = 3;
Mas[3].key = 10;
Mas[4].key = 125;
Mas[5].key = 333;
Mas[6].key = 11;
Mas[7].key = 437;
```

Data are placed in the hash table from 10 elements. Placement function: *i* = *key* % 10. Each element of the table is the pointer on stack top.

Received the hash table:

```
H[0] ← Mas[3]
H[1] ← Mas[6]
H[2] ← nullptr
H[3] ← Mas[2] ← Mas[5]
H[4] ← nullptr
H[5] ← Mas[0] ← Mas[1] ← Mas[4]
H[6] ← nullptr
H[7] ← Mas[7]
H[8] ← nullptr
H[9] ← nullptr
```

Example 19.2. The hash table uses the placement algorithm on the linked lists basis:

```
struct TNode  //  Description of the element of the stack
{
   int inf;  // Information part of structure
   TNode *a;  // Address part of structure
```

```
};

 TNode **sv_create(int m)
{
     TNode **H = new TNode* [m];
    for (int i = 0; i < m; i++) H[i] = nullptr;
    return H;
}
void sv_add(int inf, int m,  TNode** H)
{
     TNode* spt = new  TNode;
    spt->inf = inf;
    int i = abs(inf % m);
    if (H[i]) { spt->a = H[i]; H[i] = spt; }
    else  { H[i] = spt; spt->a = nullptr; }
}
 TNode* sv_seach(int inf, int m,  TNode** H)
{
    int i = abs(inf % m);
     TNode* spt = H[i];
    while (spt)
    {
        if (spt->inf == inf) return spt;
        spt = spt->a;
    }
    return nullptr;
}
void sv_delete(int m,  TNode** H)
{
     TNode* spt, * sp;
    for (int i = 0; i < m; i++)
    {
         sp = H[i];
        while (sp)  {
            spt = sp;
            sp = sp->a;
            delete spt;
        }
        }
        delete[]H;
    }
    void main ()
    {
        int n = 8;  // The number of elements in the array
        int mas [] = {5, 15, 3, 10, 125, 333, 11, 437};
        int m = 10;  // The number of elements in the hash table
          TNode** H = sv_create(m);
        for (int i = 0; i < n; i++) sv_add(mas[i], m, H);
         int key = 333;
         TNode* p = sv_seach(key, m, H);
         if (!p)  cout << "Item not found" << endl;
         else cout << p->inf << endl;
        sv_delete(m, H);
    }
```

Advantages:

1. Rather simple algorithm of the insert and search of elements.
2. The connected table cannot be crowded.

Shortcoming: bad work with unevenly placed data. For overcoming this shortcoming the technique considered above is used.

19.7. Blocks Method

The array of one-dimensional arrays of the identical size (blocks) is used.

The block number is to place the element is at the beginning. If the block overflows, then the element is placed in a special overflow block. The search is carried out in the found block and in the overflow block. The method has proven itself well when storing a hash table on a file. It writes and reads from a file can be done block by block. It is faster than element by element one.

LABS

Tasks of two levels of difficulty are in the practice. The problems with the symbol "A" have the lowest level of difficulty, the problems with the symbol "B" have a higher level of complexity.

1. Linear Algorithms Programming

A. *Enter basic data and receive result.*

A1. Create a program code of recalculation of weight from pounds to kilograms (1 pound = 0.4536 kg).

A2. Create a program code of recalculation of distance from more lovely to kilometers (1 mile = 1.609 km).

A3. Convert the dose of radioactive radiation from microsieverts to milliroentgens (1 μSv = 0.115 mr).

A4. Convert temperature from degrees Kelvin to degrees Celsius (0 ºK = = –273.1 °C).

A5. Create a program code of recalculation of volume from gallons to liters (1 gallon = 3.785 l).

A6. Create a program code of recalculation of distance from sea leagues to kilometers (1 sea league = 5.556 km).

A7. Create a program code of recalculation of weight from ounces to grams (1 ounce = 28.35 gr.).

A8. Create a program code of recalculation of distance from kabelt to meters (1 kabelt = 219.5 m).

A9. Create a program code of recalculation of distance from nautical miles to kilometers (1 nautical mile = 1.852 km).

A10. Create a program code of recalculation of length from yards to meters (1 yard = 0.9144 м).

A11. Create a program code of recalculation of volume from oil barrels to liters (1 barrel = 159 l).

A12. Create a program code of recalculation of speed from sea nodes to kilometers per hour (1 node = 1.852 km/h)

A13. Create a program code of recalculation of length from inches to centimeters (1 inch = 2.54 cm).

A14. Create a program code of recalculation of speed from miles per hour to kilometers per hour (1 mph = 1.609 km/h).

A15. Create a program code of recalculation of pressure from millimeters of mercury to pascals (1 mm Hg. = 133.3 Pas).

B. *Calculate value at the set basic data expression. Compare the received value to the specified correct result.*

B1. $$s = \frac{2\cos\left(x - \frac{2}{3}\right)}{\frac{1}{2} + \sin^2 y}\left(1 + \frac{z^2}{3 - z^2/5}\right).$$

$x = 14.26$; $y = –1\ 22$; $z = 3\ 5 \cdot 10^{-2}$. Answer of $s = 0.749155$.

B2. $$s = \frac{\sqrt[3]{9 + (x - y)^2}}{x^2 + y^2 + 2} - e^{|x-y|}\operatorname{tg}^3 z.$$

$x = –4.5$; $y = 0.75 \cdot 10^{-4}$; $z = –0.845 \cdot 10^2$. Answer of $s = –3.23765$.

B3. $$s = \frac{1 + \sin^2(x + y)}{\left|x - \frac{2y}{1 + x^2y^2}\right|}x^{|y|} + \cos^2\left(\operatorname{arctg}\frac{1}{z}\right).$$

$x = 3.74 \cdot 10^{-2}$; $y = –0.825$; $z = 0.16 \cdot 10^2$. Answer of $s = 1.05534$.

B4. $$s = |\cos x - \cos y|^{\left(1 + 2\sin^2 y\right)}\left(1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \frac{z^4}{4}\right).$$

$x = 0.4 \cdot 10^4$; $y = –0.875$; $z = –0.475 \cdot 10^{-3}$. Answer of $s = 1.98727$.

B5. $$s = \ln\left(y^{-\sqrt{|x|}}\right)\left(x - \frac{y}{2}\right) + \sin^2(\operatorname{arctg}(z)).$$

$x = –15.246$; $y = 4.642 \cdot 10^{-2}$; $z = 21$. Answer of $s = –182.038$.

B6. $$s = \sqrt{10\left(\sqrt[3]{x} + x^{y+2}\right)}\left(\arcsin^2 z - |x - y|\right).$$

$x = 16.55 \cdot 10^{-3}$; $y = –2.75$; $z = 0.15$. Answer of $s = –40.6307$.

B7. $$s = 5\operatorname{arctg}(x) - \frac{1}{4}\arccos(x)\frac{x + 3|x - y| + x^2}{|x - y|z + x^2}.$$

$x = 0.1722$; $y = 6.33$; $z = 3.25 \cdot 10^{-4}$. Answer of $s = –205.306$.

B8. $$s = \frac{e^{|x-y|}|x - y|^{x+y}}{\operatorname{arctg}(x) + \operatorname{arctg}(z)} + \sqrt[3]{x^6 + \ln^2 y}.$$

$x = –2.235 \cdot 10^{-2}$; $y = 2.23$; $z = 15.221$. Answer of $s = 39.3741$.

B9. $$s = \left|x^{\frac{y}{x}} - \sqrt[3]{\frac{y}{x}}\right| + (y - x)\frac{\cos y - \frac{z}{(y - x)}}{1 + (y - x)^2}.$$

$x = 1.825 \cdot 10^2$; $y = 18.225$; $z = –3.298 \cdot 10^{-2}$. Answer of $s = 1.21308$.

B10. $$s = 2^{-x}\sqrt{x + \sqrt[4]{|y|}}\sqrt[3]{e^{x-1/\sin z}}.$$

$x = 3.981 \cdot 10^{-2}$; $y = –1.625 \cdot 10^3$; $z = 0.512$. Answer of $s = 1.26185$.

$$\text{B11. } s = y^{\sqrt[3]{|x|}} + \frac{\cos^3(y)}{e^{|x-y|} + \frac{x}{2}} \cdot |x-y| \left(1 + \frac{\sin^2 z}{\sqrt{x+y}}\right).$$

$x = 6.251$; $y = 0.827$; $z = 25.001$. Answer of $s = 0.712122$.

$$\text{B12. } s = 2^{\left(y^x\right)} + \left(3^x\right)^y - \frac{y\left(\operatorname{arctg} z - \frac{1}{3}\right)}{|x| + \frac{1}{y^2+1}}.$$

$x = 3.251$; $y = 0.325$; $z = 0.466 \cdot 10^{-4}$. Answer of $s = 4.23655$.

$$\text{B13. } s = \frac{\sqrt[4]{y + \sqrt[3]{x-1}}}{|x-y|\left(\sin^2 z + \operatorname{tg} z\right)}.$$

$x = 17.421$; $y = 10.365 \cdot 10^{-3}$; $z = 0.828 \cdot 10^{5}$. Answer of $s = 0.330564$.

$$\text{B14. } s = \frac{y^{x+1}}{\sqrt[3]{|y-2|} + 3} + \frac{x + \frac{y}{2}}{2|x+y|}(x+1)^{-1/\sin z}.$$

$x = 12.3 \cdot 10^{-1}$; $y = 15.4$; $z = 0.252 \cdot 10^{3}$. Answer of $s = 82.8256$.

$$\text{B15. } s = \frac{x^{y+1} + e^{y-1}}{1 + x|y - \operatorname{tg} z|}\left(1 + |y-x|\right) + \frac{|y-x|^2}{2} - \frac{|y-x|^3}{3}.$$

$x = 2.444$; $y = 0.869 \cdot 10^{-2}$; $z = -0.13 \cdot 10^{3}$. Answer of $s = -0.498707$.

Example of the lab fulfilment

Condition: Create a program code for calculation of linear arithmetic expression

$$h = \frac{x^{2y} + \mathrm{e}^{y-1}}{1 + x|y - \operatorname{tg} z|} + 10 \cdot \sqrt[3]{x} - \ln(z).$$

$x = 2.45$, $y = -0.423 \cdot 10^{-2}$, $z = 1.232 \cdot 10^{3}$. Answer of $h = 6.9465$.

Example of the program code:

```
#include <iostream.h>
#include <math.h>
int main ()
{
 double x, y, z, a, b, c, h;
      cout << "Input x: ";
         cin >> x;
      cout << " Input y: ";
         cin >> y;
      cout << " Input z: ";
```

```
            cin >> z;
      a = pow(x,2*y)+exp(y-1);
      b = 1+x*fabs(y-tan(z));
      c = 10*pow(x,1/3.)-log(z);
        h = a/b+c;
     cout << "Result h= " << h << endl;
    return 0;
   }
```

2. Branching Algorithms Programming

***A.** Enter basic data. Fulfil the task.*

A1. Create a program code of the choice of the greatest of three numbers.

A2. Three numbers x, y, z are given. Find out if $x > y > z$ is true or not true. Display the answer in text form "true" or "false".

A3. Three real numbers are given. Multiply negative numbers.

A4. Four integer numbers are given. Find the sum of positive numbers.

A5. Radius of the circle and length of the party of the square are given. Find out at what figure the area is more? Output the answer in the text form "at the circle" or "at the square".

A6. Three real numbers are given. Find the sum of positive numbers.

A7. Three real numbers are given. Cube and display negative numbers.

A8. Two integer numbers are given. If both negative numbers then to calculate the sum of their modules; if only one of numbers negative then to calculate the work of numbers; if both numbers positive, then the result is equal to zero.

A9. Three integers are given. Output the numbers that are evenly divided into three.

A10. Three real numbers are given. Subtract smaller from bigger number.

A11. Four integer numbers are given. Find the work of negative numbers.

A12. Three real numbers are given. Multiply the even numbers.

A13. Four integers are given. Subtract the sum of modules of the negative numbers from the sum of the positive numbers.

A14. Three integer numbers are given. Subtract the sum of even numbers from the sum of all numbers.

A15. Four integer numbers are given. Find out whether the sum of the two first is equal to the sum of two last numbers. Output the answer "is equal" in the text form or "it is not equal" in the text form.

***B.** Calculate value according to number of option. Provide the possibility of the choice of the type of function f (x): sh(x), x^2 or ex. Display the information on the executed branch of calculations.*

B1. $$a=\begin{cases}\left(f(x)+y\right)^2-\sqrt[3]{f(x)}, & xy>0,\\ \left(f(x)+y\right)^2+\sin(x), & xy<0,\\ \left(f(x)+y\right)^2+y^3, & else.\end{cases}$$

B2. $$b=\begin{cases}\ln(f(x))+\sqrt[4]{|f(x)|}, & x/y=0,\\ \ln|f(x)/y|-y^2, & x/y<0,\\ (f(x)^2+y)^3, & else.\end{cases}$$

B3. $$a=\begin{cases}f(x)^2+\sqrt[3]{y}+\sin(y), & x-y=0,\\ \left(f(x)+y\right)^2+\ln(x), & x-y>0,\\ \left(y-f(x)\right)^2+\mathrm{tg}(y), & else.\end{cases}$$

B4. $$d=\begin{cases}\sqrt[3]{|f(x)+x|}-tg(y), & x>y,\\ (y-f(x))^3+\sin(y), & x=y,\\ y+x^3-\sqrt{f(x)}, & else.\end{cases}$$

B5. $$e=\begin{cases}\sin(f(x))/3, & xy=0,\\ \ln(|y-f(x)|), & 7<xy<10,\\ 2tg^2(x)-y, & else.\end{cases}$$

B6. $$g=\begin{cases}e^{f(x)-y}+\sqrt[3]{x}, & x/y=0,\\ x^2-\ln(y^2+x), & -5<x/y<0,\\ 2f(x)^2-y^3, & else.\end{cases}$$

B7. $$s=\begin{cases}\sin^2(x)-f(x), & x+|y|=0,\\ \sqrt[3]{|xy|}, & x+|y|<0,\\ 3f^2(x), & else.\end{cases}$$

B8. $$b=\begin{cases}\sqrt{x}/y, & x^2+y=0,\\ \cos^3(y)-f(x), & x^2+y<0,\\ \sin(\cos(2f(x))), & else.\end{cases}$$

B9. $$l = \begin{cases} 2\sin^2(\ln(|x|)), \ y = 0, \\ tg(y^2 - x), \ \ -5 < y < 0, \\ x^2 + y - 9, \ else. \end{cases}$$

B10. $$e = \begin{cases} \ln(|y| + |f(x)|), \ |xy| > 10, \\ e^{f(x)+y}, \ \ |xy| < 10, \\ \sqrt[3]{|f(x)|} + y, \ else. \end{cases}$$

B11. $$w = \begin{cases} tg^2(x) - f(x), \ xy = 0, \\ e^{2f(x)} - y^2, \ \ 0 < xy < 10, \\ \ln(|y|) + 2f(x), \ else. \end{cases}$$

B12. $$g = \begin{cases} y^2 \cdot \sin^2(x), \ y \cdot f(x) = 0, \\ tg^2(x) + f(x), \ y \cdot f(x) < 0, \\ 2f(x) - \sin(y), \ else. \end{cases}$$

B13. $$q = \begin{cases} \ln(x) - f^2(x), \ y \cdot f(x) = 10, \\ 2y - 10\sin(x), \ y \cdot f(x) < 10, \\ y^2 + f^2(x), \ \ else. \end{cases}$$

B14. $$u = \begin{cases} \sin(x) + \ln(y), \ x^2 y = 0, \\ tg^2(f(x)), \ \ 2 < x^2 y < 7, \\ f(x)^2 / 2 + x, \ else. \end{cases}$$

B15. $$w = \begin{cases} \sqrt[3]{f(x)} - xy, \ 2x / y = 0, \\ \sin^2(x) - y, \ 2x / y < 0, \\ 4y - tg(x), \ else. \end{cases}$$

3. Loop Algorithms Programming

A. *Display the table of function values y(x) for x, changing from a to b with step h = (b – a)/10. The task is to select according to number of option.*

A1. $$y(x) = \sum_{i=1}^{n} \left(\sin(ix) + \cos^2(i) \right).$$

A2. $$y(x) = \sum_{i=1}^{n} \left(5\sin(2ix) - \cos^2(x) \right).$$

A3. $y(x)=\sum_{i=1}^{n}\left(2\mathrm{tg}(ix)\cdot e^{2i}\right)$.

A4. $y(x)=\sum_{i=1}^{n}\left(15x^2-4\cos^3(ix)\right)$.

A5. $y(x)=\sum_{i=1}^{n}\left(x^{2i}\cdot\cos^2(2ix)\right)$.

A6. $y(x)=\sum_{i=1}^{n}\left(2e^{i\cdot\sin(x)}+3\sqrt{|x|}\right)$.

A7. $y(x)=\sum_{i=1}^{n}\left(2\cos(ix)\cdot ch(x)\right)$.

A8. $y(x)=\sum_{i=1}^{n}\left(e^{2\cos ix}\cdot x^{\cos(i)}\right)$.

A9. $y(x)=\sum_{i=1}^{n}\left(\sin^2(i)-3e^{ix}\right)$.

A10. $y(x)=\sum_{i=1}^{n}\left(2\mathrm{tg}^2(ix)-\sqrt{|x|}\right)$.

A11. $y(x)=\sum_{i=1}^{n}\left(2\ln(ix)-\sin^{2i}(x)\right)$.

A12. $y(x)=\sum_{i=1}^{n}\left(4\sqrt[3]{|ix|}+\sin x\right)$.

A13. $y(x)=\sum_{i=1}^{n}\left(3e^{ix}+\mathrm{ctg}(x)\right)$.

A14. $y(x)=\sum_{i=1}^{n}\left(\sqrt{|\sin 2x|}+e^{-i\sin x}\right)$.

A15. $y(x)=\sum_{i=1}^{n}\left(3x^{2i}+4e^{3i}\right)$.

***B.** Display the table of function values y(x) and its decomposition in a row of s (x) for x, changing from a to b with h step = (b – a)/10. The task is to select according to number of option in the tab. I*

Table I

Option number	a	b	Function	Function decomposition in a row Taylor	k
B1	0.1	1	$y(x)=\sin(x)$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n+1}}{(2n+1)!}$	160
B2	0.1	1	$y(x)=\mathrm{ch(x)}$	$s(x)=\sum_{n=0}^{k}\frac{x^{2n}}{(2n)!}$	100
B3	0.1	1	$y(x)=e^{x\sin(x)}$	$s(x)=\sum_{n=0}^{k}\frac{(x\cdot\sin(x))^n}{n!}$	120
B4	0.1	1	$y(x)=\cos(x)$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n}}{(2n)!}$	80
B5	0.1	1	$y(x)=\frac{\sin x}{x}$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n}}{(2n+1)!}$	140
B6	0.1	1	$y(x)=\mathrm{sh(x)}$	$s(x)=\sum_{n=0}^{k}\frac{x^{2n+1}}{(2n+1)!}$	80
B7	0.1	1	$y(x)=e^{-2e^x}$	$s(x)=\sum_{n=0}^{k}\frac{2^n(-e^x)^n}{n!}$	120
B8	0.1	1	$y(x)=5^x$	$s(x)=\sum_{n=0}^{k}\frac{x^n\ln^n(5)}{n!}$	100
B9	0.1	1	$s(x)=e^{2x}$	$s(x)=\sum_{n=0}^{k}\frac{(2x)^n}{n!}$	140
B10	0.1	0.5	$y(x)=x^2e^x$	$s(x)=\sum_{n=2}^{k}\frac{x^n}{(n-2)!}$	150
B11	0.1	1	$y(x)=x\sin(x)$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n+2}}{(2n+1)!}$	100
B12	0.1	1	$y(x)=e^{\cos(x)}$	$s(x)=\sum_{n=0}^{k}\frac{\cos^n(x)}{n!}$	80
B13	–2	–0.1	$y(x)=x\cos(x)$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n+1}}{(2n)!}$	140
B14	0.2	0.8	$y(x)=3^{x-1}$	$s(x)=\sum_{n=0}^{k}\frac{(x-1)^n\ln^n(3)}{n!}$	100
B15	0.1	0.8	$y(x)=\cos(2x)$	$s(x)=\sum_{n=0}^{k}(-4)^n\frac{x^{2n}}{(2n)!}$	180

4. One-dimensional Arrays Implementation

A. *Enter from the keyboard the array from 10 elements. Perform the task, to display the result.*

A1. The real array is given. Find the sum of the positive and the product of the negative array elements.

A2. The integer array is given. Find the product of the even and the sum of negative array elements.

A3. The real array is given. Find the difference between the sum of the positive elements and the sum of the modulus of the negative elements.

A4. The integer array is given. Find the sum of the minimum and maximum array cells.

A5. The real array is given. Display the elements whose value is greater than the average value of all elements in the array.

A6. The integer array is given. Output the numbers of the minimum and maximum elements and their value.

A7. The real array is given. Find how many elements there are between the minimum and maximum elements of the array

A8. The integer array is given. Find how many elements matter less average value of all array cells.

A9. The real array is given. Find how many elements have the value less than the average value of all array elements.

A10. The integer array is given. Replace the negative elements with the half-sum of the next elements. Do not change the end elements.

A11. An array of the real numbers is given. Count the number of the positive and sum of the negative elements.

A12. Given the array of integers. Find the number and sum of elements that have values greater than 10 and less than 100.

A13. Given the array of the real numbers. Find the amount and the product of the negative odd elements.

A14. The array of the integers is given. Display the numbers that are less than the maximum value and greater than the average value of the array elements.

A15. The array of real numbers is given. Find the array elements average and the coordinates of the minimum and maximum array elements.

B. *Enter the array size from the keyboard, select the necessary memory size for storage of array cells and enter basic data. Perform the task, to display the result.*

B1. The integer array is given. Sort the array elements non-descending from modules.

B2. The integer array is given. Transform the array as follows: move all the negative array elements to the beginning. And move all the remaining elements to the end. The initial relative position of both the negative and positive elements remains the same.

B3. The integer array is given. Find the number which is most often found in this array.

B4. The integer array is given. Find the numbers that occur in the array no more than once.

B5. The real array is given. Shift array cells cyclically on *n* of positions to the right (the value n is set from the keyboard).

B6. The integer array is given. Delete all numbers which are found in the array more than once from the array.

B7. The real array is given. Move the maximum element to the zero position, and the minimum element to the last position of the array. You should not change the relative position of the remaining elements.

B8. The real array is given. Delete all positive elements which have the negative element on the right.

B9. The integer array is given. Delete the minimum and maximum elements from the array.

B10. The real array is given. Find the sum of the elements located between the minimum and maximum array cells.

B11. The integer array is given. Find the work of the elements located between the last and penultimate positive array cells.

B12. The real array is given. Rearrange upside-down the elements located between the first positive and the last negative array cells.

B13. The integer array is given. Delete all elements standing to the element with the maximum value.

B14. The real array is given. Determine quantity of the different elements in the array.

B15. The integer array is given. Find the smallest positive element among elements with even indexes of the array.

5. Two-dimensional Arrays Implementation

A. *Enter by the keyboard the 5×5 two-dimensional elements array. Run the task, display the result.*

A1. The integer array is given. Count the number of rows that contain the null elements.

A2. The real array is given. Display coordinates of the minimum element in each column.

A3. The integer array is given. Display the number of the even element values in each row.

A4. The real array is given. Display number of the negative elements in each column.

A5. The integer array is given. Display the average value of every line.

A6. The real array is given. Find the minimum, maximum and the average value of the array elements.

A7. The integer array is given. Find the minimum value element in each row.

A8. The real array is given. Display the average value of the elements in all even lines of the array.

A9. The integer array is given. Output the maximum of the elements located in even columns of the matrix.

A10. The real array is given. Display the elements average value of the each column.

A11. The integer array is given. Find in each column the element with the maximum value.

A12. The real array is given. Display number of the positive elements in every line.

A13. The integer array is given. Print the number of the odd element values in each column.

A14. The real array is given. Display coordinates of the maximum element in every line.

A15. The integer array is given. Count the columns number with the negative elements.

B. *Enter from the keyboard the number of rows and columns of the array, allocate the required amount of memory to store the elements of the array and enter the initial data. Run the task and display the result.*

B1. The N×M matrix is given. Swap the string containing the element with the maximum value with the string containing the element with the minimum value.

B2. The N×M matrix is given. Arrange its columns by increase of their smallest elements.

B3. The N×M matrix is given. Delete the matrix column containing the element with the minimum value.

B4. The N×M matrix is given. Create a one-dimensional array. The cell value is 0 if the matrix row with the same number contains at least one zero element. Otherwise the value is 1.

B5. The N×M matrix is given. Delete the line with the maximum amount of elements.

B6. The N×M matrix is given. Determine the number of "special" elements of the matrix. An element is considered "special" if it is greater than the sum of the other elements in the column.

B7. The N×M matrix is given. Arrange lines by increase of the sum of their elements.

B8. The N×M matrix is given. Determine the number of different elements of the matrix (i. e., count the repeating elements once).

B9. The N×M matrix is given. Swap the row with the maximum element, and the row with the minimum element.

B10. The N×M matrix is given. Display all elements which are maximum in the column and at the same time minimum in the line.

B11. The N×M matrix is given. Get a one-dimensional array. Its element will be equal to the value 0 if the matrix row with the same number is sorted in ascending order. The element is 1 otherwise.

B12. The N×M matrix is given. Delete the line of the matrix containing the element with the maximum value.

B13. The N×M matrix is given. Determine the number of "special" elements of the matrix. An element is considered "special" if it is less than the sum of the remaining elements in the row

B14. The N×M matrix is given. Swap the column with the minimum element value with the column with the maximum element value.

B15. The N×M matrix is given. Arrange its lines by decrease of their maximum elements.

6. Functions Implementation

A. *Display the table of function values of y (x, n) for x, changing from a to b with step h = (b – a)/10.*

Calculate y (x, n) and place in function. Transfer parameters by the method specified in the tab. II.

Table II

Option number	*a*	*b*	*n*	Function	Transfer method parameters
1	2	3	4	5	6
A1	0.13	0.9	10	$y(x,n)=\sum_{i=1}^{n}\left(3e^{ix}+\text{ctg}(x)\right)$	According to the link
A2	0.24	1.2	8	$y(x,n)=\sum_{i=1}^{n}\left(\sqrt{\lvert\sin 2x\rvert}+e^{-i\sin x}\right)$	By value
A3	0.15	0.95	7	$y(x,n)=\sum_{i=1}^{n}\left(2\text{tg}(ix)\cdot e^{2i}\right)$	According to the pointer
A4	0.35	1.25	12	$y(x,n)=\sum_{i=1}^{n}\left(x^{2i}\cdot\cos^2(2ix)\right)$	According to the link
A5	0.22	1.1	11	$y(x,n)=\sum_{i=1}^{n}\left(2\cos(ix)\cdot\text{ch}(x)\right)$	By value
A6	0.36	0.9	6	$y(x,n)=\sum_{i=1}^{n}\left(\sin^2(i)-3e^{ix}\right)$	According to the pointer
A7	0.34	1.1	8	$y(x,n)=\sum_{i=1}^{n}\left(2\ln(ix)-\sin^{2i}(x)\right)$	According to the link
A8	0.23	0.9	5	$y(x,n)=\sum_{i=1}^{n}\left(3x^{2i}+4e^{3i}\right)$	By value

1	2	3	4	5	6
A9	0.55	1.4	15	$y(x,n)=\sum_{i=1}^{n}\left(5\sin(2ix)-\cos^2(x)\right)$	According to the pointer
A10	0.32	0.8	9	$y(x,n)=\sum_{i=1}^{n}\left(15x^2-4\cos^3(ix)\right)$	According to the link
A11	0.13	0.7	7	$y(x,n)=\sum_{i=1}^{n}\left(2e^{i\cdot\sin(x)}+3\sqrt{\lvert x\rvert}\right)$	By value
A12	0.25	0.8	6	$y(x,n)=\sum_{i=1}^{n}\left(e^{2\cos ix}\cdot x^{\cos(i)}\right)$	According to the pointer
A13	0.44	1.1	9	$y(x,n)=\sum_{i=1}^{n}\left(2\mathrm{tg}^2(ix)-\sqrt{\lvert x\rvert}\right)$	According to the link
A14	0.32	1.2	11	$y(x,n)=\sum_{i=1}^{n}\left(4\sqrt[3]{\lvert ix\rvert}+\sin x\right)$	By value
A15	0.12	1.4	18	$y(x,n)=\sum_{i=1}^{n}\left(\sin(ix)+\cos^2(i)\right)$	According to the pointer

B. *Display the table of function values and its decomposition in a row for x, changing from a to b with step h = (b – a)/10. Place calculation of y (x) and s(x) in function. Use prototypes of functions. Transfer parameters by the method specified in the tab. III. Execute calculation of the s (x) function with the given accuracy ε.*

Table III

Option num-ber	*a*	*b*	Function	Function decomposition in a row Taylor	ε	Parameter passing method
1	2	3	4	5	6	7
B1	0.8	1.8	$y(x)=\ln(x)$	$s(x)=-\sum_{n=1}^{\infty}(-1)^n\frac{(x-1)^n}{n}$	10^{-4}	According to the link
B2	0.1	0.9	$y(x)=ch^2(x)$	$s(x)=\sum_{n=1}^{\infty}\frac{2^{2n-1}x^{2n}}{(2n)!}$	10^{-5}	By value
B3	1.9	2.9	$y(x)=\frac{1}{1+x}$	$s(x)=\sum_{n=0}^{\infty}(-1)^n x^n$	10^{-6}	According to the pointer
B4	0.1	0.9	$y(x)=x\cdot\arctan(x)$	$s(x)=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+2}}{1+2n}$	10^{-4}	According to the link

1	2	3	4	5	6	7
B5	–0.1	1	$y(x)=2^{-x}$	$s(x)=\sum_{n=0}^{\infty}\frac{x^n(-\ln(2))^n}{n!}$	10^{-5}	By value
B6	–0.9	0.9	$y(x)=\cos(x-4)$	$s(x)=\sum_{n=0}^{\infty}(-1)^n\frac{(4-x)^{2n}}{(2n)!}$	10^{-3}	According to the pointer
B7	–0.5	0.5	$y(x)=\cos(\sin(x))$	$s(x)=\sum_{n=0}^{\infty}(-1)^n\frac{\sin^{2n}(x)}{(2n)!}$	10^{-4}	According to the link
B8	–0.3	0.4	$y(x)=e^x+e^{-x}$	$s(x)=\sum_{n=0}^{\infty}\frac{(-x)^n+x^n}{n!}$	10^{-5}	By value
B9	–0.1	1.3	$y(x)=2^x$	$s(x)=\sum_{n=0}^{\infty}\frac{x^n\ln^n(2)}{n!}$	10^{-3}	According to the pointer
B10	–0.5	0.5	$y(x)=e^x$	$s(x)=\sum_{n=0}^{\infty}\frac{x^{2n-1}(2n+x)}{(2n)!}$	10^{-4}	According to the link
B11	0.1	0.8	$y(x)=\ln(1+x^2)$	$s(x)=-\sum_{n=1}^{\infty}(-1)^n\frac{x^{2n}}{n}$	10^{-5}	By value
B12	1	2.5	$y(x)=\sin^2(x)$	$s(x)=-\sum_{n=1}^{\infty}(-1)^n\frac{2^{2n-1}x^{2n}}{(2n)!}$	10^{-3}	According to the pointer
B13	–1.5	1.5	$y(x)=\cos^3(x)$	$s(x)=\frac{1}{4}\sum_{n=0}^{\infty}(-1)^n\frac{(3+9^n)x^{2n}}{(2n)!}$	10^{-4}	According to the link
B14	–0.8	0.9	$y(x)=ch(x^2)$	$s(x)=\sum_{n=0}^{\infty}\frac{x^{4n}}{(2n)!}$	10^{-5}	By value
B15	–0.9	0.9	$y(x)=\arctan(x)$	$s(x)=\sum_{n=0}^{\infty}(-1)^n\frac{x^{2n+1}}{2n+1}$	10^{-3}	According to the pointer

7. Strings Implementation

***A.** Enter the string from the keyboard. Run the task, display result.*

A1. Check the brackets balance in the string (the number of opening brackets must correspond the number of closing brackets). Display test result.

A2. Count what number of words in string begins with the character of 'w'.

A3. Find and display the sequences consisting of three identical consecutive characters.

A4. Display the second sentence of the string (the characters located between the first and second point).

A5. Count the sum of the digits which are found in the string.

A6. Count the number of words in string. Separate words from each other one space. There is no space before the first word.

A7. Replace in string the character '-' with the character '*'.

A8. Display the words consisting of two characters. Separate words from each other with one space. The first and the last characters of the string are spaces.

A9. Enter the character. Determine numbers of words which begin with the entered character. Separate words from each other with one space. There is no space before the first word.

A10. Count what quantity of the letters 'a' there is in the first word of the string. Separate words from each other one space. There is no space before the first word.

A11. Display the last word of the string. The last character of the string is not the space.

A12. Display the number of words which have the last character of 'g'. The string comes to an end with gap character.

A13. Determine how many times there is the "wse" string.

A14. Replace in string the character '-' with the character '*'.

A15. Display the third word of the string. Separate words from each other with one space. There is no space before the first word.

B. *Enter the string from the keyboard. Run the task, display the result.*

B1. Display sequence number of the word of the maximum length and item number in string with which it begins. Words in string are separated by one or several spaces.

B2. Delete the penultimate word from the string. Words in string are separated by one or several spaces.

B3. Display words which begin and come to an end with the same letter. Words in string are separated by one or several spaces.

B4. Replace in string all words "C" with the "C++". Words in string are separated by one or several spaces.

B5. The string consisting of zero and units is given. Display groups of units with the maximum and minimum quantity of characters.

B6. Delete from the string the word, containing the 'r' character. Words in string are separated by one or several spaces.

B7. The string consisting of zero and units is given. Count the number of groups with five units.

B8. The string of characters consisting of any decimal digits is given. Numbers in string are separated from each other by one or several spaces. Delete even numbers from the string.

B9. Replace all consecutive spaces in the string with the single space.

B10. The string consisting of zero and units is given. Delete all groups consisting of three zero.

B11. Squeeze the word "Visual" between the second and third word of the string. Words in string are separated by one or several spaces.

B12. Interchange places the first and second word of the string. Words in string are separated by one or several spaces.

B13. Delete from the string the word, containing even quantity of characters. Words in string are separated by one or several spaces.

B14. The string of characters consisting of any decimal digits is given. Numbers in string are separated from each other by one or several spaces. Display numbers of this string in ascending order of their values.

B15. The string consisting of zero and units is given. Display group with the maximum quantity of identical characters.

8. Structures' Implementation

A. *Announce structure with the set fields. Enter the necessary list. Select memory for storage of the list dynamically. Run the task, display result.*

A1. There is the list of students. Each element of the list contains the following information: surname, year and place of birth, three exam grades for the last session. Display information about students with an average score of more than 7.

A2. There is the list of staff of the enterprise. Each element of the list contains the following information: surname, the year of birth and the year of revenues to work. Output information about employees of the firm who were born before 1980.

A3. There is the telephone database. Each element of base contains the following information: phone number, surname and subscriber's address. Output surnames of subscribers where phone numbers begin on number 5.

A4. There is the list of cars. Each element of the list contains the following information: brand, year of release, engine displacement and maximum speed. Output information about cars produced after 2000 and having a maximum speed of more than 180 km/h.

A5. There is the list of the countries of the world. Each element of the list contains the following information: the name of the country and its capital, the name of the part of the world in which the country is sutuated, the area of the country. Output information about the countries in Africa.

A6. There is the schedule of the movement of long-distance buses. Each element of the schedule contains the following information: flight number, departure time, destination point, arrival time to the destination point. Output information on all runs to the city of Mogilev.

A7. There is the list of books. Each element of the list contains the following information: name, surname of the author, year of the edition, number of pages. Output all books which title begins on letter 'A'.

A8. There is the list of participants of sports competitions. Each element of the list contains the following information: name of the team, surname of the athlete, age, height and weight. Display information about athletes who are taller than 190 cm.

A9. At the administrator of railway cash desks information on empty seats is stored in trains. Each element of the list contains the following information: departure time, destination point, number of empty seats. Output information about the trains going to Moscow on which there are empty seats.

A10. There is the list of the goods which are stored in the warehouse. Each element of the list contains the following information: name, quantity and price. Output information about goods which quantity is less than 10 pieces.

A11. At the airport there is the list of the passengers who have checked in for the flight. Each element of the list contains the following information: surname, ticket number, baggage weight. Output the list of passengers whose weight of baggage exceeds 20 kg.

A12. There is the list of participants of the competition. Each element of the list contains the following information: name of educational institution, surname and number of points scored. Output participants who have scored more than 10 points.

A13. There is the list of seeds of vegetable cultures. Each element of the list contains the following information: name of culture, number of months of crops, planting of seedling and harvesting. Output information about plants which crops time is March.

A14. There is the list of students. Each element of the list contains the following information: surname, year and place of birth, three examination grades for the last session. Output information about students born after 1995.

A15. There is the list of cars. Each element of the list contains the following information: brand, year of release, engine displacement and fuel consumption. Output information about cars with the engine displacement more than 3 liters and fuel consumption less than 10 liters to 100 km.

***B.** Announce structure with the set fields. Dynamically select memory for storage of the list. Enter data. Run the task, display result.*

B1. There is the list of students. Each element of the list contains the following information: surname, year and place of birth, three examination grades for the last session. Output information about students living in Minsk in descending order of the average score.

B2. There is the list of staff of the enterprise. Each element of the list contains the following information: surname, year of birth and year of revenues to work. Display information about the company's employees born after 1985 in descending order of work experience.

B3. There is the telephone database. Each element of base contains the following information: phone number, surname and subscriber's address. Display in alphabetical order surnames of subscribers whose phone numbers begin on number 3.

B4. There is the list of cars. Each element of the list contains the following information: brand, year of release, engine displacement and maximum speed. Output information about the cars released after 2005 in decreasing order of their maximum speed.

B5. There is the list of the countries of the world. Each element of the list contains the following information: the name of the country, year of formation of the state, the name of the part of the world in which there is the country and the area of the country. Output information about countries located in Europe, in order of increasing their area.

B6. There is the schedule of the movement of long-distance buses. Each element of the schedule contains the following information: flight number, departure time, destination point, arrival time to the destination point. Output information about bus routes to Grodno in ascending order of their departure time.

B7. There is the list of books. Each element of the list contains the following information: name, surname of the author, year of the edition, number of pages. Output in alphabetical order the titles of the books published till 1990.

B8. There is the list of participants of sports competitions. Each element of the list contains the following information: name of the team, surname of the athlete, his or her age, height and weight. Display in alphabetical order surnames of athletes whose age is younger than 18 years.

B9. At the administrator of railway cash desks information on empty seats is stored in trains. Each element of the list contains the following information: departure time, destination point, number of empty seats. Output information about trains to Brest in descending order of the number of available seats.

B10. There is the list of the goods which are stored in the warehouse. Each element of the list contains the following information: name, quantity and price. Output in alphabetical order information about goods with more than 10 and less than 100 items in stock.

B11. At the airport there is the list of the passengers who have checked in for the flight. Each element of the list contains the following information: surname, ticket number, baggage weight. Display in alphabetical order surnames of passengers whose-weight of baggage does not exceed 15 kg.

B12. There is the list of participants of the Olympic Games. Each element of the list contains the following information: name of educational institution, surname of the participant, number of points scored. Bring in decreasing order the number of points scored of the surname of participants out of BSUIR.

B13. There is the list of seeds of vegetable cultures. Each element of the list contains the following information: name of culture, number of months of crops, planting of seedling and harvesting. Output in alphabetical order information about goods with more than 10 and less than 100 items in stock.

B14. There is the list of students. Each element of the list contains the following information: surname, the year and place of birth, three examination grades for the last session. Display in alphabetical order surnames of students who have passed examinations without the two.

B15. There is the list of cars. Each element of the list contains the following information: brand, year of release, engine displacement and fuel consumption. Output information about the cars released after 2004 in ascending order of fuel consumption.

9. Files Implementation

***A.** Create the binary file, write in it ten real numbers and close the file. Open the file for reading, read the written data and perform the task. Display result also in the text file. Close all open files.*

A1. Find the sum even and quantity of negative numbers.

A2. Find quantity of the odd numbers facing positive numbers.

A3. Display positive numbers, multiple of three and all negative.

A4. Find out which of numbers (minimum or maximum) is closer to the beginning of the file.

A5. Find out whether numbers in the file are on increase of their values located.

A6. Find out what numbers are more, negative or positive.

A7. Find quantity of numbers which value is more than average value of all numbers.

A8. Find the difference between the sum of modules of positive numbers and the sum of modules of negative numbers.

A9. Find quantity of the numbers which are between the minimum and maximum numbers.

A10. Find out whether there are negative numbers which value on the module is more than average value of all numbers.

A11. Display the even numbers being after number with the maximum value.

A12. Count the sum of the numbers being between the maximum and minimum numbers.

A13. Display the negative numbers facing number with the minimum value.

A14. Find average value of positive numbers and average value of negative numbers.

A15. Find number which value is closest to average value of all numbers.

***B.** Write feature set for execution of the following tasks: creation of the binary file; data record in the file; opening of the file and reading data from it; result output to the screen; the result output in the text file. For the challenge of necessary functions use the menu. You should take the task from the corresponding option of laboratory work № 8.*

10. Recursion's Implementation

***A.** Enter the one-dimensional array from the keyboard. Solve the problem by recursive splitting the array into two parts. For control solve the problem with use of the cyclic algorithm.*

A1. Find quantity of negative array cells. At recursive splitting divide the array into two halfes.

A2. Find the sum of positive array cells. When splitting the array recursively, divide the array into the first third and the rest (2/3) of the array.

A3. Find quantity of even array cells. At recursive splitting divide the array into two halfes.

A4. Find quantity of array cells which values are more than 10 and less than 20. At recursive splitting of the array divide (2/3) arrays into the first third and the rest.

A5. Find value of the minimum array cell. At recursive splitting divide the array into two halfes.

A6. Define whether there are the negative elements in the array. At recursive splitting divide the array into the first 2/3 and other third of the array.

A7. Determine the number of array elements for which the condition sin(a[i]) > 0 is satisfied. When partitioning the array recursively, divide the array into two halves. At recursive splitting divide the array into two halves.

A8. Determine the number of array elements for which the condition 0 < cos(a[i]) < 0.5 is satisfied. At recursive splitting divide the array into the first 2/3 and other third of the array.

A9. Find the work of negative array cells. At recursive splitting divide the array into two halves.

A10. Find the sum of the array elements for which the condition $a[i]^2 > 10$ is satisfied. At recursive splitting divide the array into the first 2/3 and other third of the array.

A11. Define whether there are the even elements in the array. At recursive splitting divide the array into two halfes.

A12. Find the work of positive array cells. At recursive splitting of the array divide (2/3) arrays into the first third and the rest.

A13. Find number of the maximum array cell. At recursive splitting the array divide (2/3) arrays into the first third and the rest.

A14. Find the sum of array cells which values are more than 3 and less than 10. At recursive splitting divide the array into the first 2/3 and other third of the array.

A15. Find the work of odd array cells. At recursive splitting of the array divide (2/3) arrays into the first third and the rest.

B. *Solve the problem by two methods: using the recursion and without it.*

B1. Write function of multiplication of two numbers, using only addition operation.

B2. In an ordered array of integers a_i, i = 0...n–1, find the element number x using the binary search method: if $x \le a_{n/2}$, then $x \in \left[a_1 \ldots a_{n/2}\right]$, otherwise $x \in \left[a_{n/2+1} \ldots a_n\right]$. If element x is absent in the array, then display the corresponding message.

B3. Write function of addition of two numbers, using only operation of adding of unit.

B4. Calculate the product of two integer positive numbers $P = a \cdot b$ on the following algorithm: if b even, then it is $P = 2 \cdot (a \cdot b / 2)$, differently is $P = a + (a \cdot (b - 1))$. If b = 0, then P = 0.

B5. Count the sum of digits in decimal notation of the set number.

B6. Find function value of Akkerman *of* $A(m, n)$ which is defined for all non-negative integer arguments *of* m and n as follows: $A(0, n) = n + 1$, if $m = 0$; $A(m, 0) = A(m - 1, 1)$, if $n = 0$; $A(m, n) = A(m - 1, A(m, n - 1))$ if $m > 0$ and $n > 0$.

B7. Calculate the work of even value ($n \geq 2$) of multiplicands $y(n) = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots \cdot \frac{n}{n-1} \cdot \frac{n}{n+1}$.

B8. Check whether the set line is the palindrome.

B9. Calculate number of combinations $C_n^k = \frac{n!}{k!(n-k)!}$ on the formula: $C_n^0 = C_n^n = 1,\ C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ at $n > 1$, $0 < k < n$.

B10. Calculate $y(n) = \sqrt{1 + \sqrt{2 + \ldots + \sqrt{(n-1) + \sqrt{n}}}}$.

B11. Calculate value $x = \sqrt{a}$, using the formula $x_n = \frac{1}{2}\left(x_{n-1} + a/x_{n-1}\right)$, as initial approach to use value $x_0 = (1 + a)/2$.

B12. Calculate $y(n, k) = 1^k + 2^k + \ldots + n^k$.

B13. Calculate
$$y(n) = \cfrac{1}{n + \cfrac{1}{(n-1) + \cfrac{1}{(n-2) + \cfrac{1}{\ddots \, \ldots + \cfrac{1}{1 + \frac{1}{2}}}}}}.$$

B14. Count the number of digits in the set number.

B15. Calculate
$$y(n) = \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{\ddots \, \ldots + \cfrac{1}{(n-1) + \frac{1}{n}}}}}}.$$

11. Arrays Sorting

A. *Enter the array from n of integer numbers.* Sort numbers in non-decreasing order using the specified method. Display the result on the screen.

A1. Bubble sort method.

A2. Sheykerny sorting.

A3. Sorting by the choice.

A4. Sorting by the insert.
A5. Shell method.
A6. Bubble sort method.
A7. Cocktail shaker sort.
A8. Sorting by the choice.
A9. Sorting by the insert.
A10. Shell method.
A11. Bubble sort method.
A12. Sheykerny sorting.
A13. Sorting by the choice.
A14. Sorting by the insert.
A15. Shell method.

B. *Supplement the program written during the laboratory work № 9 with the functions of the structures array ordering in non-decreasing order of the given key. Display the result.*

B1. Key: student’s year of birth. Sorting methods: QuickSort and sorting by the choice.

B2. Key: year of revenues to work of the employee. Sorting methods: QuickSort and sorting by the insert.

B3. Key: phone number of the subscriber. Sorting methods: QuickSort and method of the Shell.

B4. Key: year of release of the car. Sorting methods: QuickSort and sorting by the choice.

B5. Key: year of formation of the state. Sorting methods: QuickSort and sorting by the insert.

B6. Key: flight number of the bus. Sorting methods: QuickSort and method of the Shell.

B7. Key: the number of pages in the book. Sorting methods: QuickSort and sorting by the choice.

B8. Key: height of the athlete. Sorting methods: QuickSort and sorting by the insert.

B9. Key: departure time of the train. Sorting methods: QuickSort and method of the Shell.

B10. Key: goods price. Sorting methods: QuickSort and sorting by the choice.

B11. Key: the baggage weight of the passenger. Sorting methods: QuickSort and sorting by the insert.

B12. Key: number of points scored participant of the Olympic Games. Sorting methods: QuickSort and method of the Shell.

B13. Key: number of month of harvesting. Sorting methods: QuickSort and sorting by the choice.

B14. Key: student’s year of birth. Sorting methods: QuickSort and sorting by the insert.

B15. Key: car engine displacement. Sorting methods: QuickSort and method of the Shell.

12. Search by Key in One-dimensional Array

A. *The array of integer numbers sorted by nondecrease is set. Display the item number with the set key or information that there is no such element in the array. Search of the message by the specified method.*

A1. Search method: linear. Key: 70.
A2. Search method: binary. Key: 17.
A3. Search method: linear with the barrier. Key: 2.
A4. Search method: binary. Key: 84.
A5. Search method: linear. Key: 12.
A6. Search method: binary. Key: 25.
A7. Search method: linear with the barrier. Key: 44.
A8. Search method: binary. Key: 74.
A9. Search method: linear. Key: 41.
A10. Search method: binary. Key: 7.
A11. Search method: linear with the barrier. Key: 28.
A12. Search method: binary. Key: 82.
A13. Search method: linear. Key: 93.
A14. Search method: binary. Key: 27.
A15. Search method: linear with the barrier. Key: 31.

B. *Add the program written at execution of laboratory work № 10 as functions of the elements search on the key in the structures array. Find the element with the specified search method set by the key* (*for simplification it is supposed that at the array there are no more than one such element*). *If the element is not found, then display the corresponding message.*

B1. Display the surname of the student who was born in 1980. Search methods: linear with the barrier and binary.

B2. Display the surname of the employee who was employed in 1999. Search method: interpolation.

B3. Display the surname of the subscriber on whom it is registered phone number 7972474. Search methods: linear and binary.

B4. Display the maximum speed of the car released in 1996. Search methods: linear with the barrier and binary.

B5. Display the name of the state formed in 1927. Search method: interpolation.

B6. Display the bus destination point with flight number 295. Search methods: linear with the barrier and binary.

B7. Display the title of the book in which 1575 pages. Search methods: linear and binary.

B8. Display the surname athlete who is 197 cm tall. Search method: interpolation.

B9. Display the destination point of the train which goes 11 hours. Search methods: linear with the barrier and binary.

B10. Display the description of goods with the price equal to 265,000 rub. Search methods: linear and binary.

B11. Display the passenger surname with the baggage weights equal to 58 kg. Search method: interpolation.

B12. Output the surname of the competition participant who scored 212 points. Search methods: linear with the barrier and binary.

B13. Display the name of the crop that is harvested in June (the sixth month of the year). Search methods: linear and binary.

B14. Display the average grade of the exam for a student born in 1991. Search methods: Linear Barrier and Binary.

B15. Display the car model with the engine size of 1998 cm^2. Search method: interpolation.

13. Stacks Implementation

A. *Create the stack consisting of n of integer numbers. Run the task, display the result. The dynamically allocated memory is released as the result of execution.*

A1. Find the minimum element of the stack.

A2. Find out whether there are in the stack negative numbers.

A3. Find the difference between the sum of the even and the sum of the odd stack elements.

A4. Find the product of the odd elements of the stack.

A5. Find number of the second (from top) the odd element of the stack.

A6. Find average value of all elements of the stack.

A7. Find the product of three first positive elements of the stack.

A8. Find the difference between the first and last elements of the stack.

A9. Find the sum of three last elements of the stack.

A10. Find quantity of negative elements of the stack.

A11. Find the sum of three first and the work of other elements of the stack.

A12. Check if there are numbers on the stack greater than 250.

A13. Check if there are larger quantity the negative or positive elements in the stack.

A14. Find the maximum element of the stack.

A15. Find the sum of positive elements of the stack.

B. *Create a stack of n integers. Run a task. Do not move the information part in RAM. Output the result to the screen. Free all dynamically allocated memory at the end.*

B1. Delete all odd numbers from the stack.

B2. Trade places the minimum and maximum elements of the stack.

B3. Transform the stack so that the order of the elements is reversed.

B4. Trade places the second and penultimate elements of the stack.

B5. Add the element with value 88 before each negative element.

B6. Add the element with value 77 before the penultimate element of the stack.

B7. Remove the every third element of the stack.

B8. Find the average of all stack elements. Remove all elements from the stack with a value less than the average.

B9. Delete every third element of the stack.

B10. Delete all negative numbers from the stack.

B11. Delete all elements of the stack located before the minimum element of the stack.

B12. Delete all elements located between the first and last negative elements of the stack.

B13. Add the element with value 33 after the maximum element of the stack.

B14. Trade places the first positive and penultimate negative stack elements.

B15. Delete all elements which values are in range from 0 to 10 from the stack.

14. Two-linked Lists Implementation

A. *Create the two-linked list consisting of n of integer numbers. Run the task. Do not move the RAM information part. Display result. Release all dynamically selected memory at the end.*

A1. Delete the minimum element of queue.

A2. Add between two in a row going negative elements of queue the element with value 99.

A3. Add the element with value 55 after each negative element of queue.

A4. Trade places the first and last elements of queue.

A5. Delete all elements facing the first negative element.

A6. Delete negative elements of queue.

A7. Trade places the last and maximum elements of queue.

A8. Add after each odd element of queue the element with value 0.

A9. Delete the second and penultimate elements of queue.

A10. Delete all even elements of queue.

A11. Add the element with value 77 after the first and before the last queue elements.

A12. Delete even elements of queue.

A13. Trade places the first and minimum elements of queue.

A14. Delete all elements standing after the minimum element of queue.

A15. Delete the maximum element of queue.

B. *Perform the task according to option. Not to move information part in random access memory. Display result. Release all dynamically selected memory at the end.*

B1. Create the two-linked list consisting of *n* of integer numbers. Retrieve from the first list and move all negative numbers to the second list.

B2. Create the two-linked list consisting of *n* of integer numbers. Remove from the list all elements between the maximum element and the minimum element.

B3. Create the two-linked list consisting of *n* of integer numbers. Move the elements repeating in the first list more than once to the second list.

B4. Create the two-linked list consisting of *n* of integer numbers. Convert it to two lists: the first list should contain the even numbers only, the second should contain the odd numbers.

B5. Create the two-linked list consisting of *n* of real numbers. Sort the elements of a list in reverse order.

B6. Create two doubly linked lists of n integers in non-decreasing order. Move all data to the third list, removing duplicate values.

B7. Create the two-linked list consisting of *n* of integer numbers. Move the elements which are between the minimum and maximum elements of the first list to the second list.

B8. Create two doubly linked lists of n integers in non-decreasing order. Move to the third list elements with values which meet both in the first and in the second lists.

B9. Create the two-linked list consisting of *n* of integer numbers. Remove negative elements and move the even ones to the second list.

B10. Create the two-linked list consisting of *n* of characters of the Latin alphabet and characters of arithmetic operations. Move characters of arithmetic operations to the second list.

B11. Create two two-linked lists consisting of *n* of characters of the Latin alphabet. Move all data to the third list so that lowercase characters are on the left side of the list and uppercase characters are on the right side.

B12. Create the two-linked list consisting of *n* of integer numbers. Move to the second list elements with values greater than the average value of the first list elements.

B13. Create the two-linked list consisting of *n* of characters of the Latin alphabet. Delete from the list elements with the values repeating more than once.

B14. Create two doubly linked lists of n integers in non-decreasing order. Convert them to a third list ordered non-ascending.

B15. Create the two-linked list consisting of *n* of characters of the Latin alphabet. Convert it to two lists: the first list must contain uppercase characters, the second list must contain lowercase.

15. Tree Data Structures

A. *Create the balanced tree of search consisting of integer numbers. Display information, using the direct, return and symmetric bypass of the tree. Run the task, display the result. Release all dynamically selected memory at the end.*

A1. Find quantity of leaves of the tree.

A2. Find quantity of the nodes having only one descendant at the left.

A3. Display values of the nodes which are sheets of the tree.

A4. Find the number of sheets in the right subtree.

A5. Define tree degree.

A6. Define quantity of the nodes of the tree having depth equal 3.

A7. Find quantity of the nodes having one descendant.

A8. Find the sum of values of internal nodes of the tree.

A9. Display values of the nodes having only one descendant on the right.

A10. Find maximum (on value) the element in the left subtree.

A11. Find quantity of nodes in the left subtree.
A12. Find minimum (on value) the element in the right subtree.
A13. Find the sum of values of nodes with powers greater than 2.
A14. Find quantity of internal nodes of the tree.
A15. Determine tree depth.

B. *Create the balanced tree of search consisting of integer numbers. Display information, using the direct, return and symmetric bypass of the tree. Run the task, display the result. Release all dynamically selected memory at the end.*
B1. Swap nodes with the minimum and maximum keys in the left subtree.
B2. Swap the node with the maximum key and the root of the tree.
B3. Find quantity of leaves at each level of the tree.
B4. Delete all nodes having negative keys in the tree.
B5. Swap nodes with the minimum and maximum keys.
B6. Delete all nodes having only one descendant at the left in the tree.
B7. Delete all tree nodes with the key value greater than 5.
B8. Remove the node with the minimum key value and all its descendants from the left branch of the tree.
B9. Remove all tree nodes with key value equal to 25.
B10. Find node with value closest to the average value of all tree keys.
B11. Remove the node with the minimum key value and all its descendants from the right branch of the tree.
B12. Delete all nodes having only one descendant on the right in the tree.
B13. Delete all nodes having even keys in the tree.
B14. Delete from the tree a branch with a node with a given key.
B15. Delete from the tree the node with the set key.

16. Algebraic Expressions Calculation

A. *Enter the specified arithmetic expression and the necessary data. Transform record of arithmetic expression to the form of reverse Polish notation. Calculate the arithmetic expression. Display the result. The task is selected according to the variant number.*

A1. $a \cdot b \cdot (c-d)+f$.
A2. $a-b/c \cdot d/f$.
A3. $a+b \cdot (c-d)/f$.
A4. $a \cdot b-c \cdot (d+f)$.
A5. $(a-b) \cdot (c-d) \cdot f$.
A6. $(a+b)/(c-d) \cdot f$.
A7. $(a+b+c) \cdot d-f$.
A8. $a \cdot (b/c-d/f)$.
A9. $a \cdot (b+c-d)/f$.

A10. $a-b\cdot(c-d+f)$.

A11. $a/(b\cdot c-d)+f$.

A12. $(a-b)/(c-d)\cdot f$.

A13. $a/b+c/d-f$.

A14. $(a+b-c)\cdot d/f$.

A15. $a\cdot b/(c-d+f)$.

B. *Enter the specified arithmetic expression and the necessary data. Transform record of arithmetic expression to the form of reverse Polish notation (use the ^ sign to denote the exponentiation operation). Calculate the arithmetic expression. Display the result. The task is selected according to the variant number.*

B1. $(x-y)^w\cdot\dfrac{c+k}{f-k}$.

B2. $\dfrac{f+s}{f-y}+\dfrac{y+s^w}{x-s}$.

B3. $a\cdot x^w-\dfrac{b+x}{y}$.

B4. $x\cdot\dfrac{y+a}{y+b^w}-c$.

B5. $\dfrac{x^w}{x-y}\cdot s+b^2$.

B6. $\dfrac{c+k\cdot s}{f-k\cdot s^w}+a$.

B7. $b-s\cdot\dfrac{x}{x^w+y^w}$.

B8. $\dfrac{c^w\cdot d^w}{k^w\cdot(k+c)}$.

B9. $x^w-y^w+\dfrac{a+y}{a-x}$.

B10. $x-y\cdot\dfrac{(x+y)^w}{x+k}$.

B11. $(a-b)^w\cdot\dfrac{x^w}{x^w+y}$.

B12. $\dfrac{x-c}{c+y^w}\cdot x-y$.

B13. $\dfrac{x}{f-k^w}+xy-c$.

B14. $ax^w+(cy-a)\cdot y$.

B15. $a^w\cdot\dfrac{x+k^w}{y-k}+s$.

17. Hashing Implementation

A. *Enter the array from n of integer numbers from the set range. Create the hash table from M of elements. Perform search of the element in the hash table. Display the initial array, the hash table and search result. The task is selected according to the variant number in the tab. IV.*

Table IV

Number option	*n*	Range of values	*M*	Scheme of hashing
A1	12	23000–45000	15	With linear addressing
A2	8	53000–78000	10	On the basis of linked lists
A3	15	12000–34000	20	With linear addressing
A4	9	11000–53000	10	On the basis of linked lists
A5	16	45000–76000	20	With linear addressing
A6	12	24000–54000	10	On the basis of linked lists
A7	8	32000–68000	10	With linear addressing
A8	14	26000–77000	10	On the basis of linked lists
A9	9	38000–58000	15	With linear addressing
A10	11	24000–79000	10	On the basis of linked lists
A11	12	27000–58000	15	With linear addressing
A12	7	47000–89000	10	On the basis of linked lists
A13	11	44000–73000	15	With linear addressing
A14	9	39000–76000	10	On the basis of linked lists
A15	12	23000–58000	15	With linear addressing

B. *Announce and enter the array of structures from n of elements. Create the hash table from M of elements. Perform search of the element in the key in the hash table. Display the initial array, the hash table and all fields of the found structure. Select the task in accordance with the option number in the tab. V.*

Table V

Number option	*n*	Fields of structure	Key field	M	Scheme of hashing
1	2	3	4	5	6
B1	6	Surname, number of group, assessment	Assessment	15	From square addressing

1	2	3	4	5	6
B2	8	Make of the car, maximum speed, year of release	Year of release	10	From any addressing
B3	7	Surname, number of group, year of birth	Birth year	20	With double hashing
A4	9	Surname, phone number, address	Number phone	10	On the basis linked lists
B5	9	Title of the book, number of pages, year of the edition	Number of pages	20	From square addressing
B6	7	Description of goods, price, day of release	Price	10	From any addressing
B7	8	Destination point, flight number, departure time	Flight number	10	With double hashing
B8	6	Surname, weight, height	Weight	10	On the basis linked lists
B9	9	Surname, quantity of points, the taken place	Quantity points	15	From square addressing
B10	8	Surname, weight, height	Height	10	From any addressing
B11	7	Surname, quantity of points, the taken place	The taken place	15	With double hashing
B12	7	Destination point, flight number, departure time	Departure time	10	On the basis linked lists
B13	8	Description of goods, price, day of release	Day of release	15	From square addressing
B14	9	Title of the book, number of pages, year of the edition	Year of the edition	10	From any addressing
B15	8	Make of the car, maximum speed, year of release	Maximum speed	15	With double hashing

APPLICATIONS

1. Console Mode of the Visual C++ 6.0 Environment

A program created in the Visual C++ environment is always a separate project. Project **(project)** - a set of interrelated source files for solving a specific problem. The project includes both files created by the programmer and files automatically created and edited by the programming environment.

To create **a new project** you need:

– choose **File – New**;

– in the window that opens, on the Projects tab, select the **Win32 Console Application project type**;

– in the **Project Name** field, enter the project name, for example, maylab1;

– in the Location field, enter the name of the directory where the project will be located and the full path to it, for example, **D:\WORK\mylab1**. The directory can also be selected using the Choose Directory dialog by clicking on the … button;

– specify the type of project being created is **Win32 Console Application**;

– click on the **OK** button;

– in the opened window of the application wizard **Win32 Console Application – Step 1 of 1** select **An empty project** (empty project) and click on the Finish button;

– in the **New Project Information** window that opens, click the **OK** button.

To work with a console application, you must create a **new file** or add an existing file with the program text.

To create **a new file** you need:

– choose **File – New**;

– in the window that opens, on the Files tab, select the C++ Source File file type;

– in the **File name field**: enter the **file name**. For convenience, it is desirable to enter a name that matches the name of the project, for example, maylab1;

– click on the **OK** button.

To add a file with the text of the program to the project, you must:

– copy the existing **file (cpp extension**) to the working folder of the project;

– in the Workspace window**, FileView tab**, right-click on the Source Files folder;

– in the opened Insert Files... dialog box, select the file to be added and click the OK button.

A project folder typically contains five files and one subfolder. The files have the following purpose.

A file with the **dsw extension** (for example, mylab1.dsw) is a project file that combines all the files included in the project.

A file with the dsp extension (for **example mylab1.dsp**) is intended for building a separate project or subproject.

The file with the **opt extension** (for example **mylab1.opt**) contains all the settings for this project.

A file with the **ncb extension** (for example, mylab1.ncb) is a service file.

A file with the **cpp extension** (eg mylab1.cpp) is a program text file.

2. Program Execution

For compilation, configuration and start of the program on execution the following items of the Build submenu are used:

1. **Compile (Ctrl + F7)** is compilation of the selected file. Results of compilation are displayed in the Output window.

2. **Build (F7)** is configuration of the project. All files that have changed since the last link are compiled. If there are no linking errors, the programming environment creates an executable file with the extension **.exe**. It is running.

3. **Rebuild All** is rearrangement of the project. All project files are compiled regardless of changes.

4. **Execute (Ctrl + F5)** is execution of the executable file created as a result of configuration of the project. Modified files are recompiled and relinked.

The appropriate messages about detected syntax errors are displayed. In this case it is necessary to correct consistently errors and to compile the project again.

After completion of work the project can be closed, having selected **the File** – **to Close the solution** or to close the MVC application ++.

For opening of the project saved earlier it is necessary to select **the File** – **to open the project or the solution**.

3. Program Debugging

If there are no syntax program errors, but result of program execution is incorrect, it is necessary to look for logical errors. For search of logical errors the built-in debugger is used.

For step-by-step program execution it is necessary to key **F10**. By each clicking the current line is executed. If it is necessary to check step by step the text of the caused function, then it is necessary to click **F11**. For the early output from function click **Shift + F11**. If you need to debug from a certain place in the program, then the cursor is placed in the corresponding line of the program and the key combination **Ctrl + F10** is pressed.

Another way to debug is to set *the program breakpoints*. To do this, place the cursor on the desired line and press **F9**. The breakpoint is indicated by a red circle on a special field located to the left of the program text window. To delete a breakpoint, press **F9** again in the required line. The quantity of points of interruption in the program can be any.

For program execution to the point of interruption it is necessary to click **F5**. For continuation of debugging **F5** is keyed (for program execution to the following point of interruption) or keys for step-by-step debugging are used.

The yellow arrow in the field to the left of the window of the text of the program indicates the line which will be executed on the following step of debugging.

To control the values of variables the following method is used: move the mouse pointer to this variable and hold it for a few seconds. A window with the current value of this variable will appear on the screen next to the variable name. These variable values will be displayed in the windows below. The lower left window displays the values of the last variables used by the program. In the lower right window (Watch) you can set the names of the variables whose values you want to control.

References

1. Microsoft [Electronic resource]. – 2022. – Access mode : https://docs.microsoft.com/en-us/cpp/cpp/?view=msvc-160.

2. Lafore, R. Object-Oriented Programming in C++. Fourth Edition // R. Lafore. – Indianapolis, Indiana : SAMS, 2002. – 1012 p.

3. Stroustrup, B. The C++ Programming Language. Fourth Edition / B. Stroustrup. – Boston : Addison-Wesley, 2013. – 1368 p.

4. Stroustrup, B. Programming – Principles and Practice Using C++ / B. Stroustrup. – Boston : Addison-Wesley, 2014. – 1312 p.

5. Kernighan, B. C Programming Language / B. Kernighan, D. Ritchie. – New Jersey : Prentice Hall, 1988. – 274 p.

6. Josuttis, N. C++ Standard Library. Tutorial and Reference / N. Josuttis. – Boston : Pearson, 2012. – 1198 p.

7. Stroustrup, B. A Tour of C++ Second Edition / B. Stroustrup. – Boston : Pearson Education, 2018. – 255 p.

8. Filipek, B. C++17 in detail / B. Filipek. – Victoria : Leanpub, 2019. – 299 p.

9. Guntheroth, K. Optimized C++: Proven Techniques for Heightened Performance / K. Guntheroth. – Sebastopol : O'Reilly Media, 2016. – 387 p.

10. Lippman, C. C++ Primer / S. Lippman, J. LaJoie, B. Moo. – Boston : Addison-Wesley, 2012. – 963 p.

Св. план 2021, поз. 10

Учебное издание

Навроцкий Анатолий Александрович
Гуринович Алевтина Борисовна

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

ALGORITHMS AND DATA STRUCTURES

УЧЕБНОЕ ПОСОБИЕ

Редактор *Е. С. Юрец*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 30.09.2022. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 8,95. Уч.-изд. л. 9,5. Тираж 50 экз. Заказ 152.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск