

Alytis Gruodis
Sergėjus Ivanikovas

C++ PROGRAMMING
FROM BASICS

Textbook

Vilnius 2012

UDK 004.42

Gr-675 Gruodis, Alytis

C++ Programming from Basics / Alytis Gruodis, Sergėjus Ivanikovas. - Vilnius: Vilniaus verslo kolegija, 2012. - 296 p.
(Textbook)
ISBN 978-609-95431-0-9

Textbook is aimed for students and all computer-mindeds who would like to be familiarize to C++ programming. Course basics from zero as well as applied console and windows solutions are presented and discussed for educational purposes.

This textbook was prepared using $\text{\LaTeX}$.
This PDF was generated by *TeXworks*, version 0.4.3 r.857 (MiKTeX),
distributed under the GNU General Public License, version 2.
which is freely available under an open source license from <http://miktex.org/>.



Mokomoji knyga parengta ir išleista projekto
Vilniaus verslo kolegijos studijų programų tarptautiškumo didinimas
lėšomis. Projekto Nr.: VP1-2.2-SMM-07-K-02-041.
Projektą iš dalies finansuoja Europos socialinis fondas.

Approbation -	Recommended for publishing by Department of Information Technologies (Vilnius Business College), N2012-005
Reviewers -	Assoc. Prof. Dr. Romanas Tumasonis Lect. Violeta Jadzgevičienė
Language editors -	Giedrė Būdienė (Chapter 1, 3-4) Jelena Ivanikova (Chapter 2, 5)
Typesetter -	Alytis Gruodis
Publisher -	Vilniaus verslo kolegija
Cover design & printing -	UAB <i>ART SET</i> , Vilnius

ISBN 978-609-95431-0-9

© Alytis Gruodis, 2012.
© Sergėjus Ivanikovas, 2012.
© Vilniaus verslo kolegija, 2012.

About the authors

Alytis Gruodis (born 1963, PhD 1995)

Associated Professor at Vilnius university, Department of General Physics and Spectroscopy, and also lecturer at Vilnius Business College, Department of Information Technologies.

Main scientific interest includes molecular modelling by means of quantum chemistry methods. Coauthor of 40 scientific publications.

Sergėjus Ivanikovas (born 1981, PhD 2010)

Lecturer at Lithuanian University of Educational Sciences, Department of Mathematics and Informatics and also lecturer at Vilnius Business College, Department of Information Technologies.

Main scientific interest includes multidimensional function optimization, parallel computing, application of information technologies in education. Coauthor of 11 scientific publications.

About textbook

Computer programming is one of the exciting work and leisure time activities, carried out by many enthusiasts. This is a hobby, formed from the school time, for many programmers. With the development of computers and operating systems, many programmers are familiarized with the several generations of computer hardware and software as well as variety of programming techniques.

A question about the choice of programming languages belongs to the field of "topic" questions in the teaching processes. In order to manage a programming language, it is necessary to learn the language syntax and expressions, to develop the ability to think algorithmically and to use the modern programming paradigms.

We have gained a lot of teaching experience in programming over the last decade at universities and colleges. We have become real enthusiasts in C++ programming. Even though like any programmers, we have an experience from the past to program using *Fortran* and *Pascal*, but we appreciate that *C++* as the best due to user-friendly expressions. Market analysis and choice of developers indicate that nowadays the most popular languages are languages created in C style only.

We have decided to write the textbook which could be useful for students as well as for computer-minded audience who would like to be familiarized with C++ programming. This textbook contains five chapters: the first - *Preface and history* (written by AG and SI), the second - *C++ basics* and the fifth - *Graphical programming using wxWidgets* (both written by SI), the third - *Algorithms* and the fourth - *Graphical programming using windows.h* (both written by AG).

The first chapter - *Preface and history* is devoted to describe the behaviour of programming language, main programming paradigms, family of C programming language and programming environment used for samples in this textbook.

The second chapter - *C++ basics* contains a C++ language tutorial. The main concepts and control structures of C++ programming language are presented here.

The third chapter - *Algorithms* is devoted to describe the practical tasks how to use 1D, 2D, 3D, etc, dynamic arrays, how to implement it into the matrix manipulation. Several popular sorting algorithms are presented and discussed. The three types of containers from STL are presented in typical situations. Finally, several simplified applied algorithms (prime numbers, random numbers, Vernam encryption) are presented for analysing purposes.

The fourth chapter - *Graphical programming using windows.h* is devoted to observe the properties of *Windows* graphics. The creation of windows, organization of menu bar and drawing of graphical primitives - such algorithms are presented in order to estimate *Windows* graphics possibilities, when any IDE platform is necessary.

The fifth chapter - *Graphical programming using wxWidgets* observes the possibilities of using *wxWidgets* to create a graphical programs. *wxWidgets* projects are presented and the

most common controls have been reviewed.

For advanced readers, an additional literature is recommended. First of all, textbooks for beginners written by famous programmers Liberty [24], [25], Lippman [26], [21], [27], Prata [36], Schildt [40], [42], [43], could be very useful in order to recognize the language syntax and to form good programming style. Secondly, improving of own programming code through algorithm optimization is always acceptable due to protection of computer resources. Several publication of Meyers [30], [31], Eckel [12], [4] Deitel [10], Qualline [37] are significant for programmers who would like to understand the best world programming tendencies. And finally, several publications could be recommended as specialized textbooks for applied scientific programming - written by Cheng [7], Bertly [6], Bulka [28], for solving of debugging problems - written by Pappas [8].

To make this textbook easy to read, we have used several fonts for different text types.

WinMain()	programming code in typewriter style
<i>Matrices</i>	term first time, in italic style
<i>LINUX</i>	proper name, in italic style

All algorithms and portions of source code, presented here, are tested by authors using non-commercial software *wxDev-C++ 7.4.2*. This IDE [1] can be downloaded from the internet <http://wxdsgn.sourceforge.net/>. All code presented here can be used without any restrictions anywhere "as it is". Authors assume no liability for hardware and/or software failures, and also for any financial disadvantages using presented code material.

This book was carefully produced including an additional support material containing source code. Nevertheless, editors and authors do not warrant the information contained therein to be free of errors. Readers are advised to keep in mind that all programming items may inadvertently be inaccurate. For communication to authors, you are wellcome to use the following addresses: Alytis.Gruodis@gmail.com or Ivanikovas@gmail.com.

Enjoy C++ !

Vilnius, May 1, 2012

Alytis Gruodis
Sergėjus Ivanikovas

Abbreviations

1D	one-dimensional, depending on one parameter
2D	two-dimensional, depending on two parameters
3D	tree-dimensional, depending on three parameters
ANSI	American National Standards Institute
AOP	Aspect-Oriented Programming, paradigm of programming
Big-O	Complexity notation
CLI	Common Language Infrastructure
DC	Descriptor
DEQUE	Double-Ended QUEue
FAT	File Allocation Table
FIFO	First In, First Out
GPL	General Public License
GUI	Graphical User Interface
I/O	Input and Output
IDE	Integrated Development Environment
LIFO	Last In, First Out
ND	N-dimensional, depending on N parameters
NT	New Technology (File system)
OOP	Object-Oriented Programming, paradigm of programming
OS	Operating System
RC	Resource Script
RGB	Red Green Blue, colour coding
RNG	Random Number Generator
STL	Standard Template Library

Contents

1	Preface and history	9
1.1	Programming paradigms	9
1.2	Family of C programming languages	12
1.3	Programming environment	13
2	C++ basics	17
2.1	The structure of program	17
2.2	Variables and data types	19
2.3	Constants	21
2.4	Basic input and output	22
2.5	Operators	28
2.6	Macros	30
2.7	Control structures	32
2.8	Functions	39
2.9	Global, local and static variables	46
2.10	Arrays	49
2.11	Character sequences	53
2.12	Pointers	55
2.13	Data structures	60
2.14	Classes	66
2.15	Friendship and inheritance	71
2.16	Polymorphism	75
2.17	Input/Output with files	77
3	Algorithms	87
3.1	Dynamical arrays	87
3.1.1	One-dimensional (1D) array	88
3.1.2	Two-dimensional (2D) array	91
3.1.3	Three-dimensional (3D) array	94
3.2	Matrices	97
3.2.1	Definition and notation	98
3.2.2	Matrix manipulation: OOP in first style	100
3.2.3	Matrix manipulation: OOP in second style	112
3.3	Containers	121
3.3.1	Complexity	122
3.3.2	Sequence containers	122

3.3.3	Associative containers	132
3.3.4	Container adaptors	143
3.4	Sorting	148
3.4.1	Exchanging	151
3.4.2	Selection	161
3.4.3	Insertion	170
3.4.4	Partitioning	177
3.4.5	Merging	179
3.5	Applied	181
3.5.1	Prime numbers	181
3.5.2	Random numbers	185
3.5.3	Encryption	191
4	Graphical programming using <i>windows.h</i>	197
4.1	Graphical interface	198
4.1.1	Window formation	198
4.1.2	Dialog and manipulator operations	205
4.2	Figure drawing	215
5	Graphical programming using <i>wxWidgets</i>	235
5.1	Graphic objects. Operating by events	235
5.2	Creation of graphical project	237
5.3	Creation of the first program	240
5.4	Basic control. Input/output of text	242
5.5	Properties	246
5.5.1	The geometrical properties	246
5.5.2	The properties of colours	248
5.5.3	Some most often used properties	249
5.6	Timer and scrollbars	252
5.7	Selection controls	263
5.8	Keyboard events	264
5.9	Mouse events	267
5.10	Graphical controls and methods	270
5.11	Menu, dialog boxes	276

Chapter 1

Preface and history

On the choice
of the correct language -

I speak Spanish to God,
Italian to women,
French to men,
And German to my horse.

Charles V ^a

This textbook is designed to teach the beginners without any previous programming experience. The university or college students will receive the knowledges and skills how to solve regular practical tasks in console or in graphical window - using C/C++ programming language.

1.1 Programming paradigms

Programming language. An artificial language containing instructions how to communicate to computer, particularly computer system, is called the *programming language*. Optimal design of programming language has been under question about 60 years due to psychological human aspects. Programming language is devoted for programmers, who need to create abstract algorithms and/or electronic equipment controlling programs. Therefore the programming language has to be user-friendly, algorithmic, constructed using items of popular human language, for example English.

Most programming languages describe sequence of commands in an *imperative* style, although some languages, these that support functional programming, use alternative forms of description in *non-imperative* style.

Usage of existing programming languages as well as constructing of new programming languages depends on many sophisticated factors belonging to the current era:

^aCharles V (1500 - 1558) - ruler of the Holy Roman Empire from 1519.

- i) facilities of hardware (amount of memory, number of processors and registers, quantity and quality of servers, etc);
- ii) practical tasks to be solved (arithmetical, physical or data-mining);
- iii) education of programmers (mathematical or informatics);
- iv) dissatisfaction of programmers (in language expressions, style and hardware possibility);
- v) dominant programming styles and substyles in the market (OOP or procedural);
- vi) outlook for the future times (to be improved... something).

Which programming language is the best? The question belongs to the field of incorrect questions. A universal language should be *eclectic*. It means, that many user-friendly expressions and paradigms should be included in renewed version or in a new dialect. Big number of pragmatic and social factors determine the language usage. Significant factors, which are described below, could prelimit the usage of one or several languages as well as language dialects:

- a) language prevalence in the world and particularly in the local place;
- b) attitude of boss;
- c) possibility to use modern tools;
- d) previous *run-in*, etc.

The purpose of programming languages could be described in several positions. Firstly, programming language is the powerful tool for task and management programming: possibility to describe the algorithms and to design the high-level projects. Secondly, programming language is some sort of "communication language" between programmers because it presents possibility of conceptual representation of the relationship (paradigms).

Fig. 1.1 represents schematic behaviour of programming as a human cultural phenomena. First of all, urbanistic society must be ready to solve some complicated mathematical/arithmetical tasks, and/or public demand for information management must occur. Secondly, all types of practical tasks must be expressed in abstracted form such a mind (logical sequence of ideas). Thirdly, abstracted form of task (structured algorithm) must be presented graphically as a chart. Programming languages occur in this stage of development in order to make an algorithmisation process easier and errorless.

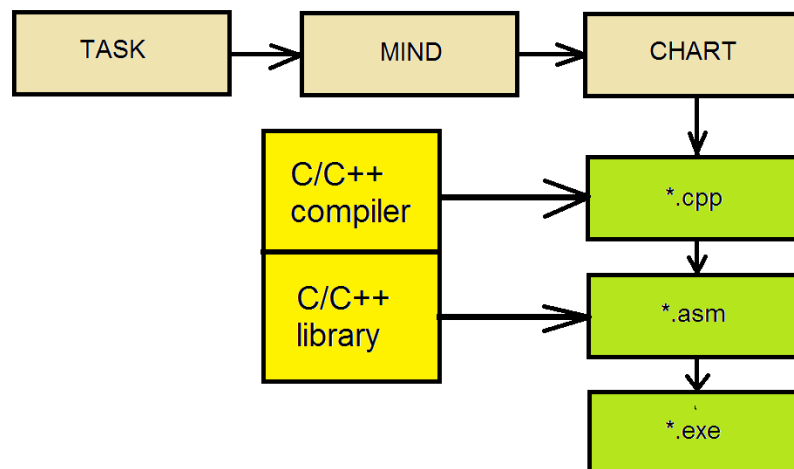


Fig. 1.1. Programming as a cultural phenomena.

C/C++ compiler converts source code (*.cpp), containing some sophisticated expression in human readable form, into an object containing Assembly code (*.asm). C/C++ linker converts *.asm file into executable *.exe by adding the libraries and standard templates. Several significant paradigms of programming - procedural, structured, object-oriented, aspect-oriented, generic - will be observed below.

Procedural (imperative) programming. Program consist of a set of instructions. Each operator prescribes to processor the work routine. Paradigm of *procedural (imperative) programming* was established for *Assembler*, and also early *Fortran* versions. It was quite useful for short programs. Generally, all programs of such type are of *single-entry/single-exit* type.

Structured programming. Program consists of a blocks (functions, subroutines, libraries) and block-files, containing structurized data. Paradigm of *structured programming* was established for *Pascal*, *C*, *Fortran*, *Basic*. It was useful for a quite long programs (for example, until fifty thousand lines of code). It is easy to read, understand, test and modify the programming code. Each action could be changed by set of two to three action. Simple procedure scheme could be inserted. For structural programming see Berry publication [29].

Object-oriented programming, OOP. Program consist of a real objects, constructed from abstract classes using templates. Classes contain the data of different types associated to the class-functions or methods. Encapsulation, inheritance, polimorfism allow to manipulate the objects associated by different degree of dependence. The essence of OOP is to create a real-world image in the structure of the program code. *Object-oriented programming, OOP* is a programming paradigm, when the program is modelled as a collection of objects associated with each other.

Paradigm of OOP was constructed in 1966 by Dhal ^b ir Nygaard ^c. Paradigm of OOP was established for *C++*, *C#*, *Java*, *Visual Basic*. It was very useful for a very long programs (for example, until million lines of code). For OOP see Coplien publication [9].

Aspect-oriented programming, AOP. *Aspect-oriented programming, AOP* is a programming paradigm which aims to increase program modularity by allowing the separation of *cross-cutting* concerns. AOP allows us the ability to apply aspects that alter the behaviour to classes or objects - independent of any inheritance hierarchy. We can then apply these aspects either during runtime or compile time. Popularity of AOP increases due to activity of the *Microsoft .NET* development community.

Generic programming, GP. *Generic programming, GP* is a modern style of programming when all algorithm or part is presented in non-specified items or terms. Such terms must be specified with initiation for specific types provided as parameters before its usage. Such software entities are known as generics in *Eiffel*, *Java*, *C#*, etc., and also in templates of *C++* and *D*. For generic programming see Czarnecki publication [13].

^bOle-Johan Dahl (1931 - 2002) - Norwegian computer scientist, Professor of Oslo university. He is known by pioneering concepts of class, inheritance, dynamic object, devoted for *SIMULA* programming language.

^cKristen Nygaard (1926 - 2002) - Norwegian computer scientist, Professor of Aarhus and Oslo universities. He is known as programming language pioneer (initiator of object-oriented programming paradigm) and politician.

1.2 Family of C programming languages

A full list of programming languages and dialects, similar or related to *C*, includes up to hundred positions. The main positions are occupied by *C*, *C++*, *C#*.

C(1973) was developed by Ritchie^d as a general-purpose system programming language. From 1980 to 2010 *C* was titled as a programming language, one of the mostly spread in the world. *C* compilers have been created almost for all types of computers. The first widely available book on the *C* programming language is known as *K&R C* (Kernighan and Ritchie) [38] [39].

C was standartized by the American National Standards Institute (ANSI) as *ANSI C* 1986). Most part of *C* code for current working computers is of *ANSI C* type. The usage of *ANSI C* guarantees the code portability (possibility to compile correctly on any hardware platform containing *C* software)

C++ means an implementation of *C* by *Object-Oriented programming*, *OOP* paradigms. According to that, compatibility of *C* and *C++* persists. *C++* started as a preprocessor for *C* and is currently nearly a superset of *C*. *C++*, created firstly as *C with classes* (1979), assumes the standartized implementation as *C++ std* (1995) due to Stroustrup^e [46] [47] activity. *C++* (named according to the incrementing operator) is becoming very popular in the world due to the well-balanced user-friendly command system.

C# (2000) is a OOP general-purpose programming language developed by *Microsoft* within its *.NET* initiative. *C#* is one of the programming languages designed for the *Common Language Infrastructure*, *CLI*. Several joined and new type paradigms were used: generic, object-oriented (class-based), and component-oriented programming.

Many later languages were created using *C* manner, including: *D*, *Go*, *Java*, *JavaScript*, *Perl*, *PHP*, *Python*, and *Unix's C Shell*. Similar syntax (main part) as well as expressional forms could be found.

C and *C++* are useful programming languages especially for large projects. There are many opinions in order to estimate the functionality or quantity of user-friendly expression in both languages.

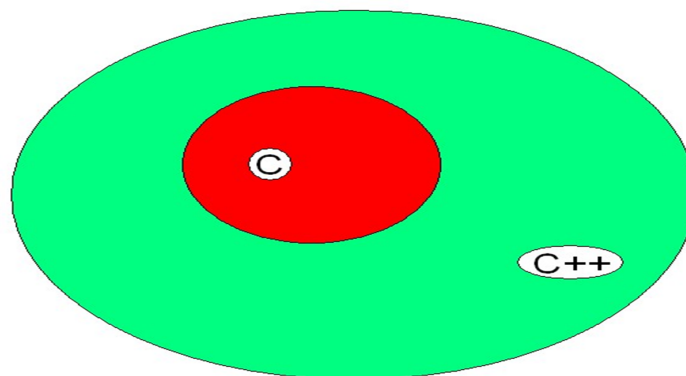


Fig. 1.2. *C++* as a superset of *C*.

^dDennis MacAlistair Ritchie (1941 - 2011) - an American computer scientist. He created the *C* programming language at *Bell Labs* between 1969 and 1973 and, with long-time colleague Ken Thompson, the *UNIX* operating system.

^eBjarne Stroustrup (born 1950) - a Danish computer scientist, mostly known for the creation and the development of the widely used *C++* programming language. Professor at Texas A&M University.

Main *C++* priorities to *C* could be formulated as follows.

1. *C++* shows about 10 % higher code productivity, in comparison to *C* - due to optimized code of libraries and usage of statements according to OOP paradigms.
2. It is easier to modify very large programs of *C++*, neither *C* - due to user-friendly expressions.
3. It is easy to use the specific libraries in *C++* due to the class existing, which is absent in *C*.
4. Class templates (present in *C++*, but absent in *C*) automatically modifies programming code.
5. *C++* programs could be very large in comparison to *C* - due to possibility to separate class method declaration from method interface.
6. Overloading of functions and operators allow to use the same names and symbols for different functions - methods.
7. Existing of different namespaces in *C++* allows to use the names without any danger of merging and allows to avoid the overlapping names, as could occur in *C*.
8. Using the references in arguments of functions or methods and returned values in *C++* is more preferable and user-friendly than pointers only in *C*.

1.3 Programming environment

Generally it is possible to write programs using text editor (for example *Notepad*) and then to use special compiler to create an executable program. But nowadays one can use integrated development environments (IDEs) to facilitate the program creating process.

An IDE is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, built automation tools and a debugger.

Usually there are several different IDEs for certain programming language. With *C++* we can use different IDEs: *C++Builder*, *Code::Blocks*, *Dev-C++*, *Microsoft Visual Studio*. Some of them are commercial, others are not. There are differences in functionality, additional features, but all of them help to create working program. *WxDev-C++* IDE was used to write code and create programs for this book.

Dev-C++ is a free IDE, developed by Laplace ^f, distributed under the *General Public License*, *GPL* for programming in *C* and *C++*. *MinGW*, a free compiler, bundled with it. It runs exclusively on *Microsoft Windows*. The *Dev-C++* project was not active from February 22, 2005 to June 2011. In 2006 forum post lead developer Colin Laplace stated, that he was busy with real-life issues and did not have time to continue development of *Dev-C++*. However, in June 30, 2011 an non-official version 4.9.9.3 of *Dev-C++* was released by an independent programmer, featuring the more recent GCC 4.5.2 compiler, Windows' SDK resources (Win32 and D3D), numerous bugfixes, and improved stability. In August 27, after five years of officially being in a beta stage, version 5.0 was released by the developer from Orwell's blog. *wxDev-C++* is also a free IDE that is based on the *Dev-C++*.

This version has additional features not found in the original *Dev-C++*. One of these is a visual RAD designer that works like *C++ Builder* to create *wxWidgets* applications. *wxDev-C++* supports both *Microsoft* and GNU *MinGW* compilers.

^fColin Laplace, programmer.

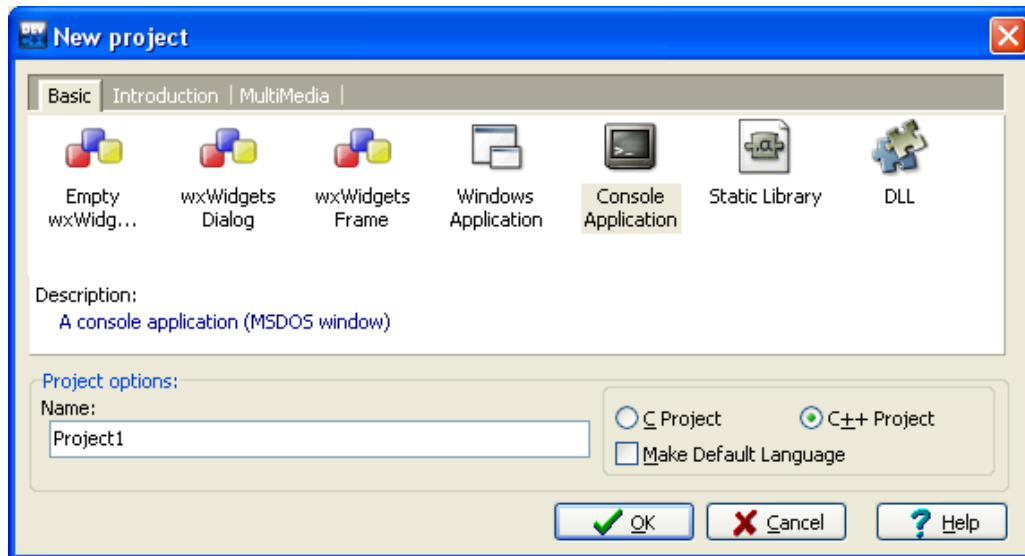
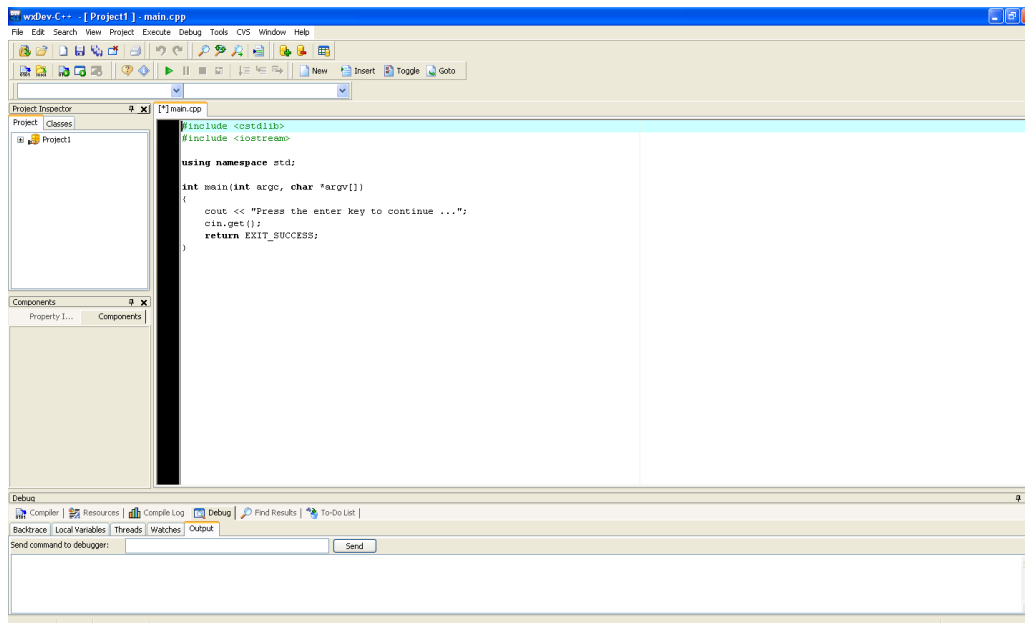


Fig. 1.3. New project menu.

Fig. 1.4. *wxDev-C++* window.

The latest version is 7.4.2, released in June 2012.

Modern IDE is a complex software system which has different functions. The scope of this book is *C++* programming. Necessary IDE functions will be also overviewed.

Program creation process is started by creating a new project in *wxDev-C++* (*File - New - Project*). Project of *Console Application* type will be used while studying *C++* basics (see Fig. 1.3). Then we will proceed with *Windows Application* and *wxWidgets* projects. This project consists of several different files and folders (project file, program code file, makefile, objects and output folders etc.). The number of files depends on project type and complexity. Project file of **.dev* extension stores the main information about project. Program file of **.cpp* extension stores code as the program text. Executable file of **.exe* extension is present in the *Output* folder or in its sub-directory.

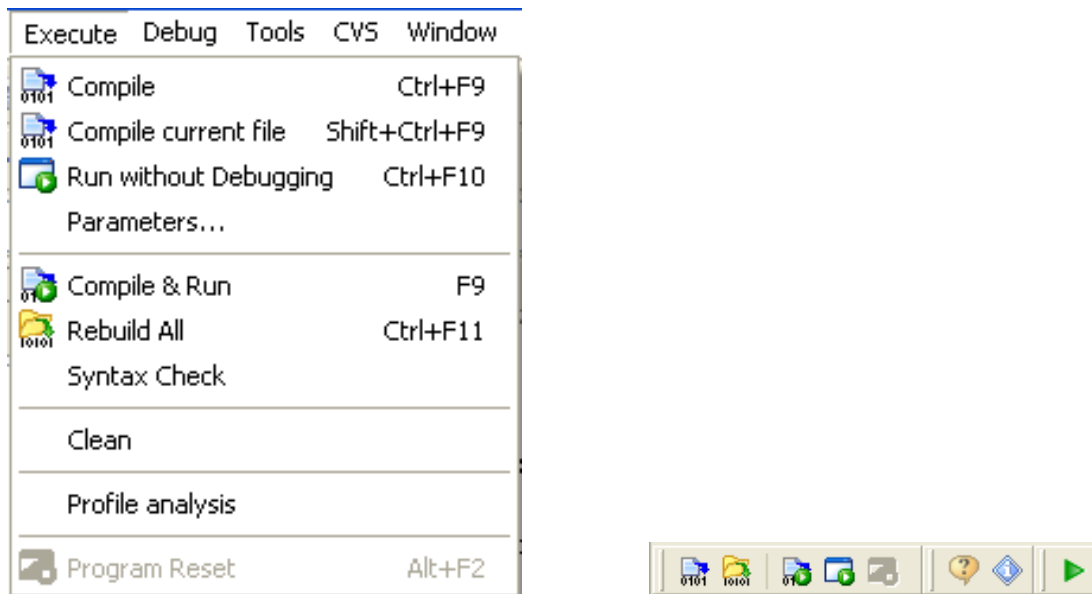


Fig. 1.5. Execute menu and corresponding command buttons.

In order to start programming one must choose a folder and name the project file and press *Save* button. The main program screen appears (see Fig. 1.4). On the top part of the window there is a main command bar, lower we can see command buttons. There is a Project inspector on the left part of the window and we see coding field (`main.cpp`) in the centre. We can see sample program text in the coding field. This program can be compiled and run immediately. Usually writing programs in C++ one must compile the program before running it. There are several ways to compile and run the program in *wxDev-C++*. The first way is to use the main menu command *Execute*, or we can use corresponding command buttons (see Fig. 1.5).

Another way is to use hot keys (*F9 Compile&Run*, *Ctrl+F9 Compile*, *Ctrl+F11 Rebuild All*, *F8 Run*, etc.). Compilation regime *Rebuild All* is more preferable to the others.

Figs. 1.6÷1.9 represent the progress of compilation.

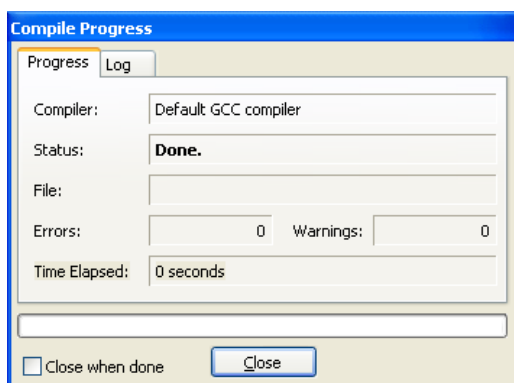


Fig. 1.6. *Compile Progress* window.



Fig. 1.7. Command button for executing - green triangle.

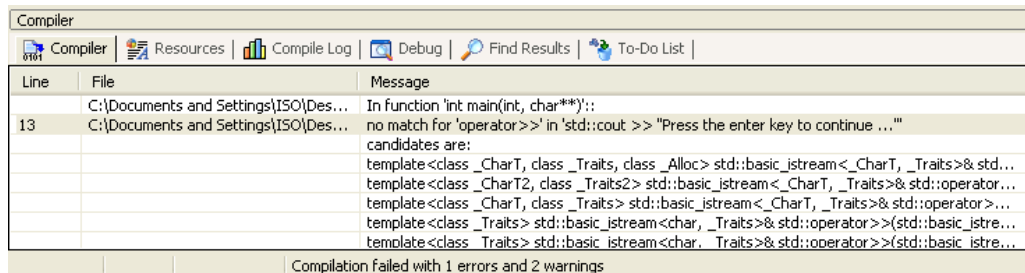


Fig. 1.8. Compiler message window.

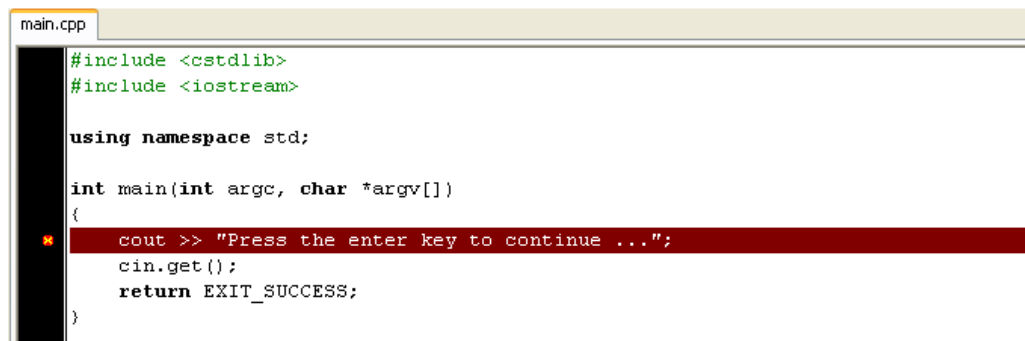


Fig. 1.9. A faulty program line.

The main program file must be saved before compiling the program for the first time (*Save File* dialog appears). Then *Compile Progress* window appears. *Done* status will appear if the program is compiled successfully (see Fig. 1.6). *Close* button should be pressed and then the program can be run.

The compiler message will appear in the lower part of the window if the compilation process fails (see Fig. 1.8). The faulty program line will be highlighted if there is a problem in program text (see Fig. 1.9). One must debug the program, fix all the errors and then recompile it. The program can be executed pressing *F8* or a command button (green triangle, Fig. 1.7.).

A message about program debugging appears before running the program as presented in Fig. 1.10. The default answer is *Yes*. This will enable the program debugging mode and recompile the project. It is possible to choose not to show the message again by putting the tick in lower left corner. The sample program will show as a console window - see Fig. 1.11.

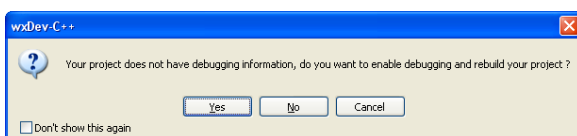


Fig. 1.10. The message about debugging.

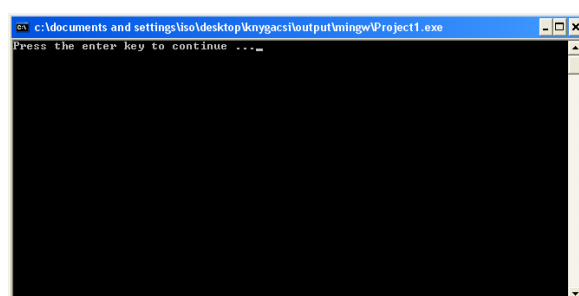


Fig. 1.11. The sample program window.

Chapter 2

C++ basics

C combines the power of Assembler
with the portability of Assembler...

Anonymous,
alluding to Bill Thacker.

2.1 The structure of program

The best way to start learning a programming language is probably by writing a program. Lets start writing our first C++ program by modifying the *wxDev-C++* sample - see Table 2.1. Table 2.2 represents output of program.

This program is one of the simplest programs, that can be written in C++, but it already contains the basic components, that every C++ program has got. Look line by line at the program code.

```
// a simple program
```

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behaviour of the program. The programmer can use them to include short explanations of the program code. Only one line (or part of line) could be commented using double slash (//). */**/* signs could be used to comment several code lines (*/** beginning of the comments, **/* the end of comments). It is important to remember, that we can not put one multiline comment into another.

```
/*This is /*a wrong type of*/ comments*/
```

```
#include <cstdlib>
```

```
#include <iostream>
```

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compilers preprocessor. In this case the directives: first `#include <cstdlib>` and second `#include <iostream>` tell the preprocessor to include the appropriate libraries. This specific files (*cstdlib* and *iostream*) include the declarations of the input-output libraries in C++. Later we will use one more directive for the preprocessor (`#define`).

Table 2.1.	Simple program.
<pre> // a simple program #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { cout << "Hello World! This is my first C++ program" << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.2.	Output of program.
<pre> Hello World! This is my first C++ program Press the enter key to continue ... </pre>	

```
using namespace std;
```

Namespaces allow to group entities like classes, objects and functions under a name. In this way the global scope can be divided in "sub-scopes", each one with its own name. All the elements of the standard C++ library are declared within the namespace called `std`. So, in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library. [45]

Global variables can be declared after the specification of namespace.

```
int main(int argc, char *argv[])
```

This line corresponds to the beginning of the definition of the `main()` function. The `main()` function is the point where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For the same reason, it is essential that all C++ programs have a `main()` function.

The word `main` is followed by expression `(int argc, char *argv[])` because it is a function declaration. In C++ each function has brackets after its name. It does not matter, weather function has any parameters or not. Parameters `argc` and `argv[]` are simply naming conventions. They have meaning **argument count** and **argument vector**. These arguments are used to parse command line parameters to a program. Argument `argc` contains the amount of arguments, separated by spaces, that were entered when the program was invoked. Argument `argv` contains the actual arguments themselves.

```
cout << "Hello World! This is my first C++ program" << endl;
```

After the name of the function and its parameters we can find the body of the function enclosed in brackets `{...}` . The information contained within these brackets is what the function does when it is executed.

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program. Name `cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the *Hello World! This is my first C++ program* sequence of characters) into the standard output stream (`cout`, which usually corresponds to the screen).

Stream `cout` is declared in the `iostream` standard file within the `std` namespace. That is why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Statement `endl` in the end of the line moves the cursor to the next line of the screen. We put semicolon in the end of each C++ program line. It is also important to remember that C++ compiler is case-sensitive. So, if one write `Cout` instead of `cout` he will have a mistake.

```
cout << "Press the enter key to continue ...";
```

This statement prints the sentence *Press the enter key to continue ...* to the program window.

```
cin.get();
```

This statement is used to stop the execution of the program (program will wait ENTER button to be pressed).

```
return EXIT_SUCCESS;
```

The return statement causes the main function to finish. Return may be followed by a return code (in our example is followed by the return code with a value of `EXIT_SUCCESS`). In general, the return code can be any integer number. This code can be used for program debugging.

2.2 Variables and data types

The variable is a PC memory area assigned to store particular data type. Using variables it is possible to store different values of the same data type in the same memory area at different times. It is important to understand what types of variables are used in C++ and how they are stored.

Each variable needs an identifier that distinguishes it from the others. A valid identifier is a sequence of one or more letters, digits or underscore characters (`_`). Neither spaces nor punctuation marks or symbols can be a part of an identifier. Only letters, digits and single underscore characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (`_`), but in some cases these may be reserved for compiler specific keywords or external identifiers. Identifier can not begin with a digit. Another important rule is that identifier name can not match any keyword of the C++

language or your compiler's specific ones, which are reserved keywords. It is very important to remember that the C++ language is a case sensitive language. It means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters.

By creating a variable programmer assigns memory area to store its value. It is important to specify what kind of data will be stored. The computer memory is organized in bytes. A byte is the minimum amount of memory that can be managed in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In order to store more complex data we must have additional data types that use groups of bytes to store the information. [45]

The main C++ variable types are following: `char`, `int`, `float`, `double`. The characteristics of each data type are presented in Table 2.3. The size and range depends on the compiler and on the system the program is compiled for. The most common values are presented.

There is also a `void` (empty) data type. It is used to create the function which does not return any value. Unsigned data type shifts the range of values. For example, if we know that we will use only positive integers, we can use the unsigned type, and the potential number range will be from 0 to 4294967295. Floating-point, i.e., real number storage in memory is different from integer numbers, because the first ones are stored in two parts: separate stored decimal and exponent value.

In order to use a variable in C++, firstly we should declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like `int`, `bool`, `float`...) followed by a valid variable identifier. For example:

```
float a;  int number;
```

These are two valid declarations of variables. The first one declares a variable of type `float` with the identifier `a`. The second declares a variable of type `int` with the identifier `number`.

Table 2.3. Characteristics of the main C++ data type.			
Name	Description	Size, B	Range for signed and unsigned
<code>char</code>	character or short integer	1	(-128 ÷ 127); (0 ÷ 255)
<code>short int</code> <code>short</code>	short integer	2	(-32768 ÷ 32767); (0 ÷ 65535)
<code>int</code>	integer	4	(-2147483648 ÷ 2147483647); (0 ÷ 4294967295)
<code>long int</code> <code>long</code>	long integer	4	(-2147483648 ÷ 2147483647); (0 ÷ 4294967295)
<code>bool</code>	boolean	1	TRUE (1) or FALSE (0)
<code>float</code>	floating point	4	± 3.4e ± 38 (7 digits)
<code>double</code>	double precision, floating point	8	± 1.7e ± 308 (15 digits)
<code>long double</code>	long double precision, floating point	8	± 1.7e ± 308 (15 digits)
<code>wchar_t</code>	wide character	2/4	one wide character

Once declared, the variables `a` and `number` can be used within the rest of their scope in the program. Variables can be declared anywhere in the program code, but before the first use of the variable.

Usually most variables are declared at the beginning of program, outside all functions (global variables) or at the beginning of function (local variable). Declaring more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

The integer data types `short`, `long` and `int` can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier `signed` or the specifier `unsigned` before the type name. By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed. Variable value can be set just after the declaration. It is called variable initiation. Initiation (initialization) is the assignment of variable value, declaring it. For example, variables can be described as follows:

```
int      k = 3;
float length = 2.18;
```

Initialization allows to shorten the program.

2.3 Constants

The sample "Hello World" program has just printed several sentences on the screen. Quotes were used to mark the text we would like to print:

```
cout << "Hello World! This is my first C++ program" << endl;
```

The text in quotes is string constant in this case. We use quotes to mark strings in C++. Single quotation marks are used to mark a separate symbol ('a'). String and symbolic constant, also Boolean constants are non-numeric constants.

Numeric constants are also used in C++ language. There are integer and floating point constants. Integer constants identify integer decimal values. In addition to decimal numbers, C++ allows the use of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (a zero character). In order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
const int e1 = 2012; // decimal
const int e2 = 03734; // 0, octal
const int e3 = 0x7DC; // 0x, hexadecimal
const int e4 = 0x7dc; // 0x, hexadecimal
```

Note that we can use both lower-case or capital letters typing x and corresponding letters in hexadecimal number. Printing the constant its value is automatically converted into decimal number.

Floating point constants express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses "by ten at the Xth height", where X is an integer value that follows the e character), or both a decimal point and an e character.

```
const double e5 = 9.87654;      // 9.87654
const double e6 = 5.97219e24;   // Earth mass, kg
const double e7 = 1.674e-27;    // mass of the hydrogen atom, kg
const double e8 = 1e100;       // gugol
```

Real numbers corresponding Earth mass, mass of the hydrogen atom (both in kg), and famous abstract number *gugol* are presented in usual notation:

$$e6 = 5.97219 \cdot 10^{24} ; \quad e7 = 1.674 \cdot 10^{-27} ; \quad e8 = 1 \cdot 10^{100} \quad (2.1)$$

We can make the program more readable by changing numeric values to identifiers. Besides, the use of constants enables us to change the parameter value easily. So, constant in programming is the identifier associated with fixed values. In C++ language we use two ways to define constants.

We can use macros to define constants. The `#define` preprocessor directive is used in this case. For example, we write:

```
# define VILNIUS 1323
# define EUR 3.45
# define ERROR "My error..."
```

We do not put the semicolon at the end of line, because this is not a C++ statement but a preprocessor directive. Compiling a program the defined constant name in program text will be replaced with the corresponding value. Once the constants are defined, you can use them in the rest of the code instead of writing their values. Constant names are usually written in capital letters.

We can use `const` command to define declared constants. The profile of the formal syntax is as follows:

```
const <data type> <constant name> = <constant value>
```

For example:

```
const char FIRST_LETTER = 'A';
const int MONTH = 30;
const float PI = 3.14;
```

2.4 Basic input and output

Using the standard input and output library, programmer is able to interact with the user by printing messages on the screen and getting the user's input from the keyboard. C++ uses a convenient abstraction called streams to perform input and output operations in sequential

media, such as the screen or the keyboard. A stream is an object, where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know, will it accept or provide characters sequentially.

The standard C++ library includes the header file `iostream`, where the standard input and output stream objects are declared.

By default, the standard output of a program is the screen, and the C++ stream object defined to access is `cout`. Function `cout` is used in conjunction with the insertion operator, which is written as `<<` (two "less than" signs).

```
cout << "My text";    // prints "My text" on screen
cout << 12.56;        // prints number 12.56 on screen
int x=7; cout << x;   // prints number 7 (the content of x) on screen
```

The insertion operator (`<<`) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

The last statement would print the message *Hello, I am a C++ statement* on the screen. The utility of repeating the insertion operator (`<<`) is useful when we want to print out a combination of variables and constants or more than one variable.

It is important to notice that `cout` does not add a line break after its output unless we indicate it explicitly, therefore, the following statements:

```
cout << "This is a first sentence. ";
cout << "This is a second sentence. ";
```

will be shown on the screen one following the other without any line break between them:

```
This is a first sentence. This is a second sentence.
```

even though we had written them in two different insertions into `cout`. In order to perform a line break on the output we must explicitly insert a new-line character into `cout`. In C++ a new-line character can be specified as `\n` (backslash, n):

```
cout << "First line.\n";
cout << "Second line.\nThird line.";
```

This produces the following output:

```
First line.
Second line.
Third line.
```

(`\n`) - combination of backslash and n is called *escape code*. These are special characters that are difficult or impossible to express otherwise in the source code of a program. All of them are preceded by a backslash (`\`). Here you have a list of the most recent escape codes.

<code>\n</code>	newline
<code>\b</code>	backspace
<code>\'</code>	single quote (')
<code>\?</code>	question mark (?)

<code>\t</code>	tab
<code>\a</code>	alert (beep)
<code>\"</code>	double quote (")
<code>\\</code>	backslash (\)

In order to add a newline, you may also use the `endl` manipulator. For example:

```
cout << "First line." << endl;
cout << "Second line." << endl;
```

would print out:

```
First line.
Second line.
```

Statement `cout` has two useful methods, which allow formatted printing.

Method `width()` defines the width of the output field. We must use this method before the output statement. We can set the width of the output field only for one output.

Method `precision()` defines the number of significant digits for real numbers (floating-point numbers). Precision remains valid for all output sentences used after its declaration, until we set a new value. The syntax of these methods is as follows:

```
cout.width      (<number of positions>);
cout.precision (<digit count>);
```

A sample program of formatted printing is presented in Table 2.4. The output of the program is shown in Table 2.5. As you can see, `cout.width(9)` reserves 9 positions for output and affects only one element (the second `cout` is not affected). The statement `cout.precision(4)` makes program to print only 4 decimal digits of `x` value.

Analogically, `cout.precision(3)` makes program to print only 3 decimal digits of `x` value. The value of `x` remains the same in both cases, only output is changed (the value of `x` is rounded). Note, that precision method affects the mantissa (decimal) part of a floating-point real number and do not take into account the value of exponent. We have different number of decimal digits while working with larger `x` values using the same precision.

We use system command `system("PAUSE")` at the end of the program to hold on the program window before exit.

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (`>>`) on the `cin` stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int day;
cin >> day;
```

The first statement declares a variable of type `int` called `day`, and the second one waits for an input from `cin` (the keyboard) in order to store it in this integer variable.

Table 2.4.	Formatted output.
<pre> //Formatted printing #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { float x = 0.12345; // initialization of x cout << 123456789 << endl; // first line cout << x << endl; // output of x cout.width(9); // width is set to 9 positions cout << x << endl; // x value is printed to the reserved field cout << x << endl; // width value is no longer valid cout.precision(4); // precision is set to 4 digits cout << x << endl; // x value with 4 decimal digits (rounded) cout.precision(3); // precision is set to 3 digits cout << x << endl; // x value with 3 decimal digits (rounded) x=1.2345; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) x=12.345; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) x=123.45; // new x value cout << x << endl; // x value with 3 decimal digits (rounded) system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.5.	Output of program.
<pre> 123456789 0.12345 0.12345 0.12345 0.1235 0.123 1.23 12.3 123 Press any key to continue ... </pre>	

Function `cin` can only process the input from the keyboard once the `RETURN` key has been pressed. Therefore, even if you request a single character, the extraction from `cin` will not process the input until the user presses `RETURN` after the character has been introduced.

We must always consider the type of the variable that is used as a container with `cin` extractions. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters. So the user must provide a valid data to the program. The input of invalid data may cause your program to misoperate.

We can also use `cin` to request more than one datum input from the user:

```
cin >> x >> y;
```

is equivalent to:

```
cin >> x;
cin >> y;
```

In both cases the user must give two data, one for variable `x` and another one for variable `y` that may be separated by any valid blank separator: a space, a tab character or a newline.

We can also use input and output functions inherited from the C language. These functions require the inclusion of `stdio.h` header file to the program. Function `scanf()` is used for data entry. The syntax is as follows:

```
scanf ("%f", &x);
```

Here is the address operator `&`, i.e., `&x` means the address of the variable `x` in the computer memory. Specifier `%f` means that floating-point number will be entered. Similarly, we can define another format, for example:

<code>%c</code>	character	<code>%s</code>	string
<code>%d</code>	integer	<code>%ld</code>	long
<code>%lf</code>	double		

If we specify `%s` the address operator (`&`) is not required because the string is an array and the array name represents its address.

Function `printf()` is often used for formatted printing. This function offers more possibilities for formatted printing than the output stream `cout`. The first parameter of `printf()` function is a string-constant, indicating the desired text and format settings for parameters we want to print, after this string we specify the variables and values to be printed. We use the same format specifiers as with `scanf()` function.

To define the format of floating-point number we use `%n.mf` structure where `n` is the width of the output field and `m` is the number of decimal digits. In this case the number of decimal digits shown do not depend on the number as it was with `precision` method. We can insert `-` before `n` to enable left alignment. It is also possible to use escape codes to improve the formatting.

Table 2.6 represents the program, where the possibilities of `printf()` function are shown. Table 2.7 is devoted for program output.

Table 2.6.	Formatted printing.
	<pre> //Formatted printing #include <cstdlib> #include <iostream> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { float x = 0.12345; // initialization of x printf("123456789123456789\n"); // first line printf("%f\n",x); // output of x printf("%9.5f\n",x); // 9 positions, 5 decimal places printf("%-9.5f%9.5f\n",x,x); // left and right alignment printf("%9.4f\n",x); // x value with 4 decimal digits printf("%9.3f\n",x); // x value with 3 decimal digits x=1.2345; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits x=12.345; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits x=123.45; // new x value printf("x=%7.3f\n",x); // x value with 3 decimal digits printf("1:%f\t2:%f\n3:%f\t4:%f\n",x,x,x,x); //tabulation system("PAUSE"); return EXIT_SUCCESS; } </pre>
Table 2.7.	Output of program.
	<pre> 123456789123456789 0.123450 0.12345 0.12345 0.12345 0.1235 0.123 x= 1.235 x= 12.345 x=123.450 1:123.449997 2:123.449997 3:123.449997 4:123.449997 Press any key to continue ... </pre>

2.5 Operators

The purpose of most programs is data entry and manipulation with data. For that purpose, C++ integrates operators. Operators are symbols or words that make the program to do something with the variables or expression. C++ has many different operators. The most important of them are the arithmetic (+ - * / %), compound assignment (+= -= *= /= %=), increase and decrease (++ --), relational and equality (> >= < <= == !=), logical (&& || !), and conditional operator (? :). The assignment operator (=) assigns a value to a variable.

```
y = 7;
```

This statement assigns the integer value 7 to the variable `y`. The left part of the assignment has to be a variable whereas the right part may contain either a constant, a variable, the result of an operation or any combination of those.

The most important rule while assigning is the right-to-left rule: **the assignment operation always takes place from right to left, and never the other way.**

```
x = y;
```

This statement assigns to variable `x` (the left value) the value contained in variable `y` (the right value). The value that was stored until this moment in `x` is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of `y` to `x` at the moment of the assignment operation. Therefore a later change of `y` will not affect the new value of `x`.

The five arithmetical operations supported by the C++ language are: + addition; - subtraction; * multiplication; / division; % modulo. Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be used to see is modulo; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:

```
a = 11 % 3;
```

the variable `a` will contain the value 2, since 2 is the remainder from dividing 11 between 3.

Writing program in C++ we can shorten the expression `x=x+a` like this: `x+=a`. This operator is called *compound assignment operator*. There are more of them:

+=	<code>x=x+a;</code>	<code>x+=a;</code>
-=	<code>x=x-a;</code>	<code>x-=a;</code>
*=	<code>x=x*a;</code>	<code>x*=a;</code>
/=	<code>x=x/a;</code>	<code>x/=a;</code>
%=	<code>x=x%a;</code>	<code>x%=a;</code>

Shortening some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to `+=1` and to `-=1`, respectively. A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (`++a`) or after it (`a++`). Although in simple expressions like `a++` or `++a` both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In

the case that the increase operator is used as a prefix (**++a**) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (**a++**) the value stored in **a** is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression. [45]

In order to evaluate the comparison between two expressions we can use the relational and equality operators (**==**, **!=**, **>**, **<**, **>=**, **<=**). The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

It is important to remember, that the operator **=** (one equal sign) is not the same as the operator **==** (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (**==**) is the equality operator that compares whether both expressions in the two sides of it are equal to each other.

Logical expressions are those expressions that contain logical operators, logical variables and/or the relational operators. The result of logical expression is true or false. We can use the following logical operators: **&&** (*AND*), **||** (*OR*), **!** (*NOT*).

Operator **!** (*NOT*) has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false.

The logical operators **&&** (*AND*) and **||** (*OR*) are used evaluating two expressions to obtain a single relational result. **&&** (*AND*) operation results true if both its two operands are true, and false otherwise. **||** (*OR*) operation results true if either one of its two operands is true, thus being false only when both operands are false themselves.

The conditional operator evaluates an expression returning a **value1** if that expression is true and a **value2** if the expression is evaluated as false. Its format is:

```
condition ? value1 : value2
```

Conditional operator enables us to shorten the program.

Type casting operators allow to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (**()**):

```
int i;
float f = 3.14;
i = (int) f;
```

Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

Note, that in C++ the type of expression result is determined according to the types of arguments. For example, if an integer is divided by integer, the answer is truncated and remains integer. Such program line:

```
cout << 5/2 <<endl;
```

will print out the answer 2. If we want to see the right answer, we must change the type of arguments:

```
cout << 5.0/2 <<endl;
```

or

```
cout << (float)5/2 <<endl;
```

First time we changed the data type of first argument (5.0 is a real number). Second time we used type casting. Using type casting we can also convert char to integer and integer to char. For example, such two program lines:

```
cout << (char) 97 << endl;
cout << (int) 'b' << endl;
```

will print out letter 'a' and number 98 (ASCII code of letter 'b'), respectively.

It is useful to remember the priority of operator. From greatest to lowest priority, the priority order is as follows:

- a) increase (increment) (++), decrease (decrement) (--) and logical *NOT* (!);
- c) type casting;
- c) multiplicative operators (*, /, %);
- d) additive operators (+, -);
- e) relational operators (>, >=, <, <=);
- f) equality operators (==, !=);
- g) logical *AND* (&&);
- h) logical *OR* (||);
- i) conditional operator (?:);
- j) assignment operators (=, +=, -=, *=, /=, %=).

We can change the priority of evaluation by using brackets.

2.6 Macros

In order to create simple functions it is also possible to use **define** preprocessor directive. These functions are usually called *macros*.

A macros is a finite fragment of the code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. It is possible to define any valid identifier as a macro, even if it is a C++ keyword. The preprocessor does not know anything about keywords. This can be useful if you want to hide a keyword such as `constant` from an older compiler that does not understand it.

We have already mentioned the possibility of using **define** directive to describe constants. For example:

```
#define LT 3.45
```

defines the macro named `LT` as an abbreviation for the value `3.45`. If somewhere after this **define** directive a C++ statement of the form comes,

```
money = eur * LT;
```

the C++ preprocessor will recognize and expand the macro `LT`. The C++ compiler will see this command line as if you had written

```
money = eur * 3.45;
```

By convention, macro names are written in uppercase. It is easier to read a program when it is possible to tell at once which names are macros.

The macro's body ends at the end of the **define** line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will come out on one line. For example:

```
#define NUMBERS 1, \
                2, \
                3
int x[ ] = { NUMBERS };
```

It corresponds to the statement:

```
int x[] = { 1, 2, 3 };
```

It is also possible to define macros which use looks like a function call. These are called function-like macros. Use the same **define** directive to define a function-like macro. But you should put a pair of parentheses immediately after the macro name. A function-like macro is only expanded if its name appears with a pair of parentheses after it. If you put spaces between the macro name and the parentheses in the macro definition, that does not define a function-like macro. We can even pass the parameters to macro. For example:

```
#define SUM(a, b) a+b
```

Using this macro, the output statement

```
cout << SUM(5, 3) << endl;
```

will print out value 8. Note, that output statement

```
cout << SUM(5, 3)*7 << endl;
```

will print out value 26, as we have no brackets in macro definition. So, we have an expression like this $5+3*7$.

It is also possible to use Assembler code in C++ program. The usage of Assembler allows to speed up the program, or to create specific programmer implemented functionality. For example, we can create function `add()` using Assembler. Function `asm` contains Assembler code.

```
int add(int a, int b)
{
    int c;
    asm(
        "addl %%ebx, %%eax;"
        : "=a" (c)
        : "a" (a),
          "b" (b)
        );
    return c;
}
```

This function will implement addition operation for integer numbers. The output sentence

```
cout << add(5, 3) << endl;
```

will print out 8.

2.7 Control structures

A program is usually not limited to a linear sequence of instructions. During the process it may bifurcate, repeat the code or take decisions.

For that purpose C++ provides control structures, that serve to specify, what has to be done by our program, when and under which circumstances.

In this section we are going to study such control structures as conditional structures, loops and some others.

In real life we have to change our activities according to the circumstances. For example, if it is raining outside, we choose proper clothes, perhaps, we will not go to the picnic. While programming with C++ we must also consider different conditions. C++ language, as many others programming languages, offers two main conditional structures: `if()` structure and `switch()` structure. So, the program can choose, what action to perform according to certain conditions.

The `if` keyword is used to execute a statement or a block of commands only if a condition is fully filled. Its form is:

```
if (condition) statement
```

where *condition* is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure. Note, that condition should always be in brackets. The statement may be a single command or a block of commands (if we want to execute more than one command).

Block of commands or compound statement is a group of statements, which are separated by semicolons (;), like all C++ statements, but grouped together in a block enclosed in braces: `{ }`:

```
{ statement1; statement2; statement3; }
```

For example, the following code fragment prints `y` value if `x` is positive:

```
if (x > 0)      cout << "y = " << y << endl;
```

We can specify, what must be done, if the condition is not fully filled by using the keyword `else`. This keyword is appended to `if()` as follows:

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

For example:

```
if  (x % 2 == 0)    cout << "x is an even number";
else                cout << "x is an odd  number";
```

We can add several `if-else` statements one to another if we want to evaluate several conditions. For example:

```
if      (x < 0)      cout << "x is a negative number";
else if (x > 0)      cout << "x is a positive number";
else                cout << "x is 0";
```

If we want to check several possible constant values of an expression, we can use `switch()` statement. Its form is the following:

```
switch (expression)
{
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    ...
    default:
        default statement;
}
```

Statement `switch()` evaluates expression and checks if it is equivalent to `value1`. If it is, it executes `statement1` (here can be several statements) until it finds the `break` statement. When it finds this break, the program jumps to the end of the `switch()` selective structure. If expression was not equal to `value1`, it will be checked against `value2`. If it is equal to this, it will execute `statement2`, until a `break` keyword is found, and then will jump to the end of the `switch()` selective structure. Finally, if the value of expression does not match any of the previously specified values (you can include as many case labels, as many values you want to check), the program will execute the statements included after the `default` label, if it exists (since it is optional).

In `switch()`, each `case` must be ended with `break`, otherwise the program will not stop after executing the statements, that must be executed for a specific condition, and will check out other values. We must also remember, that we can use only discrete expression in `switch` statement, and all the values also must be discrete.

As an example we will present a simple calculator (see Table 2.8). This program allows to enter two numbers and an operation sign. Afterwards it prints the result.

We often need to repeat some actions several times, perhaps changing only some details. Cyclic control structures enable us to repeat some of the steps many times and to organize cyclic computations.

C++ programming language has three types of cyclic structures (loops): `while()`, `do-while()` and `for()`.

The `while()` loop repeats a statement or a block while the condition set in expression is true. Its format is:

```
while (expression) statement
```

Note, that if the expression is false from the beginning, the `while` loop statement will not be executed at all. Its functionality is only to repeat the statement, while the condition set in expression is true.

For example, this program finds the first three-digit number which is divided by 17 (Table 2.9). The idea of the program is to multiply the number 17 by different numbers (we start with 1 and increase the value of second multiplier during each loop iteration) until we will get a number which is greater or equal to 100.

Table 2.8. Simple calculator.

```

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

int main(int argc, char *argv[])
{
    double x,y;  char s;
    cout << "Enter x: " ; cin >> x;
    cout << "Enter the operation sign (+, -, *, /): " ;
    cin >> s;
    cout << "Enter y: " ;
    cin >> y;

    switch (s)
    {
        case '+':
        {
            cout << x << "+" << y << "=" << x+y << endl;  break;
        }
        case '-':
        {
            cout << x << "-" << y << "=" << x-y << endl;  break;
        }
        case '*':
        {
            cout << x << "*" << y << "=" << x*y << endl;  break;
        }
        case '/':
        {
            if(y != 0)  cout << x << "/" << y << "=" << x/y << endl;
            else        cout << "Can not divide by 0\n";
            break;
        }
        default:
            cout<<"Invalid operation sign\n";
    };
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Creating a `while` loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. If we want to have the eternal loop, we can write `while(true)...` or `while(1)....`

Table 2.9.	Program.
	<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int a, b, c; a=17; b=1; c = a * b; while (c < 100) { b++; c = a * b; } cout << b << " " << c << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>

Sometimes we need to check the condition after the loop statement (in order to execute the loop statement at least once). We can use the **do-while** loop in this case. The functionality of this loop is exactly the same as the **while** loop, except that condition in the **do-while** loop is evaluated after the execution of statement instead of before. Its format is:

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

The following example program will ask for the year of Vilnius establishment until the right answer will be entered (Table 2.10).

The last, but not the least is **for()** loop. The main function of this loop is to repeat the statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So, this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration. Its format is:

```
for (initialization; condition; increase) {statement;}
```

It works in the following way [45].

1. Initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. Condition is checked. If it is true the loop continues, otherwise the loop ends and the statement is skipped (not executed).
3. The statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. Finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Table 2.10.	Vilnius establishment.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int a=0; do { cout << "Enter the year of Vilnius establishment \n"; cin >> a; } while (a != 1323); cout << "You are right! Vilnius was established in 1323" << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example, we could write:

```
for (; n<10; ) {...}
```

if we wanted to specify no initialization and no increase; or

```
for (; n<10; n++) {...}
```

if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before). Even the condition could be refused. In this case we will have open loop

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

This is the infinite loop.

Optionally, using the comma operator (,) we can specify more than one expression in initialization and increase fields of **for** loop. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. Note, that if we use a condition in **for** loop, it should be only one.

For example, we have a deposit in a bank and we want to know how much money we will have after a year, two, etc, 10 years. A program using a **for** loop is presented in Table 2.11.

It is possible to insert one loop into another. For example, this program will print out the multiplication table (see Table 2.12).

The inner loop prints one line, while the outer loop repeats new line printing with another value of *i*. Sometimes, working with loops or other structures we need to break the operation, or to skip the iteration or even to end the program.

Table 2.11.	Deposit.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { float s=1000; // the value of the deposit float p= 3; // the interest rate for(int y=1; y<=10; y++) { s *= 1.0 + p/100; cout << "After " << y << " year(s) we will have: " << s << endl; } system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.12.	Multiplication table.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { for(int i=1; i<10; i++) { for(int j=1; j<10; j++) { cout.width(3); cout<<i*j; } cout<<endl; } cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Statements `break`, `continue` and `exit` are used in these cases. It is possible to leave a loop even if the condition for its end is not fulfilled using a `break` statement. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we can use infinite loop to give the user an opportunity to perform some actions as many times as he wants (Table 2.13).

Table 2.13. Infinite loop management.

```

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

int main(int argc, char *argv[])
{
    int x;    char y;
    for(;;)
    {
        cout << "Enter x value:\n";                cin >> x;
        cout << x << "^2=" << x*x << endl;
        cout << "Do You want to continue? (y/n)\n";  cin >> y;
        if (y == 'n') break;
    }
    system("PAUSE");  return EXIT_SUCCESS;
}

```

Table 2.14. Infinite loop.

```

// Infinite loop with continue and break statements
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x;    char y;
    for(;;)
    {
        cout << "Enter x value:\n";                cin >> x;
        if (x==0) continue;
        cout << "1/" << x << "=" << 1.0/x << endl;
        cout << "Do You want to continue? (y/n)\n";  cin >> y;
        if (y == 'n') break;
    }
    system("PAUSE");  return EXIT_SUCCESS;
}

```

The `continue` statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, this program will skip the iteration if 0 will be entered (see Table 2.14).

The purpose of exit is to terminate the current program with a specific exit code. This

function is defined in the `cstdlib` library. Its prototype is:

```
void exit (int exitcode);
```

The `exitcode` is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means, that the program finished normally and any other value means that some error or unexpected results happened.

2.8 Functions

With the help of functions we divide program into smaller parts. Thus we can structure programs in a more modular way, accessing all the potential that structured programming can offer us in C++.

A *function* is a group of statements, that is executed when it is called from some point of the program. The format of function definition is as follows:

```
type name ( parameter1, parameter2, ...) { statements }
```

where following parameters are used:

- type* - data type specifier of the data returned by the function;
- name* - identifier by which it will be possible to call the function;
- parameters* - created as many as needed. Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: `int x`) which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas;
- statements* - function's body. It is a block of statements surrounded by braces `{ }`.

Let us create a first program with function as presented in Table 2.15. The result is presented in Table 2.16.

Let us analyze the program code. A C++ program always begins its execution by the `main` function. The first line of the `main()` function initializes integer variables `k` and `l` (`k=2`, `l=7`) and declares integer variable `m`. The second line calls function `multi()`. Being attentive we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself:

```
int multi (int x, int y);
m = multi(5,3);
```

The parameters and arguments have clear correspondence. Within the `main` function we called to function `multi` passing two values: 5 and 3, that correspond to the `int x` and `int y` parameters declared for function `multi`. Calling the function the value 5 is assigned to functions local variable `x` and the value 3 is assigned to functions local variable `y`.

At the point where the function is called from within `main`, the control is lost by `main` and passed to function `multi`. Function `multi` declares another local variable (`int r`), and by means of the expression `r=x*y`, it assigns to `r` the result of `x` multiplied by `y`, because the actual parameters passed for `x` and `y` are 5 and 3 respectively, the result is 15.

Statement `return r` finalizes the work of function `multi`, and returns the control back to the function that called it in the first place (in this case, `main`).

Table 2.15.	Multiplication function
<pre> #include <cstdlib> #include <iostream> using namespace std; int multi (int x, int y) // multiplication function { int r; r = x * y; return r; } int main(int argc, char *argv[]) { int k=2, l=7, m; m = multi(5,3); cout << "5*3=" << m << '\n'; cout << "8*4=" << multi(8,4) << "\n"; cout << "2*7=" << multi(k,l) << "\n"; m = 5 + multi(k,l); cout << "5+2*7=" << m << "\n"; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.16.	Output of program.
<pre> 5*3=15 8*4=32 2*7=14 5+2*7=19 Press the enter key to continue ... </pre>	

At this moment the program follows its regular course from the same point at which it was interrupted by the call to `multi`. In addition, because the return statement in function `multi` specified a value: the content of variable `r` (**r**eturn `r`), which at that moment had a value of 15. This value becomes the value of evaluating the function call. This value is assigned to the variable `m` in our program. The line

```
cout << "5*3=" << m << "\n";
```

outputs `m` value to the screen (`5*3=15`).

Then we call function `multi()` several more times:

```

cout << "8*4=" << multi(8,4) << '\n';
cout << "2*7=" << multi(k,l) << '\n';
m= 5 + multi(k,l);
cout << "5+2*7=" << m << '\n';

```

We have used some different calling methods. Calling function `multi()` for the second time, we do not assign the function value to a variable. We just printed the returned value. Calling function `multi()` for the third time, we used variables `k` and `l` to pass the parameters. In this case the values passed to function `multi()` are the values of `k` and `l`, that are 2 and 7 respectively. The fourth call to function `multi` is made from the expression

```
m= 5 + multi(k,l);
```

The value of function is used to evaluate the value of the whole expression $(5+2*7)$. Sometimes we need to create a function with no parameters, or the function which returns nothing. In first case we simply live the parameter list of the function empty:

```

int myfunction()
{
    return 2012;
}

```

In the second case we should use the `void` type. This is a special specifier that indicates absence of type (see Table 2.17).

It is necessary to remember that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement.

Table 2.17.

Program.

```

// Sample program with void type function
#include <cstdlib>
#include <iostream>
using namespace std;

void line() // line printout
{
    cout << "-----\n";
}

int main(int argc, char *argv[])
{
    line();
    cout << "    Hello, World\n";
    line();
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

What you should know about C++ functions?

1. Do not insert one function into another. All functions are equivalent and accessible to all. Only the function `main()` is required for each program. The only difference of `main()` function is that the program begins execution from it.
2. We should define the type of function before the function description.
3. The parentheses are always necessary calling the function, even if the parameter list is empty.
4. Function parameters are declared as follows: (*type1 parameter1, type2 parameter2, ...*), i.e. types can not be grouped together, as declaring variables.
5. If we add `const` before the parameter type, it is the order to the compiler that the function can not modify the parameter value.
6. The keyword `return` transmits function return value, if it exists.
7. If the return parameter type `void`, you can not use the keyword `return`. The exception is when you need to leave the function before it is fully implemented. In this case the `return` keyword without any parameters is used.
8. Using `return` keyword we can transmit only one value from the function. If we need to return more values, we must use other mechanisms.
9. Calling a function, the types of the parameter values must be compatible.
10. Functions (or its prototype) must be defined before the first call of it. The function prototype is similar to the first line of the function. We can omit the parameter names defining the function prototype.

Form of function prototype is:

```
type name ( argument_type1, argument_type2, ... );
```

Until now, in all the functions we have seen, the arguments passed to the functions have been passed by value. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves.

For example, suppose that we called our first function `multi` using the following code:

```
int k=5, l=3, m;
m = multi(k, l);
```

What we were doing in this case was to call to function `multi()` passing the values of `k` and `l`, i.e. 5 and 3 respectively, but not the variables `k` and `l` themselves.

In this way, when the function `multi()` is called, the value of its local variables `x` and `y` become 5 and 3 respectively, but any modification to either `x` or `y` within the function `multi` will not have any effect in the values of `k` and `l` outside it, because variables `k` and `l` were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function `triple` of the following example (see Table 2.18). The output of the program is presented in Table 2.19.

The first thing that should call your attention is that in the declaration of function `triple()` the type of each parameter was followed by an ampersand sign (`&`). This ampersand is what specifies that their corresponding arguments are to be passed by reference instead of by value.

Table 2.18. Passing parameters by reference.

```

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

void triple(int& x, int& y, int& z)
{
    x *= 3; y *= 3; z *= 3;
}

int main(int argc, char *argv[])
{
    int k=1, l=3, m=7;
    triple(k, l, m);
    cout << "k=" << k
         << ", l=" << l
         << ", m=" << m << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

Table 2.19. Output of program.

```

k=3, l=9, m=21
Press the enter key to continue ...

```

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

In other words, we associate **x**, **y** and **z** with the arguments passed on the function call (**k**, **l** and **m**) and any change that we do on **x** within the function will affect the value of **k** outside it. Any change that we do on **y** will affect **l**, and the same with **z** and **m**.

That is why our program's output, that shows the values stored in **k**, **l** and **m** after the call to function `triple()`, shows the values of all the three variables of `main` tripled.

If declaring the following function `void triple(int& x, int& y, int& z)` we had declared it this way: `void triple(int x, int y, int z)`; i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of **k**, **l** and **m** without having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the sum and difference for two numbers (see Table 2.20). The output of the program is presented in Table 2.21.

Table 2.20.	Passing by reference.
-------------	-----------------------

```
// Function returns more than one value
#include <cstdlib>
#include <iostream>
using namespace std;

void count(int x, int y, int& sum, int& dif)
{
    sum = x + y;
    dif = x - y;
}

int main(int argc, char *argv[])
{
    int k=9, l=5, m, n;
    count(k, l, m, n);
    cout << "k+l=" << m << "; k-l=" << n << endl;
    cout << "Press the enter key to continue ... "; cin.get();
    return EXIT_SUCCESS;
}
```

Table 2.21.	Output of program.
-------------	--------------------

```
k+l=14; k-l=4
Press the enter key to continue ...
```

Declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example, see sample program in Table 2.22.

As we can see in the body of the program there are two calls to function `divide()`. In the first one:

```
cout << divide (12) << endl;
```

we have only specified one argument, but the function `divide()` allows up to two.

So the function `divide()` has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with `int b=2`, not just `int b`). Therefore the result of this function call is 6 (12/2).

In the second call:

```
cout << divide (20,4) << endl;
```

Table 2.22.

Ignored default value.

```

// Default values for function arguments
#include <cstdlib>
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
    int r;
    r = a / b;
    return r;
}

int main(int argc, char *argv[])
{
    cout << divide (12)    << endl;
    cout << divide (20,4) << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

there are two parameters, so the default value for **b** (`int b=2`) is ignored and **b** takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4). [45]

Describing the default values of the arguments, we should follow following rules.

1. If we set default value to some parameter, we have to set default values to all the parameters after it.
2. We should not set default values to the functions parameters in random order.
3. The list of functions parameters must be divided into two parts: parameters, that do not have default values, and parameters, that have the default values.
4. While calling the function, we must assign the values to all the parameters, that do not have default values.
5. We can omit parameters with default values.
6. If we omit the value of the parameter, we have to omit all the rest of the parameters after it.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. These functions are called *overloaded* functions.

For example, let us define two functions with the same name, **triangle**, but one of them accepts two parameters of type float and calculates triangle area using formula

$$S = \frac{1}{2} \cdot a \cdot h \quad (2.2)$$

Table 2.23.	Overloaded functions.
<pre> #include <cstdlib> #include <iostream> #include <math.h> using namespace std; float triangle (float a, float h) { return 0.5*a*h; } float triangle (float a, float b, float c) { float p=0.5*(a+b+c); return sqrt(p*(p-a)*(p-b)*(p-c)); } int main(int argc, char *argv[]) { cout << triangle(7,4) << endl; cout << triangle(3,4,5) << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

while another one accepts tree parameters of type float and calculates triangle area using Herons formula:

$$S = \sqrt{p * (p - a) * (p - b) * (p - c)} \quad ; \quad p = 0.5 * (a + b + c) \quad (2.3)$$

The compiler knows which one to call in each case by examining the number of arguments passed when the function is called (see Table 2.23).

Notice, that a function can not be overloaded only by its return type. At least one of its parameters must have a different type.

We can use the `inline` specifier before the function definition. This specifier indicates the compiler that the inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behaviour of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in the running time. Most compilers optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

2.9 Global, local and static variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them - so called local variables.

Table 2.24.

Local variables.

```

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

void modify (int x, int y)
{
    x *= 2;  y += 5;
    cout << "x=" << x << ", y=" << y << endl;
    x++;  y++;
}

int main(int argc, char *argv[])
{
    int x=3, y=7;
    cout << "x=" << x << ", y=" << y << endl;
    modify(x,y);
    cout << "x=" << x << ", y=" << y << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

Table 2.25.

Output of program.

```

x=3, y=7
x=6, y=12
x=3, y=7
Press the enter key to continue ...

```

For example, in a sample program (see Table 2.24) we use two variables `x` and `y` in `main()` function. The same variables are used in function `modify()`. Function `main()` sets the values of its local variables `x` and `y` (`x=3, y=7`) and prints out these values. Then the `modify()` function is called. The values of the variables `x` and `y` are assigned to the functions `modify()` local parameters `x` and `y`. The names of the variables in `main()` and in `modify()` functions are the same, but these variables are different. Function `main()` has its own local `x` and `y` and `modify()` has its own `x` and `y`. Function `modify()` changes the values of the parameters and prints them out to the screen.

After calling `modify()` function, `main()` function outputs the values of `x` and `y` once more (the values are not changed, see Table 2.25).

There is the possibility to declare global variables. These variables are visible from any point of the code, inside and outside all functions. In order to declare global variables we simply have to declare the variable outside any function or block; that means, directly in the body of the program (just after `namespace` definition).

Table 2.26.	Global variables.
<pre> #include <cstdlib> #include <iostream> using namespace std; int x, y; void modify() { x *= 2; y += 5; cout << "x=" << x << ", y=" << y << endl; x++; y++; } int main(int argc, char *argv[]) { x=3; y=7; cout << "x=" << x << ", y=" << y << endl; modify(); cout << "x=" << x << ", y=" << y << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.27.	Output of program.
<pre> x=3, y=7 x=6, y=12 x=7, y=13 Press the enter key to continue ... </pre>	

Let us implement previous program using global variables - see Table 2.26. Now two global variables `x` and `y` are used. Function `main()` sets the values of variables `x` and `y` (`x=3, y=7`) and prints out these values - see Table 2.27.

Then the `modify()` function is called. Function `modify()` changes the values of the parameters and prints them out to the screen. After calling `modify()` function, the `main()` function outputs the values of `x` and `y` once more (the values are changed).

In some cases the local value of a variable must be preserved between the function calls. The static variables are used in these cases. These variables can be accessed only within the function where they were defined, but the value of the variable remains after the function execution.

The memory for static variables is reserved at the beginning of the program and they are kept there during the program running time, although they are available only in the function in which they are created.

Table 2.28.	Static variables.
<pre> #include <cstdlib> #include <iostream> using namespace std; double average(double x) { static double sk = 0; static double suma = 0; ++sk; suma += x; return suma/sk; } int main(int argc, char *argv[]) { cout << "average=" << average(1) << endl; cout << "average=" << average(2) << endl; cout << "average=" << average(4) << endl; cout << "average=" << average(10) << endl; cout << "average=" << average(11) << endl; cout << "Press the enter key to continue ..."; cin.get(); return EXIT_SUCCESS; } </pre>	

Table 2.29.	Output of program.
<pre> average=1 average=1.5 average=2.33333 average=4.25 average=5.6 Press the enter key to continue ... </pre>	

After the function completes its work, static variables retain their values.

For example, let us create an average counting program using static variables (see Table 2.28). The values of the variables `sk` and `suma` are preserved between function calls and enables us to calculate the average easily. Table 2.29 shows output of program.

2.10 Arrays

An array is a series of elements of the same type placed in the contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

Like a regular variable, it is necessary to declare an array before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where *type* is a valid type (like `int`, `float`, etc), *name* is a valid identifier and the *elements* field (which is always enclosed in square brackets `[]`), specifies how many of these elements the array has to contain. The elements of the array are numbered from 0. So, if we want to declare an array of 4 integer numbers, we can do it as follows:

```
int numbers [4];
```

NOTE: The elements field within brackets `[]` which represents the number of elements the array is going to include, must be a constant value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length a dynamic memory is needed.

When declaring a local array its elements will not be initialized to any value by default, so their content will be undetermined until we store some value in them. The elements of global and static arrays, are automatically initialized with their default values, that means they are filled with zeros (working with numeric data types).

In both cases, local and global, when we declare an array, we have the possibility to assign initial values to each element of the array by enclosing the values in braces `{ }`. For example:

```
int numbers [4] = { 7, 15, 22, 19 };
```

The amount of values between braces must not be larger than the number of elements that we declare for the array between square brackets `[]`.

Initializing an array, C++ makes the possibility of leaving the square brackets empty `[]`. In this case, the compiler will assume a size for the array that matches the number of values included between braces `{ }`:

```
int numbers [] = { 7, 15, 22, 19 };
```

It is possible to access the value of any of the elements of array by specifying its index. It enables us to use array elements as if it was a normal variable. The format of accessing array element is as follows:

```
name [index];
```

Also it is possible to access third element of array `numbers` like this:

```
numbers[2];
```

If we want to set the value 37 to the second element of array `numbers`, we should write:

```
numbers[1] = 37;
```

Notice that the third element of `numbers` is specified `numbers[2]`, since the first one is `numbers[0]`, the second one is `numbers[1]`, and therefore, the third one is `numbers[2]`. The last element will be `numbers[3]`. If we write `numbers[4]`, we will access the fifth element of `numbers` and therefore exceed the size of the array.

Table 2.30.

The sum of array's elements.

```

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

int main(int argc, char *argv[])
{
    int mas[7] = {5, 9, 12, 3, 7, 2, 18};
    int    sum = 0;
    for(int i=0; i<7; i++)
    {
        sum += mas[i];
    }
    cout << "sum=" << sum << endl;
    cout << "Press the enter key to continue ...";
    cin.get();
    return EXIT_SUCCESS;
}

```

It is important to understand that in the C++ language we do not receive notification, if the array indexes exceed the specified array length. The value will be placed in the memory, which is not reserved for it and can cause runtime errors.

Brackets `[]` perform two different tasks while working with arrays: one is to specify the size of arrays when they are declared; and the second one is to specify indexes for concrete array elements.

We can operate with arrays, working with their elements like with stand alone variables. For example, we can calculate the sum of the arrays elements (see Table 2.30), or we can find the minimal element of the array (see Table 2.31).

Array input and output are also presented in this program.

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a bidimensional table made of elements, all of which are of the same uniform data type.

The way to declare bidimensional array in C++ is:

```
int matrix[4][3];
```

This will declare a matrix with 4 rows and 3 columns. To access the second element of the third row of this array, we should write:

```
matrix[2][1];
```

Multidimensional arrays are not limited to two indexes (i.e., two dimensions). They can contain as many indexes as needed. But be careful! The amount of memory needed for an array increases rapidly with each dimension.

For example:

```
char century [100][365][24][60][60];
```

Table 2.31. The minimal element of the array.

```

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

int main(int argc, char *argv[])
{
    int mas[20];
    int n,min;
    cout << "How meny elements do you want to enter?\n";
    cin >> n;
    for(int i=0; i<n; i++)          // array input
    {
        cout << "mas[" << i << "]=";
        cin >> mas[i];
    }
    cout << "Entered array:\n";
    for(int i=0; i<n; i++)          // array output
    {
        cout << mas[i] << "  ";
    }
    cout << endl;
    min = 0;
    for(int i=1; i<n; i++)          // search of minimum element
    {
        if (mas[i] < mas[min]) min = i;
    };
    cout << "Minimal array element: " << mas[min] << endl;
    cout << "Its index is: " << min << endl;
    system("PAUSE"); return EXIT_SUCCESS;
}

```

declares an array with a char element for each second in a century, that is more than 3 billion characters. So this declaration would consume more than 3 GB (gigabytes) of memory!

At some moment it is necessary to pass an array to a function as a parameter. In C++ it is not possible to pass a complete block of memory by value as a parameter to a function, but it is possible to pass its address. In practice that has almost the same effect and this operation is much faster and more efficient. [45]

In order to accept arrays as parameters the only thing that is necessary to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void brackets []. For example, the following function:

```
void procedure (int arg[])
```

accepts a parameter of type "array of int" called **arg**. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

In a function declaration it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type [] [depth] [depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray [] [3] [4])
```

Notice that the first brackets [] are left empty while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension. [45]

2.11 Character sequences

C++ Standard Library implements a powerful string class, which is very useful to handle and manipulate strings of characters. And, as strings are in fact sequences of characters, they can be represented as plain arrays of char elements. The difference between char sequence and other types of arrays is that the '\0' (backslash, zero) symbol is always placed at the end of the symbol sequence. This symbol is known as a null character and is used to signal the end of the valid sequence.

For example, the following array:

```
char myword [30];
```

It can store character sequences up to 29 characters long. But it is also possible to store shorter sequences. Sample program using character sequence is presented in Table 2.32. We use character sequence to enter the name and then the program prints out greetings. Note, that runtime errors can occur if the length of entered character sequence will be bigger than specified array size.

It is possible to initiate character sequence in several forms using string literal:

```
char myword[80] = "Some text here";
char sentence[ ] = "Vilnius is the capital of Lithuania.";
```

It is possible to work with character sequences as with ordinary arrays, but also specific functions can be used to manage them. Header file `string.h` must be included to operate with string functions. To copy the contents of one character sequence into another it is necessary to call the function `strcpy (str1, str2)`. This function copies the contents of one character sequence `str2` into another `str1`:

```
char planet[50];
strcpy (planet, "Earth");
```

Unfortunately it is impossible to write sentence `planet="Earth"` working with character sequences. We can use such assignment only initializing the value.

Table 2.32.	Character sequence.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { char name[30]; cout << "Enter your name: "; cin >> name; cout << "Hello, " << name << "!" << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

It is possible to delete specified character using `strcpy()` function:

```
strcpy (&planet[2], &planet[3]);
```

This command will delete the third character (the result will be **Eath**).

It is possible to determine the length of the character sequence using `strlen()` function, for example,

```
cout << strlen(myword);
```

will print out 14 (spaces are included, but null character does not).

We can compare two character sequences using `strcmp()` function:

```
int strcmp (str1, str2);
```

This function compares character sequences and returns a negative number if the `str1` is less than the `str2`, returns 0 if the character sequences are equal, and returns a positive number if the `str1` is greater than the `str2` (numerical codes of characters are compared).

Function `strlwr(str1)` changes in capital letters into small. Function `strupr(str1)` changes small letters into capital letters.

Note, that `cin` is able to input character sequence until the first space. We can use

```
cin.getline(str1,sizeof(str1));
```

to enter whole string. We can also use `gets(str1)` for input and `puts(str1)` for output. Both these functions are inherited from C. We must include `stdio.h` header file to use these functions.

It is also possible to create string array:

```
char names [20][10];
```

This array can store up to 20 words of 9 characters. Finally, sequences of characters, stored in char arrays, can easily be converted into string objects just by using the assignment operator:

```

string mystring;
char myntcs[]="some text";
mystring = myntcs;

```

2.12 Pointers

We must not care about the physical location of our data within memory while using variables. We simply use its identifier whenever we want to refer to our variable. The memory of the computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (1B, one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number, as the previous one, plus one.

In this way, each cell can be easily located in the memory, because it has the unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776, we know, that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be. Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address, where our variable is being stored during runtime in order to operate with relative positions to it [45].

The address, that locates a variable within memory, is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example, see Table 2.33. Table 2.34 shows output of program.

Sometimes it is convenient to store the address of one variable in another variable. The variable, that stores the reference to another variable, is what we call a pointer. Pointer is a very powerful feature of the C++ language, that has many uses in advanced programming. Now we will see how this type of variable is used and declared.

Table 2.33.

Variable address.

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

int main(int argc, char *argv[])
{
    int x=7;
    cout << "x=" << x << ", &x=" << &x << endl;
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 2.34. Output of program.

```
x=7, &x=0x22ff4c
Press any key to continue ...
```

Due to the ability of a pointer to refer directly to the value, that it points to, it becomes necessary to specify in its declaration, which data type a pointer is going to point to. It is not the same thing to point to a char, as to point to an int or a float. The declaration of pointers follows this format:

```
type * name;
```

where type is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
int    * number;
char   * character;
float  * greatnumber;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact, all of them are pointers and all of them occupy the same amount of space in the memory (the size in memory of a pointer depends on the platform, where the code is going to run).

Nevertheless, the data to which they point to, do not occupy the same amount of space, nor are of the same type: the first one points to an `int`, the second one to a `char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type, they point to.

Using a pointer, we can directly access the value, stored in the variable, which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

It is necessary to emphasize, that the asterisk sign (*), that we use when declaring a pointer, means only that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator which is also written with an asterisk (*). They are simply two different things, represented with the same sign.

Let us analyze the code in Table 2.35. Table 2.36 shows output of program. In this program we initialize the variable `x` of type `int` as (`int x=7`) and declare integer type pointer `p` as (`int *p`). Then the address of variable `x` is saved in `p` as (`p=&x`). Finally, we perform the output. Note, that `x` and `*p` outputs the same value, and also `&x` and `p` are the same. We can also find the address of pointer itself (`&p`).

It is important to clarify the difference between the address (`p`) and the value, stored in the memory under this address (`*p`).

Notice the difference between the reference and dereference operators: `&` is the reference operator and can be read as "address of", `*` is the dereference operator and can be read as "value pointed by".

Thus, they have complementary (or opposite) meanings. A variable referenced with `&` can be dereferenced with `*`. Since we have got the address of the variable, we can use pointer to access it.

```
int  x;
int *p;
    p = &x;
    *p = 12;
    *p-= 7;
```

Table 2.35.	Pointer.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x=7; int *p; p = &x; cout << "x=\t" << x << ", &x=\t" << &x << endl; cout << "*p=\t" << *p << ", p=\t" << p << ", &p=\t" << &p << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.36.	Output of program.
<pre> x= 7, &x= 0x22ff4c *p= 7, p= 0x22ff4c, &p= 0x22ff48 Press any key to continue ... </pre>	

The meaning of an expression which used the dereference operator (*) is different from one, that does not. When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so, in fact, they are of the same concept.

For example, supposing these two declarations:

```
int numbers [20];
int *p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a constant pointer. We can use pointer `p` to access the elements of `numbers` array:

```
numbers[index] == *(numbers + index) == *(p + index)
&numbers[index] == (numbers + index) == (p + index)
```

Working with multidimensional arrays we can use such dependences:

```
int matrix[n][m];
matrix[index1][index2] == (*(matrix + index1) + index2)
&matrix[index1][index2] == *(matrix + index1) + index2
```

Only addition and subtraction operations are allowed to be conducted with pointers, the others make no sense in the world of pointers. But both addition and subtraction have a different behaviour with pointers according to the size of the data type to which they point.

Some data types occupy more memory, others require less memory space. For example, let's assume that in a given compiler for a specific machine, `char` takes 1 byte, `short` takes 2 bytes and `long` takes 4 bytes.

Suppose, that we define three pointers in this compiler:

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively. So if we write:

```
mychar++;
myshort++;
mylong++;
```

variable `mychar`, as you may expect, would contain the value 1001. But not so obviously, `myshort` would contain the value 2002, and `mylong` would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer. [45]

This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Both the increase (`++`) and decrease (`--`) operators have greater operator precedence than the dereference operator (`*`), but both have a special behaviour when used as suffix (the expression is evaluated with the value, it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++ ;
```

Because `++` has greater precedence than `*`, this expression is equivalent to `*(p++)`. Therefore, what it does is to increase the value of `p` (so it now points to the next element), but because `++` is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased). Notice the difference with:

```
(*p)++ ;
```

Table 2.37.	Variable address.
<pre> #include <cstdlib> #include <iostream> using namespace std; int main(int argc, char *argv[]) { int x[] = {7, 9, 13, 19, 8, 2, 11}; int *p; p = x; for(int i=0; i<7; i++, p++) cout << *p << '\t'; cout << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.38.	Output of program.					
<pre>7 9 13 19 8 2 11 Press any key to continue ...</pre>						

Here, the expression would have been evaluated as the value pointed by `p` increased by one. The value of `p` (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer). [45]

It is very convenient to use pointer arithmetic with arrays - see Table 2.37, output - Table 2.38. Increase operator enables us to address all the elements of the array.

We can use addresses and pointers working with functions. We have already discussed the possibility of passing parameters by value or by reference. Let us review the material adding the pointers. Table 2.39 represents three different function styles. Table 2.40 shows output of program.

We will create three different functions `int cube1()`, `void cube2()`, `void cube3()` for calculating cube of a number according to simple arithmetic expression:

$$y = x^3 \quad (2.4)$$

Function `int cube1(int x)` is presented in traditional function style, one parameter passed by value, result will be received using `return` statement. Function `void cube2(int &x)` is written in reference style: parameter passed by reference, and result will be returned through the function parameter. Function `void cube3(int *x)` is written in pointer style: parameter passed using pointer, and result will be returned using pointer. Reference style is the most preferable style in C++ programming, in comparison to pointer style which is typical in C programming.

Table 2.39.	fun.cpp
<pre> #include <cstdlib> #include <iostream> using namespace std; int cube1 (int x); void cube2 (int &x); void cube3 (int *x); int main(int argc, char *argv[]) { // It is possible to pass // a number cube1(2) int a = 2; cout << a << " "; cout << cube1(a) << endl; // working with cube2 // we must pass a variable a = 3; cout << a << " "; cube2(a); cout << a << endl; // working with cube3 // we must pass an address a = 4; cout << a << " "; cube3(&a); cout << a << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	<pre> int cube1(int x) { // parameter passed by value int y = x*x*x; return y; // return to get the result } void cube2(int &x) { // parameter passed by reference int y = x*x*x; x = y; // return the result // through the function parameter } void cube3(int *x) { // parameter passed using pointer int y = *x * *x * *x; *x = y; // return the result // using pointer } </pre>

Table 2.40.	Output of program fun.cpp.
2 8	
3 27	
4 64	

2.13 Data structures

We have already learned how groups of sequential data can be used in C++. But this is somewhat restrictive, since in many occasions what we want to store are not only sequences of elements all of the same data type, but sets of different elements with different data types.

A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. Data structures are declared in C++ using the following syntax:

```

struct structure_name
{
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    ...
}    object_names;

```

where `structure_name` is a name for the structure type, `object_names` can be a set of valid identifiers for objects that have the type of this structure. Within braces `{ }` there is a list with the data members, each one is specified with a type and a valid identifier as its name. [45]

The first thing we have to know, is that a data structure creates a new type. Once a data structure is declared, a new type with the identifier specified as `structure_name` is created and can be used in the rest of the program, as if it was any other type. For example:

```

struct product
{
    int weight;
    float price;
};
product apple;
product banana, melon;

```

We have first declared a structure type called `product` with two members: `weight` and `price`, each of a different fundamental type. We have then used this name of the structure type (`product`) to declare three objects of that type: `apple`, `banana` and `melon`, as we would have done with any fundamental data type.

Once declared, `product` has become a new valid type name like the fundamental ones `int`, `char` or `short` and from that point on we are able to declare objects (variables) of this compound new type, like we have done with `apple`, `banana` and `melon`.

Right at the end of the `struct` declaration and before the ending semicolon, we can use the optional field `object_name` to directly declare objects of the structure type. For example, we can also declare the structure objects `apple`, `banana` and `melon` at the moment we define the data structure type this way:

```

struct product
{
    int weight;
    float price;
}
    apple, banana, melon;

```

It is important to differentiate clearly between what is the structure type name, and what is an object (variable) that has this structure type. We can instantiate many objects (i.e. variables, like `apple`, `banana` and `melon`) from a single structure type `product`).

Once we have declared our three objects of a determined structure type (`apple`, `banana` and `melon`) we can operate directly with their members. To do that, we use a dot (`.`) inserted between the object name and the member name. For example, we could operate with any of these elements, as if they were standard variables of their respective types:

Table 2.41.	Structure sample program.
<pre>#include <cstdlib> #include <iostream> #include <string.h> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { struct product { char name[100]; float price; int amount; } p1, p2; strcpy(p1.name,"CPU"); p1.price = 350.20; p1.amount = 70; cout << "Enter the name of the product: "; gets (p2.name); cout << "Enter the price of the product: "; cin >> p2.price; cout << "Enter the amount of the product: "; cin >> p2.amount; cout << "Available products:\n"; printf("%s\t%7.2f\t%d\n", p1.name, p1.price, p1.amount); printf("%s\t%7.2f\t%d\n", p2.name, p2.price, p2.amount); system("PAUSE"); return EXIT_SUCCESS; }</pre>	

Table 2.42.	Output of program.
<pre>Enter the name of the product: HDD Enter the price of the product: 215.81 Enter the amount of the product: 150 Available products: CPU 350.20 70 HDD 215.80 150 Press any key to continue ...</pre>	

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Each one of these has the data type, corresponding to the member they refer to variables: `apple.weight`, `banana.weight` and `melon.weight` are of type `int`, while `apple.price`, `banana.price` and `melon.price` are of type `float`.

Let's see a real example where you can see, how a structure type can be used in the same way as fundamental types (see Table 2.41). In this program we use `gets()` to input a string and `printf()` to perform the output. Table 2.42 shows output of program.

Data structures can be used to represent databases, especially if we consider the possibility of building arrays of them (see Table 2.43). Like any other type, structures can be pointed by its own type of pointers:

Table 2.43.	Structure array.
<pre> #include <cstdlib> #include <iostream> #include <string.h> #include <stdio.h> using namespace std; int main(int argc, char *argv[]) { struct product { char name[100]; float price; int amount; } p[100]; int n; cout << "Enter the amount of different products" << " you would like to specify:\n"; cin >> n; for(int i=0; i<n; i++) { cout << "Enter the name of the product: "; cin >> p[i].name; cout << "Enter the price of the product: "; cin >> p[i].price; cout << "Enter the amount of the product: "; cin >> p[i].amount; }; cout << "Available products:\n"; for(int i=0; i<n; i++) { printf("%s\t%7.2f\t%d\n", p[i].name, p[i].price, p[i].amount); }; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 2.44. Combinations of pointers and structure members.		
Expression	What is evaluated	Equivalent
<code>a.b</code>	Member <code>b</code> of object <code>a</code>	
<code>a->b</code>	Member <code>b</code> of object pointed by <code>a</code>	<code>(*a).b</code>
<code>*a.b</code>	Value pointed by member <code>b</code> of object <code>a</code>	<code>*(a.b)</code>

```

struct product
{
    char name[100];
    float price;
    int amount;
}
    p1, *p;
p = &p1;
p -> price = 150;

```

Here `p1` is an object of structure type `product`, and `p` is a pointer to point to objects of structure type `product`. The previous code includes an important introduction: the arrow operator (`->`). This is a dereference operator, that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference.

In the example we used: `p->price` which is equivalent to: `(*p).price`. Both expressions are valid and both mean, that we are evaluating the member `price` of the data structure pointed by a pointer, called `p`. Table 2.44 summarizes possible combinations of pointers and structure members.

Structures can also be nested so that a valid element of a structure can also be in its turn another structure. It is possible to create connected list of structures using pointers and dynamic memory allocation as presented in Tables 2.45a, 2.45b.

Table 2.45a.	List of structures using pointers.
--------------	------------------------------------

```

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

struct person
{
    char    name[50];
    int     age;
    person *next;
    // We will use this pointer to access the next person
};

```

Table 2.45b.

List of structures using pointers.

```

int main(int argc, char *argv[])
{
    person  *p1 = NULL,
            *p2 = NULL;
    int      n = 0;
    char t;
    cout << "Would you like to enter a person? (y/n)\n";    cin >> t;
    while((t=='y')||(t=='Y'))
    {
        if (p1==NULL)
        {
            // dynamic memory allocation for pointer p1
            p1 = (person *)malloc(sizeof(person));
            p2 = p1;
        }
        else
        {
            p2 -> next = (person *)malloc(sizeof(person));
            // dynamic memory allocation for pointer p2
            p2 = p2 -> next;
        };
        // function malloc allocates the requested size of memory
        // we use type casting to convert general pointer
        // returned by malloc to specified type of pointer
        cout << "Enter the person's name: ";    cin >> p2 -> name;
        cout << "Enter the person's age: ";    cin >> p2 -> age;
        p2 -> next = NULL;
        n++;
        cout << "Would you like to enter a person? (y/n)\n"; cin >> t;
    }
    cout << n << " persons are entered.\n";
    cout << "The list of persons:\n";
    p2 = p1; // we will start from the beginning of the list
    while(p2 != NULL)
    {
        cout << p2 -> name << '\t' << p2 -> age << endl;
        p2 = p2 -> next; //we move to the next person
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

2.14 Classes

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword `class`, with the following format:

```
class class_name
{
    access_specifier_1:  member1;
    access_specifier_2:  member2;
    ...
}
    object_names;
```

where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All of this is very similar to the declaration on data structures. But in this case we can include also functions and members. A new thing is called access specifier. An access specifier is one of the following three keywords: `private`, `public` or `protected`. These specifiers modify the access rights, that the members following them acquire:

- i) `private` members of a class are accessible only from within other members of the same class or from their friends;
- ii) `protected` members are accessible from members of the same class and from their friends, but also from members of their derived classes;
- iii) finally, `public` members are accessible from anywhere, where the object is visible.

By default, all members of a class declared with the `class` keyword have `private` access for all its members. Therefore, any member, that is declared before one other class specifier automatically has `private` access. For example:

```
class CRectangle
{
    int x, y;
public:
    void set_values (int,int);
    int area (void);
}
    rect;
```

Declares a class (i.e., a type) called `CRectangle` and an object (i.e., a variable) of this class called `rect`. This class contains four members: two data members of type `int` (member `x` and member `y`) with `private` access (because `private` is the default access level) and two member functions with `public` access: `set_values()` and `area()`, of which for now we have only included their declaration, not their definition.

After the previous declarations of `CRectangle` and `rect`, we can refer within the body of the program to any of the `public` members of the object `rect` as if they were normal functions

or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we were doing with plain data structures before. For example:

```
rect.set_values (2,5);
S = rect.area();
```

The only members of `rect` that we cannot access from the body of our program outside the class are `x` and `y`, since they have private access and they can only be referred from within other members of that same class.

The complete example of class `CRectangle` is presented in Table 2.46.

The most important new thing in this code is the operator of scope (`::`, two colons) included in the definition of `set_values()`. It is used to define a member of a class from outside the class definition itself.

You may notice, that the definition of the member function `area()` has been included directly within the definition of the `CRectangle` class because of its extreme simplicity, whereas `set_values()` has only its prototype declared within the class, but its definition is outside it. In this outside definition, we must use the operator of scope (`::`) to specify, that we are defining a function, that is a member of the class `CRectangle` and not a regular global function.

The scope operator (`::`) specifies the class which member being declared belongs to, granting exactly the same scope properties, as if this function definition was directly included within the class definition. For example, in the function `set_values()` of the previous code, we have been able to use the variables `x` and `y`, which are private members of class `CRectangle`, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second case it will be a normal (not-inline) class member function, which, in fact, supposes no difference in behaviour.

Table 2.46.

Class `CRectangle`.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: void set_values (int,int); int area () { return (x*y); } };</pre>	<pre>void CRectangle::set_values (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect; rect.set_values (3,4); cout << "area: " << rect.area(); system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	--

Members `x` and `y` have private access (remember, that if nothing else is said, all members of a class defined with keyword `class` have private access). By declaring them private, we deny access to them from anywhere outside the class.

This makes sense, since we have already defined a member function to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. It is difficult to see any utility in protecting these two variables in such simple example as given, but in greater projects it may be very important, that values cannot be modified in unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. Each object has its own variables and, in some way, has also its own function members, that use its object's own variables to operate.

That is the basic concept of object-oriented programming, OOP. Data and functions are both members of the object. We no longer use sets of global variables, that we pass from one function to another as parameters, but, instead, we handle objects, that have their own data and functions embedded as members. Notice, that we have not had to give any parameters to `rect.area()`. The member function directly used the data members of the object `rect`.

Objects often need to initialize variables or assign dynamic memory during the process of their creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function `area()` before having called function `set_values()`? Probably, we would have gotten an undetermined result since the members `x` and `y` would have never been assigned a value.

In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and can not have any return type, not even `void`.

We are going to implement `CRectangle` including a constructor (see Table 2.47). As you can see, the result of this example is identical to the previous one. But now we have removed the member function `set_values()`, and have included instead a constructor, that performs a similar action: it initializes the values of `x` and `y` with the parameters, that are passed to it.

Table 2.47.	Class CRectangle.
<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: CRectangle (int,int); int area () { return (x*y);} };</pre>	<pre>CRectangle::CRectangle (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect (3,4); cout << "rect area: " << rect.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>

Notice, that these arguments are passed to the constructor at the moment, at which the objects of this class are created:

```
CRectangle rect (3,4);
```

Constructors can not be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

You can also see how, neither the constructor prototype declaration (within the class) nor the latter constructor definition, includes a return value; not even void.

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator `delete`.

The destructor must have the same name as the class, but preceded with a tilde sign (`~`) and it must also return no value. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

Like any other function, a constructor can also be overloaded with more than one function that has the same name, but different types or number of parameters.

Remember, that for overloaded functions the compiler will call the one, whose parameters match the arguments, used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one, that matches the arguments passed on the object declaration (see Table 2.48).

Table 2.48.

Class CRectangle.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CRectangle { int x, y; public: CRectangle (); CRectangle (int,int); int area () {return (x*y);} }; CRectangle::CRectangle () { x = 1; y = 1; }</pre>	<pre>CRectangle::CRectangle (int a, int b) { x = a; y = b; } int main(int argc, char *argv[]) { CRectangle rect1 (3,4); CRectangle rect2; cout << "rect1 area: " << rect1.area() << endl; cout << "rect2 area: " << rect2.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	---

In this case, `rect2` was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both `x` and `y` with a value of 1.

Important: Notice, if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rect2;    // right
CRectangle rect2();  // wrong!
```

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So, you have to declare all objects of that class according to the constructor prototypes, you have defined for the class.

But the compiler not only creates a default constructor if you do not specify your own. It provides three special member functions in total, that are implicitly declared if you do not declare your own. These are the copy constructor, the copy assignment operator, and the default destructor. The copy constructor and the copy assignment operator copy all the data, contained in another object to the data members of the current object.

It is perfectly valid to create pointers, that point to classes. We simply have to consider, that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer.

For example:

```
CRectangle *prect;
```

is a pointer to an object of class `CRectangle`. As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (`->`) of indirection. Next, you have a summary of how can you read some pointer and class operators (`*`, `&`, `.`, `->`, `[ ]`) that can be used with classes as presented in Table 2.49. Expression can be read as combinations of pointer and class operators.

Table 2.49. Combinations of pointer and class operators.	
Expression	Can be read as
<code>*x</code>	pointed by <code>x</code>
<code>&x</code>	address of <code>x</code>
<code>x.y</code>	member <code>y</code> of object <code>x</code>
<code>x->y</code>	member <code>y</code> of object pointed by <code>x</code>
<code>(*x).y</code>	member <code>y</code> of object pointed by <code>x</code>
<code>x[0]</code>	first object pointed by <code>x</code>
<code>x[1]</code>	second object pointed by <code>x</code>
<code>x[n]</code>	(n+1)th object pointed by <code>x</code>

2.15 Friendship and inheritance

Private and protected members of a class, usually, cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends. Friends are functions or classes declared with the **friend** keyword.

Declaring an external function as friend of a class, allows this function to have an access to the private and protected members of this class. That is done by declaring a prototype of this external function within the class, and preceding it with the keyword **friend** (see Table 2.50).

The **sum()** and **dif()** functions are **friends** of **complex** class. From within these functions we have been able to access the members **a** and **b** of different objects of type **complex**, which are private members. Notice that neither in the declaration of **sum()** nor in its later use in function **main()** it is considered **sum** a member of class **complex**. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can be used, for example, to conduct operations between two different classes. In general, the use of friend functions is out of an object-oriented programming methodology, so it is better to use members of the same class to perform operations with them whenever it is possible.

Just as there is the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

Consider that friendship is not corresponded if we do not explicitly specify it. For example, if first class is considered as a friend class by second one, but the first class does not consider the second one to be a friend, so the first class can access the protected and private members of the second one but not the reverse way.

Another property of friendship is that it is not transitive: The friend of a friend is not considered to be a friend unless is explicitly specified.

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own ones. For example, we suppose that we want to declare a series of classes that describe different geometrical objects (circle, cylinder). They have certain common properties, such as radius (see Table 2.51).

This could be represented in the world of classes with a class **circle** from which we would derive the **cylinder** class.

Classes that are derived from the others inherit all the accessible members of the base class. That means that if the base class includes the member **A** and we derive it to another class with another member called **B**, the derived class will contain both members **A** and **B**.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

where identifier **derived_class_name** is the name of the derived class and identifier **base_class_name** is the name of the class on which it is based.

Table 2.50. Friend functions.

```

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

class complex
{
    int a, b;
public:
    complex() {a=1; b=1;}
    complex (float x, float y) {a=x; b=y;}
    void set (float x, float y) {a=x; b=y;}
    void print() {cout << a << " + " << b << "*i";}
    friend complex sum (complex, complex);
    friend complex dif (complex, complex);
};

complex sum (complex k, complex l)
{
    complex m;
    m.a = k.a + l.a;
    m.b = k.b + l.b;
    return m;
}

complex dif (complex k, complex l)
{
    complex m;
    m.a = k.a - l.a;
    m.b = k.b - l.b;
    return m;
}

int main(int argc, char *argv[])
{
    complex z1, z2, z3;
    z1.set(7,9);
    z2.set(3,6);
    cout << "z1= ";    z1.print();  cout << endl;
    cout << "z2= ";    z2.print();  cout << endl;
    z3=sum(z1,z2);
    cout << "z1+z2= ";  z3.print();  cout << endl;
    z3=dif(z1,z2);
    cout << "z1-z2= ";  z3.print();  cout << endl;
    system("PAUSE");  return EXIT_SUCCESS;
}

```

The `public` access specifier may be replaced by any one of the other access specifiers - `protected` and `private`. This access specifier limits the most accessible level for the members inherited from the base class.

The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

The objects of the class `cylinder` contain members inherited from `circle`. The `protected` access specifier is similar to `private`. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members. Since we wanted `radius` to be accessible from members of the derived class `cylinder` and not only by members of `circle`, we have used `protected` access instead of `private`.

Table 2.51.

Derived classes.

```

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

class circle
{
protected:
    float r;
public:
    void set(float x) {r=x;}
    float area() { return 3.14*r*r;}
};

class cylinder:protected circle
{
    float h;
public:
    void set(float x, float y) {r=x; h=y;}
    float volume() {return 3.14*r*r*h;}
};

int main(int argc, char *argv[])
{
    circle c;
    c.set(5);
    cout << "area= " << c.area() << endl;
    cylinder cc;
    cc.set(5,3);
    cout << "volume= " << cc.volume() << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.52. Different access types.

Access	Public	Protected	Private
Members of the same class	yes	yes	yes
Members of derived classes	yes	yes	no
Not members	yes	no	no

We can summarize the different access types according to who can access them in the following way as presented in Table 2.52.

”Not members” represent any access from outside the class, such as from `main()`, from another class or from a function. For example see Table 2.53.

A derived class inherits every member of a base class except:

- i) its constructor and its destructor;
- ii) its operator `=()` members;
- iii) its friends.

Although the constructors and destructors of the base class are not inherited by themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

Table 2.53. Derived classes and constructors.

```

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

class circle
{
protected:
    float r;
public:
    circle() {r=1;}
    circle(float x) {r=x;}
    void set(float x) {r=x;}
    float area() {return 3.14*r*r;}
};

class cylinder:protected circle
{
    float h;
public:
    cylinder():circle() {h=1;}
    cylinder (float x, float y) : circle(x) {h=y;}
    void set (float x, float y) {r=x; h=y;}
    float volume() {return 3.1415926 * r * r * h;}
};

int main(int argc, char *argv[])
{
    circle c(5);
    // c.set(5);
    cout << "area= "
         << c.area() << endl;
    cylinder cc(5,3);
    // cc.set(5,3);
    cout << "volume= "
         << cc.volume() << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    derived_constructor_name (parameters) :
        base_constructor_name (parameters) {...}
}
```

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by separating the different base classes with commas in the derived class declaration.

2.16 Polymorphism

One of the key features of derived classes is that a pointer to a derived class is a type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings *Object Oriented Methodologies* to its full potential.

We are going to start by writing a program about the rectangle and the triangle (see Table 2.54).

Table 2.54.

Pointers to base class.

<pre>#include <cstdlib> #include <iostream> using namespace std; class CPolygon { protected: int width, height; public: void set_values (int a, int b) { width=a; height=b; } }; class CRectangle: public CPolygon { public: int area () { return (width * height); } };</pre>	<pre>class CTriangle: public CPolygon { public: int area () { return (width * height / 2); } }; int main(int argc, char *argv[]) { CRectangle rect; CTriangle trgl; CPolygon *ppoly1 = &rect; CPolygon *ppoly2 = &trgl; ppoly1 -> set_values (4,5); ppoly2 -> set_values (4,5); cout << rect.area() << endl; cout << trgl.area() << endl; system("PAUSE"); return EXIT_SUCCESS; }</pre>
--	---

Table 2.55.	Virtual members.
-------------	------------------

```

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

class circle
{
protected:
    float r;
public:
    circle() {r=1;}
    circle(float x) {r=x;}
    void set(float x) {r=x;}
    virtual float area() {return 3.14*r*r;}
};

class cylinder:public circle
{
    float h;
public:
    cylinder ():circle() {h=1;}
    cylinder (float x, float y):circle(x) {h=y;}
    void set (float x, float y) {r=x; h=y;}
    float area()    {return 2 * 3.14 * r * h;}
    float volume() {return      3.14 * r * r * h;}
};

int main(int argc, char *argv[])
{
    circle c(5);
    circle *pp;
    pp=&c;
    cout << "area= " << pp->area() << endl;
    cylinder cc(5,3);
    pp=&cc;
    cout << "cylinder side area= " << pp->area() << endl;
    system("PAUSE");    return EXIT_SUCCESS;
}

```

Table 2.56.	Output of program.
-------------	--------------------

```

area= 78.5
cylinder side area= 94.2
Press any key to continue ...

```

In function `main()`, we create two pointers that point to objects of class `CPolygon` (`ppoly1` and `ppoly2`). Then we assign references to `rect` and `trgl` to these pointers, and as both are objects of the classes derived from `CPolygon`, both are valid assignment operations.

The only limitation in using `*ppoly1` and `*ppoly2` instead of `rect` and `trgl` is that both `*ppoly1` and `*ppoly2` are of type `CPolygon*` and therefore we can only use these pointers to refer to the members that `CRectangle` and `CTriangle` inherit from `CPolygon`. For that reason when we call the `area()` members at the end of the program we have had to use directly the objects `rect` and `trgl` instead of the pointers `*ppoly1` and `*ppoly2`. [45]

A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword `virtual`. For example see Table 2.55. Table 2.56 represents output of program.

Two classes (`circle` and `cylinder`) have the same members: `r`, `set()` and `area()`.

The member function `area()` has been declared as virtual in the base class because it is later redefined in derived class. You can verify if you want that if you remove this virtual keyword from the declaration of `area()` within `circle()`, and then you run the program and the result will be 78.5 instead of 94.2. That happens because instead of calling the corresponding `area()` function for each object (`circle::area()`, `cylinder::area()`). Function `circle::area()` will be called in all cases since the calls are via a pointer whose type is `circle *`.

Therefore, the `virtual` keyword allows a member of a derived class with the same name as one in the base class to be appropriately called from a pointer and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the example above.

A class, that declares or inherits a virtual function, is called a polymorphic class.

2.17 Input/Output with files

C++ provides the following classes to perform input and output (i/o) of characters to/from files:

- i) `ofstream`: stream class to write to files;
- ii) `ifstream`: stream class to read from files;
- iii) `fstream`: stream class to read from and write to files both.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects of these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes, that are related to our file streams.

And, in fact, we can use our file streams in the same way we already used to use `cin` and `cout`, with the only difference, that we have to associate these streams with physical files. Let's see the example (Table 2.57).

This code creates a file called `example.txt` and inserts a sentence into it in the same way we used to do with `cout`, but using the file stream `myfile` instead.

We can also use input/output functions inherited from C. Let us observe the work with text files.

The first operation generally performed on an object of one of these classes is to associate it to a real file.

This procedure is known as to open a file.

Table 2.57. Basic file operations.

```

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

int main(int argc, char *argv[])
{
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    system("PAUSE"); return EXIT_SUCCESS;
}

```

An open file is represented within the program by a stream object (an instantiation of one of these classes, in the previous example it was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

<code>ios::in</code>	open for input operations.
<code>ios::out</code>	open for output operations.
<code>ios::binary</code>	open in binary mode.
<code>ios::ate</code>	set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
<code>ios::app</code>	all output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
<code>ios::trunc</code>	if the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator `OR` (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```

ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);

```

Each one of the `open()` member functions of the classes devoted to file operation `ofstream`, `ifstream` and `fstream` has a default mode, that is used if the file is opened without a second argument as presented in Table 2.58.

Table 2.58. Default mode parameters.

Class	Default mode parameter
<code>ofstream</code>	<code>ios::out</code>
<code>ifstream</code>	<code>ios::in</code>
<code>fstream</code>	<code>ios::in ios::out</code>

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function. The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter, the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

As the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms used to open a file are valid and equivalent. If you want to check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a `bool` value of `TRUE` in the case that indeed the stream object is associated with an open file, or `FALSE` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

When we finished input and output operations on the file we must close it, so that its resources become available again. In order to do that, we have to call the stream's member function `close()`. This member function takes no parameters and its function is to flush the associated buffers and to close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes. In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`. Text file streams are those where we do not include the `ios::binary` flag in their opening mode.

These files are designed to store text and thus all values that we input from or output to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value. Data output operations on text files can be performed in the same way we operated with `cout` (see Table 2.59).

Data input from a file can also be performed in the same way that we did with `cin` (see Table 2.60).

Table 2.59. Writing to a text file.

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

int main(int argc, char *argv[])
{
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n" << "This is another line.\n";
        myfile.close();
    }
    else
        cout << "Unable to open file";
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 2.60. Reading from a text file.

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

int main(int argc, char *argv[])
{
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( myfile.good() )
        {
            getline (myfile,line);
            cout << line << endl;
        };
        myfile.close();
    }
    else
        cout << "Unable to open file";
    system("PAUSE"); return EXIT_SUCCESS;
}
```

This last example reads a text file and prints out its content on the screen. Notice, how we have used a new member function, called `good()`, that returns `TRUE` in the case that the stream is ready for input/output operations. We have created a while loop that finishes, when indeed `myfile.good()` is no longer `TRUE`, which will happen either if the end of the file has been reached or if some other error occurred.

In addition to `good()`, which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a `bool` value):

- `bad()` - returns `TRUE` if a reading or writing operation fails. For example, in the case when we are trying to write to a file that is not open for writing or if the device, where we are trying to write, has no space left.
- `fail()` - returns `TRUE` in the same cases as `bad()`, but also in the case when a format error happens, like when an alphabetical character is extracted, when we are trying to read an integer number.
- `eof()` - returns `TRUE` if a file open for reading has reached the end.
- `good()` - this is the most generic state flag: it returns `FALSE` in the same cases in which calling any of the previous functions would return `TRUE`.

In order to reset the state flags checked by any of these member functions, we have just seen, we can use the member function `clear()`, which takes no parameters.

All i/o streams objects have, at least, one internal stream pointer: `istream`, like `istream`, has a pointer, known as the `get()` pointer, that points to the element to be read in the next input operation.

Objects of `ofstream`, like `ostream`, has a pointer, known as the `put()` pointer, that points to the location, where the next element has to be written.

Finally, objects of `fstream` inherits both, the `get()` and the `put()` pointers, from `iostream` (which is itself derived from both `istream` and `ostream`).

These internal stream pointers, that point to the reading or writing locations within a stream, can be manipulated using the following member functions:

- `tellg()` and `tellp()` - these two member functions have no parameters and return a value of the member type `pos_type`, which is an integer data type representing the current position of the `get` stream pointer (in the case of `tellg`) or the `put` stream pointer (in the case of `tellp`);
- `seekg()` and `seekp()` - these functions allow to change the position of the `get` and `put` stream pointers.

Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions `tellg` and `tellp`: the member type `pos_type`, which is an integer value. The other prototype for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the position of the `get` or `put` pointer is set to an offset value relative to some specific point determined by the parameter `direction`. Offset is of the member type

`off_type`, which is also an integer type; and `direction` is of type `seekdir`, which is an enumerated type (`enum`) that determines the point from where offset is counted from and that can take any of the following values:

- `ios::beg` - offset counted from the beginning of the stream,
- `ios::cur` - offset counted from the current position of the stream pointer,
- `ios::end` - offset counted from the end of the stream.

The following example uses the member functions we have just seen to obtain the size of a file (see Table 2.61).

It is also possible to operate with different portions of data while working with files. Let us create a sample program, which rewrites a matrix from one file to another. We will perform this task in three different ways: operating by characters (see Table 2.62), by strings (see Table 2.63) and by numbers (see Table 2.64).

It is not so easy to obtain the end of line working with numbers. That is why we use for loops to read the data. As it was mentioned earlier, we can also use file input/output function inherited from C. We must include header files `stdio.h` and `conio.h` to use these functions. The process is rather similar. First of all we define a pointer to a file and then open it.

```
FILE * pointertofile;  
pointertofile = fopen (<file name>, <file use TYPE>);
```

After the assignment of the file name, we need to define file usage type one of the following:

- "r" - open for reading, the file has to be created;
- "w" - open for writing, and, if the file already exists, its contents will be cleaned, rewritten;
- "a" - opening file for appending, data will be added at the end of the file, if the file does not exists, it will be created;
- "r+" - open file for reading or writing, the file has to be created;

Table 2.61	Obtaining file size.
------------	----------------------

```
#include <cstdlib>  
#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main(int argc, char *argv[])  
{  
    long begin,end;  
    ifstream myfile ("example.txt");  
    begin = myfile.tellg();  
    myfile.seekg (0, ios::end);  
    end = myfile.tellg();  
    myfile.close();  
    cout << "size is: " << (end-begin) << " bytes.\n";  
    system("PAUSE"); return EXIT_SUCCESS;  
}
```

Table 2.62.

Rewriting the matrix
while working with characters.

```

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

int main(int argc, char *argv[])
{
    char s;
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    while(f.get(s))              // we read till end of file
        g.put(s);               // we write characters to the second file
    f.close();                   // first file is closed
    g.close();                   // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.63.

Rewriting the matrix
while working with strings.

```

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

int main(int argc, char *argv[])
{
    char s[100];
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    while(f.getline(s,100)!=NULL) // we read till end of file
        g << s << endl;        // we write strings to the second file
    f.close();                   // first file is closed
    g.close();                   // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.64.	Rewriting the matrix while working with numbers.
-------------	---

```

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

int main(int argc, char *argv[])
{
    int s;
    fstream f,g;                // the stream objects are created
    f.open("data.txt",ios::in);  // we open the first file for reading
    g.open("result.txt",ios::out); // we open the second file for writing
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            f >> s;              // we read numbers
            g << s << " ";       // we write numbers to the second file
        };
        g << endl;
    };
    f.close();                  // first file is closed
    g.close();                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

"w+" - open file for reading or writing, and if the file already exists, its contents will be rewritten;

"a+" - open for reading and appending, if the file does not exist, it will be created.

We should not forget to close file using `fclose()` function. Let us observe the same three programs with matrix copying while working with these functions. We will perform this task in three different ways: operating by characters (see Table 2.65), by strings (see Table 2.66) and by numbers (see Table 2.67).

Table 2.65.

Rewriting the matrix
while working with characters using C functions.

```

#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

int main(int argc, char *argv[])
{
    char s;
    FILE *f, *g;                // the pointers are declared
    f=fopen("data.txt","r");     // we open the first file for reading
    g=fopen("result.txt","w");   // we open the second file for writing
    while((s=getc(f))!=EOF)      // we read characters
        putc(s,g);              // we write characters to the second file
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.66.

Rewriting the matrix
while working with strings using C functions.

```

#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;

int main(int argc, char *argv[])
{
    char s[100];
    FILE *f, *g;                // the pointers are declared
    f=fopen("data.txt","r");     // we open the first file for reading
    g=fopen("result.txt","w");   // we open the second file for writing
    while(fgets(s,100,f)!=NULL) // we read strings
        fputs(s,g);             // we write strings to the second file
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Table 2.67.

Rewriting the matrix
while working with numbers using C functions.

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

int main(int argc, char *argv[])
{
    int s;
    FILE *f, *g;                // the pointers are declared
    f = fopen("data.txt","r");   // we open the first file for reading
    g = fopen("result.txt","w"); // we open the second file for writing
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            fscanf (f,"%d",&s); // we read numbers
            fprintf(g,"%d ",s); // we write numbers to the second file
        };
        fprintf(g,"\n");
    };
    fclose(f);                  // first file is closed
    fclose(g);                  // second file is closed
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Chapter 3

Algorithms

3.1 Dynamical arrays

Arrays represent the programming features for manipulation of data sets. When related data items are extended in one or several dimensions (measured by natural number) we have the structures representing mathematical vectors, plane surfaces, cubes, etc. All elements in arrays have the same type and the same inner structure. Each element of array is represented by the name of array where the position of element in array is presented by index in parentheses: []. Using the arrays allows to programmers to rewrite the repeatable code in a user-friendly manner.

Two different techniques of array organization - static and dynamic - could be used due to memory allocation processes.

Static array. Static array could be created in any place of program using declaration of array with fixed number of items. For example, static array `iXXX` of integer type (4B) is declared with memory allocation for ten elements (4x10=40B):

```
int iXXX[10];
```

Static array belongs to the simple methods of item manipulation and is useful for beginners. But it is necessary to point out that memory allocation for static array occurs with the declaration of array and is valid until total end of program. It means that so called significant memory leak could occur, even in case of large array usage.

Dynamic array. Dynamic array could be created in any place of program using special programming technique: a) declaration of pointer of requested data type; b) allocation of array with fixed number of items using operator `new`. For example, dynamic array `iYYY` of integer type will be created and memory for ten integers (4x10=40B) will be allocated:

```
int *iYYY;  
iYYY = new int[10];
```

If you need to allocate memory for one element, you can use another notation which allows to allocate memory and to initialize the variable also - by value in parenthesis (), for example, 1687:

Table 3.1. Arrays.

Method	Name	Stable	File name
Dynamic	1D array	Yes	dim01.cpp
Dynamic	2D array	Yes	dim02.cpp
Dynamic	3D array	Yes	dim03.cpp

```
int *iZZZ;
iZZZ = new int(1687);
```

The size of array (static as well as dynamic) could be measured by using operator `sizeof()` which returns natural number in Bytes corresponding to the amount of allocated memory.

```
cout << " Size of iYYY = " << sizeof (iYYY) << endl;
```

Memory allocation for dynamic array occurs with the declaration of array (using operator `new`) and is valid until total end of program as well as until using operator `delete` in any place:

```
delete iYYY;
```

It is obvious that term *dynamic* represents the programming technique for array creating and array destroying in any place according to the program requirement (not only at the end as in case of *static*).

In this section, typical dynamic one-dimensional (1D), two-dimensional (2D) and three-dimensional (3D) arrays are presented - see Table 3.1. Another similar implementations are presented in topic Schildt [40], [42] and Alexandrescu [2] publications.

3.1.1 One-dimensional (1D) array

Table 3.2 represents programming implementation of one-dimensional (1D) array. Table 3.3 represents the output of the program. Programming code contains two part: `main()` function and interface of class `DDD1` together with interface realizations through methods.

Function `main()` contains three sentences made-up in *Object-Oriented programming, OOP* manner: creating object, input of data, and output of data.

```
DDD1 argus;
argus.input();
argus.show();
```

Interface of class `DDD1` contains descriptions of variables as well as methods. Variable section contains declaration of private variables (pointer to the array `*iA` and size of 1D array `nA`, both variables of integer type). Method section includes declaration of prototypes of special methods `DDD1::DDD1()` and `DDD1::~~DDD1()` (constructor and destructor, respectively) and methods `void DDD1::show()` and `void DDD1::input()`.

Constructor `DDD1::DDD1()` is devoted for allocation of memory of dynamic array. The loop `do..while()` is made for *online* input of array size (using operator `cin()`). One condition must be satisfied: variable `nA` (corresponding to array size) must be positive non-zero number. After satisfaction of the mentioned condition, operator `new` allocates the memory of 1D array `iA` :

```
iA = new int [nA];
```

Table 3.2. 1D array.

Program dim01.cpp.

```

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

class DDD1
{
private:
    int *iA;
    int nA;
public:
    DDD1(); // constructor
    ~DDD1(); // destructor
    void show();
    void input();
};

DDD1::DDD1() // constructor
{
    do
    {
        cout << "1D array: size = ";
        cin >> nA;
    }
    while (nA <= 0);
    iA = new int [nA];
}

int main(int argc, char *argv[])
{
    DDD1 argus;
    argus.input();
    argus.show();
    system("PAUSE");
    return EXIT_SUCCESS;
}

DDD1::~~DDD1() // destructor
{
    delete iA;
}

void DDD1::show()
{
    cout << "Output of 1D array:\n";
    cout << "      ";
    for (int j=0; j<nA; j++)
        cout << j << " ";
    cout << endl;
    cout << "      ";
    for (int j=0; j<nA; j++)
        cout << "####";
    cout << endl;
    cout << "      # ";
    for (int i=0; i<nA; i++)
    {
        cout << iA[i] << " ";
    };
    cout << endl;
}

void DDD1::input()
{
    int j;
    cout << "Input of 1D array:\n";
    for (int i=0; i<nA; i++)
    {
        cout << i << " : ";
        cin >> j; iA[i]=j;
    };
}

```

Also destructor `DDD1::~~DDD1()`, in opposite to constructor `DDD1::DDD1()`, erases the dynamic array using operator `delete`:

```
delete iA;
```

Method `void DDD1::input()` contains one loop:

Table 3.3.	Output of program dim01.cpp.
<pre> 1D array: size = 4 Input of 1D array: 0 : 8 1 : 8 2 : 9 3 : 9 </pre>	<pre> Output of 1D array: 0 1 2 3 ##### # 8 8 9 9 Press any key to continue ... </pre>

```

for (int i=0; i<nA; i++)
{
    ...
};

```

where *on-line* data input (using input stream `cin`) is realized - see Table 3.3. Output pairs - 1D array index and array member separated by colon (:) - are presented on the left.

Method `void DDD1::show()` is devoted for data output purposes: two loops of the same type allows to print index of array and member of array, as presented in Table 3.3 on right. Array index always starts from zero, according to the general C/C++ standard.

Table 3.4 represents memory allocation of `iA` array. According to the variable declaration, `iA[ ]` array was allocated of integer type (standard amount of memory 4B for each member) - see Fig. 3.1. *Resource allocator* (OS program, working in *Kernel* mode) will present the initial address (pointer to the array `iA` or pointer to the zeroth member `&iA[0]`). Possible initial address (for example 0012FF3C) is presented here in hexadecimal format. Operator `new` requests memory starting from mentioned, for each member four Bytes. In all modern operating systems (OS, for example, *Windows7*, *Linux*, etc) you can not request the memory manually due to the strong control restriction, *Resource allocator* will do it automatically.

Table 3.5 represents memory paging for 4GB memory distribution as a typical distribution for Pentium computers. All advanced programs *Win32-api* created by user can operate in address space $4\text{MB} \div 2\text{GB}$.

Several aspects of memory allocation are widely described by Allen [19], by Silberschatz [14], by Sutter [3].

Table 3.4. 1D array.	Memory allocation for <code>iA</code> array.		
Pointer	Memory for integer, 4B portion	Member index	Members, <code>iA[]</code>
0012FF7C	F7C, F7D, F7E, F7F	0	8
0012FF80	F80, F81, F82, F83	1	8
0012FF84	F84, F85, F86, F87	2	9
0012FF88	F88, F89, F8A, F8B	3	9

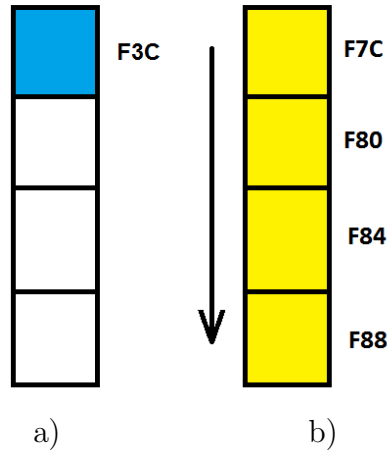


Fig. 3.1. Memory allocation for 1D array:

a) declaration of pointer using `int *iA;`b) creation of 1D array using `iA=new int[4].`Arrow visualizes allocation process using operator `new` .

3.1.2 Two-dimensional (2D) array

Table 3.6 represents programming implementation of two-dimensional (2D) array. Table 3.7 represents output of program.

Programming code contains two parts: `main()` function and interface of class `DDD2` together with interface realizations through methods. Function `main()` contains three sentences made-up in OOP manner: creating object, input of data, and output of data.

Interface of class `DDD2` contains descriptions of variables as well as methods. Variable section contains declaration of private variables (double pointer to the array `**iB` or `*(**iB)` and two numbers expressing size of 2D array, all three variables of integer type).

The method section includes declaration of prototypes of special methods `DDD2::DDD2()` and `DDD2::~~DDD2()` (constructor and destructor, respectively) and methods `void DDD2::show()` and `void DDD2::input()`.

Constructor `DDD2::DDD2()` is devoted for allocation of memory of dynamic array. Two loops `do..while()` are made for *online* input of array sizes (using input stream `cin`). One condition must be satisfied: variables `nA` and `nB` (array size) must be positive non-zero number. After satisfaction of the mentioned condition, several processes of memory allocation begin. The first step is devoted for memory allocation for 1D array of reference `iB[ ]` using operator `new`:

Table 3.5. Memory paging.		4GB.
Area	Objects	Comments
3GB÷4GB	Kernel, GDI, Ring0	OS only
2GB÷3GB	Commdlg.dll, windows dialog	Shared
4MB÷2GB	Win32-api, Win16-api	Shared
1MB÷4MB	Win16-api	advanced
0MB÷1MB	MS-DOS-api, real mode drivers	advanced

Table 3.6. 2D array.

Program dim02.cpp.

```

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

// 2D dynamic array
class DDD2
{
private:
    int **iB;
    int  nA,nB;
public:
    DDD2(); // constructor
    ~DDD2(); // destructor
    void show();
    void input();
};

void DDD2::input()
{
    int m;
    cout << "Input of 2D array:\n";
    for (int i=0; i<nA; i++)
    {
        for (int j=0; j<nB; j++)
        {
            cout << i << "  "
                << j << "  : ";
            cin >> m;
            iB[i][j]=m;
        }
    }
}

int main(int argc, char *argv[])
{
    DDD2 argus;
    argus.input();
    argus.show();
    system("PAUSE");
    return EXIT_SUCCESS;
}

DDD2::DDD2() // constructor
{
    do
    {
        cout << "2D array: size1 = ";
        cin >> nA;
    }
    while (nA <= 0);
    do
    {
        cout << "2D array: size2 = ";
        cin >> nB;
    }
    while (nB <= 0);
    iB = new int*[nA];
    for (int i=0; i<nA; i++)
        iB[i] = new int[nB];
}

DDD2::~DDD2() // destructor
{
    delete iB;
}

void DDD2::show()
{
    cout << "Output of 2D array:\n";
    cout << "      ";
    for (int j=0; j<nB; j++)
        cout << j << "  ";
    cout << endl;
    cout << "      ";

    for (int j=0; j<nB; j++)
        cout << "####";
    cout << endl;
    for (int i=0; i<nA; i++)
    {
        cout << i << "  #  ";
        for (int j=0; j<nB; j++)
        {
            cout << iB[i][j] << "  " ;
        }
        cout << endl;
    }
}

```

Table 3.7.	Output of program dim02.cpp.
<pre> 2D array: size1 = 2 2D array: size2 = 3 Input of 2D array: 0 0 : 7 0 1 : 7 0 2 : 8 1 0 : 8 1 1 : 6 1 2 : 6 </pre>	<pre> Output of 2D array: 0 1 2 ##### 0 # 7 7 8 1 # 8 6 6 Press any key to continue ... </pre>

```
iB = new int*[nA];
```

The second step is realized in 1D loop, where operator `new` creates the 2D array `iB[ ][ ]` for each reference `iB[ ]` - see Fig. 3.2.

```

for (int i=0; i<nA; i++)
    iB[i] = new int[nB];

```

In that case, full allocation process is realized through two loops, and two indexes allow to address and access the each member of 2D array.

Also destructor `DDD2::~DDD2()`, in opposite to constructor `DDD2::DDD2()`, erases the dynamic array using operator `delete`:

```
delete iB;
```

Method `void DDD2::input()` contains loop in loop:

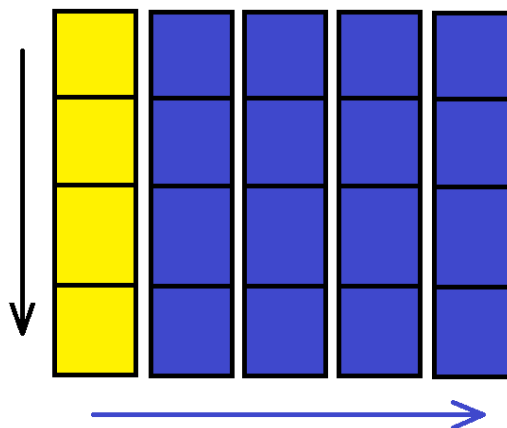


Fig. 3.2. Memory allocation for 2D array.

Declaration of pointer using `int **iB;` - top left, the same as in Fig. 3.1.

Creation of 1D array using `iB=new int*[4];` - vertical arrow, the same as in Fig. 3.1.

Creation of 2D array using `iB[]=new int[5];` - horizontal arrow.

```

for (int i=0; i<nA; i++)
{
    for (int j=0; j<nB; j++)
    {
        ...
    }
}

```

where *online* data input (using input stream `cin`) is realized - see Table 3.5. Output triads - two indexes of 2D array and array member separated by colon (:) - are presented on the left.

Method `void DDD2::show()` is devoted for data output purposes: several loops of the same type allows to print indexes of array and member of array, as presented in Table 3.7, on the right. Two indexes, according to the matrix notation, could be named as *column-index* and *row-index*.

3.1.3 Three-dimensional (3D) array

Tables 3.8a, 3.8b represent programming implementation of three-dimensional (3D) array. Table 3.9 represents output of program. Programming code contains two parts: the `main()` function and interface of class `DDD3` together with interface realizations through methods.

Function `main()` contains three sentences made-up in OOP manner: creating object, input of data, and output of data.

Interface of class `DDD3` contains descriptions of variables as well as methods. Variable section contains declaration of private variables as a triple pointer to the array `***iC` or `*(**iC)` and three numbers of integer type `nA`, `nB`, `nC` representing 3D array size. Method section includes declaration of prototypes of special methods `DDD3::DDD3()` and `DDD3::~~DDD3()` (constructor and destructor, respectively) and methods `void DDD3::show()` and `void DDD3::input()`.

Constructor `DDD3()` is devoted for allocation of memory of dynamic array. The loop `do..while()` is made for *online* input of array size (using input stream `cin`). Variables `nA`, `nB` and `nC` (array sizes) must be positive non-zero number. After satisfaction of mentioned condition operator `new` allocates the memory of dynamic array in tree steps.

The first step is devoted for creating 1D array: operator `new` creates the 1D array `iC[ ]` for variable `iC` (two stars operation) :

```
iC = new int**[nA];
```

The second step is realized in 1D loop, where operator `new` creates the 2D array `iC[ ][ ]` for each reference `iC[ ]`.

```

for (int i=0; i<nA; i++)
    iC[i] = new int*[nB];

```

The third step is realized in 2D loop, where operator `new` creates the three-dimensional 3D array `iC[ ][ ][ ]` for each reference `iC[ ][ ]`.

Table 3.8a. 3D array.

Program dim03.cpp.

```

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

class DDD3
{
private:
    int ***iC;
    int nA,nB,nC;
public:
    DDD3(); // constructor
    ~DDD3(); // destructor
    void show();
    void input();
};

void DDD3::show()
{
    cout << "Output of 3D array:\n";
    for (int k=0; k<nC; k++)
    {
        cout << k << " layer\n";
        cout << " ";
        for (int j=0; j<nB; j++)
            cout << j << " ";
        cout << endl;
        cout << " ";

        for (int j=0; j<nB; j++)
            cout << "####";
        cout << endl;
        for (int i=0; i<nA; i++)
        {
            cout << i << " # ";
            for (int j=0; j<nB; j++)
            {
                cout << iC[i][j][k]
                    << " ";
            };
            cout << endl;
        };
    }; // end for k
}

DDD3::DDD3() // constructor
{
    do
    {
        cout << "3D array: size1 = ";
        cin >> nA;
    }
    while (nA <= 0);
    do
    {
        cout << "3D array: size2 = ";
        cin >> nB;
    }
    while (nB <= 0);
    do
    {
        cout << "3D array: size3 = ";
        cin >> nC;
    }
    while (nC <= 0);
    iC = new int**[nA];
    for (int i=0; i<nA; i++)
        iC[i] = new int*[nB];
    for (int i=0; i<nA; i++)
    {
        for (int j=0; j<nB; j++)
        {
            iC[i][j] = new int[nC];
        };
    };
}

DDD3::~DDD3() // destructor
{
    delete iC;
}

```

Table 3.8b. 3D array.

Program dim03.cpp.

```

int main(int argc, char *argv[])
{
    DDD3 argus;
    argus.input();
    argus.show();
    system("PAUSE");
    return EXIT_SUCCESS;
}

void DDD3::input()
{
    int m;
    cout << "Input of 3D array:" << endl;
    for (int k=0; k<nC; k++)
    {
        cout << k << " layer" << endl;
        for (int i=0; i<nA; i++)
        {
            for (int j=0; j<nB; j++)
            {
                cout << i << " "
                    << j << " : ";
                cin >> m;
                iC[i][j][k]=m;
            }
        }
    }
}

```

```

for (int i=0; i<nA; i++)
{
    for (int j=0; j<nB; j++)
    {
        iC[i][j] = new int[nC];
    }
}

```

In that case full allocation process is realized through three loops and three indexes allows to address the each member of 3D array.

Also destructor `DDD3::~DDD3()`, in opposite to constructor `DDD3::DDD3()`, erases the dynamical array using operator `delete`:

```
delete iC;
```

Method `void DDD3::input()` contains three loops:

```

for (int k=0; k<nC; k++)
{
    for (int i=0; i<nA; i++)
    {
        for (int j=0; j<nB; j++)
        {
            ...
        }
    }
}

```

Table 3.9.	Output of program dim03.cpp.
<pre> 3D array: size1 = 2 3D array: size2 = 3 3D array: size3 = 2 Input of 3D array: 0 layer 0 0 : 1 0 1 : 1 0 2 : 1 1 0 : 1 1 1 : 1 1 2 : 1 1 layer 0 0 : 2 0 1 : 2 0 2 : 2 1 0 : 2 1 1 : 2 1 2 : 2 </pre>	<pre> Output of 3D array: 0 layer 0 1 2 ##### 0 # 1 1 1 1 # 1 1 1 1 layer 0 1 2 ##### 0 # 2 2 2 1 # 2 2 2 Press any key to continue ... </pre>

where *online* data input (using input stream `cin`) is realized - see Table 3.9. Output quarterons - three indexes of 3D array and array member separated by colon (:) - are presented on the left.

Method `void DDD3::show()` is devoted for data output purposes: three loops of the same type allows to print index of array and member of array, as presented in Table 3.9, on the right. Using names from matrix notation, it is convenient to name first index as column index, second index - as row index, and third index - as layer index.

3.2 Matrices

C takes the point of view
that the programmer is always right.

Michael DeCorte

Matrix could be described as a N -dimensional array of numbers (in some cases, also expressions, symbols, large complicated objects, such as tables etc). Usually, when we talk about matrices, two-dimensional (2D) array, arranged in rows and columns, takes place. Each item of matrix (called an element) is individual, and each element is managed using the two indexes belonging to row and column.

Matrices are very important as a mathematical tool in different scientific fields. In computer graphics, matrices are used for realization of visual reality effects: projecting a three-dimensional (3D) image onto a two-dimensional (2D) screen, and creating semi-realistic motion

in many computer games. In physics, matrices are used to describe optical effects of light propagation in media, etc. Usage of matrices in algorithms allows simplifying the computations in significant order. Several aspects of matrix usage are described by Yang [53].

3.2.1 Definition and notation

Mathematically, matrix notation is very simple. Matrix A (m -by- n matrix) represents a table with determined m rows (index i varies from 0 until $(m-1)$) and n columns (index j varies from 0 until $(n-1)$). Items (elements of a matrix) are often denoted by a variable with two subscripts: for example $a_{1,1}$:

$$A_{m,n} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix} \quad (3.1)$$

Formal rules. Copying the matrix to matrix belongs to one of the simplest mathematical operations. If A is an m -by- n matrix and E is an m -by- n matrix then operation of copying can be performed:

$$E_{i,j} = A_{i,j} \quad (3.2)$$

Matrices of the same size (m -by- n matrix) can be added or subtracted - element by element with the same indexes. The sum C of two m -by- n matrices as well as the difference D of two m -by- n matrices A and B are calculated entrywise:

$$C_{i,j} = (A + B)_{i,j} = A_{i,j} + B_{i,j} \quad (3.3)$$

$$D_{i,j} = (A - B)_{i,j} = A_{i,j} - B_{i,j} \quad (3.4)$$

Fig. 3.3 and Fig. 3.4 represent schemes of the member-to-member operations, where addition and subtraction can be performed practically.

Multiplication of two matrices $[A,B]$ is more complicated. This operation could be performed in case only if the number of columns of the left matrix A is equal to the number of rows of the right matrix B . If A is an m -by- n matrix and B is an n -by- l matrix, then their matrix product F is the m -by- l matrix.

2	3	+	7	9	=	2+7	3+9
4	1		4	5		4+4	1+5
2	5		1	2		2+1	5+2

Fig. 3.3. Matrix addition.

2	3	-	7	9	=	2-7	3-9
4	1		4	5		4-4	1-5
2	5		1	2		2-1	5-2

Fig. 3.4. Matrix subtraction.

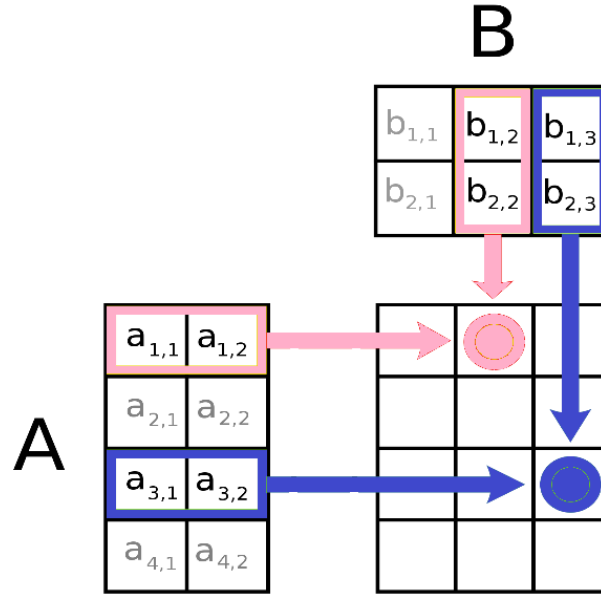


Fig. 3.5. Multiplication scheme.

Items of F matrix are given by dot product of the corresponding row of A and the corresponding column of B :

$$F_{i,j} = [A, B]_{i,j} = \sum_{r=1}^n A_{i,r} B_{r,j} \quad (3.5)$$

Fig. 3.5 and Fig. 3.6 represents scheme of the member-to-member operations, where multiplication can be performed practically.

In this section, several programming implementations of matrix copying, addition, subtraction and multiplication are presented - see Table 3.10. Two programs written in different programming style are observed and discussed. Two different programming style were used: function-like style and object-like style. Actually, object-like style is by far more preferable - due to simplified user-friendly notation.

$$\begin{array}{|c|c|} \hline \mathbf{2} & \mathbf{3} \\ \hline \mathbf{4} & \mathbf{1} \\ \hline 2 & 5 \\ \hline \end{array} * \begin{array}{|c|c|c|} \hline 1 & 5 & 3 \\ \hline 2 & 9 & 6 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \mathbf{2*1+3*2} & \mathbf{2*5+3*9} & \mathbf{2*3+3*6} \\ \hline \mathbf{4*1+1*1} & \mathbf{4*5+1*9} & \mathbf{4*3+1*6} \\ \hline 2*1+5*2 & 2*5+5*9 & 2*3+5*6 \\ \hline \end{array}$$

Fig. 3.6. Matrix multiplication.

Table 3.10. Matrices.

Method	Name	Stable	File name
OOP-1	function-style	Yes	mmm01.cpp
OOP-2	object-style	Yes	mmm02.cpp

3.2.2 Matrix manipulation: OOP in first style

Table 3.11 represents `main()` function of matrix manipulation buildt-up in traditional OOP function-like style. Tables 3.12a,b,c,d,e represent header file containing interface and realization of class `MATRIX1`. Table 3.13 represents output of mentioned program.

Function `main()` represents several routines for matrix manipulation. Declaration of class object is related to the description of matrix parameters: number of rows, number of column and matrix name.

```
// matrix: 3 rows, 2 columns, A
MATRIX1 A(3,2,'A');
```

Methods `void MATRIX1::Input()` and `void MATRIX1::Show()` realize data input and data output. Methods `void MATRIX1::Addition()` and similar `void MATRIX1::Subtraction()`, also `void MATRIX1::Copy()` and complicated `void MATRIX1::Multiplication()` realize corresponding mathematical operations.

Table 3.11. Matrix.	Program mmm01.cpp.
<pre>#include <cstdlib> #include <iostream> #include <iomanip> using namespace std; #include "mmm01.h" int main() { // MATRIX: 3 rows, 2 columns MATRIX1 A(3,2,'A'); A.Input(); A.Show(); MATRIX1 B(3,2,'B'); B.Input(); B.Show(); // Addition, Subtraction MATRIX1 C(3,2,'C'); C.Addition(A,B); C.Show(); MATRIX1 D(3,2,'D'); D.Subtraction(A,B); D.Show(); system("PAUSE"); // Copying and multiplication MATRIX1 E(3,2,'E'); E.Input(); E.Show(); MATRIX1 F(3,2,'F'); F.Copy(E); F.Show(); MATRIX1 G(2,3,'G'); G.Input(); G.Show(); MATRIX1 H(3,3,'H'); H.Multiplication(F,G); H.Show(); system("PAUSE"); return EXIT_SUCCESS; }</pre>	

Table 3.12a.

Program mmm01.h.

```

class MATRIX1
{
protected:
    int      row_t,
           column_t;
    char    name_t;
private:
    int **arrayPtr2;
    bool process;
public:
    MATRIX1(int n_row, int n_col, char c_name); // constructor
    ~MATRIX1();                               // destructor
    void Initial_value(int ia);
    void Input();
    void Show();
    void          Copy (MATRIX1 &X);
    void          Addition (MATRIX1 &X, MATRIX1 &Y);
    void    Subtraction (MATRIX1 &X, MATRIX1 &Y);
    void    Multiplication (MATRIX1 &X, MATRIX1 &Y);
    bool Is_allowed(int z1, int x1, int y1,                // rows
                   int z2, int x2, int y2, char process); // columns
}; // end class MATRIX

void MATRIX1::Addition(MATRIX1 &X, MATRIX1 &Y)
// Addition : Z=X+Y
{
    bool b1; int s1,s2,s3;
    b1 = Is_allowed( row_t, X.row_t, Y.row_t,
                    column_t, X.column_t, Y.column_t, '+');
    if (!b1) exit(0);

    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1=X.arrayPtr2[i1][i2];
            s2=Y.arrayPtr2[i1][i2];
            s3 = s1 + s2;
            arrayPtr2[i1][i2] = s3;
        }
    };
    cout << name_t << " = " << X.name_t << " + "
          << Y.name_t << endl << endl;
}

```

Table 3.12b.

Program mmm01.h.

```

void MATRIX1::Subtraction(MATRIX1 &X, MATRIX1 &Y)
// Substraction : Z=X-Y
{
    bool b1;  int s1,s2,s3;
    b1 = Is_allowed(   row_t,   X.row_t,   Y.row_t,
                      column_t, X.column_t, Y.column_t, '-');
    if (!b1) exit(0);

    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1=X.arrayPtr2[i1][i2];
            s2=Y.arrayPtr2[i1][i2];
            s3 = s1 - s2;
            arrayPtr2[i1][i2] = s3;
        };
    };
    cout << name_t << " = " << X.name_t << " - "
          << Y.name_t << endl << endl;
}

void MATRIX1::Copy(MATRIX1 &X)
// copy : Z=X
{
    bool b1; int s1;
    b1 = Is_allowed(   row_t,   X.row_t, 0,
                      column_t, X.column_t, 0, 'c');
    if (!b1) exit(0);

    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1 = X.arrayPtr2[i1][i2];
            arrayPtr2[i1][i2] = s1;
        };
    };
    cout << name_t << " = " << X.name_t << endl << endl;
}

```

Table 3.12c.

Program mmm01.h.

```

void MATRIX1::Multiplication(MATRIX1 &X, MATRIX1 &Y)
{
    bool b1;  int temp_s, s1,s2,s3;
    b1 = Is_allowed(   row_t,   X.row_t,   Y.row_t,
                      column_t, X.column_t, Y.column_t, '*');
    if (!b1) exit(0);

    for (int i=0; i<X.row_t; i++)
    {
        for (int k=0; k<Y.column_t; k++)
        {
            temp_s = 0;
            for (int j=0; j<X.column_t; j++)
            {
                s1 = X.arrayPtr2[i][j];
                s2 = Y.arrayPtr2[j][k];
                s3 = s1 * s2;
                temp_s += s3;
            }; //end for j
            arrayPtr2[i][k] = temp_s;
        }; //end for k
    }; //end for i
    cout << name_t << " = " << X.name_t << " * "
          << Y.name_t << endl << endl;
}

MATRIX1::MATRIX1( int  n_row,      int  n_col,   char  c_name) :
                  row_t (n_row), column_t (n_col), name_t (c_name)
// constructor
{
    arrayPtr2 = new int*[n_row];
    for (int i=0; i<n_row; i++)
        arrayPtr2[i] = new int[n_col];
    process = true;
    cout << " MATRIX " << name_t
          << " ( " <<   row_t << " X "
          << column_t << " )\n\n";
    // MATRIX validation
    Initial_value(0);
}

MATRIX1::~MATRIX1() // destructor
{
    delete arrayPtr2;
}

```

Table 3.12d.

Program mmm01.h.

```

bool MATRIX1::Is_allowed(int z1, int x1, int y1,           // rows
                          int z2, int x2, int y2, char process) // columns
// Z = X (process) Y
{
    char *r1 = "addition";      char *r2 = "substraction";
    char *r3 = "multiplication"; char *r5 = "copy";
    char *nea= ". Different parameters of matrices ...\n"; char *ne = "No ";
    switch (process)
    {
        case ('c'): // matrix copy
            if ((z1 == x1) && (z2 == x2)) return true;
            else
            {
                cout << z1 << " " << x1 << endl;
                cout << z2 << " " << x2 << endl;
                cout << ne << r5 << nea;    system("PAUSE"); return false;
            };
            break; // *****
        case ('*'): // multiplication
            if ((x1 == y2) && (x1 == z1) && (x1 == z2) // square
                && (x2 == y1)) return true; // any
            else
            {
                cout << x1 << " " << y2 << " "
                    << z1 << " " << z2 << " square\n";
                cout << x2 << " " << y1 << " part\n";
                cout << ne << r3 << nea;    system("PAUSE"); return false;
            };
            break; // *****
        case ('+'): // addition
            if ((x1 == y1) && (y1 == z1) &&
                (x2 == y2) && (y2 == z2)) return true;
            else
            {
                cout << ne << r1 << nea;    system("PAUSE"); return false;
            };
            break; // *****
        case ('-'): // subtraction
            if ((x1 == y1) && (y1 == z1) &&
                (x2 == y2) && (y2 == z2)) return true;
            else
            {
                cout << ne << r2 << nea;    system("PAUSE"); return false;
            };
            break;
    }; // end switch
    return false; // no processes?
}

```

Table 3.12a represents an interface of class **MATRIX1**. Class contains several types of class members: protected, private and public.

Private members (accessible for class methods only !) represents a double pointer ****arrayPtr2** for 2D array realization of integer type and variable **process** of boolean type - for allowing/forbidding of operation.

Protected members (like private but including some exceptions) represents parameters of matrix - number of rows and columns **row_t** and **column_t** of integer type and name of matrix **name_t** of character type. Such elements are accessible in the class which inherits the present class. In the inherited class, such elements are valid according to the rules for private members.

Public members (accessible without any restrictions) represent special methods (constructor **MATRIX1::MATRIX1()** and destructor **MATRIX1::~~MATRIX1()**) as well as user-created methods for input **void MATRIX1::Input()** and output **void MATRIX1::Show()**. Matrix initialization is realized in method **void MATRIX1::Initial_value()**. Several methods are presented for matrix manipulation **void MATRIX1::Copy()**, **void MATRIX1::Addition()**, very similar to previous **void MATRIX1::Subtraction()**, **void MATRIX1::Multiplication()**, and allowing/forbidding of operation **void MATRIX1::Is_allowed()**.

Generally, class **MATRIX1** uses dynamic memory allocation for 2D array, as it was described previously in subsection 3.1.2 (programming code presented in Table 3.6). Table 3.12c represents constructor **MATRIX1::MATRIX1()** for memory allocation purposes.

Matrix memory requesting occurs on time of object initialization (see function **main()**). Significant values - number of row, number of column, matrix name - must be transmitted from function **main()** to class constructor **MATRIX1::MATRIX1()**. Usage of protected class members is devoted for such complicated data exchange.

Several parameters - number of row **n_row**, number of column **n_col**, name of matrix **c_name** - are local variables in constructor **MATRIX1::MATRIX1()**. In order to use it in class namespace, association to the protected class members is necessary using column **(:)** technique.

```
// constructor
MATRIX1(    int n_row,        int n_col,    char c_name) :
            row_t (n_row), column_t (n_col), name_t (c_name)
```

After mentioned operation, local variable **n_row** corresponds to class protected variable **row_t**, local variable **n_col** - to class protected variable **column_t**, local variable **c_name** - to class protected variable **name_t**.

Allocation of memory of 2D dynamical array could be done in two steps. The first step is devoted for creating 1D array **arrayPtr2[]** of references:

```
arrayPtr2 = new int*[n_row];
```

The second step is realized in 1D loop, where operator **new** creates the 2D array **arrayPtr2[][]** for each reference **arrayPtr2[]**.

```
for (int i=0; i<n_row; i++)
    arrayPtr2[i] = new int[n_col];;
```

In that case, full allocation process is realized through two loops, and two indexes allows to address the each member of 2D array.

Table 3.12e.

Program mmm01.h.

```

void MATRIX1::Input()
{
    int ia;  cout << name_t << " matrix input\n";
    for (int i1=0; i1<row_t; i1++)
    {
        cout << (i1+1) << " row : \n";
        for (int i2=0; i2<column_t; i2++)
        {
            cout << (i2+1) << " : " ; cin >>ia;
            arrayPtr2[i1][i2]=ia;
        };
    };
    cout << endl;
}

void MATRIX1::Initial_value(int ia)
{
    for (int i1=0; i1<row_t; i1++)
        for (int i2=0; i2<column_t; i2++)
            arrayPtr2[i1][i2] = ia;
}

void MATRIX1::Show()
{
    cout << name_t << " MATRIX : (operation - "
        << process << " )" << endl;
    cout << " ";
    for (int i2=0; i2<column_t; i2++)
        cout << setw(4) << (i2+1);
    cout << endl;
    cout << " -----" << endl;

    for (int i1=0; i1<row_t; i1++)
    {
        cout << setw(4) << (i1+1) << " % ";
        for (int i2=0; i2<column_t; i2++)
            cout << setw(4) << arrayPtr2[i1][i2];
        cout << endl;
    };
    cout << endl;
}

```

After finishing the memory allocation process, matrix initialization could be performed. This operation is necessary, even if the matrix output would be performed without any mathematical manipulation. Elements of matrix will be filled by random established bits, which can create some difficulties for output processes. Class method `void MATRIX1::Initial_value()`, see Table 3.12e, requires any integer for input (for example, zero, million, etc), which will be assigned to each member of 2D array. Assignment operation will be done using loop in loop.

```
for (int i1=0; i1<row_t; i1++)
{
    for (int i2=0; i2<column_t; i2++)
    {
        ...
    }
}
```

Also destructor `MATRIX1::~~MATRIX1()`, which works in opposite to constructor `MATRIX1::MATRIX1()`, erases the dynamic array using operator `delete`:

```
delete arrayPtr2;
```

Table 3.12e represents two class methods for input/output realization `void MATRIX1::Input()`, `void MATRIX1::Show()`, which are very similar to previously described methods belonging to the `DDD2` class (2D arrays, see subsection 3.1). Input and output operations are done using loop in loop technique, similar to `bool MATRIX1::Initial_value()` method.

In order to make some matrix manipulation operation, it is necessary to check if it allows to make any mathematical operation according to the formal rules (see subsection 3.2.1). If dimensions of matrices do not correspond to each other for addition, subtraction and copying operations, formal rules can not be satisfied and part of matrix can be left empty or not to be filled. For multiplication operation, in case of nonadequacy in dimensions, interruption of loops occurs and operation must be stopped. For preventing such algorithm termination, it is necessary to formalize the process of operation allowing/forbidding.

Matching of matrix dimensions must be calculated formally. Class method `bool MATRIX1::Is_allowed()` (see Table 3.12d) is devoted for such purposes. Formally, initial *X* and *Y* matrices and final *Z* matrix are related through relationship, when formal process corresponds to one of the operators: addition (+), subtraction (-), multiplication (*), copying (c).

$$Z = X \text{ (process) } Y$$

Class method `bool MATRIX1::Is_allowed()` requires several input parameters of dimensionality (dimensions of final *Z* matrix and initial *X* and *Y* matrices, index 1 corresponds to row, index 2 corresponds to column) and matrix identification in output process (variable `process`, which corresponds to the arithmetical notation of operator (+-*c)). Method `bool MATRIX1::Is_allowed()` returns *TRUE* value in case of satisfying all condition for selected operator - one of (+-*c) - and *FALSE* value in case of any inadequacy.

```
bool Is_allowed(int z1, int x1, int y1,           // rows
                int z2, int x2, int y2, char process) // columns
// Z = X (process) Y
```

Method `Is_allowed()` contains `switch()` operator, where several `case()` branches allows to select required algorithm. Operator `break` allows to leave `switch()` operator in case if one of requested conditions is satisfied - operator `+-*c` was found. In case of unknown operator, method `Is_allowed()` returns *FALSE* value.

```
switch (process)
{
    case ('+'): // addition
        {...}
        break;
    case ('-'): // subtraction
        {...}
        break;
    case ('*'): // multiplication
        {...}
        break;
    case ('c'): // copy
        {...}
        break;
}; // end switch
```

Final decision about permission of addition operation will be done in one of `case()` branch using operator `if()`. Formal rules will be implemented through parameters of matrix dimension - row to row, column to column. Parameters of three matrices must be checked. Comparison of satisfying parameters according to the formal rules gives *TRUE* or *FALSE* value. In case of *FALSE* value operation is interrupted.

```
case ('+'): // addition
if ((x1 == y1) && (y1 == z1) &&
    (x2 == y2) && (y2 == z2)) return true;
```

Final decision about permission of multiplication operation will be done in one of `case()` branch using operator `if()`. Parameters of three matrices must be checked.

```
case ('*'): // multiplication
if ((x1 == y2) && (x1 == z1) && (x1 == z2) && // square
    (x2 == y1)) return true;                // any
```

Final decision about permission of copying operation will be done in one of `case()` branch using operator `if()`. Parameters of two matrices must be checked.

```
case ('c'): // matrix copy
if ((z1 == x1) && (z2 == x2)) return true;
```

Class method of matrix manipulation `void MATRIX1::Copy()` - see Table 3.12b - is one of the simplest methods. Method `void MATRIX1::Copy()` requires one input parameter - pointer to object `&X` of `MATRIX1` class, which contains 2D array with all necessary attributes, and also this method operates by own object of `MATRIX1` class, which also contains 2D array with all necessary attributes. This own object will be formed as an output object.

```
Copy(MATRIX1 &X)
// copy : Z=X
```

After satisfying the formal rules for requested operation through method `void MATRIX1::Is_allowed()`, loop in loop technique will be used for accessing of matrix elements:

```
for (int i1=0; i1<X.row_t; i1++) // rows
{
    for (int i2=0; i2<X.column_t; i2++) // columns
    {
        ...
    };
};
```

Data exchange between X and Z matrices will be done through local variable `s1`. An element of input matrix `X.arrayPtr2[ ][ ]` belonging to the object X is attributed to variable `s1`, and then variable `s1` is attributed to the output matrix belonging to own object, two-dimensional 2D array `arrayPtr2[ ][ ]`.

```
int s1;
s1 = X.arrayPtr2[i1][i2];
arrayPtr2[i1][i2] = s1;
```

Two class methods of matrix manipulation `void MATRIX1::Addition()`, `void MATRIX1::Subtraction()` - see Table 3.12b - are very similar each to other. Method `void MATRIX1::Addition()` requires two input parameters - pointer to object `&X` of `MATRIX1` type and pointer to object `&Y` of `MATRIX1` type, which contain 2D arrays with all necessary attributes, and also this method operates by own object of `MATRIX1` type, which also contains 2D array with all necessary attributes. This own object will be formed as output object.

```
void Addition(MATRIX1 &X, MATRIX1 &Y);
// Addition : Z=X+Y
```

After satisfying the formal rules for requested operation through method `bool MATRIX1::Is_allowed()`, loop in loop technique will be used for accessing of matrix elements:

```
for (int i1=0; i1<X.row_t; i1++) // rows
{
    for (int i2=0; i2<X.column_t; i2++) // columns
    {
        ...
    };
};
```

Data exchange between X, Y and Z matrices will be done through local variables `s1`, `s2`, `s3`. An element of first input matrix `X.arrayPtr2[ ][ ]` belonging to the object X is attributed to variable `s1`, the same operation is performed with the element of next input matrix called `Y.arrayPtr2[ ][ ]`. Addition is performed in the simplest arithmetic way: `s3=s1+s2`. Subtraction is performed in the same way: `s3=s1-s2`. After that, variable `s3` is attributed to the output matrix belonging to own object, `arrayPtr2[ ][ ]`.

```

int s1,s2,s3;
s1 = X.arrayPtr2[i1][i2];
s2 = Y.arrayPtr2[i1][i2];
s3 = s1 + s2;
arrayPtr2[i1][i2] = s3;

```

Class method of matrix manipulation `void MATRIX1::Multiplication()` - see Table 3.12c - must be titled as quite complicated. Method `Multiplication()` requires two input parameters - pointer to object `&X` of `MATRIX1` type, pointer to object `&Y` of `MATRIX1` type, which contains input 2D arrays with all necessary attributes, and also this method is operated by own object of `MATRIX1` class, which also contains 2D array with all necessary attributes. This own object will be formed as the output object.

```

void Multiplication(MATRIX1 &X, MATRIX1 &Y);
// Multiplication : Z=X*Y

```

After satisfying the formal rules for the requested operation through method `bool MATRIX1::Is_allowed()`, *loop in loop in loop* - triple loop technique will be used for accessing of matrix elements and operation formation. The second loop is devoted for the possibility to form the sum - row-to-column - using temporary variable `temp_s`.

```

for (int i=0; i<X.row_t; i++) // rows
{
    for (int k=0; k<Y.column_t; k++) // columns
    {
        temp_s=0;
        for (int j=0; j<X.column_t; j++) // columns
        {
            ...
        };
        arrayPtr2[i][k] =temp_s;
    };
};

```

The third loop (local variable `j` is devoted for selecting elements from row of first input matrix - object `X.arrayPtr2[ ][ ]` and column of second input matrix - object `Y.arrayPtr2[ ][ ]`.

Local variables `s1`, `s2` are used. Multiplication will be performed as simple arithmetical operation `s3=s1*s2`. Variable `s1` must be added to `temp_s`. After finishing third loop, variable `temp_s` is attributed to the output matrix belonging to own object, `arrayPtr2[ ][ ]`, no any object attributes.

Table 3.13.

Output of program mmm01.cpp.

```

MATRIX A ( 3 X 2 )

A matrix input
1 row :
1 : 1
2 : 1
2 row :
1 : 1
2 : 1
3 row :
1 : 1
2 : 1

A MATRIX : (operation - 1 )
      1  2
-----
1 %   1  1
2 %   1  1
3 %   1  1

MATRIX B ( 3 X 2 )

B matrix input
1 row :
1 : 2
2 : 2
2 row :
1 : 2
2 : 2
3 row :
1 : 2
2 : 2

B MATRIX : (operation - 1 )
      1  2
-----
1 %   2  2
2 %   2  2
3 %   2  2

MATRIX C ( 3 X 2 )

C = A + B

C MATRIX : (operation - 1 )
      1  2
-----
1 %   3  3
2 %   3  3
3 %   3  3

MATRIX D ( 3 X 2 )

D = A - B

D MATRIX : (operation - 1 )
      1  2
-----
1 %  -1 -1
2 %  -1 -1
3 %  -1 -1

Press any key to continue ...

MATRIX E ( 3 X 2 )

E matrix input
1 row :
1 : 4
2 : 4
2 row :
1 : 4
2 : 4
3 row :
1 : 4
2 : 4

E MATRIX : (operation - 1 )
      1  2
-----
1 %   4  4
2 %   4  4
3 %   4  4

MATRIX F ( 3 X 2 )

F = E

F MATRIX : (operation - 1 )
      1  2
-----
1 %   4  4
2 %   4  4
3 %   4  4

MATRIX G ( 2 X 3 )

G matrix input
1 row :
1 : 5
2 : 5
3 : 5
2 row :
1 : 5
2 : 5
3 : 5

G MATRIX : (operation - 1 )
      1  2  3
-----
1 %   5  5  5
2 %   5  5  5

MATRIX H ( 3 X 3 )

H = F * G

H MATRIX : (operation - 1 )
      1  2  3
-----
1 %  40 40 40
2 %  40 40 40
3 %  40 40 40

Press any key to continue ...

```

```

int s1,s2,s3,temp_s;
for (int i=0; i<X.row_t; i++)
{
    for (int k=0; k<Y.column_t; k++)
    {
        temp_s = 0;
        for (int j=0; j<X.column_t; j++)
        {
            s1 = X.arrayPtr2[i][j];
            s2 = Y.arrayPtr2[j][k];
            s3 = s1 * s2;
            temp_s += s3;
        }; //end for j
        arrayPtr2[i][k] = temp_s;
    }; //end for k
}; //end for i

```

Table 3.13 represents output of matrix manipulation program. Left column represents creation of matrix A, B, C, D of the same dimension, filling by values and outputting, and then two operations: addition and subtraction - $C=A+B$, $D=A-B$. Right column represents creation of matrix E, F, G, H of the different dimensionality, filling by values and outputting, and then two operations: copying and multiplication - $F=E$, $H=F*G$.

3.2.3 Matrix manipulation: OOP in second style

Table 3.14 represents `main()` function of matrix manipulation buildt-up in OOP operator-overloading style. Similar but more complicated matrix implementations are presented in Shi publication [16]. Tables 3.15a,b,c,d represent header file containing class **MATRIX2** interface and realization. Table 3.16 represents output of mentioned program.

Function `main()` represents several routines for matrix manipulation. Declaration of class object is related to the description of matrix parameters: number of rows, number of column and matrix name. It is necessary to point out, that declaration of objects belonging to the **MATRIX2** class (this subsection) and **MATRIX1** class (see subsection 3.2.2) contains any differences.

```

// matrix: 3 rows, 2 columns, A
MATRIX2 K(3,2,'K')

```

Table 3.15a represents interface of class **MATRIX2**. Class contains several types of class members: protected, private and public.

Private members (accessible for class methods only !) represent a double pointer `**arrayPtr2` for 2D array realization of integer type and variable `process` of boolean type - for allowing/forbidding of operation.

Protected members (like private but including some exceptions) represent parameters of matrix - number of rows and columns `row_t` and `column_t` of integer type and name of matrix `name_t` of character type.

Public members (accessible without any restrictions) consist of special methods and user-created methods.

Special methods represent constructor `MATRIX2::MATRIX2()` and destructor

Table 3.14. Matrix manipulation.

Program mmm02.cpp.

```

#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

#include "mmm02.h"

int main()
{
    // A MATRIX1: 3 rows, 2 columns
    // Copy
    MATRIX2 K(3,2,'K');    K.Input();    K.Show();
    MATRIX2 L(3,2,'L');    L = K;        L.Show();

    // Addition
    MATRIX2 M(3,2,'M');    M.Input();    M.Show();
                                M += K;    M.Show();

    // Subtraction
    MATRIX2 N(3,2,'N');    N.Input();    N.Show();
                                N -= L;    N.Show();

    system("PAUSE"); return 0;
}

```

MATRIX2::~MATRIX2().

User-created methods represent several methods: i) for data input and output realization `void MATRIX2::Input()`, `void MATRIX2::Show()`; ii) for matrix initialization `void MATRIX2::Initial_value()`; iii) for allowing/forbidding of operation `bool MATRIX2::Is_allowed()`; iv) for mathematical operations as three overloaded operators `void MATRIX2::operator=()`, `void MATRIX2::operator+=()`, `void MATRIX2::operator-=()`.

Constructor `MATRIX2()` and destructor `~MATRIX2()`, user created methods `Input()`, `Show()`, `Initial_value()`, `Is_allowed()` were described in Subsection 3.2.2. In class `MATRIX2`, they have used without any changes.

Table 3.15e represents interface realisation of class `MATRIX2` containing method `operator=()`.

Method `operator=`, belonging to `MATRIX2` namespace (second word `MATRIX2`), requires one input parameter - pointer to object `&X` of `MATRIX2` type (third word `MATRIX2`), which contains 2D array with all necessary attributes. This method operates by own object of `MATRIX2` class, which also contains 2D array with all necessary attributes. This own object will be formed as an output object of `MATRIX2` type (first word `MATRIX2`)

```

MATRIX2 & MATRIX2::operator=(MATRIX2 &X)
// = : Z=X

```

First of all, it is necessary to check the type of attribution. Because `operator=` is used for data exchange between different operands, it is necessary to prevent data exchange between operands themselves. In that case, `if()` condition must be not satisfied.

```
// A <- A copying forbidden
if (this == &X) return *this;
```

Loop in loop gives the possibility to access each indexed element of 2D array

```
for (int i1=0; i1<X.row_t; i1++) // rows
{
    for (int i2=0; i2<X.column_t; i2++) // columns
    {
        ...
    };
};
```

Data exchange between X object and own object will be proceeded through local variable `s1`. Element of input matrix `X.arrayPtr2[ ][ ]`, belonging to the object X is attributed to variable `s1`, and then variable `s1` is attributed to the output matrix belonging to own object, `arrayPtr2[ ][ ]`, containing any object attributes.

```
int s1;
s1 = X.arrayPtr2[i1][i2];
arrayPtr2[i1][i2] = s1;
```

After finishing loop processes, own object created and manipulated by method `operator=` must be returned to the `main()` function. Operator `return *this` performs this action.

```
MATRIX2 & MATRIX2::operator=(MATRIX2 &X)
{
    {...}
    return *this; // return left side
}
```

Table 3.15b represents interface realisation of class `MATRIX2` containing methods `operator+=` and `operator-=`. Such two methods - overloaded operators are very similar, and significant differences occur at arithmetical level of operation only.

Method `operator+=` requires one input parameter - pointer to object `&X` of `MATRIX2` class, which contains 2D array with all necessary attributes, and also this method operates by own object of `MATRIX2` class, which also contains 2D array with all necessary attributes. This own object will be formed as an output object.

```
MATRIX2 &operator+=(MATRIX2 &X)
// += : Z=Z+X
```

After satisfying the formal rules for requested operation through method `bool MATRIX2::Is_allowed()`, loop in loop technique will be used for accessing of matrix elements and operation formation.

```
for (int i1=0; i1<X.row_t; i1++) // rows
{
    for (int i2=0; i2<X.column_t; i2++) // columns
    {
        ...
    };
};
```

Data exchange between **X** object and own object will be done through local variables **s1** and **s2**. An element of input matrix **X.arrayPtr2[][]**, belonging to the object **X**, is attributed to variable **s1** and an element of own matrix **arrayPtr2[][]**, belonging to the own object, is attributed to variable **s2**. Addition process could be done using expression **s2+=s1** or **s2=s2+s1**, and subtraction - **s2-=s1** or **s2=s2-s1**. After that, variable **s2** is attributed to the output matrix belonging to own object, **arrayPtr2[][]**.

```
int s1,s2;
s1 = X.arrayPtr2[i1][i2];
s2 = arrayPtr2[i1][i2];
s2 += s1;
arrayPtr2[i1][i2] = s2;
```

After finishing loop processes, own object, created and manipulated by method **operator=**, must be returned to the **main()** function. Each object contains special pointer named **this**. Method **operator=** returns reference to object - operator **return** performs this action.

```
MATRIX2 & MATRIX2::operator+=(MATRIX2 &X)
{
    {...}
    return *this; // return left side
}
```

Table 3.15a.

Program mmm02.h.

```
class MATRIX2
{
protected:
    int row_t, column_t;
    char name_t;
private:
    int **arrayPtr2;
    bool process;
public:
    MATRIX2(int n_row, int n_col, char c_name);
    ~MATRIX2(); // destructor
    void Initial_value(int ia);
    void Input();
    void Show();
    bool Is_allowed(int z1, int x1, int y1, // rows
                    int z2, int x2, int y2, char process); // columns
    MATRIX2 &operator = (MATRIX2 &X);
    MATRIX2 &operator += (MATRIX2 &X);
    MATRIX2 &operator -= (MATRIX2 &X);
}; // end class MATRIX1
```

Table 3.15b.

Program mmm02.h.

```

MATRIX2 & MATRIX2::operator+=(MATRIX2 &X)
{
    bool b1; int s1,s2;
    b1 = Is_allowed(    row_t,    row_t,    X.row_t,
                      column_t, column_t, X.column_t,  '+');
    if (!b1) exit(0);

    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1 = X.arrayPtr2[i1][i2];
            s2 =    arrayPtr2[i1][i2];
            s2 += s1;
            arrayPtr2[i1][i2] = s2;
        };
    };
    cout << name_t << " = " <<    name_t << " + "
          << X.name_t << endl << endl;
    return *this;
}

MATRIX2 & MATRIX2::operator-=(MATRIX2 &X)
{
    bool b1; int s1,s2;
    b1 = Is_allowed(    row_t,    row_t,    X.row_t,
                      column_t, column_t, X.column_t,  '-');
    if (!b1) exit(0);
    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1 = X.arrayPtr2[i1][i2];
            s2 =    arrayPtr2[i1][i2];
            s2 -= s1;
            arrayPtr2[i1][i2] = s2;
        };
    };
    cout << name_t << " = " <<    name_t << " - "
          << X.name_t << endl << endl;
    return *this;
}

```

Table 3.15c.

Program mmm02.h.

```

bool MATRIX2::Is_allowed(int z1, int x1, int y1,           // rows
                          int z2, int x2, int y2, char process) // columns
// Z = X (process) Y
{
    char *ne = "No ";
    char *r1 = "addition";
    char *r2 = "substraction";
    char *r3 = "multiplication";
    char *r5 = "copy";
    char *nea= ". Different parameters of matrices ...\n";
    switch (process)
    {
        case ('*'): // multiplication
            if ((x1 == y2) && (x1 == z1) &&           // square
                (x1 == z2) && (x2 == y1)) return true; // any
            else
            {
                cout << x1 << " " << y2 << " "
                     << z1 << " " << z2 << " square\n";
                cout << x2 << " " << y1 << " part\n";
                cout << ne << r3 << nea;    system("PAUSE"); return false;
            };
            break; // *****
        case ('+'): // addition
            if ((x1 == y1) && (y1 == z1) &&
                (x2 == y2) && (y2 == z2)) return true;
            else
            {
                cout << ne << r1 << nea;    system("PAUSE"); return false;
            };
            break; // *****
        case ('-'): // substraction
            if ((x1 == y1) && (y1 == z1) &&
                (x2 == y2) && (y2 == z2)) return true;
            else
            {
                cout << ne << r2 << nea;    system("PAUSE"); return false;
            };
            break; // *****
    }; // end switch
    return false; // which process?
}

```

Table 3.15d.

Program mmm02.h.

```

MATRIX2::MATRIX2(   int  n_row,      int  n_col,   char  c_name) :
                    row_t (n_row), column_t (n_col), name_t (c_name)
{
    arrayPtr2 = new int*[n_row];
    for (int i=0; i<n_row; i++)
        arrayPtr2[i] = new int[n_col];
    process = true;
    cout << " MATRIX " << name_t
          << " ( " <<   row_t << " X "
              << column_t << " )\n\n";
    // matrix initialization
    Initial_value(0);
}

MATRIX2::~~MATRIX2()
{
    delete arrayPtr2;
}

void MATRIX2::Initial_value(int ia)
{
    for (int i1=0; i1<row_t; i1++)
        for (int i2=0; i2<column_t; i2++)
            arrayPtr2[i1][i2] = ia;
}

void MATRIX2::Show()
{
    cout << name_t << " MATRIX : (operation - "
          << process << " )\n";
    cout << " ";
    for (int i2=0; i2<column_t; i2++)
        cout << setw(4) << (i2+1);
    cout << endl;
    cout << " ----- \n";

    for (int i1=0; i1<row_t; i1++)
    {
        cout << setw(4) << (i1+1) << " % ";
        for (int i2=0; i2<column_t; i2++)
            cout << setw(4) << arrayPtr2[i1][i2];
        cout << endl;
    };
    cout << endl;
}

```

Table 3.15e.

Program mmm02.h.

```

MATRIX2 & MATRIX2::operator=(MATRIX2 &X)
{
    // A <- A copying forbidden
    if (this == &X) return *this;
    int s1;

    for (int i1=0; i1<X.row_t; i1++) // rows
    {
        for (int i2=0; i2<X.column_t; i2++) // columns
        {
            s1=X.arrayPtr2[i1][i2];
            arrayPtr2[i1][i2] = s1;
        };
    };
    cout <<  name_t << " = "
         << X.name_t << endl << endl;
    return *this; // return left side
}

void MATRIX2::Input()
{
    int ia;  cout << name_t << " matrix input\n";
    for (int i1=0; i1<row_t; i1++)
    {
        cout << (i1+1) << " row : \n";
        for (int i2=0; i2<column_t; i2++)
        {
            cout << (i2+1) << " : " ; cin >>ia;
            arrayPtr2[i1][i2]=ia;
        };
    };
    cout << endl;
}

```

Table 3.16 represents output of matrix manipulation program. Left column represents creation of matrix K, L, M of the same dimension, filling by values and outputting, and then two operations: copying, addition - $L=K$, $M=M+K$. Right column represents creation of matrix N, filling by values and outputting, and then subtraction - $N=N-L$.

Table 3.16. Output of program mmm02.cpp.

```

MATRIX K ( 3 X 2 )

K matrix input
1 row :
1 : 1
2 : 1
2 row :
1 : 1
2 : 1
3 row :
1 : 1
2 : 1

K MATRIX : (operation - 1 )
      1  2
-----
1 %   1  1
2 %   1  1
3 %   1  1

MATRIX L ( 3 X 2 )

L = K

L MATRIX : (operation - 1 )
      1  2
-----
1 %   1  1
2 %   1  1
3 %   1  1

MATRIX M ( 3 X 2 )

M matrix input
1 row :
1 : 2
2 : 2
2 row :
1 : 2
2 : 2
3 row :
1 : 2
2 : 2

M MATRIX : (operation - 1 )
      1  2
-----
1 %   2  2
2 %   2  2
3 %   2  2

M = M + K

M MATRIX : (operation - 1 )
      1  2
-----
1 %   3  3
2 %   3  3
3 %   3  3

MATRIX N ( 3 X 2 )

N matrix input
1 row :
1 : 4
2 : 4
2 row :
1 : 4
2 : 4
3 row :
1 : 4
2 : 4

N MATRIX : (operation - 1 )
      1  2
-----
1 %   4  4
2 %   4  4
3 %   4  4

N = N - L

N MATRIX : (operation - 1 )
      1  2
-----
1 %   3  3
2 %   3  3
3 %   3  3

Press any key to continue ...

```

3.3 Containers

Standard Template Library, *STL* is a C++ standard library of container classes, algorithms, and iterators, as described in topic Jossutis [18] publication. STL provides many of the basic algorithms and data structures of computer science.

Initiated by Stepanov ^a, STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template - for detail explanation see excellent Meyers [32] and Austern [5] publications.

Container represents an object created and supported by operating system (OS) for storage of other objects using mathematical set of elements. The container manages the storage space of elements depending of OS requesting and provides member functions which are devoted to access the container elements. Accessing could be performed directly or through *iterators*. Properties of iterators as a reference objects are similar to pointers - see Musser [34] and Josuttis [18] publications.

Containers correspond to the well known programming structures: dynamic array corresponds to a vector, linked lists corresponds to a list, trees corresponds to a set, associative arrays corresponds to a map, etc.

STL, like many class libraries, includes container classes for storing other objects: **vector**, **list**, **deque**, **set**, **multiset**, **pair**, **map**, **multimap**, and several classes of hash-like type: **hash_set**, **hash_multiset**, **hash_multimap** and **hash_map**. Each of these classes is a template, and can be instantiated to contain any type of object.

In this section, sequence containers, associative containers and container adaptors are presented - see Table 3.17.

Member-functions or methods of containers are similar to each other in many cases. Decision of usage of different type of containers depends on functionality offered by the container as well as on efficiency (complexity). The main part of problems contains operations of element manipulation: inserting/removing, accessing, search, indexing, etc - see Plauger [35], Halpern [15], Ford [50] publications.

Different containers (according to the its own data manipulating, sorting, arraying properties) are defined in headers. Name of container follows name of header. It means that vector is defined in header `<vector>`, list - in header `<list>`, etc.

All containers satisfy the requirements of the *Container* concept.

Table 3.17. Containers.

Type	Name	Stable	File name
Sequence container	vector	Yes	con01.cpp
	vector	Yes	con02.cpp
	list		con03.cpp
Associative container	pair	Yes	con10.cpp
	set	Yes	con11.cpp
	map	Yes	con12.cpp
Container adaptor	stack	Yes	con21.cpp
	queue		con22.cpp

^aAlexander Alexandrovich Stepanov (born 1950) - primary designer and implementer of the C++ Standard Template Library, which he started to develop around 1992 while employed at *Hewlett Packard* Labs.

According to this standard, all containers have methods `begin()`, `end()`, `size()`, `max_size()`, `empty()` and `swap()`.

3.3.1 Complexity

Efficiency of algorithm depends on many parameters such as: i) size of the data array (n); ii) chaotic behaviour of input data; iii) local or global complexity of data, etc. In computer science, *Big-O* notation $O(n)$ is used to classify the algorithms by how they respond to changes in input size. Computational complexity (expressed in qualitative terms *worst*, *average* and *best* behaviour) allows to estimate the sophisticated quality of algorithm. Respond means requirements of processing time or operating memory space.

If performance or complexity of an algorithm could be estimated by parameter $O(n)$, performance will grow linearly and in direct proportion to the size of the input data set. $O(n^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set.

3.3.2 Sequence containers

Let's assume the existing of so called linear list - 1D array as a finite set of elements, associated by one pointer. Together with the management functions - methods and methods for external communications, this template behaviour could be titled as container `vector` or container `list` (no restriction for element analysis, management and processing) - see Vandervoorde [17], Koenig [33] publication. Table 3.18 represents the members of sequence containers: `vector` and `list`.

Differences. The container `vector` has low memory usage (starting from memory amount required for one member and increasing twice on-demand) and all elements are stored contiguously. A user can denote initial capacity for the container `vector`. `Vector` is useful to quickly and easily allocate the requested memory for specific (random) data storage [50].

Container implementation of `vector` type consists of a pointer to a dynamically allocated array. Function `size()` and function `capacity()` return value of actual number of elements and value of size of internal array respectively. Reallocation of `vector` occurs, when the size of `vector` becomes larger than its capacity after inserted new elements.

When `vector` reallocates new region of storage, it moves all previously inserted elements into new memory region and clears old place.

Container implementation of `list` type represents an implementation of a doubly linked list. Data storage is provided non-contiguously. Frequent reallocation of memory is not necessary because container of `list` type does not allocate memory for future, only current usage. The list allocates and deallocates memory as needed only: by inserting and removing elements, respectively. For inserting operation, list is very efficient, complexity contains only $O(1)$. Element shifting operation is absent like with `vectors`.

Accessing of elements is quite complicated operation of $O(n)$ type. Accessing operation requires traversal routine to find the node that needs to be accessed.

Methods. Containers of `vector` and `list` type contain special members - constructors and destructors, which starts to work at the moment of object declaration.

Table 3.18. Members of sequence containers.				
Members		Complexity	vector	list
general	constructor	*	constructor	constructor
	destructor	O(n)	destructor	destructor
	operator=	O(n)	operator=	operator=
	assign	O(n)	assign	assign
	get_allocator	O(1)	get_allocator	get_allocator
capacity	size	*	size	size
	max_size	*	max_size	max_size
	empty	O(1)	empty	empty
	reserve	O(1)	reserve	
	capacity	O(1)	capacity	
iterators	begin	O(1)	begin	begin
	rbegin	O(1)	rbegin	rbegin
	end	O(1)	end	end
	rend	O(1)	rend	rend
element access	at	O(1)	at	
	operator[]	*	operator[]	
	data	O(n)	data	
	front	O(1)	front	front
	back	O(1)	back	back
modifiers	insert	*	insert	insert
	erase	*	erase	erase
	swap	O(1)	swap	swap
	clear	O(n)	clear	clear
	push_front	O(1)		push_front
	pop_front	O(1)		pop_front
	push_back	O(1)	push_back	push_back
	pop_back	O(1)	pop_back	pop_back
	resize	O(n)	resize	resize
operations	splice	*		splice
	remove	*		remove
	remove_if	*		remove_if
	reverse	*		reverse
	unique	*		unique
	sort	*		sort

Special member - constructor constructs the container from different sources. Methods `operator=` and `assign()`, presented in both templates, assigns values to the container.

Operator `get_allocator` returns the allocator which was used to allocate memory for the elements.

Methods related to the capacity such as `size()`, `max_size()`, `empty()` are present in both containers. Method `size()` returns the number of elements in the container. Method `max_size()` returns the maximum possible number of elements in the container. Method `empty()` checks if the container is empty or not. Several methods related to the capacity

are included in **vector** only.

Method **reserve()** reserves storage, and method **capacity()** returns the number of elements which could fill the currently allocated storage (not number of real elements in this container). In some cases both numbers can be equal to each other (when requested vector is fully filled but not overfilled).

Iterators as the main tool for container manipulating are implemented in both containers. Methods **begin()** and **end()** return an iterator to the beginning of the container and to the end of the container respectively. Methods **rbegin()** and **rend()** return a reverse iterator to the reverse beginning of the container and to the reverse end of the container respectively.

Several methods related to the element access are implemented in both containers. Methods **front()** and **back()** are devoted to access the first and the last element of container respectively. For vector only, methods **at()** and **operator[]** are devoted to access of specified element with and without bounds checking respectively and method **data()** accesses the underlying array.

The main part of modifiers is the same for both containers: **vector** and **list**. Methods **erase()** and **clear()** erases elements and clears content, respectively. Methods **insert()** and **resize()** inserts elements and changes the number of stored elements until requested one, respectively. Method **swap()** swaps the contents with another container of the same type. Methods **push_back()** and **pop_back()** inserts elements to the end and removes the last element, respectively. For **list** only, methods **push_front()** and **pop_front()** insert an elements to the beginning and remove the first element, respectively.

There are several operations realized in **list** only. Method **merge()** merges two sorted lists. Method **splice()** moves elements from another list. Method **remove()** removes elements equal to the given value. Method **remove_if()** removes elements satisfying specific criteria. Method **reverse()** reverses the order of the elements. Method **unique()** removes consecutive duplicate elements. Method **sort()** sorts the elements.

VECTOR. Table 3.19 represents programming implementation of container **vector**, and Table 3.20 - output of working program.

Function **main()** represents several routines for container **vector** manipulation. Declaration of object **argo** of class **SEQUENCE1** type is made in traditional OOP manner. Methods **increase()** and **decrease()** realize data input and data output. Methods **show1()** and **show2()** realize output operations.

```
SEQUENCE1 argo;
    argo.increase(2,5);  argo.show1();
SEQUENCE1 tore;
    tore.increase(3,6);  tore.show2();
    tore.decrease(2);    tore.show2();
```

Table 3.19 represents interface of class **SEQUENCE1**. Class contains one type of class members: public.

Public members (accessible without any restrictions) represents an 1D array **vAA** of **vector** **<int>** type, three iterators of **vector** **<int>** type, special method - constructor **SEQUENCE1()** as well as user-created methods for vector manipulation **increase()**, **decrease()**, for output realization **show1()**, **show2()**.

Table 3.19. Container vector.

Program con01.cpp.

```

#include <cstdlib>
#include <iostream>
#include <vector>
using namespace std;

class SEQUENCE1
{
public:
    vector<int> vAA;
    vector<int>::iterator iter;
    vector<int>::iterator iter_ini;
    vector<int>::iterator iter_fin;
    SEQUENCE1();
    void increase(int ia1, int ia2);
    void decrease(int ia1);
    void show1();
    void show2();
};

SEQUENCE1::SEQUENCE1() // constructor
{
    vAA.clear();
}

int main(int argc, char *argv[])
{
    SEQUENCE1 argo;
    argo.increase(2,5);  argo.show1();
    SEQUENCE1 tore;
    tore.increase(3,6);  tore.show2();
    tore.decrease(2);    tore.show2();
    system("PAUSE");
    return EXIT_SUCCESS;
}

void SEQUENCE1::show1()
{
    cout << "Show1" << endl;
    iter_ini = vAA.begin();
    iter_fin = vAA.end();
    for (iter = iter_ini;
         iter < iter_fin;
         iter++)
        cout << *iter << endl;
    cout << endl;
}

void SEQUENCE1::show2()
{
    cout << "Show2" << endl;
    int iSize=vAA.size();
    for (int i=0; i<iSize; i++)
        cout << i << "  #  "
            << vAA[i] << endl;
    cout << endl;
}

void SEQUENCE1::increase(int ia1,
                        int ia2)
{
    cout << "Input from " << ia1
        << " until " << ia2
        << endl << endl;
    for (int i=ia1; i<=ia2; i++)
        vAA.push_back(i);
}

void SEQUENCE1::decrease(int ia1)
{
    cout << "remove " << ia1
        << " elements." << "\n\n";
    for (int i=1; i<=ia1; i++)
        vAA.pop_back();
}

```

Constructor `SEQUENCE1::SEQUENCE1()` is devoted for clearing of vector content using expression `vAA.clear()`; . It is necessary to point out, that initially, before some manipulations, container must be empty. Method `void SEQUENCE1::increase()` is devoted for including of requested interval (from `ia1` until `ia2`). This process is organized through one loop. Vector method `push_back(i)` inserts integer number as elements to the end.

Table 3.20. Output of program con01.cpp.

Input from 2 until 5

Show1

2

3

4

5

Input from 3 until 6

Show2

0 # 3

1 # 4

2 # 5

3 # 6

remove 2 elements.

Show2

0 # 3

1 # 4

Press any key to continue ...

```
for (int i=ia1; i<=ia2; i++)
    vAA.push_back(i);
```

Method `void SEQUENCE1::decrease()` is devoted for removing requested number of elements from the end. This process is organized through one loop. Vector method `pop_back(i)` removes requested times the elements from the end.

```
for (int i=1; i<=ia1; i++)
    vAA.pop_back();
```

Method `void SEQUENCE1::show1()` is devoted for output of vector content using traditional iterator technigues. First of all, It is necessary to "measure" the real amount of vector. Methods `void begin()` and `void end()` return initial and final iterator values, respectively.

```
iter_ini = vAA.begin();
iter_fin = vAA.end();
```

Secondly, one loop performs the output of reference to iterator `*iter` when loop variable `iter` varies form initial to final value.

```
for (iter = iter_ini; iter < iter_fin; iter++)
    cout << *iter << endl;
```

Method `void SEQUENCE1::show2()` is devoted for output of vector content using operator `[ ]`, well known for array manipulation. First step is directed to the establishing of amount of vector using method `size()`.

```
int iSize=vAA.size();
```

Secondly, one loop performs the output of vector member using operator `[ ]` from traditional array manipulation technique.

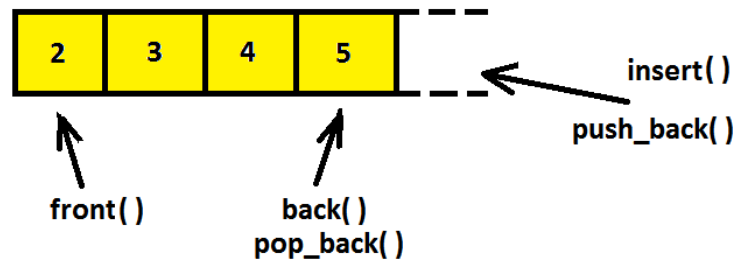


Fig. 3.7. Container vector. Data organization scheme.

```
for (int i=0; i<iSize; i++)
    cout << i << "  #  " << vAA[i] << endl;
```

Table 3.20 represents output of program. It is obvious, that inserting the members into container vector occurs at the end - see Table 3.20, left Show1 part. Removing of members occurs from the end - see Table 3.20, right Show2 part. No any differences in output design of methods Show1 and Show2 occur, where different output technique - reference to iterator or operator `[ ]` - was used.

Fig. 3.7 illustates data organization scheme for container **vector**. Linear dependence is manipulated from the end of array.

Table 3.21. Container vector.

Program con02.cpp.

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

vector<int> eCC;
void paramet(int j)
{
    cout << setw(3) << j << "  , size = " << setw(3) << eCC.size()
        << "  , capacity = " << setw(3) << eCC.capacity()
        << endl;
}

int main(int argc, char *argv[])
{
    paramet(0);
    for (int i=1; i<=18; i++)
    {
        eCC.push_back(i);    paramet(i);
    };
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 3.22. Output of program con02.cpp.

```

0 , size = 0 , capacity = 0
1 , size = 1 , capacity = 1
2 , size = 2 , capacity = 2
3 , size = 3 , capacity = 4
4 , size = 4 , capacity = 4
5 , size = 5 , capacity = 8
6 , size = 6 , capacity = 8
7 , size = 7 , capacity = 8
8 , size = 8 , capacity = 8
9 , size = 9 , capacity = 16
10 , size = 10 , capacity = 16
11 , size = 11 , capacity = 16
12 , size = 12 , capacity = 16
13 , size = 13 , capacity = 16
14 , size = 14 , capacity = 16
15 , size = 15 , capacity = 16
16 , size = 16 , capacity = 16
17 , size = 17 , capacity = 32
18 , size = 18 , capacity = 32
Press any key to continue ...

```

Table 3.21 represents another methodologically important programming implementation of vector. Table 3.22 represents an output of this program. Object `eCC` in the global space is declared as container of `vector` type created using template `<int>`.

Data input is organized in loop `for()` using operator `push_back()`. After each input of loop variable, size and capacity of vector is outputted using operators `size()` and `capacity()`, respectively. Table 3.22 shows, that initially size of vector is equal to zero (zero elements), but after starting loop activity, the size of vector increases continuously. Increasing is linear, step by step, depending on the loop activity (increment operation of loop variable `i++`).

Capacity of vector increases not continuously, but with a jump. Initially capacity is equal to zero (no allocated memory for vector). For first element, capacity increases from zero to one. When all amount of existing vector is filled by elements, capacity of vector increases twice. First increasing was made for capacity of 1 element, then for capacity of 2 elements, then 4, 8, 16, 32, ... and so on. All operations are provided automatically by OS, and user can not influence it.

Increasing of capacity is operation made up by OS, because it is necessary to allocate the new twice increased amount of memory and to remove all previous vector content to new one.

LIST. Table 3.23 represents programming implementation of `list` container and Table 3.24 - output of working program. Function `main()` represents several routines for the container `list` manipulation. Declaration of object `tore` of class `SEQUENCE2` type is made in traditional OOP manner. Several methods for list increasing `void SEQUENCE2::increase1()` and `void SEQUENCE2::increase2()`, also methods for list decreasing `void SEQUENCE2::decrease1()`

Table 3.23. Container list.

Program con03.cpp.

```

#include <cstdlib>
#include <iostream>
#include <list>
using namespace std;

class SEQUENCE2
{
public:
    list<int> vBB;
    list<int>::iterator iter;
    list<int>::iterator iter_ini;
    list<int>::iterator iter_fin;
    SEQUENCE2();
    void increase1(int ia1, int ia2);
    void decrease1(int ia1);
    void increase2(int ia1, int ia2);
    void decrease2(int ia1);
    void show1();
};

void SEQUENCE2::show1()
{
    cout << "Show1" << endl;
    iter_ini = vBB.begin();
    iter_fin = vBB.end();
    for (iter = iter_ini;
         iter != iter_fin;
         iter++)
        cout << *iter << endl;
    cout << endl;
}

int main(int argc, char *argv[])
{
    SEQUENCE2 te;
    te.increase1(5,8); te.show1();
    te.decrease1(2);   te.show1();
    te.increase2(0,4); te.show1();
    te.decrease2(1);   te.show1();
    system("PAUSE");
    return EXIT_SUCCESS;
}

void SEQUENCE2::increase1(int ia1,
                          int ia2)
{
    cout << "Input (back) from "
          << ia1
          << " until "
          << ia2 << endl;
    for (int i=ia1; i<=ia2; i++)
        vBB.push_back(i);
    cout << endl;
}

void SEQUENCE2::decrease1(int ia1)
{
    cout << "Remove (back) "
          << ia1
          << " elements...\n\n";
    for (int i=1; i<=ia1; i++)
        vBB.pop_back();
}

void SEQUENCE2::increase2(int ia1,
                          int ia2)
{
    cout << "Input (front) from "
          << ia1
          << " until "
          << ia2 << endl;
    for (int i=ia1; i<=ia2; i++)
        vBB.push_front(i);
    cout << endl;
}

void SEQUENCE2::decrease2(int ia1)
{
    cout << "Remove (front) " << ia1
          << " elements..." << "\n\n";
    for (int i=1; i<=ia1; i++)
        vBB.pop_front();
}

SEQUENCE2::SEQUENCE2() // constructor
{
    vBB.clear();
}

```

Table 3.24.	Output of program con03.cpp.
Input (back) from 5 until 8	Input (front) from 0 until 4
Show1	Show1
5	4
6	3
7	2
8	1
	0
Remove (back) 2 elements...	5
	6
Show1	Remove (front) 1 elements...
5	
6	Show1
	3
	2
	1
	0
	5
	6
	Press any key to continue ...

and void SEQUENCE2::decrease2() are devoted for data input and data output. Method void SEQUENCE2::show1() realizes output operations.

```
SEQUENCE2 tore;
    tore.increase1(5,8); tore.show1();
    tore.decrease1(2);   tore.show1();
    tore.increase2(0,4); tore.show1();
    tore.decrease2(1);   tore.show1();
```

Table 3.23 represents interface of class SEQUENCE2. Class contains public class members. Public members (accessible without any restrictions) represent an 1D array vBB of list <int> type, three iterators of list <int> type, special method - constructor SEQUENCE1() as well as user-created methods for list manipulation void SEQUENCE2::increase(), void SEQUENCE2::decrease(), for output realization void SEQUENCE2::show1(), void SEQUENCE2::show2().

Constructor SEQUENCE2::SEQUENCE2() is devoted for clearing of list content using expression vAA.clear();. It is necessary to point out, that initially, before some manipulations, container must be empty.

Method void SEQUENCE2::increase1() is devoted for including of requested interval (from ia1 until ia2). This process is organized through one loop. List method push_back(i) inserts integer number as elements to the end.

```
for (int i=ia1; i<=ia2; i++)
```

```
vBB.push_back(i);
```

Method `void SEQUENCE2::increase2()` is also devoted for including of requested interval (from `ia1` until `ia2`). This process is organized through one loop. List method `push_front(i)` inserts integer number as elements to the beginning.

```
for (int i=ia1; i<=ia2; i++)
    vBB.push_front(i);
```

Method `void SEQUENCE2::decrease1()` is devoted for removing requested number of elements from the end. This process is organized through one loop. Vector method `pop_back(i)` removes requested times the elements from the end.

```
for (int i=1; i<=ia1; i++)
    vBB.pop_back();
```

Method `void SEQUENCE2::decrease2()` is devoted for removing requested number of elements from the beginning. This process is organized through one loop. Vector method `pop_front(i)` removes requested times the elements from the end.

```
for (int i=1; i<=ia1; i++)
    vBB.pop_front();
```

Method `void SEQUENCE2::show1()` is devoted for output of vector content using traditional iterator technique. First of all, it is necessary to "measure" the real amount of vector. Methods `void begin()` and `void end()` return initial and final iterator values, respectively.

```
iter_ini = vBB.begin();
iter_fin = vBB.end();
```

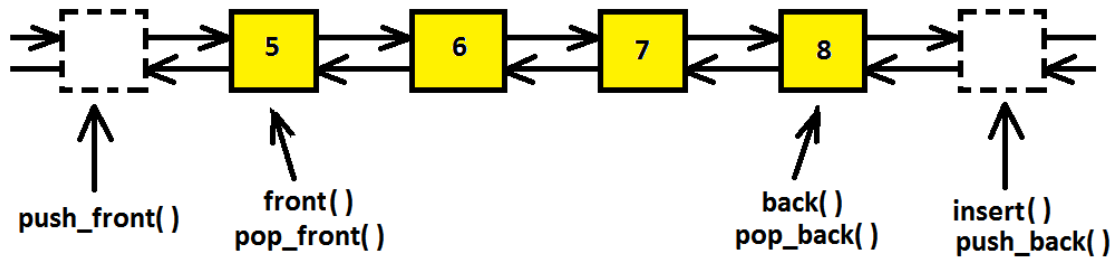
Secondly, one loop performs the output of reference to iterator `*iter` when loop variable `iter` varies from initial to final value. Operator `[]` is absent in list implementation.

```
for (iter = iter_ini; iter != iter_fin; iter++)
    cout << *iter << endl;
```

Table 3.24 represents output of program. Using of container list in vector manner is possible, and input (back) and remove (back) operations show that. The operating point of container list like vector is at the end in this case - see left part of Table 3.24. But also container list could be filled from beginning - and two operations show that: input (front) and remove (front). In that case operating point is at the beginning - see right part of Table 3.24.

Fig. 3.8 illustrates data organization scheme for container `list`. Linear dependence is manipulated from the end as well as top of array.

Container `list` is implemented as a doubly linked list of items. Each element has its own segment of memory. Also, each element (item) refers to its predecessor and its successor.

Fig. 3.8. Container `list`. Data organization scheme.

3.3.3 Associative containers

Let's assume the existing of so called an associative array, map, or dictionary - as a finite set of elements composed of a collection of key pairs. Each possible key pair appears at the most once in the collection. Together with the methods as a management functions and methods for external communications, this template behaviour could be titled as a container set or a container map (no restriction for element analysis, management and processing).

According to the associative container definition, each key in map and set must be unique. Element composition is presented in non-linear way. In a set, each element is key; there are no mapped values. In a map, each element is composed from a key and a mapped value. In both associative containers `set` and `map`, ordering of elements occurs after each inserting. Internally, the elements are always sorted from lower to higher following a specific strict weak ordering criterion - this criterion belongs to container construction.

The simplest associative container is `pair` container. As a basic container, it can be used to store any of two elements, called first and second. The container of `pair` type is very universal, can do basic features like assignment and comparisons only. Container `pair` is a basic ingredient of the abstract containers `map`, `multimap` and `hash_map`.

Table 3.25 represents the members of associative containers: `set` and `map`.

Differences. Container `set` is devoted for storage of unique elements, but the elements themselves are treated as keys. Efficient accessing of `set` elements is organized by their key. Sets are typically implemented as *binary search trees*, BST. In that case, associative container `set` significantly differs from sequence containers. Accessing of elements in vector is realized by their relative or absolute position. For `set` container, no two repeatable elements could be inserted. For repeatable elements in associative container, it is possible to use another type of container `multiset`.

Container `map` stores elements formed by the combination of a key value and a mapped value. Key value is generally used for element identifying purposes, while the mapped value is any value associated to this key. Due to identifying purposes, key value must be unique - no repeatable key values could be inserted into map container. If associative container is necessary for repeatable key values, such as multiple elements with equivalent keys, another type of container such as `multimap` is possible.

Methods. Containers `set` and `map` contain special members - constructors and destructors, which starts to work from object declaration moment: constructs the container.

Table 3.25. Members of associative containers.				
Members		Complexity	set	map
general	constructor	*	constructor	constructor
	destructor	$O(n)$	destructor	destructor
	operator=	$O(n)$	operator=	operator=
	get_allocator	$O(1)$	get_allocator	get_allocator
capacity	size	*	size	size
	max_size	*	max_size	max_size
	empty	$O(1)$	empty	empty
iterators	begin	$O(1)$	begin	begin
	rbegin	$O(1)$	rbegin	rbegin
	end	$O(1)$	end	end
	rend	$O(1)$	rend	rend
elements	operator[]	*		operator[]
	at	$O(1)$		at
modifiers	insert	*	insert	insert
	erase	*	erase	erase
	swap	$O(1)$	swap	swap
	clear	$O(n)$	clear	clear
observers	key_comp	$O(1)$	key_comp	key_comp
	value_comp	$O(1)$	value_comp	value_comp
operations	find	$O(\log n)$	find	find
	count	$O(\log n)$	count	count
	lower_bound	$O(\log n)$	lower_bound	lower_bound
	upper_bound	$O(\log n)$	upper_bound	upper_bound
	equal_range	$O(\log n)$	equal_range	equal_range
lookup	count		count	count
	find		find	find
	equal_range		equal_range	equal_range
	lower_bound		lower_bound	lower_bound
	upper_bound		upper_bound	upper_bound

Operator= , presented in both templates, assigns values to the container. **Operator get_allocator** returns the allocator which was used to allocate memory for the elements.

Methods related to the capacity such as **size()**, **max_size()**, **empty()** are present in both containers. Method **size()** returns the number of elements in this container. Method **max_size()** returns the maximum possible number of elements in the container. Method **empty()** checks the statement if the container is empty or not.

Prototypes of pointers - so called iterators (as main tool for container manipulation) are realized in both containers. Methods **begin()** and **end()** return an iterator to the beginning of the container and to the end of the container respectively. Methods **rbegin()** and **rend()** return a reverse iterator to the reverse beginning of the container and to the reverse end of the container respectively.

Several methods related to the element access are realized for **map** container only. Methods

`at()` and `operator[ ]` are devoted for access of specified element with and without bounds checking, respectively. For the `set` container such methods are absent.

The main part of modifiers is the same for both containers: `set` and `map`. Methods `erase()` and `clear()` erases elements and clears content, respectively. Method `insert()` inserts elements. Method `swap()` swaps the contents with another container of the same type.

Observers `key_comp()` and `value_comp()` are presented for both containers: `set` and `map`. Method `key_comp()` returns the key comparison function. Method `value_comp()` returns the value comparison function.

It is necessary to point out, that this method in container `set` is equivalent to `key_comp()`, since the elements are composed from a key only.

Several lookup methods are realized in both containers: `set` and `map`. Method `count()` returns the number of elements matching specific key. Method `find()` finds an element with specific key. Method `equal_range()` returns a range of elements matching specific key. Method `lower_bound()` returns an iterator to the first element with a key not less than the given value. Method `upper_bound()` returns an iterator to the first element with a key greater than a certain value.

PAIR. Container `pair` holds only two values of the same or different type. Table 3.26 represents programming implementation of `pair` container, and Table 3.27 - output of working program. Function `main()` represents typical routine for container `pair` manipulation, which is described in `<utility>` library.

The great advantage of pairs is that they have built-in operations to compare themselves. Pairs are compared in order from first to second element. If the first elements are not equal, the result will be based on the comparison of the first elements only. The second elements will be compared only if the first ones are equal. The array of pairs can easily be sorted by STL internal functions.

There are several possibilities how to input data into `pair` container. First of all, it is possible to use constructor of container by object declaration. Two data items of different type that of string and integer are delimited by comma.

```
pair <string,double> car1 ("Nissan Juke",52990);
```

Secondly, operators `first` and `second` could be used for initialization of object components of previously declared `pair` object by corresponding variables.

```
pair <string,double> car2;  car2.first  = "Nissan Qashqai";
                           car2.second = 58900;
```

Thirdly, function `make_pair` constructs a pair object with its first element set to first variable and its second element set to second variable. Type and order of variables must correspond to the inner structure of object declared as a `pair` container.

```
pair <string,double> car3;  car3 = make_pair ("Nissan Primastar",77500);
```

Output of container components is possible using operators `first` and `second`

```
cout << "The price of " << car1.first << " " << car1.second << " Lt." << endl;
```

Table 3.26. Container pair.	Program con10.cpp.
<pre> #include <cstdlib> #include <iostream> #include <utility> #include <string> using namespace std; int main () { pair <string,double> car1 ("Nissan Juke",52990); pair <string,double> car2; pair <string,double> car3; car2.first = "Nissan Qashqai"; car2.second = 58900; car3 = make_pair ("Nissan Primastar",77500); char *t=" - "; char *tt = " Lt."; cout << "The price of " << car1.first << t << car1.second << tt << endl; cout << "The price of " << car2.first << t << car2.second << tt << endl; cout << "The price of " << car3.first << t << car3.second << tt << endl; system("PAUSE"); return 0; } </pre>	

Table 3.27.	Output of program con10.cpp.
<pre> The price of Nissan Juke - 52990 Lt. The price of Nissan Qashqai - 58900 Lt. The price of Nissan Primastar - 77500 Lt. Press any key to continue ... </pre>	

SET. Table 3.28 represents programming implementation of **set** container, and Table 3.29 shows output of working program.

Function **main()** represents several routines for container **set** manipulation. Function **void show()** is devoted for output purposes.

Function **void show(set<int> sXX)** requires one input parameter - an object of container **set** type constructed from integers.

Mentioned function contains three declarations of iterators **iter**, **iter_ini**, **iter_fin** and following "measuring" of size of container **sXX** by means of operators **begin()** and **end()**.

Table 3.28. Container set.

Program con11.cpp.

```

#include <cstdlib>
#include <iostream>
#include <set>
using namespace std;

int main(int argc, char *argv[])
{
    // filling of set
    set<int> sAA;
    sAA.insert(19); sAA.insert(20);
    sAA.insert(10); sAA.insert(11);
    show(sAA);

    // pointers used as iterators
    int iBB[] = {17,14,15,16,13};
    set<int> sBB (iBB,iBB+4);
    show(sBB);

    // a copy of set
    set<int> sCC (sBB);
    show(sCC);

    // a copy of set using iterators
    set<int> sDD (sBB.begin(), sBB.end());
    sDD.insert (sAA.begin(), sAA.end());
    show(sDD);

    system("PAUSE"); return EXIT_SUCCESS;
}

void show(set<int> sXX)
{
    set<int>::iterator iter;
    set<int>::iterator iter_ini;
    set<int>::iterator iter_fin;
    iter_ini = sXX.begin();
    iter_fin = sXX.end();
    for (iter = iter_ini;
         iter != iter_fin;
         iter++)
        cout << *iter << endl;
    cout << endl;
}

```

```

set<int>::iterator iter;
set<int>::iterator iter_ini;
set<int>::iterator iter_fin;
iter_ini = sXX.begin();
iter_fin = sXX.end();

```

Using iterator one-dimensional loop technique, each member of container could be outputted using `*iter` reference expression

```

for (iter = iter\_ini; iter != iter\_fin; iter++)
    cout << *iter << endl;
cout << endl;

```

Function `main()` is made in imperative style. Declaration of object `sAA` of integer type, several insert operations (constant integers 10,11, etc) and output using `show()` function contain first block of program:

Table 3.29.	Output of program con11.cpp.
10	10
11	11
19	14
20	15
	16
14	17
15	19
16	20
17	
	Press any key to continue ...
14	
15	
16	
17	

```

set<int> sAA;
sAA.insert(10); sAA.insert(11);
sAA.insert(19); sAA.insert(20);
show(sAA);

```

Table 3.29, left part, first output represents this action.

Second block of program is devoted for special manipulation technique with *pointers as iterators*.

```

// pointers used as iterators
int iBB[] = {17,14,15,16,13};
set<int> sBB (iBB,iBB+4);
show(sBB);

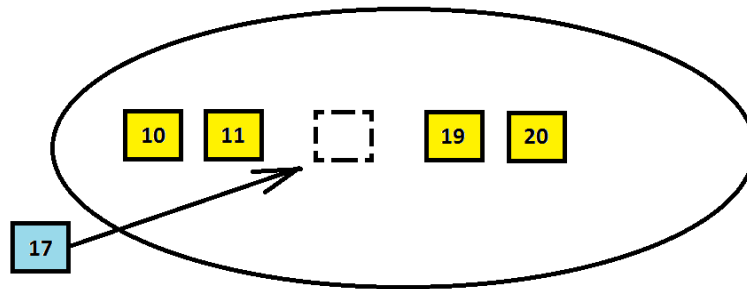
```

Static 1D array is declared and initialized using one sentence: `int iBB[] = {17,14,15,16,13};`. Declaration of `sBB` container occurs with the initialization by four members from `iBB` array using pointers. Presenting of array name without operator `[]` allows to operate by pointer to array member, but not directly to content of array member. Arithmetical expressions of `(iBB)`, `(iBB+4)` type allow to recalculate the array members through pointers.

Sentence `set<int> sBB (iBB,iBB+4)` represents declaration of container `sBB` as well as task for constructor of class `set` to input the content of array (array members) from `iBB[0]` until `iBB[0+3]`: `{17,14,15,16}` only. This task is requested through pointers of corresponding members.

Table 3.29, left part, second output represents this action.

Copying of container belongs to the one of the simplest operations. It is necessary to declare the new container object `sCC` together with task to constructor to fill the content of new object `sCC` by container `sBB`.

Fig. 3.9. Container `set`. Data organization scheme.

```
// a copy of set
set<int> sCC (sBB);
show(sCC);
```

Table 3.29, left part, third output represents this action.

Container `set` for full or partially limited copying also allows to use iterators also. Expression `set<int> sDD (sBB.begin(), sBB.end())` represents declaration of container `sDD` as well as task to constructor to fill content of newly created container by values pointed by iterators belonging to object `sBB`. "Measuring" of size `sBB` is made using operators `begin()` and `end()`. Therefore, analogically to this, operator `insert (sAA.begin(), sAA.end())` uses two iterators belonging to object `sAA`.

```
// a copy of set using iterators
set<int> sDD (sBB.begin(), sBB.end());
sDD.insert (sAA.begin(), sAA.end());
show(sDD);
```

Table 3.29, right part represents this action.

Fig. 3.9 illustrates data organization scheme for container `set`. Set represents an linear ordered structure, inserting operation is always made in a proper place.

MAP. Tables 3.30a, 3.30b, 3.30c represent programming implementation of `map` container, and Table 3.31 shows output of working program. Function `main()` represents two typical routines for container `map` manipulation.

First part of `main()` function is devoted to use container `map` in number-pointed manner. Declaration of container object `People1` is made using template `<int,string>` which means that data pair, consisting of integer and string, represents one associative member of container `map`. Thus iterator `iter` of the same type must be declared.

```
map<int,string> People1;
map<int,string>::iterator iter;
```

Lets start to fill the container with associative data. For example, local telephone number 7401 (data of integer type) assigned to the employer Mike Q. (data of string type) represents one pair of data. Assignment could be made using array index notation, `operator[ ]` to object `People1`.

Table 3.30a. Container map.

Program con12.cpp.

```

#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <map>
#include <string>
using namespace std;

void map1();    // 1st block
void map2();    // 2nd block

int main(int argc, char *argv[])
{
    map1();     // 1st block
    map2();     // 2nd block

    system("PAUSE");
    return EXIT_SUCCESS;
}

```

```

// cell number, int; employer name, string
// Assignment: array index notation
People1 [7401] = "Mike Q.";
People1 [7220] = "Charlie B.";
People1 [7228] = "David R.";
People1 [7224] = "Harry D.";
People1 [7400] = "John S.";

```

Output of one fixed container member could be made using array index notation - see Table 3.25, left part, first line.

```
cout << "People1 [7220] = " << People1[7220] << "\n\n";
```

Amount of container is possible to measure using operator `size()`- see Table 3.29, left part, second line.

```
cout << "Map size = " << People1.size() << "\n\n";
```

Full output of container `map` is easy to do using traditional iterator technique through one loop. Operators `begin()` and `end()` was used for establishing of real, current amount of container, and loop must be done between initial `People1.begin()` and final `People1.end()` iterators in `for()` loop:

Table 3.30b. Container map.

Program con12.cpp.

```

void map1()
{
    map<int,string> People1; // 1st
    map<int,string>::iterator iter;

    // cell number, int; name, string;
    // Assignment: array index notation
    People1 [7401] = "Mike Q.";
    People1 [7220] = "Charlie B.";
    People1 [7228] = "David R.";
    People1 [7224] = "Harry D.";
    People1 [7400] = "John S.";

    cout << "People1 [7220] = " << People1[7220] << "\n\n";
    cout << "Map size = " << People1.size() << "\n\n";

    for ( iter = People1.begin(); iter != People1.end(); ++iter)
    {
        cout << (*iter).first << " ==> "
            << (*iter).second << endl;
    };
    cout << endl;
}

```

```

for ( iter = People1.begin(); iter != People1.end(); ++iter)
{
    ...
};
cout << endl;

```

Inner part of loop represents output of reference to pointer **iter* described as the first part *(*iter).first* and the second part *(*iter).second*.

According to the declaration of object `map<int,string> People1`, object *People1* belonging to the container `map` was created using template `<int,string>`. So, associative parts (integer and string) are included into member, and complex pointers allows to distinguish both part of member according to the declaration position.

```

cout << (*iter).first << " ==> "
    << (*iter).second << endl;

```

Described manipulations are presented in Table 3.29, on left bottom.

The second part of the `main()` function is devoted to use container `map` in string-pointed manner. Declaration of container object *People2* is made using template `<string,int>` which means that data pair, consisting of string and integer, represents one associative member of container `map`. Also iterator *iter2* of the same type must be declared.

Table 3.30c. Container map.

Program con12.cpp.

```

void map2()
{
    map<string,int> People2; // 2nd
    map<string,int>::iterator iter2;

    // name, string; cell number, int;
    // Assignment: array index notation
    People2 ["Hank Q." ] = 6250;
    People2 ["Charlie G."] = 6290;
    People2 ["Kent Z." ] = 6271;
    People2 ["Harry D." ] = 6222;
    People2 ["Dave Q." ] = 6211;

    // Assignment: insert() and STL pair
    People2.insert(std::pair<string,int>("Stephen F.",6003));
    // Assignment: insert() and "value_type()"
    People2.insert(map<string,int>::value_type("Michael I.",6009));
    // Assignment: insert() and "make_pair()"
    People2.insert(std::make_pair("Larry N.",6005));

    cout << "Map size = " << People2.size() << "\n\n";
    for( iter2 = People2.begin(); iter2 != People2.end(); ++iter2)
    {
        cout << setw(11) << (*iter2).first << " <== "
              << (*iter2).second << endl;
    };
    cout << endl;
}

```

Table 3.31.

Output of program con12.cpp.

People1 [7220] = Charlie B.

Map size = 8

Map size = 5

7220 ==> Charlie B.

7224 ==> Harry D.

7228 ==> David R.

7400 ==> John S.

7401 ==> Mike Q.

Charlie B. <== 6290

Dave Q. <== 6211

Hank Q. <== 6250

Harry D. <== 6222

Kent Z. <== 6271

Larry N. <== 6005

Michael I. <== 6009

Stephen F. <== 6003

Press any key to continue ...

```
map<string,int> People2;
map<string,int>::iterator iter2;
```

Let's start to fill the container with associative data. In that case, employer name **Hank Q.** (data of string type) is associated to the local telephone number **6250** (data of integer type). There are several techniques of data insertion into container. One of the simplest way is when assignment could be made by using array index notation, operator `[ ]` to object **People2**.

```
// name, string; cell number, int;
// Assignment: array index notation
People2 ["Hank Q." ] = 6250;
People2 ["Charlie G." ] = 6290;
People2 ["Kent Z." ] = 6271;
People2 ["Harry D." ] = 6222;
People2 ["Dave Q." ] = 6211;
```

The next possibility is to use operator `insert()` where input parameter represent an output of object of `pair()` from STL:

```
// Assignment: insert() and STL pair
People2.insert(std::pair<string,int>("Stephen F.",6003));
```

Manipulator `pair<string,int>("Stephen F.",6003)` associates string to integer using template `<string,int>` and moves it to `insert()` operator.

The third possibility is to use operator `insert()` where input parameter represents an output of object of `value_type()` from STL:

```
// Assignment: insert() and "value_type()"
People2.insert(map<string,int>::value_type("Michael I.",6009));
```

Manipulator `map<string,int>::value_type("Michael I.",6009)` associates string to integer using template `<string,int>` from `map` and moves it to `insert()`.

The fourth possibility is to use operator `insert()` where input parameter represents an output of object of `make_pair()` from STL:

```
// Assignment: insert() and "make_pair()"
People2.insert(std::make_pair("Larry N.",6005));
```

Manipulator `make_pair("Larry N.",6005)` associates string to integer using template `<string,int>` from STL and moves it to `insert()`.

Full output of container `map` is easy to do using traditional iterator technique as it was described previously.

```
for( iter2 = People2.begin(); iter2 != People2.end(); ++iter2)
{
    cout << setw(11) << (*iter2).first << " <== "
        << (*iter2).second << endl;
};
```

Output of second block is presented in Table 3.31, right part.

Fig. 3.10 illustrates data organization scheme for container `map`. Data set represents a non-linear pair-ordered structure, inserting operation is always made in a proper place - according to the first parameter!

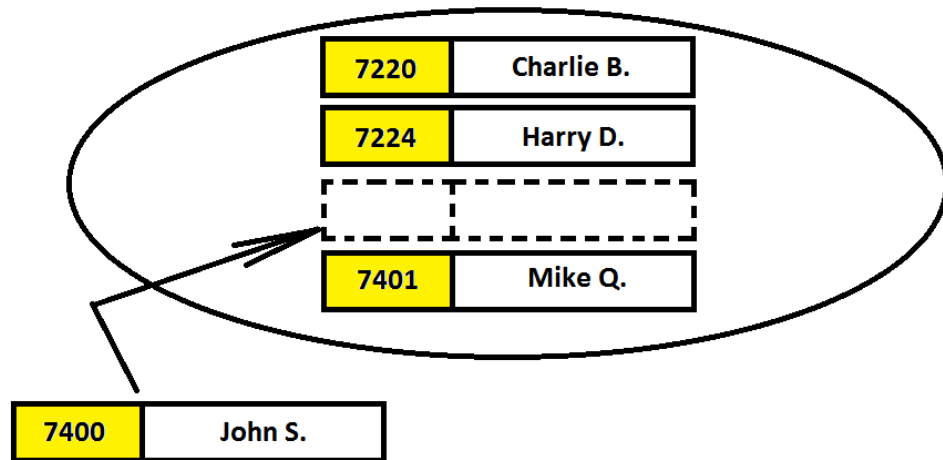


Fig. 3.10. Container map. Data organization scheme.

3.3.4 Container adaptors

Linear lists could be created from the elements associated by one pointer. Management procedures are very simple. Access of elements could be provided using `operator[]` only. It is possible to supplement the typical linear lists by additional management procedures and external communications, in order to make several standard modifications.

1. *Stack*. New elements can be written and read at the bottom of the list only (last accessed element) according to paradigm *LIFO* (Last In, First Out).
2. *Queue*. New elements can be written at the bottom of the list, and only to the first element is available for reading - according to paradigm *FIFO* (First In, First Out).
3. *Deque* (double-ended queue). New elements can be written and read at the bottom as well as at the top of list.

Such reorganized structure of list type are widely used not only for advanced applications but also for a systematic programs. For example, the stack is useful for storing the parameters from the interrupted process. Sometimes local data could be organized using stack and deque. Queues are popular for buffer memory organizing, when it is necessary to collect the data from equipment of low activity.

Table 3.32 represents the members of container adaptors: `stack` and `queue`.

Differences. The `stack` class supports a last-in, first-out (*LIFO*) data structure. Analogy in physics could be formulated as follows: a stack of cubes in vertical direction. Elements (cubes) may be inserted or removed only from the top of the stack. Top of the `stack` represents the last element at the end of container. The reason to use the `stack` is associated with the restriction to accessing only the top element.

The `queue` class supports a first-in, first-out (*FIFO*) data structure. Analogy in social life could be formulated as follows: array of people in one line standing and waiting the access to supermarket teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected.

Table 3.32. Members of container adaptors.

Members		Complexity	stack	queue
general	constructor	*	constructor	constructor
capacity	size	*	size	size
	max_size	*	max_size	max_size
	empty	O(1)	empty	empty
iterators	begin	O(1)	begin	begin
	rbegin	O(1)	rbegin	rbegin
	end	O(1)	end	end
	rend	O(1)	rend	rend
element	top	*	top	
access	front	O(1)		front
	back	O(1)		back
modifiers	push	O(1)	push	push
	pop	O(1)	pop	pop

The reason to use the **queue** is associated with the restriction of accessing the front and back elements only.

Methods. Methods related to the capacity such as **size()**, **max_size()**, **empty()** are present in both containers. Method **size()** returns the number of elements in the container. Method **max_size()** returns the maximum possible number of elements in the container. Method **empty()** checks if the container is empty or not. Iterators are the main tool for container manipulating realized in both containers. Methods **begin()** and **end()** return an iterator to the beginning of the container and to the end of the container respectively. Methods **rbegin()** and **rend()** return a reverse iterator to the reverse beginning of the container and to the reverse end of the container respectively.

Element access for **stack** container is realized using operator **top()** only (according to the stack ideology), which returns the last element. Element access for **queue** container is realized using operators **front()** and **back()**, which return the first and last elements.

Modifiers **push()** and **pop()** are realized for both containers, which allow to insert and to extract the members at the end of the container.

STACK. Table 3.33 represents programming implementation of stack and Table 3.34 represents the output of working program. Function **main()** contains declaration of of stack object **deor** of integer type.

```
stack <int> deor;
```

Filling by elements is implemented by two loops using operator **push()**.

```
for (int i=0; i<=3; i++) deor.push(i*10);
for (int i=5; i<=7; i++) deor.push(i*10);
```

Loop **while()** was used for stack output. If loop condition is not satisfied (in case the stack is not empty as **(!deor.empty() == true)** or **(deor.empty() == false)**) output will be provided from the end using operator **top()** and following erasing of stack members my means of operator **pop()**.

Table 3.33. Container stack.	Program con21.cpp.
<pre> #include <cstdlib> #include <iostream> #include <stack> using namespace std; int main(int argc, char *argv[]) { stack <int> deor; for (int i=0; i<=3; i++) deor.push(i*10); for (int i=5; i<=7; i++) deor.push(i*10); while (!deor.empty()) { cout << deor.top() << " "; deor.pop(); }; cout << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 3.34.	Output of program con21.cpp.
<pre> 70 60 50 30 20 10 0 Press any key to continue ... </pre>	
<pre> while (!deor.empty()) { cout << deor.top() << " "; deor.pop(); }; </pre>	

Table 3.34 represents output of working program. Output sequence is presented in the opposite direction to input according to the input scheme in two loops: 0, 10, 20, 30, 50, 60, 70.

Fig. 3.11 illustrates data organization scheme for container **stack**. Linear dependence is manipulated from the end of array.

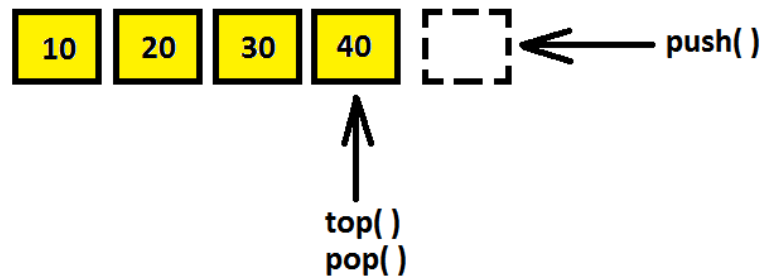


Fig. 3.11. Container **stack**. Data organization scheme.

QUEUE. Table 3.35 represents program implementation of **queue** container, and Table 3.36 represents the output of working program.

First block of the **main()** function contains declaration of queue object **nora1** of integer type

```
queue <int> nora1;
```

Data filling was provided in one-dimensional loop using operator **push()**:

```
for (int i=0; i<=4; i++)
    nora1.push(i*10);
```

Loop **while()** was used for queue output. It looks quite similar to the stack implementation. If the loop condition is not satisfied (in case the stack is not empty as **(!nora1.empty() == true)** or **(nora1.empty() == false)**) output will be provided from the front (first element) using operator **front()** and following erasing of stack members by means of operator **pop()**.

```
while (!nora1.empty())
{
    cout << nora1.front() << " ";
    nora1.pop();
};
```

The first line of Table 3.34 represents a queue which repeats the number input order in loop.

The second block of the **main()** function contains declaration of queue object **nora2** of integer type

```
queue <int> nora2;
```

Data filling was provided in one-dimensional loop using operator **push()**:

```
for (int j=5; j<=9; j++)
    nora2.push(j*10);
```

Using of operator **back()** allows to receive the last previously pushed member (to receive the pointer of mentioned member) which will be erased using operator **pop()** at least.

```
while (!nora2.empty())
{
    cout << nora2.back() << " ";
    nora2.pop();
};
```

Table 3.35. Container queue.	Program con22.cpp.
<pre> #include <cstdlib> #include <queue> #include <iostream> using namespace std; int main(int argc, char *argv[]) { queue <int> nora1; // 1st for (int i=0; i<=4; i++) nora1.push(i*10); while (!nora1.empty()) { cout << nora1.front() << " "; nora1.pop(); }; cout << endl; queue <int> nora2; // 2nd for (int j=5; j<=9; j++) nora2.push(j*10); while (!nora2.empty()) { cout << nora2.back() << " "; nora2.pop(); }; cout << endl; system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 3.36.	Output of program con22.cpp.
<pre> 0 10 20 30 40 90 90 90 90 90 Press any key to continue . . . </pre>	

The second line of Table 3.36 represents a queue which repeats the last element only (because first, after first, etc, elements outgoing).

Fig. 3.12 illustrates data organization scheme for container `queue`. Linear dependence is manipulated from the end of the array.

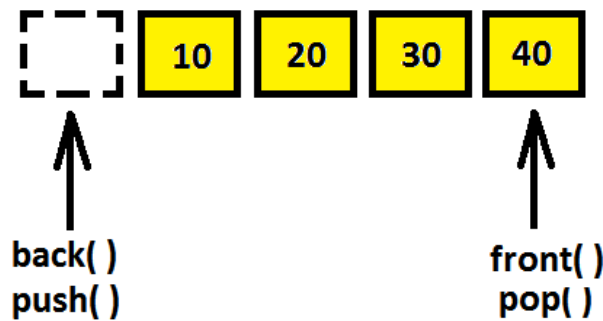


Fig. 3.12. Container queue. Data organization scheme.

3.4 Sorting

The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

Donald Knuth ^b

Let's assume the vector (one dimensional (1D) array) filled with elements. *Sorting* algorithm is an algorithm devoted to reordering of vector elements due to certain rule (increasing or decreasing value of parameter). In many cases, lexicographical order (for letters, items etc) and numerical order (for numbers, expressions, etc) could be titled as *ordering rules*. Sorting must be provided for two important purposes:

- i) to create user-friendly, human-readable basis sets (ordered arrays are more understandable in comparison to unordered);
- ii) to create basis sets for another programs which will use the ordered arrays several times and more effective.

General notes. Table 3.37 shows a list of sorting methods and their implementations. Twelve algorithms belonging to the *exchanging*, *selection*, *insertion*, *partitioning* and *merging* methods, represent a "gold fund" of the classical sorting algorithms. For more detailed explanations see Liberty [24], Sedgewick [44] publications. While selecting the algorithms for practical purposes, it is necessary to pay an attention to stability of algorithms, which depends on the data structure, chaos order, data complexity, etc. Problems of task standardisation in the initial stage of program designing predetermines the mobility, portability, possibilities for implementation into another projects in future etc. For standardisation problems see Sutter [48], [49], [3] publications.

Table 3.38 represents the possibility to compare mentioned algorithms using complexity $O(n)$ criteria for different cases: the best, average and the worst. This criterion is presented in the interval between $O(n)$ (ideal case) and $O(n^2)$ (worst case).

^bDonald Ervin Knuth (born 1938) - computer scientist and Professor Emeritus at Stanford University, author of the multi-volume work "The Art of Computer programming". Knuth was called the "father" of the analysis of algorithms.

Table 3.37. Sorting methods and its implementations.			
Method	Name	Stable	File name
Exchanging	Bubble_sort	Yes	sor01.cpp
Exchanging	Cocktail_sort	Yes	sor02.cpp
Exchanging	Comb_sort	No	sor03.cpp
Exchanging	Odd_even_sort	Yes	sor04.cpp
Selection	Heap_sort	No	sor10.cpp
Selection	Selection_sort	No	sor11.cpp sor12.cpp
Insertion	Shell_sort	No No	sor20.cpp sor21.cpp
Insertion	Insertion_sort	Yes	sor22.cpp
Partitioning	Quick_sort	Depends	sor30.cpp
Merging	Merge_sort	Yes	sor40.cpp

For typical sorting algorithms, quality could be expressed through good behaviour of $O(n \log n)$ and bad behaviour of $O(n^2)$. In average case, an ideal but not possible behaviour for a sort is $O(n)$. Comparison-based sorting algorithms evaluate the elements of the list via an abstract key comparison operation. Such algorithms need at least $O(n * \log n)$ comparisons for most inputs.

All twelve algorithms, belonging to the mentioned five methods (see Tables 3.39, 3.41, etc.) are presented in this section according to one-style manner. All projects contain function `main()`, function `show()`, and corresponding sorting function `Something_sort()` according to scheme in Table 3.37, where `something_` must be exchanged by `Bubble_`, `Comb_`, etc. Several projects contain also function `swap()`.

Function `void show()` is prepared for output of sorted/unsorted data array. Function `void show(int *AA, int k, char a)` requires three input parameters: pointer to array (presented here as a reference `*AA`; number of array members `k`, integer; parameter of sorting phase `a`, character: initial - 'I', final - 'F', partially sorted - 'T').

Function contains the declaration of file variable `fout1` associated to the output file `rez.txt`.

```
ofstream fout1("rez.txt", ios::out | ios::binary);
```

Table 3.38. Sorting methods: complexity $O(n)$ as a quality parameter.				
Name	Best	Average	Worst	Memory
Bubble_sort	n	n^2	n^2	1
Cocktail_sort	n	n^2	n^2	1
Comb_sort	n	$n \log n$	n^2	1
Odd_even_sort	n	n^2	n^2	1
Heap_sort	$n \log n$	$n \log n$	$n \log n$	1
Selection_sort	n^2	n^2	n^2	1
Shell_sort	n	$n(\log n)^2$	$n(\log n)^2$	1
Insertion_sort	n	n^2	n^2	1
Quick_sort	$n \log n$	$n \log n$	n^2	$\log n$
Merge_sort	$n \log n$	$n \log n$	$n \log n$	n

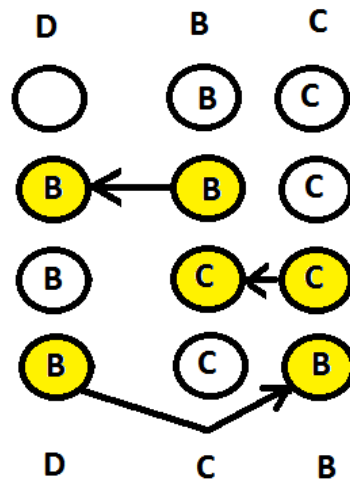


Fig. 3.13. Swapping operation.

Function contains 1D loop where indexed member of array `AA[ ]` will be outputted into file using operator `fout1` as well as the screen using operator `cout`.

```
for (int i=0; i<k; i++)
{
    cout << " " << AA[i];
    fout1 << " " << AA[i];
};
```

Function `swap()` is presented for the elementary swapping of two objects when data type is established as stabilar (for example, integer)

```
void swap(int &B, int &C)
{
    int D;
    D=B; B=C; C=D;
}
```

or data type will be determined in processing. Exchange schema is visualized in Fig. 3.13. Function requires two parameters - pointers to objects `BB` and `CC`. In that case, usage of template with declaring of abstract class `X` is necessary:

```
template <class X> void swap(X &BB, X &CC)
{
    X something;
    something = BB;
    BB = CC;
    CC = something;
}
```

In order to simplify the data section, function `main()` contains declaration and initialization of the static array `iAA[ ]` containing disordered integer values. Number of `iAA` array members is presented as user-requested variable `nAA`.

```
int iAA[] = {80,89,57,53,35,11,42,29,88,16};
int nAA = 10;
```

Function `show()` with input parameter 'I' and input parameter 'F' shows unordered (initial) and ordered (final) arrays. Function `Something_sort(iAA,nAA)` performs corresponding task of different kind of sorting.

```
show(iAA,nAA,'I');
Something_sort(iAA,nAA);
show(iAA,nAA,'F');
```

3.4.1 Exchanging

Exchanging method represents several programming implementations, where order exchange processes takes place. Several algorithms such as *Bubble-sort*, *Cocktail-sort*, *Comb-sort*, *Odd-even-sort*, will be described below.

Bubble-sort. Table 3.39 represents programming implementation of *Bubble-sort* algorithm.

Bubble-sort belongs to the one of the oldest and simplest sorting algorithms (first publication [11] of Demuth at 1956). *Bubble-sort* is called as a *model algorithm* for introductory computer science students - because it is a useful tool to analyze the concept of an algorithm: easy to understand and to implement.

Complexity of algorithm in worst case $O(n^2)$ means that its efficiency could be quite satisfactory for small n values only. But in other cases, especially with the dramatically increasing of n , complexity increases in orders. In that case, algorithms like *insertion sort* are usually considerably more efficient.

Bubble-sort is a sorting algorithm which works in accordance with such three conditions: i) continuously steps through a list, ii) swapps two items in right order; iii) repeats two previous operations until all items appear in the requested order.

Function `Bubble_sort()` contains two loops one inside other: `do...while (k>1)` loop and inside for (`int i=0; i<k-1; i++`) ... loop. First loop requests the passes through area of data, and second loop arranges the data in ascending or descending direction using swapping operation.

According to Knuth note in [20], efficiency of *bubble-sort* is quite low. For example, let's assume an array containing N members where one element only is out of order. For such case, *bubble-sort* will take $2 * N$ time for reordering. However in case of two unsorted elements, *bubble-sort* will take only at most $3 * N$ time. It could be concluded that *bubble-sort* is inefficient, but very stable algorithm.

Inner loop `for()` arranges the values according to the inner loop parameter `kk`. After finishing of inner loop, first loop parameter `k` becomes to equal to `kk`.

```
kk=0;
for (int i=0; i<k-1; i++)
{
    if (AA[i] > AA[i+1])
    {
        swap(AA[i],AA[i+1]);
        kk=i+1;
    };
};
k=kk;
```

Table 3.39. Bubble_sort.	Program sor01.cpp.
<pre> #include <cstdlib> #include <iostream> #include <fstream> using namespace std; void swap(int &B, int &C) { int D; D=B; B=C; C=D; } void Bubble_sort(int *AA, int m) { int kk, a1,a2; int k=m; do { kk=0; for (int i=0; i<k-1; i++) { // ascending if (AA[i] > AA[i+1]) { swap(AA[i],AA[i+1]); kk=i+1; } }; k=kk; show(AA,m,'T'); } while (k>1); } </pre>	<pre> int main(int argc, char *argv[]) { int iAA[]={80,89,57,53,35, 11,42,29,88,16}; int nAA=10; show(iAA,nAA,'I'); Bubble_sort(iAA,nAA); show(iAA,nAA,'F'); system("PAUSE"); return EXIT_SUCCESS; } void show(int *AA, int k, char a) { ofstream fout1("rez.txt", ios::out ios::binary); cout << a << " "; fout1 << a << " "; for (int i=0; i<k; i++) { cout << " " << AA[i]; fout1 << " " << AA[i]; }; cout << endl; fout1 << endl; fout1.close(); } </pre>

Table 3.40.	Output of program sor01.cpp.
I	80 89 57 53 35 11 42 29 88 16
T	80 57 53 35 11 42 29 88 16 89
T	57 53 35 11 42 29 80 16 88 89
T	53 35 11 42 29 57 16 80 88 89
T	35 11 42 29 53 16 57 80 88 89
T	11 35 29 42 16 53 57 80 88 89
T	11 29 35 16 42 53 57 80 88 89
T	11 29 16 35 42 53 57 80 88 89
T	11 16 29 35 42 53 57 80 88 89
T	11 16 29 35 42 53 57 80 88 89
F	11 16 29 35 42 53 57 80 88 89

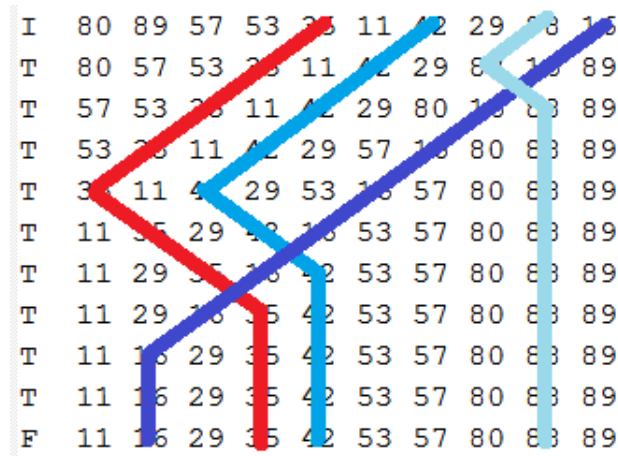


Fig. 3.14a. Visualization of dynamics. Output file of *Burple sort* (Table 3.40).

It means, that area of passes decreases with each succesfull inner loop. Algorithm works until no swaps have occurred on the last pass.

Table 3.40 represents output of working program. Initial (I) as unsorted and final (F) as sorted data are presented at first and last line. Intermediated (T) as partially sorted data are presented in lines between mentioned. "Bubble" effect is easy to understand by analysing the place of current member, for example 35 and 42. Fig. 3.14a represents visualization of sorting dynamics. "Bubble" represents a temporary group of members (local ordered data), which are travelling together some part of way until full reordering.

Cocktail sort. *Cocktail-sort* belongs to the group of sofisticated algorithms. known as *Bidirectional bubble sort* and *cocktail shaker sort*.

Main differences between *bubble sort* and *cocktail sort* are in algorithm implementation: *cocktail sort* sorts in both directions on each pass through the list.

Table 3.41 represents programming implementation of *Cocktail-sort* algorithm. Function `Cocktail_sort` contains `do...while()` loop and two inside loops. Parameter `bAction` of boolean type is treated as a main decision parameter and main exit parameter. Exit from loop `do...while()` occurs in one case only: if no swaps occurred. If no elements have been swapped, requested array is sorted (ordered).

```
if (bAction==false) return;
```

First inner loop (incremented value of loop variable, `i++`) is devoted for testing whether the two elements are in the wrong order. In that case swap of elements is necessary. Second inner loop (decremented value of loop variable, `i--`) is devoted for testing whether the two elements are in the wrong order. In that case swap of elements is also necessary. Difference between the two very similar conditions is easy by comparing dynamics of loop variable or, in other words, direction of array checking: direct or reversing respectively.

Table 3.41. Cocktail_sort.

Program sor02.cpp.

```

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

void swap(int &B, int &C)
{
    int D;
    D=B; B=C; C=D;
}

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
                11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Cocktail_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE");
    return EXIT_SUCCESS;
}

void Cocktail_sort(int *AA, int m)
{
    bool bAction;
    do
    {
        bAction=false;
        for (int i=0; i<=(m-2); i++)
        {
            if (AA[i] > AA[i+1])
            {
                swap(AA[i],AA[i+1]);
                bAction = true;
            };
        };
        show(AA,m,'t');
        if (bAction==false) return;
        // exit the outer loop here
        // if no swaps occurred.

        bAction=false;
        for (int i=(m-2); i>=0; i--)
        {
            if (AA[i]>AA[i+1])
            {
                swap(AA[i],AA[i+1]);
                bAction=true;
            };
        };
        show(AA,m,'T');
    }
    while (bAction==true);
    // if no elements have been swapped,
    // then the list is sorted
}

```

```

do
{
    bAction=false;
    for (int i=0; i<=(m-2); i++) {...};

    if (bAction==false) return;

    bAction=false;
    for (int i=(m-2); i>=0; i--) {...};
}
while (bAction==true);

```

The first inner loop is presented in direction where loop variable increases. In case when two neighbour elements are in the wrong order, swap operation orders it. After that, parameter `bAction` becomes true value.

```

for (int i=0; i<=(m-2); i++)
{
    // test whether the two elements are in the wrong order
    if (AA[i] > AA[i+1])
    {
        swap(AA[i],AA[i+1]);
        bAction = true;
    };
};

```

The second inner loop is presented in direction, when loop variable decreases.

In case when two neighbour elements are in the wrong order, swap operation orders it. After that, parameter `bAction` becomes true value.

```

bAction=false;
for (int i=(m-2); i>=0; i--)
{
    // test whether the two elements are in the wrong order
    if (AA[i]>AA[i+1])
    {
        swap(AA[i],AA[i+1]);
        bAction=true;
    };
};

```

Table 3.42 represents output of working program. Initial (I) as unsorted and final (F) as sorted data are presented at first and last line. Intermediated (T) as a partially sorted data are presented in lines between mentioned. "Bubble" effect is very small, comparing to the case of *bubble-sort*. Fig. 3.14b represents visualization of sorting dynamics. "Bubble" represents a group of members of negligible amount, which are swinging at several local places. *Cocktail-sort* algorithm is more efficient comparing with *bubble-sort*, due to double loop of intercrossing directions.

Table 3.42.	Output of program sor02.cpp.									
I	80	89	57	53	35	11	42	29	88	16
t	80	57	53	35	11	42	29	88	16	89
T	11	80	57	53	35	16	42	29	88	89
t	11	57	53	35	16	42	29	80	88	89
T	11	16	57	53	35	29	42	80	88	89
t	11	16	53	35	29	42	57	80	88	89
T	11	16	29	53	35	42	57	80	88	89
t	11	16	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

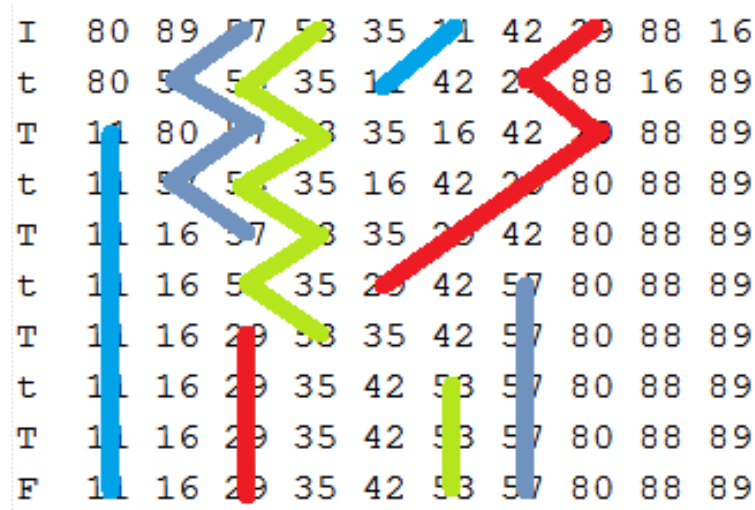


Fig. 3.14b. Visualization of dynamics. Output file of *Cocktail-sort* (Table 3.42).

Comb-sort. *Comb-sort* is a sorting algorithm of the end of XX century. Table 3.43 represents programming implementation of *Comb-sort* algorithm, and Table 3.44 shows output of working program. Firstly originally designed by Włodzimierz Dobosiewicz in 1980 and later popularized by Stephen Lacey and Richard Box in 1991, *Comb-sort* becomes quite popular due to its simplicity and original manipulating ideas.

Comb-sort improves on *Bubble-sort*, and rivals algorithms like *Quick-sort*. The basic idea is to eliminate the small values near the end of the array, since these essentially slow the sorting in a *Bubble-sort*.

The main idea of algorithm is to construct a gap for the working area. Initially, `iGap` is equal to the number of array elements, but in the loop, it will be decreased.

```
iGap = int((double)iGap / dG);
if (iGap<1) iGap=1;
```

This gap will be used in `for()` loop as an parameter for limiting the number of cycles. Loop variable will be increased with jump (`i+iGap`).

Table 3.43. Comb_sort.

Program sor03.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
                11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Comb_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE");
    return EXIT_SUCCESS;
}

void Comb_sort(int *AA, int m)
{
    double dG = 1.247330950103979;
    int iGap=m; bool bDone=false;
    do
    {
        iGap = int((double)iGap / dG);
        if (iGap<1) iGap=1;
        bDone= false;
        for (int i=0;
            (i+iGap) <= m;
            i++)
        {
            if (AA[i] > AA[i+iGap])
            {
                swap(AA[i],
                    AA[i+iGap]);
                bDone=true;
            };
        };
        show(AA,m,'T');
    }
    while (!( (iGap == 1)&&
        (bDone == false)));
}

void swap(int &B, int &C)
{
    int D; D=B; B=C; C=D;
}

for (int i=0; (i+iGap) <= m; i++)
{
    ...
};

```

Finally, element exchange condition is checked using operator `if()`. Swap operation will be done between elements, separated by gap `AA[i]` and `AA[i+iGap]`, but not between neighbour elements. At the end, the gap converges to one, and swap operation will be done between neighbour elements.

Table 3.44.	Output of program sor03.cpp.										
I	80	89	57	53	35	11	42	29	88	16	
T	80	16	26	53	35	11	42	29	88	89	
T	42	16	26	53	35	11	80	29	88	89	
T	35	11	26	29	42	16	57	53	88	89	
T	29	11	16	35	42	26	57	53	88	89	
T	16	11	29	26	42	35	57	53	80	89	
T	11	16	26	29	35	42	53	57	80	88	
T	11	16	26	29	35	42	53	57	80	88	
F	11	16	26	29	35	42	53	57	80	88	

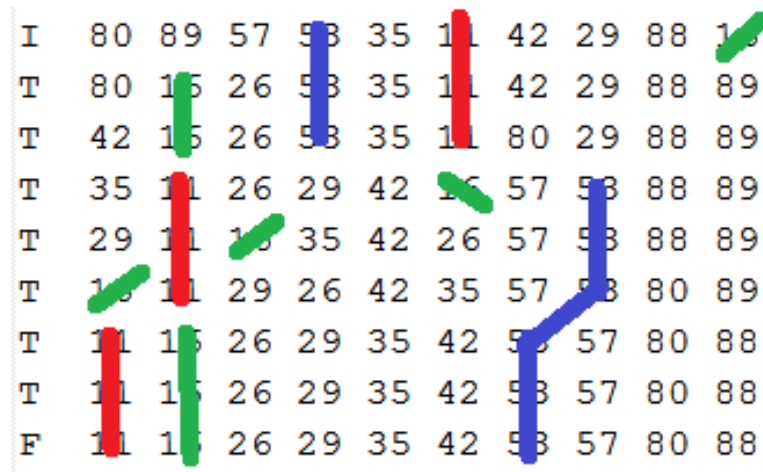


Fig. 3.15. Visualization of dynamics. Output file of *Comb-sort* (Table 3.44).

```

if (AA[i] > AA[i+iGap])
{
    swap(AA[i], AA[i+iGap]);
    bDone=true;
};

```

Table 3.44 represents output of working program. "Bubble" effect is absent, comparing to the *Bubble-sort*.

Fig. 3.15 represents visualization of sorting dynamics. Several stabilar group of members are present in local places, but chaotic selection orders the members very rapidly. *Comb-sort* algorithm is more efficient comparing to the *Cocktail-sort*, due to floating gap.

Odd-even-sort. *Odd-even-sort* or *Odd-even transposition sort* is a not complicated sorting algorithm, developed in order to use on parallel processors. Similar to *Bubble-sort*, presented *Odd-even-sort* algorithm works as following:

- i) compares all (odd and even) - indexed pairs of adjacent elements in the array; in case if a pair is wrong ordered the elements are switched (until the second is larger than the first);

- ii) next step repeats this for (even, odd)-indexed pairs;
- iii) then it alternates between (odd, even) and (even, odd) steps;
- iv) array is sorted when all elements are ordered (the second is larger than the first, etc).

Table 3.45 represents programming implementation of *Odd-even-sort* algorithm, and Table 3.46 shows output of working program.

Loop `while()` is organized for analyzing situation when it is necessary to interrupt this algorithm - according to variable `bool bDone`. Each swapping gives `bDone=false` value, but each starting of `for()` loop gives `bDone=true` value. No swapping operations (`bDone=true` value only) means that all data is ordered and loop `while()` is necessary to quit.

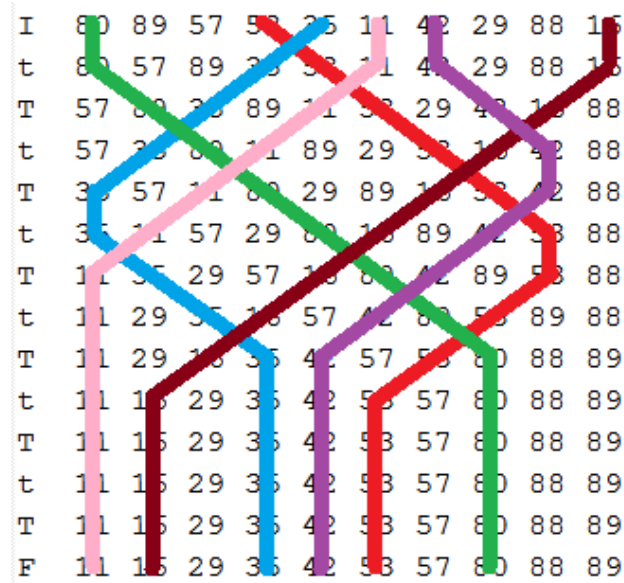
Table 3.45. Odd_even_sort.

Program sor04.cpp.

<pre> #include <cstdlib> #include <iostream> #include <fstream> using namespace std; void show(int *AA, int k, char a) { ofstream fout1("rez.txt", ios::out ios::binary); cout << a << " "; fout1 << a << " "; for (int i=0; i<k; i++) { cout << " " << AA[i]; fout1 << " " << AA[i]; }; cout << endl; fout1 << endl; fout1.close(); } int main(int argc, char *argv[]) { int iAA[]={80,89,57,53,35, 11,42,29,88,16}; int nAA=10; show(iAA,nAA,'I'); Odd_even_sort(iAA,nAA); show(iAA,nAA,'F'); system("PAUSE"); return EXIT_SUCCESS; } </pre>	<pre> void swap(int &B, int &C) { int D; D=B; B=C; C=D; } void Odd_even_sort(int *AA, int m) { bool bDone = false; while(!bDone) { bDone=true; for(int i=1; i<m-1; i+=2) { if(AA[i]>AA[i+1]) { swap(AA[i],AA[i+1]); bDone = false; }; }; show(AA,m,'t'); for(int j=0; j<m-1; j+=2) { if(AA[j]>AA[j+1]) { swap(AA[j],AA[j+1]); bDone = false; }; }; show(AA,m,'T'); }; } </pre>
---	---

Table 3.46. Output of program sor04.cpp.

I	80	89	57	53	35	11	42	29	88	16
t	80	57	89	35	53	11	42	29	88	16
T	57	80	35	89	11	53	29	42	16	88
t	57	35	80	11	89	29	53	16	42	88
T	35	57	11	80	29	89	16	53	42	88
t	35	11	57	29	80	16	89	42	53	88
T	11	35	29	57	16	80	42	89	53	88
t	11	29	35	16	57	42	80	53	89	88
T	11	29	16	35	42	57	53	80	88	89
t	11	16	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
t	11	16	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

Fig. 3.16. Visualization of dynamics. Output file of *Odd-even-sort* (Table 3.46).

Two parallel `for()` loops are organized with different loop variables: starting from one and zero (odd-even pair). Step of loop variable is equal to two. It means that both loops are sorting different elements of the same array.

```
for(int i=1; i<m-1; i+=2)
{ ... };
for(int j=0; j<m-1; j+=2)
{ ... };
```

Condition of element comparing is implemented in `if()` statement. In case, if a neighbour pair `AA[i], AA[i+1]` is in wrong order, swap operation must be done.

```
if (AA[i] > AA[i+1]) swap(AA[i],AA[i+1]);
```

Table 3.46 represents output of program, Fig. 3.16 represents visualization of sorting dynamics. "Bubble" effect is present - as a group of members at local place, which travels step by step. Dynamics of sorting is similar to the *Bubble-sort*.

3.4.2 Selection

If you think
you're a really good programmer...

Read (Knuth's)
Art of Computer Programming.

You should definitely
send me a résumé,
if you can read the whole thing.

Bill Gates ^c

Using different sorting methods, it would be possible to choose the most efficient working algorithm according to the type of unsorted data. In addition to the exchange method described previously, there are widely known several algorithms of selection method. Selection operation means that operations of element swapping (in ordering routine) will be done not in the whole amount of array but in selected region according to the certain limitations.

Heap_sort. *Heap-sort* is one of the more efficient version of *selection-sort*. In addition, *heap-sort* could be used for search of minimum value and maximum value between the array elements.

This task uses a data structure called a *heap* as a special type of *binary tree*. Heap formation must be made according to several formal rules.

1. Once the array of elements has been made into a heap, the root node is guaranteed to be the largest (or smallest) element.
2. When an element of array is removed from the array and placed at the end of the array, the heap must be rearranged - in a way, that the largest element remaining moves to the root.

Using the heap, finding of the next largest element takes $O(\log n)$ time, instead of $O(n)$ time for a linear scan which could be realized in case of simple *selection sort*. This circumstance allows *heap sort* to run in $O(n \log n)$ time, and this is also the complexity of the worst case.

Tables 3.47a, 3.47b represent *Heap-sort* implementation, Table 3.48 shows output of working program. Function `Heap_sort()` includes two loops. First loop contains a call to function `siftDown()`, when loop variable changes from half of number of arrays until zero (decrementing operation). See Table 3.48, where initial part of output file contains the rows marked with 't'.

^cWilliam Henry "Bill" Gates III (born 1955) - American business magnate, computer programmer and philanthropist. Previous chairman of computer software company *Microsoft*.

Table 3.47a. Heap_sort.

Program di01.cpp.

```

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

void swap(int &B, int &C)
{
    int D;  D=B; B=C; C=D;
}

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",  ios::out | ios::binary);
    cout  << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout  << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout  << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,  11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Heap_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE"); return EXIT_SUCCESS;
}

```

Reordering of half line from 0-th until middle-th elements occurs. This operation is devoted to build a heap out of the data.

```

for (int j = ((m/2)-1); j >= 0; j--)
    siftDown(AA, j, m);

```

The second loop contains the calls to function `swap()` and function `siftDown()`, when loop variable changes from number of arrays until one (decrementing operation).

Function `swap()` is devoted for swapping of 0-th and i -th array elements, but not neighbouring elements.

Table 3.47b. Heap_sort.

Program di01.cpp.

```

void siftDown(int *AA, int root, int bottom)
{
    int done, maxChild;  bool bDone = false;
    while ((root*2 < bottom) && (!bDone))
    {
        if          (2*root ==    bottom)      maxChild = 2*root;
        else if (AA[2*root] >  AA[2*root+1])    maxChild = 2*root;
        else                                           maxChild = 2*root+1;

        if      (AA[root] < AA[maxChild])
        {
            swap (AA[root],  AA[maxChild]);
            root = maxChild;
        }
        else      bDone=true;
        show(AA,bottom,'t');
    };
}

void Heap_sort(int *AA, int m)
{
    for (int j = ((m/2)-1); j >= 0; j--)
        siftDown(AA, j, m);

    for (int i=(m-1); i>=1; i--)
    {
        swap(AA[0],AA[i]);
        siftDown(AA, 0, i-1);
        show(AA,m,'T');
    };
}

```

It means that removing of the largest element from the heap occurs. Removed element is necessary to insert into the sorted array. For the first element, this would be position 0 of the array.

Reconstruction of heap continues, and the next largest element will be removed from heap and also inserted into the array.

See Table 3.48, where final part of output file contains the rows marked with 't' and 'T'. Fig. 3.17 visualizes ordering in heap when two swap operations are necessary. Reordering of half line from 0-th until middle-th elements occurs also for each pass, but forming of heap creates the dynamic structure, when right ordered array is placed on the right part of blue line (see Fig. 3.18).

Table 3.48. Output of program sor10.cpp.

```

I  80 89 57 53 35 11 42 29 88 16
t  80 89 57 53 88 11 42 29 35 16
t  80 89 57 53 88 11 42 29 35 16
t  80 89 88 53 57 11 42 29 35 16
t  80 89 88 53 57 11 42 29 35 16
t  80 89 88 53 57 11 42 29 35 16
t  89 80 88 53 57 11 42 29 35 16
t  89 88 80 53 57 11 42 29 35 16
t  89 88 80 53 57 11 42 29 35 16
t  88 16 80 53 57 11 42 29
t  88 80 16 53 57 11 42 29
t  88 80 57 53 16 11 42 29
T  88 80 57 53 16 11 42 29 35 89
t  80 35 57 53 16 11 42
t  80 57 35 53 16 11 42
t  80 57 35 53 16 11 42
T  80 57 35 53 16 11 42 29 88 89
t  57 29 35 53 16 11
t  57 53 35 29 16 11
T  57 53 35 29 16 11 42 80 88 89
t  53 42 35 29 16
t  53 42 35 29 16
T  53 42 35 29 16 11 57 80 88 89
t  42 11 35 29
t  42 35 11 29
T  42 35 11 29 16 53 57 80 88 89
t  35 16 11
t  35 29 11
T  35 29 11 16 42 53 57 80 88 89
t  29 16
T  29 16 11 35 42 53 57 80 88 89
t  16
T  16 11 29 35 42 53 57 80 88 89
T  11 16 29 35 42 53 57 80 88 89
F  11 16 29 35 42 53 57 80 88 89

```

```

for (int i=(m-1); i>=1; i--)
{
    swap(AA[0],AA[i]);
    siftDown(AA, 0, i-1);
};

```

Heap (as complete binary tree) is implemented compactly with an array. In that case, no special space is required for pointers, because the parent and children of each node can be found by arithmetic operation using array index.

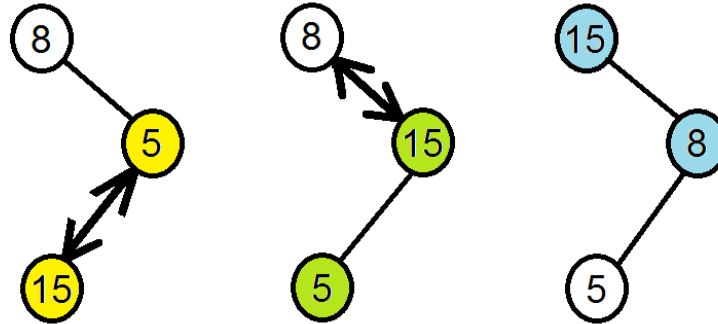


Fig. 3.17. Ordering in heap. Two swap operations.

```

void siftDown(int *AA, int root, int bottom)
{
    int done, maxChild; bool bDone = false;
    while ((root*2 < bottom) && (!bDone))
    {
        ...
    };
}

```

The `siftDown()` function extends the heap property for several elements of array: `AA[i-1]`, `AA[i]`, `AA[i+1]`. By swapping the values `AA[i]` and `AA[i+1]` the heap property for position i is established. At this point, the only problem is that the heap property might not hold for index $(i + 1)$. The sift-down function is applied tail-recursively to index $(i + 1)$ until the heap property is established for all elements.

The `siftDown()` function is very fast. In each step it only needs two comparisons and one swap. The first comparison `if () else if () else` is devoted for establishing of `maxChild` in comparison to parent - value `bottom`.

If special condition `if (AA[2*root] > AA[2*root+1])` is valid, `maxChild` occurs for current root (it means the re-establishing of heap).

```

if      ( 2*root == bottom)    maxChild = 2*root;
else if (AA[2*root] > AA[2*root+1]) maxChild = 2*root;
else                                     maxChild = 2*root+1;

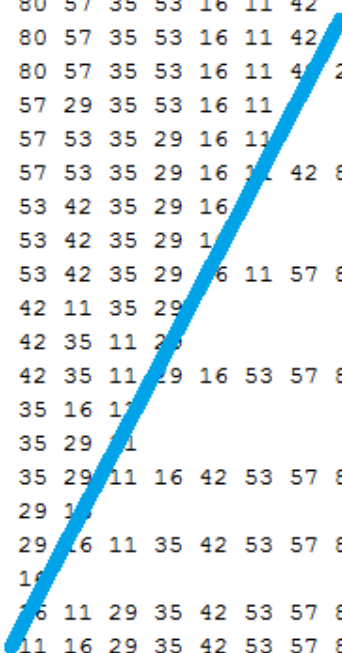
```

The second comparison is devoted for validation of root through `if` condition. If condition `(AA[root] < AA[maxChild])` is TRUE, swapping between `AA[root]` and `AA[maxChild]` is necessary.

```

if (AA[root] < AA[maxChild])
{
    swap(AA[root], AA[maxChild]);
    root = maxChild;
}
else
    bDone=true;

```



I	80	89	57	53	35	11	42	29	88	16
T	88	80	57	53	16	11	42	29	35	89
t	80	35	57	53	16	11	42			
t	80	57	35	53	16	11	42			
t	80	57	35	53	16	11	42			
T	80	57	35	53	16	11	42	29	88	89
t	57	29	35	53	16	11				
t	57	53	35	29	16	11				
T	57	53	35	29	16	11	42	80	88	89
t	53	42	35	29	16					
t	53	42	35	29	16					
T	53	42	35	29	16	11	57	80	88	89
t	42	11	35	29						
t	42	35	11	29						
T	42	35	11	29	16	53	57	80	88	89
t	35	16	11							
t	35	29	11							
T	35	29	11	16	42	53	57	80	88	89
t	29	16								
T	29	16	11	35	42	53	57	80	88	89
t	16									
T	16	11	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

Fig. 3.18. Visualization of dynamics. Output file of *Heap-sort* (Table 3.48).

Selection-sort. *Selection-sort* belongs to the algorithms of so called *in-place* comparison sort. Algorithm contains the name of corresponding method (belongs to selection method). Computational complexity is equal to $O(n^2)$, it means, that inefficiency for a large data arrays is a main so called "flip-side" effect. Therefore it is necessary to point out, that algorithm does no more than n swaps. When swapping is expensive (in terms of wasting of computer resources, especially for complex data), this algorithm is quite effective.

Selection-sort works according such three conditions:

- i) finds the minimum value in the presented array;
- ii) swaps it with the value in the first position;
- iii) repeats these steps for the remainder of array.

Table 3.49 represents programming implementation of *Selection-sort* algorithm, and Table 3.50 shows output of working program.

The first loop `for()` is devoted for finding the maximal value in the array, as value `BB`.

```
for (int i=(mm-1); i>=0; i--)
{
    if (AA[i]>BB) BB=AA[i];
    while ((mm > 0)&&(AA[mm]==BB)) mm--;
};
```

Loop `while()` is devoted for limitation of pass through loop (if number of elements in array is equal to n , number of passes is equal to the same n).

Table 3.49. Selection_sort.

Program sor11.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
                11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Selection_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE");
    return EXIT_SUCCESS;
}

void swap(int &B, int &C)
{
    int D; D=B; B=C; C=D;
}

void Selection_sort(int *AA, int m)
{
    int BB,CC; int mm=m;
    BB=AA[mm];
    for (int i=(mm-1); i>=0; i--)
    {
        if (AA[i]>BB) BB=AA[i];
        while ((mm > 0)&&
            (AA[mm] == BB)) mm--;
    };
    while (mm>0)
    {
        CC=BB; BB=AA[mm];
        for (int j=(mm-1); j>=0; j--)
        {
            if (AA[j] == CC)
            {
                swap(AA[j], AA[mm]);
                mm--;
            }
            else
            {
                if (AA[j]>BB) BB=AA[j];
            }
        };
        while ((mm > 0)&&
            (AA[mm] == BB)) mm--;
        show(AA,m,'T');
    };
}

```

```

while (mm>0)
{
    CC=BB; BB=AA[mm];
    for (int j=(mm-1); j>=0; j--)
    { ... };
    while ((mm>0)&&(AA[mm]==BB)) mm--;
};

```

Swapping occurs between current $AA[j]$ and fixed CC , and after swapping operation between $AA[j]$ and $AA[mm]$ mm decreases.

Table 3.50.	Output of program sor11.cpp.
I	80 89 57 53 35 11 42 29 88 16
T	80 16 57 53 35 11 42 29 88 89
T	80 16 57 53 35 11 42 29 88 89
T	29 16 57 53 35 11 42 80 88 89
T	29 16 42 53 35 11 57 80 88 89
T	29 16 42 11 35 53 57 80 88 89
T	29 16 35 11 42 53 57 80 88 89
T	29 16 11 35 42 53 57 80 88 89
T	11 16 29 35 42 53 57 80 88 89
F	11 16 29 35 42 53 57 80 88 89

```
if (AA[j]==CC)
{
    swap(AA[j], AA[mm]); mm--;
}
else
{
    if (AA[j]>BB) BB=AA[j];
};
```

Fig. 3.19 visualizes dynamics of sorting as presented in output file of *Selection-sort* in Table 3.50). No bubble effects is present. Some part of data is replaced to true position after one operation, some part - after several operations. Number of passes is equal to the number of elements in array.

Table 3.51 represents another programming implementation of *selection sort* and Table 3.52 shows output of program. This program is written in simplified manner.

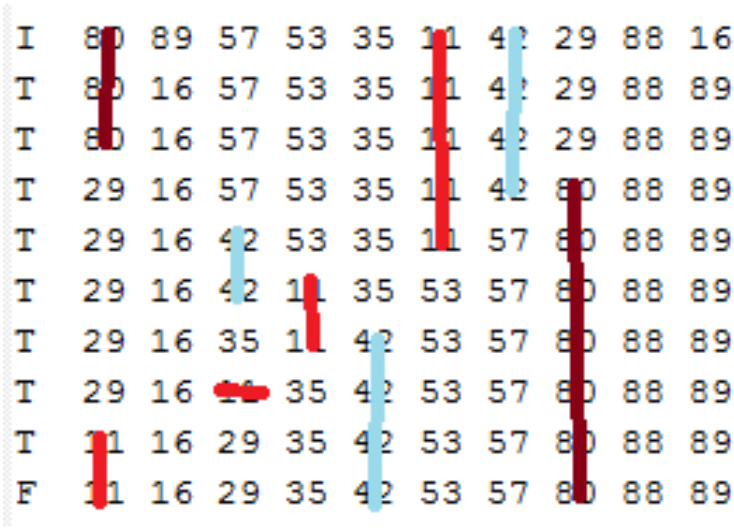


Fig. 3.19. Visualization of dynamics. Output file of *Selection-sort* (as presented in Table 3.50).

Table 3.51. Selection_sort_2.

Program sor12.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

void swap(int &B, int &C)
{
    int D; D=B; B=C; C=D;
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
        11,42,29,88,16};
    int nAA=10;
    show(iAA,nAA,'I');
    Selection_sort_2(iAA,nAA);
    show(iAA,nAA,'F');
    system("PAUSE"); return 0;
}

void Selection_sort_2(int *AA, int m)
{
    int BB;
    for (int i=0; i<=(m-1); i++)
    {
        BB=i;
        for (int j = (i+1);
            j <= (m-1);
            j ++))
        {
            if (AA[BB]>AA[j]) BB=j;
        };
        if (BB != i)
            swap (AA[i],AA[BB]);
        show(AA,m,'T');
    };
}

```

Table 3.52.

Output of program sor12.cpp.

```

I  80 89 57 53 35 11 42 29 88 16
T  11 89 57 53 35 80 42 29 88 16
T  11 16 57 53 35 80 42 29 88 89
T  11 16 29 53 35 80 42 57 88 89
T  11 16 29 35 53 80 42 57 88 89
T  11 16 29 35 42 80 53 57 88 89
T  11 16 29 35 42 53 80 57 88 89
T  11 16 29 35 42 53 57 80 88 89
T  11 16 29 35 42 53 57 80 88 89
T  11 16 29 35 42 53 57 80 88 89
T  11 16 29 35 42 53 57 80 88 89
F  11 16 29 35 42 53 57 80 88 89
Press any key to continue ...

```

3.4.3 Insertion

Next following method - insertion method - is known from sixth decade of twenty century. Insertion of elements into the occupied place is always associated with the removing operation which was not necessary in previous - exchange and selection - methods.

Shell_sort. *Shell-sort* algorithm was invented by Shell^d who improved two popular algorithms the *bubble-sort* and the *insertion-sort* by essential detail: moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Table 3.53 represents programming implementation of *Shell-sort*, 1-st version, and Table 3.54 shows the output of working program.

Function **Shell_sort-1** contains several loops. The first loop represents the non-correct usage of **for** loop according to the syntax notation, but otherwise, is correct from compiler position. The first loop is devoted for receiving the maximum allowed **h** value (less than number of elements), which will be received using increasing loop variable: **h=3*h+1**:

```
for (h=1; h<=m/9; h=3*h+1);
```

Loop **for()** of second order is devoted for main manipulating operations. Loop variable **h** decreases with step as lowest common multiple to 3. This loop contains second loop inside.

```
for (; h>0; h/=3)
{
    ...
};
```

Loop **for()** performs selecting of array members **BB=AA[i]** and assigning of the new array members **AA[j+h]=BB**. Loop variable increases homogeneously from requested **h** values until **m** values, corresponding number of array elements. Condition of assignment is realized in the loop of third level, where loop variable **j** decreases.

```
for (i=h; i<m; i++)
{
    BB=AA[i];
    for (j=i-h; j>=0; j-=h)
    {
        ...
    };
    AA[j+h]=BB;
};
```

Moving operation for insertion purposes could be realized in case if the condition of comparing is true. Then previous position of element is changed into new one: **AA[j+h] = AA[j]**.

```
if (BB < AA[j]) AA[j+h] = AA[j];
else break;
```

^dDonald L. Shell (born 1924) - American computer scientist who designed in 1959 the *Shell-sort* sorting algorithm.

Table 3.53. Shell_sort-1.

Program sor20.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
        11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Shell_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE"); return EXIT_SUCCESS;
}

void Shell_sort(int *AA, int m)
{
    int i,h,j,BB;
    for (h=1; h<=m/9; h=3*h+1);
    for (; h>0; h/=3)
    {
        for (i=h;i<m;i++)
        {
            BB=AA[i];
            for (j=i-h; j>=0; j-=h)
            {
                if (BB < AA[j])
                    AA[j+h] = AA[j];
                else break;
            };
            AA[j+h]=BB;
            show(AA,m,'T');
        };
    };
}

```

Table 3.54 represents output of working program. It is obvious that full ordering starts from the smallest element, then next smallest, and so on. Reordering in the initial part (second..fifth line marked as 'T') shows, that partial ordering is made in chaotic manner, depending on loop variable *h*.

Shell-sort-2. Second implementation of *Shell-sort* presented and discussed by Liberty [24] is similar to the previous one. Table 3.55 represents code of *Shell-sort*, 2-nd version, Table 3.56 shows output of working program.

Table 3.54. Output of program sor20.cpp.

I	80	89	57	53	35	11	42	29	88	16
T	35	89	57	53	80	11	42	29	88	16
T	35	11	57	53	80	89	42	29	88	16
T	35	11	42	53	80	89	57	29	88	16
T	35	11	42	29	80	89	57	53	88	16
T	35	11	42	29	80	89	57	53	88	16
T	35	11	42	29	80	16	57	53	88	89
T	11	35	42	29	80	16	57	53	88	89
T	11	35	42	29	80	16	57	53	88	89
T	11	29	35	42	80	16	57	53	88	89
T	11	29	35	42	80	16	57	53	88	89
T	11	16	29	35	42	80	57	53	88	89
T	11	16	29	35	42	57	80	53	88	89
T	11	16	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

This version contains declaration of static 1D array containing values for describing of changing insertion intervals. Fig. 3.20 represents array distribution in logarithmic scale.

```
int cols[] = {1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968,
              861, 336, 112, 48, 21, 7, 3, 1};
```

Such values ($HH=cols[k]$) are used in second loop of the first level, in order to determine the starting value of loop variable in loop of second order.

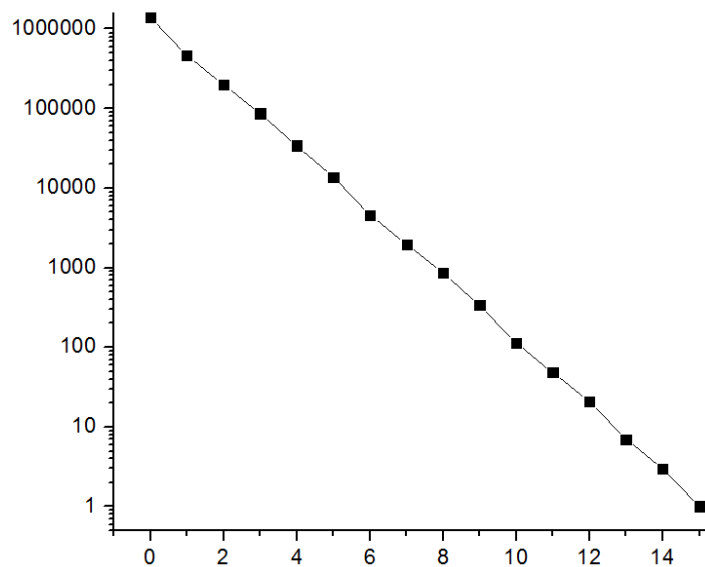


Fig. 3.20. Array of cols in lognormal distribution.

Table 3.55. Shell_sort-2.

Program sor21.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
        11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Shell_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE");
    return EXIT_SUCCESS;
}

void swap(int &B, int &C)
{
    int D; D=B; B=C; C=D;
}

void Shell_sort(int *AA, int m)
{
    int BB, j, HH;
    int cols[] = {1391376, 463792,
        198768, 86961,
        33936, 13776,
        4592, 1968,
        861, 336,
        112, 48,
        21, 7,
        3, 1};
    for (int k=0; k<16; k++)
    {
        HH=cols[k];
        for (int i=HH; i<m; i++)
        {
            BB=AA[i]; j=i;
            while ((j>=HH) &&
                (AA[j-HH]> BB) )
            {
                AA[j] = AA[j-HH];
                j -= HH;
            }
            AA[j] = BB;
        };
        show(AA,m,'T');
    };
}

```

```

for (int k=0; k<16; k++)
{
    HH=cols[k];
    for (int i=HH; i<m; i++)
        { ... };
};

```

The second version of *Shell-sort* is devoted for very large arrays, up to millions (biggest value in `cols` 1391376). The first version of *Shell-sort* is devoted for arrays of middle size, up to thousands members.

Table 3.56. Output of program sor21.cpp.

I	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	29	88	16	53	35	11	42	80	89	57
T	29	35	11	42	80	16	53	88	89	57
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

Table 3.56 represents output of working program. Second version of *Shell sort* is devoted for very large arrays. About 75% of output amount is presented without any changes, where output lines are in non-ordered form. Presented array for sorting contains 10 members only. When big values of in `cols` array are used in part of interval selecting, any changes occur. Only last three members 7,3,1 in `cols` array was used for array manipulation.

Insertion_sort. *Insertion-sort* is a simple sorting algorithm, which is very effective for the small arrays. With the increasing of array size, efficiency decreases rapidly. This is according to specificity of algorithm: new place of memory is requested. In some cases, there is possible to use the fields of shared memory (especially in arrays). Generally, *Insertion-sort* is quite expensive algorithm, because shifting all following elements over by one takes place.

For very large arrays, *shell sort* is the most efficient programming implementation in insertion method.

Insertion-sort algorithm works according such two conditions:

- i) taking items from the list - one-by-one;
- ii) inserting items in their correct position into a new sorted array.

Table 3.57 represents programming implementation of *Insertion-sort*, Table 3.58 shows output of working program.

Loop `for()` is presented for selecting of all members of array using `BB=AA[i]` (in initial part of loop) and also moving a member into the new place using expression `AA[j+1]=BB` (in final part of loop). This loop contains the loop `do{...}while()` of second order.

```
for ( int i=0; i<=(m-1); i++)
{
    BB=AA[i];    j=i-1;    just = false; ...    AA[j+1]= BB;
};
```

Table 3.57. Insertion_sort.

Program sor22.cpp.

```

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

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
        11,42,29,88,16};
    int nAA=10;
    show(iAA,nAA,'I');
    Insertion_sort(iAA,nAA);
    show(iAA,nAA,'F');
    system("PAUSE"); return EXIT_SUCCESS;
}

void Insertion_sort(int *AA, int m)
{
    int BB,j; bool just;
    for ( int i=0; i<=(m-1); i++)
    {
        BB=AA[i]; j=i-1; just = false;
        do
        {
            // ascending
            if (AA[j] > BB)
            {
                AA[j+1] = AA[j]; j--;
                if (j<0) just = true;
            }
            else just=true;
        }
        while (just==false);
        AA[j+1]= BB;
        show(AA,m,'T');
    };
};

```

Loop `do{...}while()` of second level is devoted for decision how long it is necessary to continue the looping operations. Boolean value `just` is used for such purposes. Value `FALSE` means that shifting operation occurs in the last loop and looping must be continued.

```

do
{ ... }
while (just == false);

```

Loop `do{...}while()` of second level contains `if()` statement where member shifting could be realized: `AA[j+1]=AA[j]`. Member ordering will be provided in ascending direction according to satisfying the condition `AA[j]>BB`.

Table 3.58.

Output of program sor22.cpp.

I	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	80	89	57	53	35	11	42	29	88	16
T	57	80	89	53	35	11	42	29	88	16
T	53	57	80	89	35	11	42	29	88	16
T	35	53	57	80	89	11	42	29	88	16
T	11	35	53	57	80	89	42	29	88	16
T	11	35	42	53	57	80	89	29	88	16
T	11	29	35	42	53	57	80	89	88	16
T	11	29	35	42	53	57	80	88	89	16
T	11	16	29	35	42	53	57	80	88	89
F	11	16	29	35	42	53	57	80	88	89

```
if (AA[j] > BB) // ascending
{
    AA[j+1] = AA[j]; j--;
    if (j<0) just = true;
}
else just=true;
```

Table 3.58 represents output of working program, Fig. 3.21 visualizes dynamics of *Selection-sort*. Array of ten mebers belongs to the type of small data amount, and sorting operation using this algorithm is very effective.

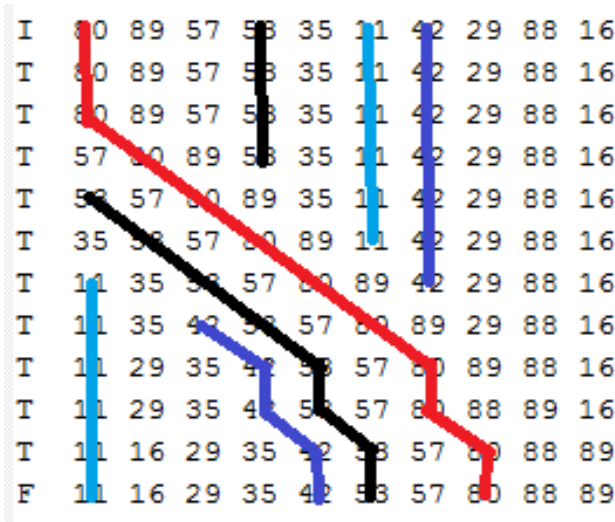


Fig. 3.21. Visualization of dynamics.
Output file of *Selection-sort* (Table 3.58).

3.4.4 Partitioning

Partitioning method represents a method containing the moving operation in the small area but not in whole array.

Quick-sort. *Quick-sort* is a divide and conquer algorithm which relies on a partition operation. An array of element is divided into two parts, and some positions called a *pivot* are selected. All elements smaller than the pivot are moved before it and all elements greater than the pivot are moved after it.

The lesser and greater subarrays are then recursively sorted. Efficient implementations of *Quick-sort* (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.

Together with its modest $O(\log n)$ space usage, *Quick-sort* is one of the most popular sorting algorithms and is available in many standard programming libraries.

The most complex issue in *Quick-sort* is to choose a good pivot element. Consistently poor choices of pivots can result in drastically slower $O(n^2)$ performance, if at each step the median is chosen as the pivot then the algorithm works in $O(n \log n)$. Finding the median however, is an $O(n)$ operation on unsorted arrays and therefore exacts its own penalty with sorting.

Table 3.59 represents programming implementation of *Quick-sort* and Table 3.60 shows output of working program. Function `Quick-sort()` contains three important parts.

The first part of function `Quick-sort()` is devoted for initial selecting of pivot as a half of arithmetical summ of `iLower` and `iHigher`.

```
int index =(int)((iLower+iHigher)/2);
int XXX   = AA[index];
```

The second part of function `Quick-sort()` is devoted to partition operation in loop `do{...}while()`. Recalculation of indeces `i` and `j` is provided in `while()` loop in order to use in swapping operation.

```
do
{
    while (AA[i]<XXX) i++;
    while (AA[j]>XXX) j--;
    if (i<=j)
    {
        swap(AA[i],AA[j]);  i++; j--;
    }
}
while (i<=j);
```

Conditions `AA[i]<XXX` and `AA[j]>XXX` must be satisfied for incremented and decremented changing of indexes, respectively.

The third part of function `Quick-sort()` is devoted for recursion. Two conditions must be satisfied for indeces `i` and `j` in order to call the same function `Quick_sort()`, but with changed input parameters. Such parameters were recalculated according to new position of pivot, and effectivity of swapping increases due to rebalancing nature.

```
if ( iLower <      j ) Quick_sort(AA, iLower, j      );
if ( i      < iHigher ) Quick_sort(AA, i,      iHigher);
```

Table 3.59. Quick_sort.	Program sor30.cpp.
<pre> #include <cstdlib> #include <iostream> #include <fstream> using namespace std; void show(int *AA, int k, char a) { ofstream fout1("rez.txt", ios::out ios::binary); cout << a << " "; fout1 << a << " "; for (int i=0; i<k; i++) { cout << " " << AA[i]; fout1 << " " << AA[i]; }; cout << endl; fout1 << endl; fout1.close(); } int main(int argc, char *argv[]) { int iAA[]={80,89,57,53,35, 11,42,29,88,16}; int nAA=10; show(iAA,nAA,'I'); Quick_sort(iAA,0,nAA-1); show(iAA,nAA,'F'); system("PAUSE"); return EXIT_SUCCESS; } </pre>	<pre> void swap(int &B, int &C) { int D; D=B; B=C; C=D; } void Quick_sort(int *AA, int iLower, int iHigher) { int i = iLower, j = iHigher; int index =(int)((iLower+iHigher)/2); int XXX = AA[index]; // partition do { while (AA[i]<XXX) i++; while (AA[j]>XXX) j--; if (i<=j) { swap(AA[i],AA[j]); i++; j--; }; show(AA,iHigher,'T'); } while (i<=j); // recursion if (iLower < j) Quick_sort(AA, iLower, j); if (i < iHigher) Quick_sort(AA, i, iHigher); } </pre>

Table 3.60 represents output of working program, Fig. 3.22 visualises the dynamics of *Quick-sort*. Using of recursion requires memory wasting (task for operating system).

Decreasing of working amount of array is associated with the properties of algorithm, when selected pivot is shifted from the middle to the one of side.

Table 3.60.	Output of program sor30.cpp.
I	80 89 57 53 35 11 42 29 88 16
T	16 89 57 53 35 11 42 29 88
T	16 29 57 53 35 11 42 89 88
T	16 29 11 53 35 57 42 89 88
T	16 29 11 35 53 57 42 89 88
T	16 11 29
T	11
T	11 16 29
T	11 16 29 35 42 57 53 89 88
T	11 16 29 35 42 57 53 89 88
T	11 16 29 35 42 57 53 80 88
T	11 16 29 35 42 57 53 80 88
T	11 16 29 35 42 53 57 80
T	11 16 29 35 42 53 57 80
F	11 16 29 35 42 53 57 80 88 89

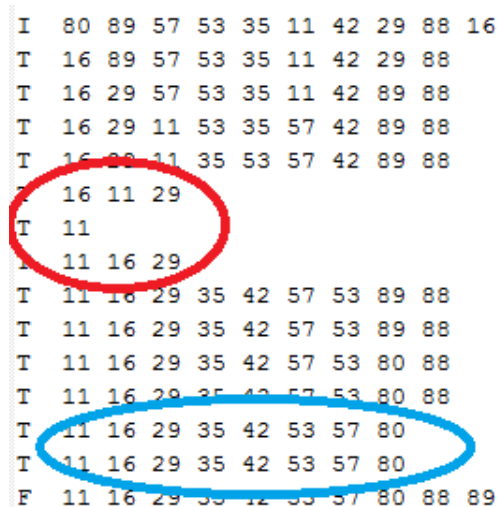


Fig. 3.22. Visualization of dynamics. Output file of *Quick-sort* (Table 3.60).

3.4.5 Merging

Merging belongs to the unique method of sorting due to specific manner of manipulation.

Merge-sort. *Merge-sort* algorithm uses the merging of the previously sorted arrays into new one, which after insertion is not chaotic, but sorted also (ordered insertion operation). Comparing of two neighbour elements and swapping; such two operations are used in the similar manner like other methods.

Along the algorithms described here, this is the first that scales well to very large arrays, because its worst-case running time is $O(n \log n)$.

Table 3.61. Merge_sort.

Program sor40.cpp.

```

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

void swap(int &B, int &C)
{
    int D;
    D=B; B=C; C=D;
}

void show(int *AA, int k, char a)
{
    ofstream fout1("rez.txt",
        ios::out | ios::binary);
    cout << a << " ";
    fout1 << a << " ";
    for (int i=0; i<k; i++)
    {
        cout << " " << AA[i];
        fout1 << " " << AA[i];
    };
    cout << endl; fout1 << endl;
    fout1.close();
}

int main(int argc, char *argv[])
{
    int iAA[]={80,89,57,53,35,
        11,42,29,88,16};
    int nAA=10;

    show(iAA,nAA,'I');
    Merge_sort(iAA,nAA);
    show(iAA,nAA,'F');

    system("PAUSE");
    return EXIT_SUCCESS;
}

void Merge(int *AA, int part1, int nnn)
{
    // Storage requirements
    int *TT;
    TT = new int[part1+1];

    int j = 0;
    // Reduction in data transfer
    while(j < part1)
    {
        TT[j] = AA[j]; j++;
    };
    // Infinity will be Sentinel
    TT[j] = 1000000000;

    int k=0; int i=0;
    while(j < nnn)
    {
        if (TT[i] <= AA[j])
            {AA[k] = TT[i]; i++;}
        else
            {AA[k] = AA[j]; j++;};
        k++;
    };
    while(i < part1)
    {
        AA[k] = TT[i]; i++; k++;
    };
    show(AA,nnn,'T');
}

void Merge_sort(int *AA, int m)
{
    int iPart,iLast; int *iP;
    if(m>1)
    {
        iPart = (int)((float)m/(float)2.0);
        Merge_sort( AA, iPart);
        iP=&AA[iPart];
        iLast = m; iLast -=iPart;
        Merge_sort( iP, iLast);
        Merge      ( AA, iPart, m);
    }
}

```

Table 3.62.	Output of program sor40.cpp.
I	80 89 57 53 35 11 42 29 88 16
T	80 89
T	35 53
T	35 53 57
T	35 53 57 80 89
T	11 42
T	16 88
T	16 29 88
T	11 16 29 42 88
T	11 16 29 35 42 53 57 80 88 89
F	11 16 29 35 42 53 57 80 88 89

Merge-sort algorithm has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in many programming languages. *Merge-sort* has been used in *Java* at least since 2000 in JDK1.3.

Table 3.61 represents programming implementation of *Merge-sort* and Table 3.62 shows the output of working program.

Table 3.62 represents output of working program, Fig. 3.23 visualises the dynamics of *Merge-sort* processes. Using of recursion requires memory wasting (task for operating system). Decreasing of working amount of array is associated with the properties of algorithm, when selected pivot is shifted from the middle to one of the sides.

```

I 80 89 57 53 35 11 42 29 88 16
T 80 89
T 35 53
T 35 53 57
T 35 53 57 80 89
T 11 42
T 16 88
T 16 29 88
T 11 16 29 42 88
T 11 16 29 35 42 53 57 80 88 89
F 11 16 29 35 42 53 57 80 88 89

```

Fig. 3.23. Visualization of dynamics. Output file of *Merge-sort* (Table 3.62).

3.5 Applied

3.5.1 Prime numbers

A *prime* number is a natural number (greater than 1) that has no positive divisors other than 1 and itself.

A set of *natural* numbers 1, 2, 3, 4, 5, 6, 7, 8, ..., etc. includes small but significant subset of *prime* numbers: 1, 2, 3, 5, 7, 11, 13, 17, 19, etc.

Eratosthenes^e proposed an effective algorithm called as a *Sieve of Eratosthenes* for search of prime numbers up to any given limit. The main idea of algorithm could be formulated as follows: marking the composites of each prime.

It is necessary to interactively mark the special numbers - as composite the multiples of each prime, starting with the multiples of 2. The sequence of numbers with the same difference, equal to that prime, will be removed (i.e. not prime). When the algorithm terminates, all the numbers in the array that are not marked are prime.

Table 3.63 represents programming implementation of *Sieve of Eratosthenes* built up in traditional OOP manner. Table 3.64 shows output of program. Function `main()` contains a creation of object `primes` of class type `SIEVE` and immediately requesting for constructor to create an 1D array containing 500 members. Two class methods such as `primes.selection()` and `primes.show2()` are devoted to prime number selection and outputting respectively.

```
SIEVE  primes(500);
        primes.selection();
        primes.show2();
```

Class `SIEVE` contains several protected members i.e. variables `int iMax_t` and public members - variables: pointer `*iP` of integer type, object `uPrimes` of vector type created using `<int>` template ; special functions constructor `SIEVE::SIEVE(int iMax)` and destructor `SIEVE::~~SIEVE()`; two class methods `void SIEVE::selection()` and `void SIEVE::show2()`.

1D array memory requesting occurs on time of object initialization (see function `main()`). Number of array elements (500) must be transmitted from `main()` to class constructor `SIEVE()`. Usage of protected class members is devoted for such complicated data exchange.

Parameter `int iMax` (number of array elements) is a local variable in constructor `SIEVE::SIEVE( )` In order to use it in class namespace, an association to the protected class members is necessary using column `(:)` technique.

```
SIEVE::SIEVE(int iMax): // constructor
    iMax_t (iMax)
```

After the operation mentioned, local variable `iMax` corresponds to class protected variable `iMax_t`.

The main part of constructor `SIEVE::SIEVE( )` contains dynamic allocation of memory using operator `new` and following initialization of 1D array in `for()` loop by number one. This initialization means, that according to the first assumption all numbers are primes (assumption before running sieve selection). An amount of memory is equal to `iMax_t`.

```
iP = new int [iMax_t];
for (int i=0; i<iMax_t; i++)
    iP[i] = 1;
```

Destructor `SIEVE::~~SIEVE( )` contains alone operator `delete` for erasing of dynamical 1D array.

```
delete iP;
```

Class method `void SIEVE::selection()` is presented for implementation of Erastotene idea. First of all it is necessary to set the limit of loop as a square root to total number of members using operator `pow()`.

^eEratosthenes of Cyrene (276 BC - 195 BC) - Greek philosopher and mathematician.

Table 3.63. Primes.

Program pri01.cpp.

```

#include <cstdlib>
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>
using namespace std;

class SIEVE
{
protected:
    int iMax_t;
public:
    int *iP;
    vector <int> uPrimes;
    SIEVE(int iMax); // constructor
    ~SIEVE();        // destructor
    void selection();
    void show2();
};

// constructor
SIEVE::SIEVE(int iMax):
    iMax_t (iMax)
{
    iP = new int [iMax_t];
    for (int i=0; i<iMax_t; i++)
        iP[i] = 1;
}

int main()
{
    SIEVE primes(500);
    primes.selection();
    primes.show2();
    system("PAUSE");
    return EXIT_SUCCESS;
}

// destructor
SIEVE::~SIEVE()
{
    delete iP;
}

void SIEVE::selection()
{
    int iLimit;
    double dA = (double)iMax_t;
    iLimit = (int)pow(dA,0.5);
    for (int i2=2; i2<iMax_t; i2++)
    {
        if (iP[i2] == 0) goto done;
        uPrimes.push_back(i2);
        // primes only
        if (i2 > iLimit) goto done;
        for (int j=i2; j<iMax_t; j+=i2)
            iP[j]=0;
    }
done: ;
}; // end for i2
}

void SIEVE::show2()
{
    int j=0;
    int iSize = uPrimes.size();
    for (int i=0; i<iSize; i++)
    {
        cout << setw(5) << uPrimes[i];
        j++;
        if (j>10) {j=0; cout << endl;}
    };
    cout << endl;
}

```

```
iLimit = (int)pow(dA,0.5);
```

Variable `iMax_t` represents requested amount of numbers to be checked. The first `for()` loop is presented for selecting all numbers starting from 2 until variable `iMax_t` by incremented step.

Table 3.64. Output of program pri01.cpp.

2	3	5	7	11	13	17	19	23	29	31
37	41	43	47	53	59	61	67	71	73	79
83	89	97	101	103	107	109	113	127	131	137
139	149	151	157	163	167	173	179	181	191	193
197	199	211	223	227	229	233	239	241	251	257
263	269	271	277	281	283	293	307	311	313	317
331	337	347	349	353	359	367	373	379	383	389
397	401	409	419	421	431	433	439	443	449	457
461	463	467	479	487	491	499				

Press any key to continue ...

```

for (int i2=2; i2<iMax_t; i2++)
{
    ...
}; // end for i2

```

Content of the first `for()` loop contains the following statements. There are two possibilities to claim that certain number is not prime.

Firstly, according to checking of statement `iP[i2] == 0`, number is fixed as a multiplication of two numbers (exactly not prime).

Secondly, there are no necessity to run loop in case, when loop variable exceeds `iLimit` variable. If such two conditions are not satisfied after finishing loop, certain number is fixed as prime, sending to the vector: `uPrimes.push_back(i2)`.

```

if (iP[i2] == 0) goto done;
uPrimes.push_back(i2); // primes
if (i2 > iLimit) goto done;
done: ;

```

The second loop, placed in the first loop, is devoted to interactive marking composites by zero (first assumption that all numbers are primes is the wrong assumption!). The multiples of each prime contain the sequences with the same difference, equal to that prime. Loop variable increases which step `j+=i2`

```

for (int j=i2; j<iMax_t; j+=i2)
    iP[j]=0;

```

Table 3.64 represents output of program. Primer number are presented as a sequence up to 500, as requested previously.

Fig. 3.24 represents the prime numbers selected using the *Sieve of Eratosthenes*, built up from the first principles. Composites are depicted in colours whereas primes left as uncoloured (white).

01	02	03	04	05	06	07	08	09	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Fig. 3.24. Sieve of Eratosthenes. Composites are depicted in colours whereas prime left as uncoloured.

3.5.2 Random numbers

A random sequence is a vague notion...
in which each term is unpredictable
to the uninitiated and whose digits
pass a certain number of tests
traditional with statisticians.

D. H. Lehmer ^f

A certain numeric sequence is said to be *statistically random* when it contains sequences which are not recognizable as regular sequences or patterns. Several physical processes represent the products characterized by parameters of statistically random type. Nuclear decay, fusion, particle scattering processes could be described using expressions of probabilities. Otherwise, statistical randomness does not necessarily imply "true" randomness, which must be objectively unpredictable. Pseudo-randomness is sufficient for many uses, such as statistics, hence the name statistical randomness.

In computer science, random numbers play an important role on environment imitation processes. General task could be formulated as follows: how to make the game unrepeatable and unpredictable?

Pure random numbers could be generated using specific hardware such as PCI/USB interface containing so called Geiger counter. Particle detector of such type is devoted for measuring of ionizing radiation in the surrounding. It detects the emission of nuclear radiation: alpha particles, beta particles or gamma rays by the ionization produced in a low-pressure gas in a Geiger tube. For the reason that nuclear decay processes have a stochastic nature, Geiger counter produces the time-resolved intensity distribution according to the decay activity in the presence locality. This decay information could be transferred after simple arithmetical operation into statistically random numbers (integers or real).

Many applications of randomness have led to the development of several different methods for generating random data. From ancient times, three general methods namely dice flipping,

^fDerrick Henry "Dick" Lehmer (1905 - 1991) - an American mathematician who devised the Lucas-Lehmer test for Mersenne primes. Working as a number theorist, he was a pioneer into early applied electronic computing.

coin flipping, shuffling of playing cards are useful for imitation purposes: see Press publication [51].

Computational algorithm, designed for generation of sequences of numbers or symbols that lack any pattern (appear random or pseudo-random) is called an *random number generator*, *RNG*.

RNG-1. One of the most common RNG is the linear congruential generator, which uses the recurrence to generate numbers $z_{i+1} = F * (z_i)$.

$$z_{i+1} = r * z_i * (1 - z_i) \quad (3.6)$$

Variable r plays the role of the coefficient for space covering. If variable z is present in the interval $z \in (0 \div 1)$, pseudo-random chaotic distribution is received in case of $r = 4.0$ and more only. In case if $r < 4.0$ - for example $r = 3.6$ - a hole in distribution is present. Hole means that several places are predictable and therefore they do not contain random numbers.

Table 3.65 represents programming implementation of the RNG-1 according to the previously presented formula. Table 3.66 shows output of program.

Being made in traditional loop form the number generation occurs in each `for()` cycle. For estimation purposes, output pairs containing cycle number `i` and generated pseudo-random number `zn` will be output to the file. At the end of each cycle, `zn` will be attributed to `z`.

```
for (int i=0; i<=1000; i++)
{
    zn = r * z * (1.0-z);
    z  = zn;
};
```

Table 3.65. RNG-1.

Program pro01.cpp.

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

int main(int argc, char *argv[])
{
    double z=0.4; double zn;
    double r=4.0; // 3.6
    ofstream fout("rez.txt", ios::out | ios::binary);
    for (int i=0; i<=1000; i++)
    {
        zn = r*z*(1.0-z);
        fout << i << " " << zn << endl;
        z=zn;
    };
    fout.close();
    system("PAUSE"); return EXIT_SUCCESS;
}
```

Table 3.66.

Output of program pro01.cpp (r=0.4).

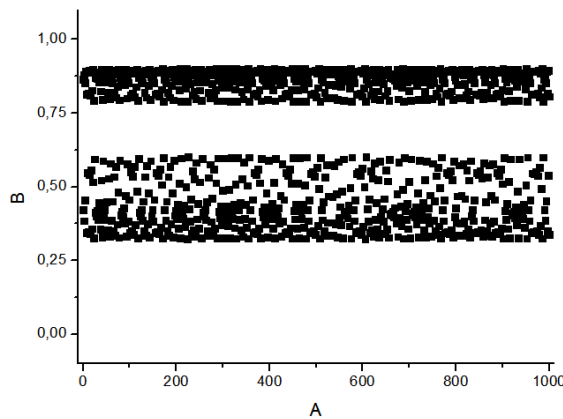
0	0.96	500	0.854761
1	0.1536	501	0.496578
2	0.520028	502	0.999953
3	0.998395	503	0.000187301
4	0.00640774	504	0.000749064
5	0.0254667	505	0.00299401
...		...	
100	0.495893	990	0.340463
101	0.999933	991	0.898191
102	0.000269858	992	0.365774
103	0.00107914	993	0.927934
104	0.0043119	994	0.26749
105	0.0171732	995	0.783757
...			

Table 3.66 represents output of program. From the first view, several sequences of unknown type could be treated as random or pseudo-random. Otherwise, serious analysis must be done. Fig. 3.25 represents random number distributions, generated by RND-1 with different r : a) $r = 0.36$; b) $r = 0.40$.

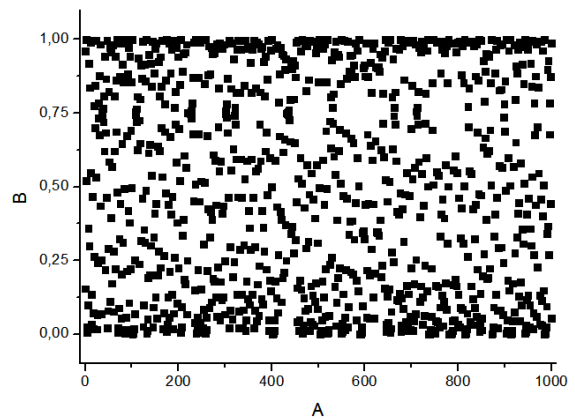
Probable numbers are presented as a variables from loop, continuously increased in interval $(1 \div 100)$. For case a), the big hole in the functional space is present. It means that some part is quite predictable. For case b), a quasi-uniform distribution is present. It means, that RND-1 algorithm using parameter $r = 0.40$ can be used for programming the serious imitation of random processes.

RNG-2. Next type of the linear congruential generator, which uses the recurrence to generate numbers $z_{i+1} = F * (z_i)$, is less popular than the previous one.

$$z_{i+1} = (P_1 * z_i + P_2) \pmod{N} \quad (3.7)$$



a)



b)

Fig. 3.25. Random number distributions, a) $r = 0.36$; b) $r = 0.40$. generated with different r .

Modular arithmetic can be handled mathematically by introducing a congruence relation on the integers. For a positive integer n , two integers a and b are said to be congruent modulo n , if their difference $(a - b)$ is an integer multiple of n . The number n is called the modulus of the congruence. Congruent condition is written in following form.

$$a \equiv b \pmod{n} \quad (3.8)$$

For example,

$$26 \equiv 5 \pmod{7} \quad (3.9)$$

because $26-5=21$, which is a multiple of 7.

Table 3.67 represents programming implementation of random number generator, RNG-2a. Main difference to previous program is presence of modulus operation `% N`. Table 3.68 represents output of program. Table 3.69 represents programming implementation of random number generator, RNG-2b.

Both programs contains the same loop, expressed using `for()` statement:

```
for (int i=0; i<=1000; i++)
{
    zn = (P1 * z0 + P2) % N; //mod
    fout1 << i << " " << z0 << " " << zn << endl;
    z0 = zn;
};
```

Member of sequence is calculated using formula expression containing `mod` operation. Cycle variable `i` is related to the function argument `z0`. Calculated function value `zn` will be outputted. After that, calculated function value `zn` will be used as a function argument for next cycle `z0 = zn`.

Differencies between RNG-2a and RNG-2b are in the functional parameters only. For program,

```
int z0 = 79, P1 = 263, P2 = 71, N = 100; // RNG-2a
int z0 = 79, P1 = 17367, P2 = 4, N = 216593; // RNG-2b
```

First of all, it is necessary to confirm the uniform covering of requested space by presented numbers. Secondly, process of space covering must be dynamically unpredictable. It means that the testing of uniform covering must be done only after finishing the generation processes but not by-the-way of running RNG.

Several routine determinations must be done. Number pairs (x, y) for 2D visualization must be created. According to the recurrent RNG idea, member of sequence $z_{i+1} = F * (z_i)$. Let's attribute z_i to x and z_{i+1} to y . After one cycle of RNG, $z_{i+2} = F * (z_{i+1})$. Let's attribute z_{i+1} to x and z_{i+2} to y . But the principle of recurrent generation means, that x from previous cycle plays the role of y in current cycle and so on. Therefore, by using this routine, number pairs (x, y) for 2D visualization will be done in each loop.

Fig. 3.26 represents random number distributions, generated by RGN-2 with different parameters, as it was previously described. Fig. 3.26a shows repeatable distribution when it is easy to guess the position of the next generated variable. Some cycle dependencies occurs according to the inner relationships of presented parameters. Fig. 3.26b shows chaotic unpredictable distribution, which is impossible for guess operation. Cycle dependencies are absent.

Table 3.67. RNG-2a.	Program pro02.cpp.
<pre> #include <cstdlib> #include <iostream> #include <fstream> using namespace std; int main(int argc, char *argv[]) { int z0 = 79, P1 = 263, P2 = 71, N = 100; int zn; ofstream fout1("rez.txt", ios::out ios::binary); for (int i=0; i<=1000; i++) { zn = (P1 * z0 + P2) % N; // mod fout1 << i << " " << z0 << " " << zn << endl; z0 = zn; }; fout1.close(); system("PAUSE"); return EXIT_SUCCESS; } </pre>	

Table 3.68.	Output of program pro02.cpp.
<pre> 0 79 48 1 48 95 2 95 56 3 56 99 4 99 8 5 8 75 ... </pre>	<pre> 100 79 48 101 48 95 102 95 56 103 56 99 104 99 8 105 8 75 ... </pre>

Table 3.69. RNG-2b

Program pro03.cpp.

```

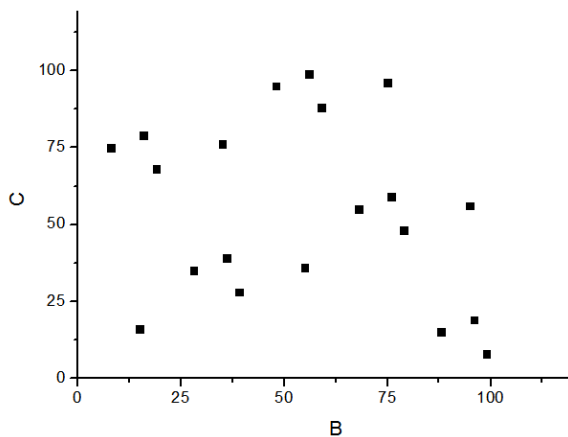
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
    int z0 = 79, P1 = 17367, P2 = 4, N = 216593;
    int zn;
    ofstream fout1("rez.txt", ios::out | ios::binary);
    for (int i=0; i<=1000; i++)
    {
        zn = (P1 * z0 + P2) % N; //mod
        fout1 << i << " " << z0 << " " << zn << endl;
        z0 = zn;
    };

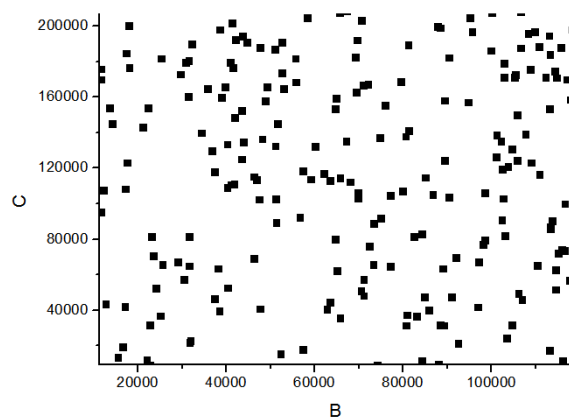
    fout1.close();

    system("PAUSE");
    return EXIT_SUCCESS;
}

```



a)



b)

Fig. 3.26. Random number distributions, generated by RNG-2:

a) $z_0=79$, $P_1=263$, $P_2=71$, $N=100$;b) $z_0=79$, $P_1=17367$, $P_2=4$, $N=216593$.

3.5.3 Encryption

In cryptography, *encryption* is the mathematical process when the easy readable and understandable information (referred to as a *plaintext*) using an algorithm (called a *cipher*) is transformed into unreadable/no-understandable form (referred to as a *ciphertext*) for any person. Only an exclusive person (legal or hacker...) possessing special knowledge (called a *key*) could perform the reverse process: to transfer the encrypted information into readable form again.

Cryptography has been known from old times as a *state-of-art* technique how to hide the important information. Today, many old algorithms like Caesar code, Vigenere code, etc, are interesting looking from historical perspectives only but do not contain any practical importance. Only Vernam[§] algorithm keeps this own significance until present time. Several aspects of Vernam algorithm are presented in Liberty publication [24].

Vernam cipher is a symmetrical stream cipher in which the item of plaintext is combined with the item of random data of the same length, in order to generate the ciphertext. Generation must be done according to the rule of Boolean "Exclusive OR" (XOR) function (symbolised by $\oplus$) presented in Table 3.70.

$$C = A \oplus B \quad (3.10)$$

The statement $A \oplus B$ is true when either A or B, but not both, are true.

The cipher is reciprocal when the identical keystream is used both to encipher the plaintext to ciphertext and to decipher the ciphertext into original plaintext:

$$Ciphertext = Plaintext \oplus Key \quad (3.11)$$

$$Plaintext = Ciphertext \oplus Key \quad (3.12)$$

Tables 3.71a, 3.71b, 3.71c represent encryption/decryption operations by means of *Vernam cipher*. Function `main()` contains following statements. Object `unco` of class type `MULTI` is organized in traditional OOP manner. Class methods are devoted to limited actions. Method `red()` is devoted to check the length of plaintext. Method `yellow()` is devoted to organize the keystream. Method `green()` is devoted to translate: the plaintext to ciphertext and also the ciphertext to plaintext again (reversible function).

```
MULTI unco;
    unco.red    (fin0);          // check length of plaintext
    unco.yellow(fout2);         // organize the keystream
    unco.green (fin1,fout3);     // plaintext to ciphertext
    unco.green (fin2,fout4);     // ciphertext to plaintext
```

Table 3.70. Truth table of "Exclusive OR" (XOR) function $C=A \oplus B$.

A	B	C	A	B	C
false	false	false	0	0	0
true	false	true	1	0	1
false	true	true	0	1	1
true	true	false	1	1	0

[§]Gilbert Sandford Vernam (1890 - 1960) - AT&T Bell Labs engineer. In 1917, he invented an additive polyalphabetic stream cipher and later co-invented an automated one-time pad cipher.

Table 3.71a. Vernam cipher.	Program cod01.cpp.
	<pre> #include <iostream> #include <fstream> #include <iomanip> #include <cmath> using namespace std; class MULTI { public: int iSize; // size of file, B char *TT; // keystream void red (ifstream &infile); void yellow(ofstream &fout2); void vulcan(char &aa, char &bb, char &cc); void green (ifstream &fin1, // input ofstream &fout3); // output }; int main() { MULTI unco; // plaintext size ifstream fin0 ("r1.txt", ios::in ios::binary); // plaintext unco.red(fin0); // check length of plaintext fin0.close(); // keystream ofstream fout2("r2.txt", ios::out ios::binary); unco.yellow(fout2); // organize the keystream fout2.close(); // encryption ifstream fin1 ("r1.txt", ios::in ios::binary); // plaintext ofstream fout3 ("r3.txt", ios::out ios::binary); // encoded unco.green(fin1,fout3); // plaintext to ciphertext fin1.close(); fout3.close(); // decryption ifstream fin2 ("r3.txt", ios::in ios::binary); // encoded ofstream fout4 ("r4.txt", ios::out ios::binary); // plaintext unco.green(fin2,fout4); // ciphertext to plaintext fin2.close(); fout4.close(); system("PAUSE"); return EXIT_SUCCESS; } </pre>

Interface of class MULTI contains public part for description the variables and methods. Variable `iSize` of integer type devoted for size of file and pointer to the `TT` object of character type devoted for keystream. Four methods `red()`, `yellow()`, `vulcan()`, `green()` are of public type also.

```
class MULTI
{
public:
    int iSize; // size of file, B
    char *TT; // keystream
    void red (ifstream &infile);
    void yellow(ofstream &fout2);
    void vulcan(char &aa, char &bb, char &cc);
    void green (ifstream &fin1, // input
                ofstream &fout3);// output
};
```

Table 3.71b. Vernam cipher.

Program cod01.cpp.

```
void MULTI::red (ifstream &infile)
{
    char aa; iSize=0; // file size, B
    if (!infile)
    {
        cout << " no input file "
              << infile << "... EXIT.\n";
        exit(0);
    }
    else
    {
        infile.seekg(iSize,ios::beg);
        while(!infile.eof())
        {
            infile.get(aa); iSize++;
            cout << aa;
            infile.seekg(iSize,ios::beg);
        };
        cout << "Size of input file = "
              << iSize << "B\n\n";
    };
    cout << endl;
}
```

Table 3.71c. Vernam cipher.

Program cod01.cpp.

```

void MULTI::yellow(ofstream &fout2)
{
    TT = new char [iSize];
    for (int i=0; i<iSize; i++)
    {
        TT[i]=(char)(fabs(rand()));
        fout2 << setw(5) << (int)TT[i];
    };
}

void MULTI::vulcan(char &aa, char &bb, char &cc)
// XOR function, bit
{
    cc = aa ^ bb;
}

void MULTI::green (ifstream &fin1, // input
                  ofstream &fout3) // output
{
    char aa,bb,cc;
    for (int i=1; i<iSize; i++)
    {
        fin1.get(aa); fin1.seekg(i,ios::beg);
        bb=TT[i];
        vulcan(aa,bb,cc);
        fout3.put(cc);
        cout << cc;
    };
    cout << "\n\n";
}

```

Method `void MULTI::red()` is devoted to establish the lenght of plaintext file. Method requires one input parameter as pointer to file name `ifstream &infile`. According to `if()` condition in case if file is absent, the statement `exit(0)` will be proceeded (program will be terminated).

```

if (!infile)
{
    cout << " no input file " << infile << "... EXIT.\n";
    exit(0);
};

```

According to `if()` condition in case if file is present, the file marker is set to the beginning (initially `iSize=0`):

```
infile.seekg(iSize,ios::beg);
```

Until condition of file end `infile.eof()` is not satisfied, loop `while()` will be proceeded. Loop contain an operator `get()` for reading single character, and after each reading operation the parameter `iSize` will be increased (incremental operation). File marker is set to the current place corresponding to the loop variable (`iSize=1,2,3,...`). When condition of file end `infile.eof() == true` is satisfied, variable `iSize` is equal to the number of Bytes in file (file length).

```
while(!infile.eof())
{
    infile.get(aa); iSize++;
    cout << aa;
    infile.seekg(iSize,ios::beg);
};
cout << "Size of input file = " << iSize << "B\n\n";
```

Method `void MULTI::yellow()` is devoted to organize the of keystream as a sequence of pseudo-random numbers. This method requires one input parameter as a pointer to the file name `ofstream &fout2`. This method must be run after method `void MULTI::red()` only, when the length of plaintext file is established. Statement `new` created dynamic 1D array of `iSize` length of character type. Loop `for()` is devoted for array initialization by random numbers using function `rand()`. Such random numbers are converted into characters due to the task requirements.

```
TT = new char [iSize];
for (int i=0; i<iSize; i++)
{
    TT[i]=(char)(fabs(rand()));
    fout2 << setw(5) << (int)TT[i];
};
```

Method `void MULTI::green()` is devoted to encryption as well as decryption operations. This method requires two input parameters - two pointers to input and output files `ifstream &fin1`, `ofstream &fout3`. This method contains 1D loop `for()` where all coding or decoding operations are performed. Until the loop variable exceed file length value, file manipulating operations will be proceeded: i) reading of current byte of plaintext from input file; ii) accessing of current byte of keystream from 1D dynamic array; iii) generating of output value by `vulcan()` function of XOR type; iv) whitening the current byte of ciphertext to output file.

```
for (int i=1; i<iSize; i++)
{
    fin1.get(aa);      // aa  plaintext  Byte
    fin1.seekg(i,ios::beg);
    bb=TT[i];          // bb  keystream  Byte
    vulcan(aa,bb,cc);  // cc  ciphertext Byte
    fout3.put(cc);
};
```

Table 3.72.	Output of program cod01.cpp.
	<pre> Honeyed seal of soft affections, Tenderest pledge of future bliss, Dearest tie of young connections, Love's first snowdrop, virgin kiss. Robert Burns Size of input file = 158B k??r,?Z,_B;nho) ?G&?mf))HIz9???e]fyxy?^I #?Mrrs;W\$gl;?j?c ??E UGV3?t?UiJ??< K?}?D? Honeyed seal of soft affections, Tenderest pledge of future bliss, Dearest tie of young connections, Love's first snowdrop, virgin kiss. Robert Burns Press any key to continue ... </pre>

Method `void MULTI::vulcan()` is devoted to realization of XOR function. Method requires three input parameters as pointers to the variables: `char &aa`, `char &bb`, `char &cc`. Actually, variable `cc` is an output variable when `aa` and `bb` are the input variables according to functionality. Function contains only one statement - *mod* function (`^`) for XOR function realization.

```

// XOR function, bit
cc = aa ^ bb;

```

Table 3.72 represents output of program. Strophe of poem as plaintext is encoded using random numbers through the XOR operation and decoded again into the same form.

Nota bene. Presented code containing implemented Vernam algorithm is intended for educational purposes only. Do not use for communication containing secure/important information over an insecure network. Keystream organisation using the `rand()` function is completely unsuitable for real protection of information, however, it is allowed for educational purposes only. In order to realize unbreakable code, you should organize:

- i) keystream generation using RNG and following verification of random number distribution in order to satisfy the several stochastic distribution rules;
- ii) exchanging of verified keystream sequences using safe channel, much safer (and also much expensive) than usually used.

Chapter 4

Graphical programming using *windows.h*

When the code and
the comments disagree,
both are probably wrong.

Norm Schryer

This chapter is devoted to beginners who decided to program the graphical applications not in console regime but in screen window. *Microsoft Corporation* introduced an OS named *Windows* in 1985 as an add-on to *MS-DOS*. Main reason of this new system was the market interest in *Graphical User Interfaces* (GUIs).

GUI programming of so called *Window* type was initiated in OS *Windows-95*, but another OS versions *Windows-98/NT/XP/Vista/7* also support C++ programming based on usage of the library `<windows.h>`. All presented below programs are tested in *Windows7* only, without any guarantee to run it in FAT-based *Windows-95/98/Me* as well as NT-based *Windows-NT/2K/XP/Vista*.

However, the significant exceptions or contradictions between graphical programming in different OS are absent and only negligible differences could occur only. For practical purposes, determination of OS *Window* in this chapter means all OS family, where GUI interface is realized.

OS *Windows* presents a GUI who is responsible to recognize the working video-adaptor / hardware graphical interface. It is not necessary for programmer to go into the question how to operate the graphical hardware, because abstract GUI interface creates the conditions and environment for programming at very low level (pixeling level).

Programming of this type must be started from very low level. In some cases, it gives a delight in possibility to control and to manage many small pieces of interface. In otherwords, it requires high qualification of programmer and very conservative view into programming. It also prevents the mobility, fast changes in code etc. For detail description, see excellent publications of LaMothe [22], [23], Williams [52], and Schildt [41].

This chapter is devoted to describe the main possibilities of window graphical interface, which is essential for manipulating by functions of GDI interface in particularly and for understanding of graphic programming altogether:

- i) formation of graphical interface while using requested parameters;
- ii) possibility to create and manipulate the program using menu buttons and mouse manipulator buttons;
- iii) possibility to make the marker positioning at requested screen place;
- iv) possibility to create graphical primitives on screen: pixel, line, rectangle, ellipse;
- v) possibility to clear requested place of screen.

4.1 Graphical interface

Graphical interface is a programming behaviour made for managing, developing the 2D/3D information on the screen of computer. Graphical interface is treated as one of the most important possibility of information exchange between human and computer. Physiologically, any human receives 80% of information through the eyes.

This section is devoted to problems of window interface creation: main window and menu bar. Mouse manipulator operations are also described.

4.1.1 Window formation

1st programming task. Programming task is presented as follows: to create window with the possibility to switch, to destroy this window and nothing else.

Tables 4.1a, 4.1b, 4.1c represent the code for formation of windows like geometrical figure on the monitor screen. Tables 4.1a, 4.1b are devoted for `WinMain()` function. General structure of programming content is presented below.

```
#include <windows.h>

int WINAPI WinMain (HINSTANCE, HINSTANCE, LPSTR, int)
{
    ...
    // Window structure
    wincl.lpfnWndProc = WindowProc1234; // function called by windows
}
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM)
{
    // how to operate?
}
```

First of all, library `<windows.h>` with implemented GUI interface is necessary. Secondly, function `int WINAPI WinMain()` is necessary. This function organizes creation of window in dynamical (not static) regime using multithreading technologies. Messages from processes (dispatching technique) allow the redrawing of window. After each cycle of windows redrawing, function `int WINAPI WinMain()` returns the management to OS.

Table 4.1a.

Program win01.cpp.

```

#include <windows.h>

// part of global objects
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "WindowsApp"; // class name

int WINAPI WinMain (HINSTANCE hThisInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR      lpszArgument,
                   int         nFunsterStil)
{
    HWND      hwnd;      // handle for window
    MSG        messages; // messages to the application are saved
    WNDCLASSEX wincl;     // data structure for the windowclass

    // Window structure
    wincl.hInstance      = hThisInstance;
    wincl.lpszClassName  = szClassName;
    wincl.lpfnWndProc     = WindowProc1234; // function called by windows
    wincl.style           = CS_DBLCLKS;     // Catch double-clicks
    wincl.cbSize          = sizeof (WNDCLASSEX);

    // Use default icon and mouse-pointer
    wincl.hIcon           = LoadIcon      (NULL, IDI_APPLICATION);
    wincl.hIconSm         = LoadIcon      (NULL, IDI_APPLICATION);
    wincl.hCursor         = LoadCursor    (NULL, IDC_ARROW);
    wincl.lpszMenuName    = NULL; // No menu
    wincl.cbClsExtra      = 0;          // No extra bytes after the window class
    wincl.cbWndExtra      = 0;          // structure or the window instance
    // Windows's default color as the background of the window
    wincl.hbrBackground   = (HBRUSH) COLOR_BACKGROUND;

    // Register the window class, and if it fails quit the program
    if (!RegisterClassEx (&wincl))
        return 0;
    // continued to Table 4.1b

```

Function `int WINAPI WinMain()` contains parameters for window structure description, and partially, parameter for associated so called *call-back* function, which is called from OS according to dispatch activity. User-created function `LRESULT CALLBACK WindowProc1234 ()` contains the main programming information "how to operate".

After describing the library `<windows.h>`, part of global objects is present. It is necessary to declare the function `LRESULT CALLBACK WindowProc1234()` (user-created function!) and also class name: `szClassName[ ]` (string in C style), which will be used in all associated files.

Table 4.1b.

Program win01.cpp.

```

// continued from Table 4.1a

// The class is registered, let's create the program
hwnd = CreateWindowEx (
    0,                                // Extended possibilites for variation
    szClassName,                     // Classname
    "Window only...",                // Title Text
    WS_OVERLAPPEDWINDOW,             // default window
    CW_USEDEFAULT,                    // Windows decides the position
    CW_USEDEFAULT,                    // where the window ends up on the screen
    800,                              // The programs width  in pixels
    600,                              // The programs height in pixels
    HWND_DESKTOP,                    // The window is a child-window to desktop
    NULL,                             // No menu
    hThisInstance,                   // Program Instance handler
    NULL                              // No Window Creation data
);

// Make the window visible on the screen
ShowWindow (hwnd, nFunsterStil);

// Run the message loop.
// It will run until GetMessage() returns 0
while (GetMessage (&messages, NULL, 0, 0))
{
    // Translate virtual-key messages into character messages
    TranslateMessage(&messages);
    // Send message to Window Procedure WindowProc1234
    DispatchMessage(&messages);
}

// The program return-value is 0 -
// The value that PostQuitMessage() gave
return messages.wParam;
}

```

```

// part of global objects
LRESULT CALLBACK WindowProc1234 (HWND, UINT, WPARAM, LPARAM);
char szClassName[ ] = "WindowsApp"; // class name

```

WinMain. Function `int WINAPI WinMain()` contains four input parameters of different types: `HINSTANCE`, `LPSTR` and `int`. Such parameters are necessary for windows manipulating operations. Types `HINSTANCE`, `LPSTR` for `WinMain()` function and `HWND`, `UINT`, `WPARAM`, `LPARAM` for callback function are described in library `<windows.h>`.

```
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR      lpszArgument,  int      nFunsterStil)
```

Function `int WINAPI WinMain()` must solve the following tasks:

- i) determination of window class;
- ii) registration of window class;
- iii) creation of window of required class;
- iv) making the window visible on the screen;
- v) starting the loop of dispatching the messages.

Let's start from determination of window class. The first block of function `int WINAPI WinMain()` contains description of handle for window, messages to the application to be saved and data structure for the window class.

```
HWND      hwnd;      // handle for window
MSG        messages; // messages to the application are saved
WNDCLASSEX wincl;    // data structure for the windowclass
```

The second block of function `int WINAPI WinMain()` contains descriptor (DC) of this application, window class name, name of user-created function called by OS *Windows*, and several window style parameters.

```
// Window structure
wincl.hInstance      = hThisInstance; // DC of this app
wincl.lpszClassName  = szClassName;   // window class name
wincl.lpfnWndProc     = WindowProc1234; // function called by windows
wincl.style           = CS_DBLCLKS;    // Catch double-clicks
wincl.cbSize          = sizeof (WNDCLASSEX);
```

The third block of function `int WINAPI WinMain()` contains parameters of icons and mouse-pointers as standard, default icon and standard pointer for cursor.

```
// Use default icon and mouse-pointer
wincl.hIcon    = LoadIcon (NULL, IDI_APPLICATION);
wincl.hIconSm  = LoadIcon (NULL, IDI_APPLICATION);
wincl.hCursor  = LoadCursor (NULL, IDC_ARROW);
```

The fourth block of function `int WINAPI WinMain()` contains parameters depending menu organization (in that case no menu) and several routine parameters for window organization: no extra bytes after the window class and structure or the window instance. Background of the window must be coloured by selected, user-defined or default colour, and in this case window's default colour could be determined.

```
wincl.lpszMenuName = NULL;      // No menu
wincl.cbClsExtra   = 0;         // No extra bytes after the window class
wincl.cbWndExtra   = 0;         // structure or the window instance
// Windows's default colour as the background of the window
wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;
```

Determination of window class is finished, and let's start to register the window class. Registration of window class must be succesful, and if it fails quitting off the program is compulsory. This condition is realized through `if` statement.

```
// Register the window class, and if it fails quit the program
if (!RegisterClassEx (&wincl)) return 0;
```

In case of successful registration of window class, creation of program (window of required class) is allowed. Function (HWND)CreateWindowEx() is used for such purposes. Function contains 12 input parameters.

The first parameter (0) is presented for advanced users, default value 0 is enough for all routine works. Extended possibilities for variation could be used.

The second parameter (szClassName) requires window class name as a string from global area (made in C style).

The third parameter ("Window only...") represents name of default window, which name user can change according to circumstances.

The fourth parameter (WS_OVERLAPPEDWINDOW) represents style of default window: it will be presented in overlapping style to the other windows.

The fifth and the sixth parameters (CW_USEDEFAULT) represents the position of starting window. In that case, OS *Windows* decides on the initial position (x_0, y_0) of window as geometrical figure on the screen. Fig. 4.1 represents distribution of window coordinates on screen. Initial point (0,0) is presented at top left.

The seventh and the eighth parameters (800,600) represents the width and height (x and y , respectively) of windows as geometrical figure in pixels. Thus it is also possible to use macros CW_USEDEFAULT as in previous positions in order to receive the default window values.

The ninth parameter (HWND_DESKTOP) is devoted to window style determination: the window is a child-window to desktop.

The tenth parameter (NULL) shows the information that menu will be not organized (no menu).

The eleventh parameter (hThisInstance) shows, which program *Instance handler* will be used (handler described in WINAPI WinMain() function as one of four input parameters).

The twelfth parameter (NULL) says that no window creation data will be used (this parameter is useful for advanced users).

Next function ShowWindow () is devoted to window drawing: it makes the window visible on the screen. Two input parameters - descriptor hwnd from function (HWND)CreateWindowEx() and descriptor nFunsterStil as one of four input parameters of WINAPI WinMain() function must be initialized.

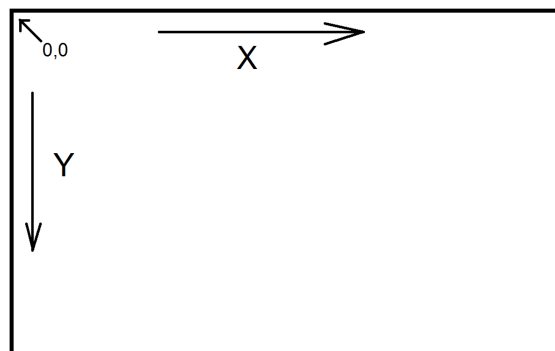


Fig. 4.1. Coordinates of window as a geometrical figure. Initial point (0,0) at top left.

```

// The class is registered, let's create the program
hwnd = CreateWindowEx (
    0,                                // Extended possibilities for variation
    szClassName,                      // Classname
    "Window only...",                // Title Text
    WS_OVERLAPPEDWINDOW,              // default window
    CW_USEDEFAULT,                    // Windows decides the position
    CW_USEDEFAULT,                    // where the window ends up on the screen
    800,                              // The programs width in pixels
    600,                              // The programs height in pixels
    HWND_DESKTOP,                    // The window is a child-window to desktop
    NULL,                             // No menu
    hThisInstance,                   // Program Instance handler
    NULL );                           // No Window Creation data
ShowWindow (hwnd, nFunsterStil); // Make the window visible on the screen

```

Finally, starting the loop of dispatching the messages must be done. Loop `while()` contains two functions.

The first function `TranslateMessage(&messages)` is devoted to translate the virtual-key messages into character messages.

The second function `DispatchMessage(&messages)` is devoted to send messages to user-created window procedure `WindowProc1234()` (in that case).

```

// Run the message loop.
// It will run until GetMessage() returns 0
while (GetMessage (&messages, NULL, 0, 0))
{
    // Translate virtual-key messages into character messages
    TranslateMessage(&messages);
    // Send message to Window Procedure WindowProc1234
    DispatchMessage(&messages);
}

```

Program finishes, when function `PostQuitMessage()` gives return message.

```

// The program return-value is 0 -
// The value that PostQuitMessage() gives
return messages.wParam;

```

CALLBACK. Table 4.1c represents user-created callback function called `WindowProc1234()`. Due to programming task (to create window with the possibility to switch, to destroy this window and nothing else) callback function `WindowProc1234()` contains one `switch()` in loop. According to handle of messages `message` two different switch cases are presented.

The first case as a regular requesting for window destroying could be organized by user (through window managing button). In that case, handle `message` containing `WM_DESTROY` occurs. Function `PostQuitMessage (0)` realizes sending a `WM_QUIT` to the message queue.

The second case as a default case is realized through function `DefWindowProc()`.

Table 4.1c. Program win01.cpp.

```

// This function is called by the Windows function DispatchMessage()
LRESULT CALLBACK WindowProc1234 (HWND hwnd,
                                UINT message,
                                WPARAM wParam,
                                LPARAM lParam)
{
    switch (message)                // handle the messages
    {
        case WM_DESTROY:
            PostQuitMessage (0);   // send a WM_QUIT to the message queue
            break;
        default:                   // for messages that we don't deal with
            return DefWindowProc (hwnd, message, wParam, lParam);
    };
    return 0;
}

```

```

switch (message)                // handle the messages
{
    case WM_DESTROY:
        PostQuitMessage (0); // send a WM_QUIT to the message queue
        break;
    default:                   // for messages that we don't deal with
        return DefWindowProc (hwnd, message, wParam, lParam);
};
return 0;

```

Fig. 4.2 represents output of program win01.cpp. Window is presented as a rectangle with background filled with gray colour.

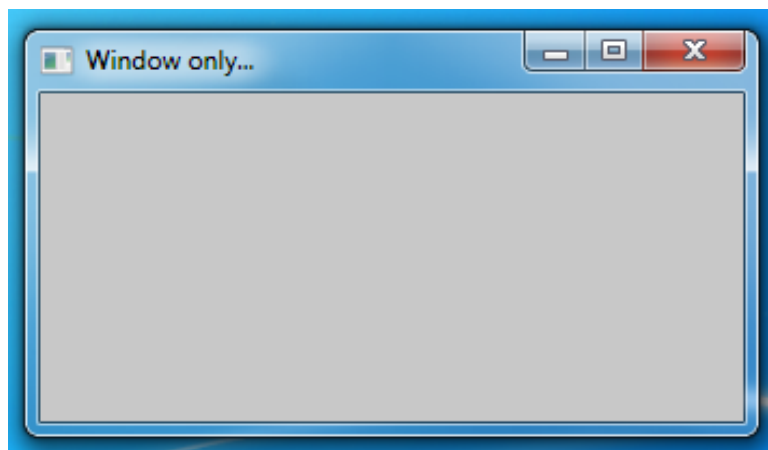


Fig. 4.2. Output of win01.cpp. Window only.

Right top angle contains three system buttons for window management: *Minimize*, *Maximize*, *Close*. Left top angle contains three system buttons built up in *Windows-95* style - *Restore*, *Move*, *Size*, *Minimize*, *Maximize*, *Close*. Such possibilities are not presented in Fig. 4.2. Window etiquette containing user-suggested phrase "Window only..." is presented on the windows top. Finishing of program and closing of window occurs by pushing red button *Close* from right top angle.

4.1.2 Dialog and manipulator operations

Dialog represents a behaviour of programming tools for exchanging management information between human and machine (computer). One of the best choices of dialog is the organization of menu buttons. Mouse plays a role of manipulator in modern computers.

2nd programming task. Programming task represents window creation, user-requested menu bar creation and realizations of mouse manipulator operations.

Tables 4.2a, 4.2b represent `int WINAPI WinMain()` function, Tables 4.2c, 4.2d represent user-created call-back function `LRESULT CALLBACK WindowProc2345()`. Table 4.3 shows corresponding header file and Table 4.4 represents resource script file.

Function `int WINAPI WinMain()` devoted for presented programming task is very similar to previous one which is presented in Tables 4.1a, 4.1b. Main code sentences and comments of code correspond to each other. Significant differences are observed below.

Additional libraries are present. In addition to `<windows.h>`, libraries for input operations `<string.h>` and `<stdio.h>` are necessary. Header file "win02.h" must be also included here (presented in Table 4.3).

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win02.h"
```

WinMain. That part of windows class parameters contains:

- i) determination of user-created call-back function `WindowProc2345()`;
- ii) determination of main menu name `MYMENU_002`.

Menu is described in resource script file as presented in Table 4.4.

```
// Windows class determination
wcl.lpfnWndProc = WindowProc2345; // function of window
wcl.lpszMenuName = "MYMENU_002"; // main menu
```

User-delimited colour of window background could be established by function `(HBRUSH)GetStockObject()`. According to user suggestions, input parameters `WHITE_BRUSH` or `BLACK_BRUSH` - prelimit white and black colour of background respectively.

```
// to fill window by white colour
wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
//wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);
```

Table 4.2a.

Program win02.cpp.

```

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win02.h"

// global objects
LRESULT CALLBACK WindowProc2345(HWND,UINT,WPARAM,LPARAM);
char szWinName[] = "MyWindow";

int WINAPI WinMain (HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR     lpzArgs,  int         nWinMode)
{
    HWND hwnd;
    MSG msg;
    WNDCLASS wcl;
    HACCEL hAccel;

    // Windows class determination
    wcl.hInstance      = hThisInst;      // descriptor of application
    wcl.lpszClassName = szWinName;      // name of windows class
    wcl.lpfnWndProc    = WindowProc2345; // function of window
    wcl.style          = 0;              // default style

    wcl.hIcon    = LoadIcon (NULL, IDI_APPLICATION); // standard icon
    wcl.hCursor  = LoadCursor (NULL, IDC_ARROW);     // standard cursor

    wcl.lpszMenuName="MYMENU_002"; // main menu
    //wcl.lpszMenuName=NULL;        // no menu
    wcl.cbClsExtra= 0; // no additional
    wcl.cbWndExtra= 0; // information

    // to fill window by white color
    wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
    //wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);

    // to register the window class
    if ( !RegisterClass (&wcl)) return 0;

    // continued to Table 4.2b

```

In case of successful registration of window class, the creation of program (window of required class) is allowed. Function (HWND)CreateWindow() is used for such purpose. Function contains eleven input parameters.

The first parameter (szWinName) requires class name, declared before WinMain() function, in global part.

Table 4.2b.

Program win02.cpp.

```

// continued from Table 4.2a

// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "MENU AND MANIPULATOR", // window header
    WS_OVERLAPPEDWINDOW, // window style
    CW_USEDEFAULT,       // X-coordinate, from Windows
    CW_USEDEFAULT,       // Y-coordinate, from Windows
    CW_USEDEFAULT,       // width, from Windows
    CW_USEDEFAULT,       // high, from Windows
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,        // use class menu
    hThisInst,           // descriptor
    NULL);               // no additional arguments

// to load accelerator
hAccel = LoadAccelerators (hThisInst,"MYMENU_002");

//to show window and redraw content
ShowWindow (hwnd,nWinMode);
UpdateWindow(hwnd);

// cycle of annonses
while ( GetMessage (&msg, NULL,0,0))
{
    if (!TranslateAccelerator (hwnd,hAccel,&msg))
    {
        TranslateMessage (&msg); // to use keyboard
        DispatchMessage (&msg); // to return direction from Windows
    }
};
return msg.wParam;
}

```

The second parameter ("MENU AND MANIPULATOR") represents string of window header. This string will be presented on window top as an window etiquette.

All the other parameters are commented in the previous subsection, for function `CreateWindowEx()`.

```

// to create window
hwnd = CreateWindow (
    szWinName,           // class name
    "MENU AND MANIPULATOR", // window header
    WS_OVERLAPPEDWINDOW, // window style
    CW_USEDEFAULT,       // X-coordinate, from Windows

```

```

CW_USEDEFAULT,      // Y-coordinate, from Windows
CW_USEDEFAULT,      // width, from Windows
CW_USEDEFAULT,      // high, from Windows
HWND_DESKTOP,       // no additional window
(HMENU) NULL,       // use class menu
hThisInst,          // descriptor
NULL);              // no additional arguments

```

Table 4.2c.

Program win02.cpp.

```

LRESULT CALLBACK WindowProc2345 (HWND  hwnd,
                                UINT   message,
                                WPARAM wParam,
                                LPARAM lParam)
{
    int response;
    switch (message)
    {
    case WM_COMMAND: //
        switch ( LOWORD (wParam))
        {
            case IDM_ALPHA:  MessageBox (hwnd, "Alpha",  "",  MB_OK);
                                break;
            case IDM_BETA:   MessageBox (hwnd, "Beta",   "",  MB_OK);
                                break;
            case IDM_GAMMA:  MessageBox (hwnd, "Gamma",  "",  MB_OK);
                                break;
            case IDM_EPSILON: MessageBox (hwnd, "Epsilon", "",  MB_OK);
                                break;
            case IDM_ZETA:   MessageBox (hwnd, "Zeta",   "",  MB_OK);
                                break;
            case IDM_ETA:    MessageBox (hwnd, "Eta",    "",  MB_OK);
                                break;
            case IDM_THETA:  MessageBox (hwnd, "Theta",  "",  MB_OK);
                                break;
            case IDM_HELP:   MessageBox (hwnd, "No Help", "Help", MB_OK);
                                break;
        };
        break;

        // continued to Table 4.2d

```

Table 4.2d.

Program win02.cpp.

```

// continued from Table 4.2c

case WM_RBUTTONDOWN: // right button of mouse
    response = MessageBox (hwnd, "Select action:",
                           "Right button", MB_ABORTRETRYIGNORE);
    switch (response)
    {
        case IDABORT:  MessageBox (hwnd, "", "Abort",  MB_OK); break;
        case IDRETRY:  MessageBox (hwnd, "", "Retry",  MB_OK); break;
        case IDIGNORE: MessageBox (hwnd, "", "Ignore", MB_OK); break;
    };

case WM_LBUTTONDOWN: // left button of mouse
    response = MessageBox (hwnd, "Continue ?",
                           "Left button", MB_ICONHAND | MB_YESNO);
    switch (response)
    {
        case IDYES:    MessageBox (hwnd, "Press", "YES", MB_OK); break;
        case IDNO:     MessageBox (hwnd, "Press", "NO",  MB_OK); break;
    };

case WM_DESTROY: // to end program
    PostQuitMessage (0); break;
default:
    return DefWindowProc (hwnd, message, wParam, lParam);
}
return 0;
}

```

Additional possibilities of manipulating could be received from keyboard. Function `LoadAccelerators()` is presented for such purpose.

This function requires two input parameters - descriptor of application `hThisInst` and menu name "MYMENU_002". Menu is presented in resource script file, where keyboard buttons are associated with the corresponding identifiers, see Table 4.4.

```

// to load accelerator
hAccel = LoadAccelerators (hThisInst,"MYMENU_002");

```

CALLBACK. Call-back function `LRESULT CALLBACK WindowProc2345()` (see Tables 4.2c, 4.2d) contains the main programming code according to user request. Managing of program could be done by several different ways: by virtual machine command, by pushing the button of mouse manipulator (right as well as left) or using using system window button *Close* for virtual machine destroying. All such cases are presented in `switch()` structure according to the input parameter `message` request.

```

switch (message)
{
    case WM_COMMAND:    {...}
    case WM_RBUTTONDOWN:{...}    // right button of mouse
    case WM_LBUTTONDOWN:{...}    // left button of mouse
    case WM_DESTROY:    {...}    // to end program
    default:
}

```

Let's go to the first case `case WM_COMMAND:` - virtual machine command. According to the user request, this part contains eight menu button positions such as: `IDM_ALPHA`, `IDM_BETA`, ..., `IDM_THETA`, `IDM_HELP`. In that case, selecting of each position is confirmed by creation of message box with corresponding etiquette. All presented identifiers must be described in header file "win02.h" (Table 4.3) which is included into the main program file (Table 4.2a) as well as in resource script file (Table 4.4).

Function `MessageBox()` requires four input parameters: i) handle of windows `hwnd`; ii) middle and top etiquettes to message box - as the strings of C type, for example "No Help" and "Help"; iii) managing button, in that case master button `MB_OK`.

User-requested programming code could be placed between `case` and `break` operators.

```

switch ( LOWORD (wParam))
{
    case IDM_ALPHA:    MessageBox (hwnd, "Alpha",    "",    MB_OK); break;
    case IDM_BETA:    MessageBox (hwnd, "Beta",    "",    MB_OK); break;
    case IDM_GAMMA:    MessageBox (hwnd, "Gamma",    "",    MB_OK); break;
    case IDM_EPSILON:  MessageBox (hwnd, "Epsilon",    "",    MB_OK); break;
    case IDM_ZETA:    MessageBox (hwnd, "Zeta",    "",    MB_OK); break;
    case IDM_ETA:    MessageBox (hwnd, "Eta",    "",    MB_OK); break;
    case IDM_THETA:    MessageBox (hwnd, "Theta",    "",    MB_OK); break;
    case IDM_HELP:    MessageBox (hwnd, "No Help", "Help", MB_OK); break;
    // case IDM_XXX:    /* any user-created code */          break;
};
break;

```

Let's go to the second case `case WM_RBUTTONDOWN` which corresponds to the manipulation by right button of mouse. First of all, it is necessary to select manipulation type. For such purposes, function `MessageBox()` through master button group `MB_ABORTRETRYIGNORE` offers the selection of three positions "Abort", "Retry", "Ignore". After that, selecting of each position by click of mouse button - `IDABORT`, `IDRETRY`, `IDIGNORE` - is confirmed by creation of message box with corresponding etiquette. User-requested programming code could be placed between `case` and `break` operators.

```

int response = MessageBox (hwnd, "Select action:", "Right button",
                           MB_ABORTRETRYIGNORE);

switch (response)
{
    case IDABORT:  MessageBox (hwnd, "", "Abort", MB_OK); break;
    case IDRETRY:  MessageBox (hwnd, "", "Retry", MB_OK); break;
    case IDIGNORE: MessageBox (hwnd, "", "Ignore", MB_OK); break;
};

```

Let's go to the third case `case WM_LBUTTONDOWN` which corresponds to the manipulation by left button of mouse. First of all, it is necessary to select manipulation type. For such purposes, function `MessageBox()` through master button group `MB_ICONHAND | MB_YESNO` offers the selection of position "Left button". After that, selecting of each position by click of mouse button -IDYES, IDNO - is confirmed by creation of message box with corresponding etiquette. User-requested programming code could be placed between `case` and `break` operators.

```
int response = MessageBox (hwnd, "Continue ?", "Left button",
                           MB_ICONHAND | MB_YESNO);

switch (response)
{
    case IDYES:    MessageBox (hwnd, "Press", "YES", MB_OK); break;
    case IDNO:     MessageBox (hwnd, "Press", "NO",  MB_OK); break;
};
```

Let's go to the fourth case `case WM_DESTROY`. Regular requesting for window destroying could be organized by user (by pushing window managing button). In that case, handle message containing `WM_DESTROY` occurs, and function `PostQuitMessage(0)` realizes the sending a `WM_QUIT` to the message queue.

Let's go to the fifth case `case default`. Default case is realized through function `DefWindowProc()`.

HEADER FILE. Table 4.3 represents the header file `win02.h` which must be included into `win02.cpp` and `win02.rc` files. Content of header file represents an association of `case` identifiers in `WM_COMMAND` expression only to the integer numbers through operator `define`. Integer number could be chosen freely, but it must be unique for each identifier.

```
#define IDM_ALPHA    100
```

Table 4.3. Header file.	Program win02.h.
<pre>// win02.h // #define IDM_ALPHA 100 #define IDM_BETA 101 #define IDM_GAMMA 102 #define IDM_DELTA 103 #define IDM_EPSILON 104 #define IDM_ZETA 105 #define IDM_ETA 106 #define IDM_THETA 107 #define IDM_HELP 108</pre>	

RC FILE. Table 4.4 represents resource script (RC) file win02.rc which is responsible for program management organization: menu buttons and accelerators from keyboard. Resource script (RC) file is written using script language, but not C/C++ language. First of all, this file contains two `include` statements pointed to library `<windows.h>` and header file `"win02.h"`.

```
#include "windows.h"
#include "win02.h"
```

The second part of resource script file contains description of menu buttons. Statement containing menu name MYMENU_002 is responsible for menu recognising in WinMain() function.

```
MYMENU_002 MENU DISCARDABLE
```

After that, distribution of menu items is presented. Each menu item must be associated with the identifier, previously described in header file. After that, this menu item could be used in call-back function. For example, menu items ALPHA and BETA could be grouped into pop-up item FIRST.

```
POPUP "&FIRST"
BEGIN
    MENUITEM "&ALPHA\tF2",          IDM_ALPHA
    MENUITEM "&BETA\tF3",          IDM_BETA
END
```

Also, pop-up item SECOND could be organized as collection of menu item GAMMA and second level pop-up item DELTA which contains menu item EPSILON and menu item ZETA

```
POPUP "&SECOND"
BEGIN
    MENUITEM "&GAMMA\tShift+G",      IDM_GAMMA
    POPUP "&DELTA"
    BEGIN
        MENUITEM "&EPSILON\tCtrl+E",  IDM_EPSILON
        MENUITEM "&ZETA\tCtrl+Z",    IDM_ZETA
    END
END
```

The third part of resource script (RC) file contains description of accelerators. Statement containing menu name MYMENU_002 is responsible for accelerators recognising in WinMain() function.

```
MYMENU_002    ACCELERATORS MOVEABLE PURE
```

Keyboard accelerators are presented using definition of three type: virtual key notation VK_F2, ASCII notation "G" or "Ê" (one key or CTRL+E, respectively). Semantics of such type associations is very strong: virtual key could be associated to the identifier, previously described in header file. After that, this accelerator could be used in call-back function.

```
VK_F2,          IDM_ALPHA,          VIRTKEY
"G",            IDM_GAMMA,          ASCII
"^E",           IDM_EPSILON,        ASCII
```

Figs. 4.3÷4.5 represent the output of win02 program. New window is created and menu containing two pop-up items is organized.

Table 4.4. RC file.

Program win02.rc.

```

#include "windows.h"
#include "win02.h"

// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

// Menu
MYMENU_002 MENU DISCARDABLE
BEGIN
    POPUP "&FIRST"
    BEGIN
        MENUITEM "&ALPHA\tF2",          IDM_ALPHA
        MENUITEM "&BETA\tF3",          IDM_BETA
    END
    POPUP "&SECOND"
    BEGIN
        MENUITEM "&GAMMA\tShift+G",      IDM_GAMMA
        POPUP "&DELTA"
        BEGIN
            MENUITEM "&EPSILON\tCtrl+E",  IDM_EPSILON
            MENUITEM "&ZETA\tCtrl+Z",      IDM_ZETA
        END
        MENUITEM "&ETA\tCtrl+F4",        IDM_ETA
        MENUITEM "&THETA\tF5",          IDM_THETA
    END
    MENUITEM "&HELP",                  IDM_HELP
END

// Accelerator
MYMENU_002 ACCELERATORS MOVEABLE PURE
BEGIN
    VK_F2,          IDM_ALPHA,          VIRTKEY
    VK_F3,          IDM_BETA,            VIRTKEY
    "G",            IDM_GAMMA,           ASCII
    "^E",           IDM_EPSILON,         ASCII
    "^Z",           IDM_ZETA,            ASCII
    VK_F4,          IDM_ETA,             VIRTKEY, CONTROL
    VK_F5,          IDM_THETA,           VIRTKEY
    VK_F1,          IDM_HELP,            VIRTKEY
END

#endif // English (U.S.) resources

```

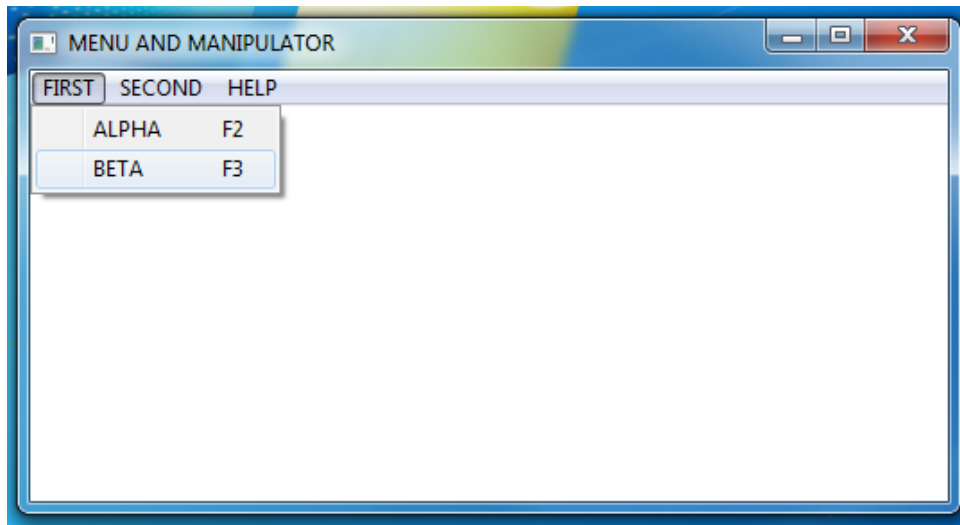


Fig. 4.3. Output of win02. Menu in window.

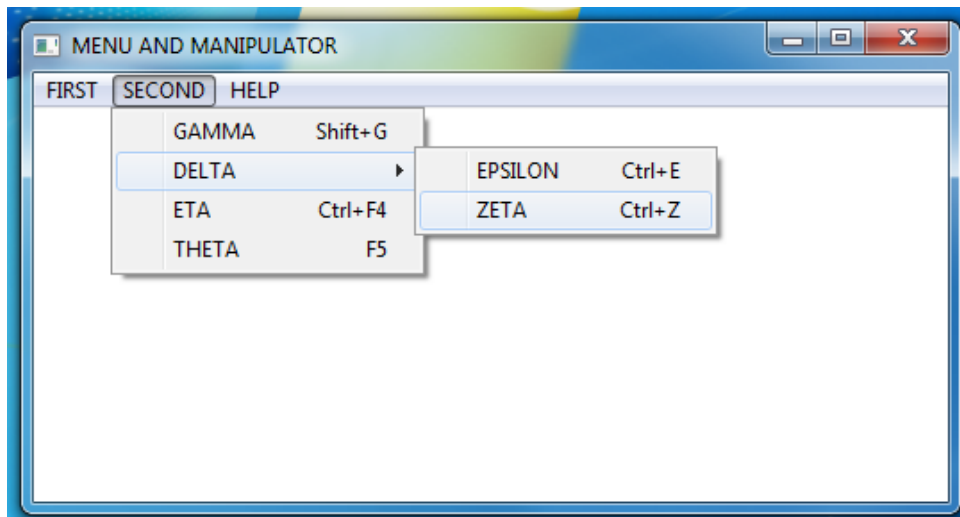


Fig. 4.4. Output of win02. Menu in window.

In Fig. 4.3, pop-up item **FIRST** is activated using mouse manipulator. According to the menu description in resource script file, two menu items **ALPHA** and **BETA** are present. Accelerators are presented on button etiquettes.

In Fig. 4.4, pop-up item **SECOND** is activated using mouse manipulator, and pop-up item of the second level **DELTA** is also activated. According to the menu description in resource script file, two columns of menu items are present. Accelerators are presented on button etiquettes.

In Fig. 4.5, right button of mouse manipulator is pushed, and corresponding message box occurs. This box contains the information for following selection of the next step: *Abort*, *Retry*, *Ignore*.

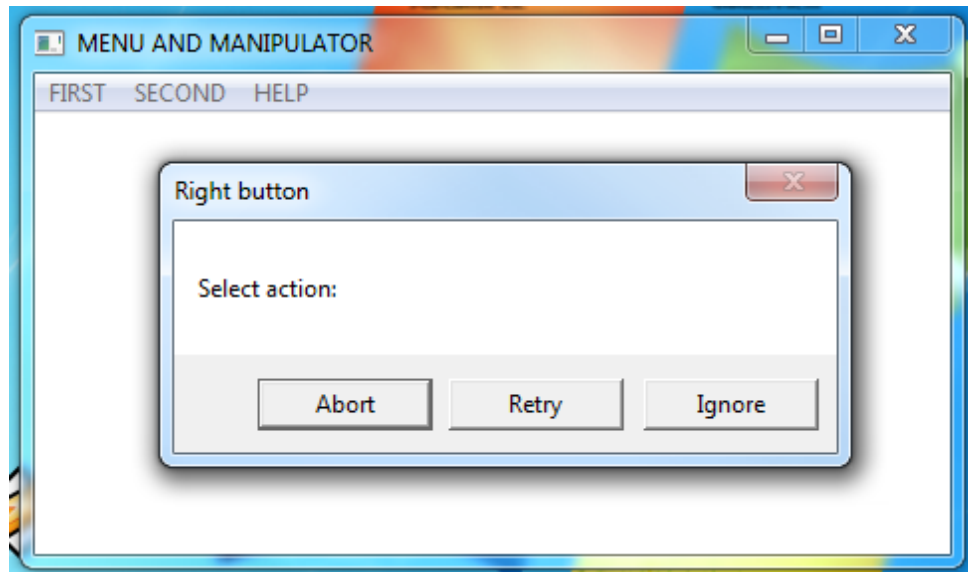


Fig. 4.5. Output of win02. Menu in window.

4.2 Figure drawing

C Code.
 C code run.
 Run code run...
 PLEASE !!!

Barbara Tongue

3rd programming task. Programming task represents window creation, user-requested menu bar creation and realization of several primitive drawings on screen such as a line, a rectangle, an ellipse and a pie. Tables 4.5a, 4.5b represent `int WINAPI WinMain ()` function, Tables 4.5c, 4.5d represent call-back function `LRESULT CALLBACK WindowProc3456()`. Table 4.6 represents corresponding header file, and Table 4.7 shows resource script file. Figs. 4.6÷4.8 show outputs of program.

WinMain. Function `int WINAPI WinMain()` prepared according to the presented programming task is very similar to the previous one, presented in Table 4.1a as well as Table 4.2a. Main code sentences and comments of code corresponds each to other. Negligible differences are observed below.

Additional library functions are necessary. In addition to `<windows.h>`, several libraries for input operations: `<string.h>` and `<stdio.h>` are necessary. Header file "win03.h" must be included here (presented in Table 4.6).

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win03.h"
```

Table 4.5a.

Program win03.cpp.

```

#include <windows.h>
#include <string.h>
#include <stdio.h>
#include "win03.h"

// global objects
LRESULT CALLBACK WindowProc3456(HWND,UINT,WPARAM,LPARAM);
char szWinName[] = "MyWindow"; // window class
int iXmax, iYmax;               // window width and hight

HDC      memdc;                 // DC of virtual window
HBITMAP  hbit;                  // rastr in virtual window
HBRUSH   hbrush, hOldbrush;     // DC of pen
HPEN     hOldpen;               // DC of old pen
HPEN     hRedpen, hGreenpen, hBluepen, // DC of color pen
         hYellowpen, hCyanopen,hMargentapen;

int WINAPI WinMain (HINSTANCE hThisInst, HINSTANCE hPrevInst,
                    LPSTR      lpzArgs,  int      nWinMode)
{
    HWND hwnd;  MSG msg;  WNDCLASS wcl;  HACCEL hAccel;

    // Windows class determination
    wcl.hInstance      = hThisInst;      // DC of application
    wcl.lpszClassName  = szWinName;      // windows class
    wcl.lpfnWndProc    = WindowProc3456; // function of window
    wcl.style          = 0;              // default style

    wcl.hIcon          = LoadIcon      (NULL, IDI_APPLICATION); // standard icon
    wcl.hCursor         = LoadCursor   (NULL, IDC_ARROW);        // standard cursor
    wcl.lpszMenuName    = "MYMENU_003"; // main menu
    //wcl.lpszMenuName = NULL;          // no menu
    wcl.cbClsExtra      = 0; // no additional
    wcl.cbWndExtra      = 0; // information

    // to fill window by color
    //wcl.hbrBackground  = (HBRUSH)GetStockObject (WHITE_BRUSH);
    wcl.hbrBackground = (HBRUSH)GetStockObject (BLACK_BRUSH);

    // to register the window class
    if ( !RegisterClass (&wcl)) return 0;

    // continued to Table 4.5b

```

Table 4.5b.

Program win03.cpp.

```

// continued from Table 4.5a

// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "FIGURES",           // header
    WS_OVERLAPPEDWINDOW, // window style
    0,    // CW_USEDEFAULT, // X-0-coordinate
    0,    // CW_USEDEFAULT, // Y-0-coordinate
    1200, // CW_USEDEFAULT, // width
    800,  // CW_USEDEFAULT, // hight
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,         // use class menu
    hThisInst,            // descriptor
    NULL);                // no additional arguments

// to show window and redraw content
ShowWindow (hwnd, nWinMode);
UpdateWindow (hwnd);

// cycle of annonses
while ( GetMessage (&msg, NULL,0,0))
{
    if (!TranslateAccelerator (hwnd,hAccel,&msg))
    {
        TranslateMessage (&msg); // to use keyboard
        DispatchMessage (&msg); // to return direction from Windows
    }
}
return msg.wParam;
}

```

Block of global objects contains the following parts: declaration and initialization of window class, declaration of integer variables devoted for visualization on screen: window width and height, declaration of descriptor (DC) of virtual window, bitmap in virtual window, DC of pen, DC of old pen, and six DCs of colour pen. All classes named HDC, HBITMAP, HBRUSH, HPEN are described in library <windows.h>.

```

char szWinName[] = "MyWindow";           // window class
int iXmax, iYmax;                         // window width and hight
HDC memdc;                               // DC of virtual window
HBITMAP hbit;                             // bitmap in virtual window
HBRUSH hbrush, hOldbrush;                 // DC of pen
HPEN hOldpen;                             // DC of old pen
HPEN hRedpen, hGreenpen, hBluepen,       // DCs of color pens
    hYellowpen, hCyanopen, hMargentapen;

```

Part of windows class parameters contains determination of user-created call-back function `WindowProc3456()` and determination of main menu name `MYMENU_003`. Menu is described in resource script file as presented in Table 4.7.

```
// Windows class determination
wcl.lpfWndProc   = WindowProc3456; // function of window
wcl.lpszMenuName = "MYMENU_003";   // main menu
```

User-delimited colour of window background could be established by function `(HBRUSH)GetStockObject()`. According to user suggestions, input parameters `WHITE_BRUSH` or `BLACK_BRUSH` predefine white and black colour respectively.

```
// to fill window by colour
// wcl.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
wcl.hbrBackground   = (HBRUSH)GetStockObject (BLACK_BRUSH);
```

In case of successful registration of window class, the creation of window of required class is allowed. Function `(HWND)CreateWindow()` was used for such purpose. Function contains eleven input parameters.

The first parameter (`szWinName`) requires class name. This parameter is declared before `WinMain()` function, in global part.

The second parameter ("`FIGURES`") represents string of window header. This string will be presented on window top as an etiquette.

The third parameter (`WS_OVERLAPPEDWINDOW`) represents style of default window: it will be presented in overlapping style to the other windows.

The fourth and the fifth parameters (`0,0`) represent the position of starting window as the geometrical figure on the screen. In that case, user-requested parameters of window will be used. According to scheme presented in Fig. 4.1, left top angle of window starts at position (`0,0`) of screen.

The sixth and the seventh parameters (`1200,800`) represent the width and height (x and y respectively) of windows as geometrical figure (in pixels).

Generally, it is not necessary to use an absolute values, because some risk to overload the real screen exist. In many cases, it is preferable to use macros `CW_USEDEFAULT` for fourth, fifth, sixth and seventh positions in order to receive the default window values.

All the other parameters are commented in the previous subsection, for function `CreateWindowEx()`.

```
// to create window
hwnd= CreateWindow (
    szWinName,           // class name
    "FIGURES",           // header
    WS_OVERLAPPEDWINDOW, // window style
    0,                   // CW_USEDEFAULT, // X-0-coordinate
    0,                   // CW_USEDEFAULT, // Y-0-coordinate
    1200,                // CW_USEDEFAULT, // width
    800,                 // CW_USEDEFAULT, // height
    HWND_DESKTOP,        // no additional window
    (HMENU) NULL,        // use class menu
    hThisInst,           // descriptor
    NULL);               // no additional arguments
```

CALLBACK. Call-back function `LRESULT CALLBACK WindowProc3456()` (see Tables 4.5c, 4.5d) contains the main program code. Managing of program could be done using several different ways: by virtual machine command, or using system window button *Close* for virtual machine destroying.

```
switch (message)
{
    case WM_CREATE: // to prepare tools
    case WM_DESTROY: // to end program
    case WM_PAINT: // to redraw window
    case WM_COMMAND: //
        switch ( LOWORD (wParam))
        {
            case IDM_01: // to draw pixel
            case IDM_02: // to draw line
            case IDM_03: // to draw ellipse
            case IDM_04: // to draw rectangle
            case IDM_05: // to clear
            case IDM_HELP: // to show comments
        }
    default:
}
```

All such cases are presented in `switch()` statement according to the input parameter `message` request. Several systemic cases represent preparation of tools (`WM_CREATE`), destruction of tools `WM_DESTROY` and usage of tools `WM_PAINT`. Several user-defined cases (through `WM_COMMAND`) represent drawing of the geometric primitives: pixel, line, ellipse, rectangle as well as window clearing and output of comments (traditional help).

Code of function `LRESULT CALLBACK WindowProc3456()` is very long compared to the previous `WindowProc1234()` function. Several code blocks belonging to the function `WindowProc3456()` are presented in partitioned form of included files (initial file divided into six parts) for the reader's convenience. It is difficult to present the whole file of quite large amount in the plain user-readable form.

```
#include "win03_create.h"
```

All cases will be described separately.

WM_CREATE. Code block, devoted to `case WM_CREATE`, is realized in file "win03_create.h" - see Table 4.5e.

Function `GetSystemMetrics()` is used in order to get window screen parameters `iXmax` and `iYmax` previously described in global section.

```
iXmax = GetSystemMetrics (SM_CXSCREEN);
iYmax = GetSystemMetrics (SM_CYSCREEN);
```

Three functions such as `CreateCompatibleBitmap()`, `GetDC()` and also `CreateCompatibleDC()` are used for receiving descriptors which will be used in operation to create bitmap in windows. `SelectObject()` function selects an object into the specified device context (according to the descriptor `hbit`).

Table 4.5c.

Program win03.cpp.

```

LRESULT CALLBACK WindowProc3456 (HWND  hwnd,      UINT  message,
                                WPARAM wParam,  LPARAM lParam)
{
    HDC hdc; PAINTSTRUCT paintstruct; int response;
    switch (message)
    {
        case WM_CREATE:
            #include "win03_create.h"
            break;

        case WM_DESTROY: // to end program
            #include "win03_destroy.h"
            break;

        case WM_PAINT: // to redraw window
            hdc = BeginPaint (hwnd, &paintstruct);    // to get DC
            BitBlt (hdc, 0, 0, iXmax, iYmax, memdc, 0, 0, SRCCOPY);
            EndPaint (hwnd, &paintstruct);           // to free DC
            break;

        case WM_COMMAND: //
            switch ( LOWORD (wParam))
            {
                case IDM_01: // pixel
                    #include "win03_command_01.h"
                    break;

                case IDM_02: // line
                    #include "win03_command_02.h"
                    break;

                case IDM_03: // ellipse
                    #include "win03_command_03.h"
                    break;

                case IDM_04: // rectangle
                    #include "win03_command_04.h"
                    break;

                case IDM_05:
                    MessageBox (hwnd, "", "Clear", MB_OK);
                    PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
                    InvalidateRect (hwnd, NULL, 1);
                    break;

                // continued to Table 4.5d
            }
    }
}

```

Table 4.5d.

Program win03.cpp.

```

        // continued from Table 4.5c

        case IDM_HELP:
            char *cc;
            cc="Pixels, Lines, Ellipses, Rectanges.\n\nEnjoy Window graphics!";
            MessageBox (hwnd,cc, "Help", MB_OK);
            break;
        };
        break;
    default:
        return DefWindowProc (hwnd, message, wParam, lParam);
    }
    return 0;
}

```

Table 4.5e.

Program win03_create.h.

```

// get window screen parameters:
iXmax = GetSystemMetrics (SM_CXSCREEN);
iYmax = GetSystemMetrics (SM_CYSCREEN);

// to create bitmap in windows
hdc  = GetDC (hwnd);
memdc = CreateCompatibleDC (hdc);
hbit  = CreateCompatibleBitmap (hdc, iXmax, iYmax);
SelectObject (memdc, hbit);

hbrush = (HBRUSH)GetStockObject (WHITE_BRUSH);
SelectObject (memdc, hbrush);
PatBlt (memdc,0,0,iXmax, iYmax,PATCOPY);

hRedpen      = CreatePen(PS_SOLID, 1, RGB(0xff, 0, 0));
hGreenpen    = CreatePen(PS_SOLID, 2, RGB( 0, 0xff, 0));
hBluepen     = CreatePen(PS_SOLID, 3, RGB( 0, 0, 0xff));
hYellowpen   = CreatePen(PS_SOLID, 4, RGB(0xff, 0xff, 0));
hCyanopen    = CreatePen(PS_SOLID, 5, RGB( 0, 0xff, 0xff));
hMargentapen = CreatePen(PS_SOLID, 6, RGB(0xff, 0, 0xff));

// too keep default pen
hOldpen = (HPEN)SelectObject (memdc, hYellowpen);
SelectObject (memdc, hOldpen);
ReleaseDC (hwnd, hdc);

```

Table 4.5f. RGB coding.

Colour	Red	Green	Blue	Yellow	Cyano	Margenta
RED	0xff	0	0	0xff	0	0xff
GREEN	0	0xff	0	0xff	0xff	0
BLUE	0	0	0xff	0	0xff	0xff

```
// to create bitmap in windows
hdc      = GetDC (hwnd);
memdc    = CreateCompatibleDC (hdc);
hbit     = CreateCompatibleBitmap (hdc, iXmax, iYmax);
SelectObject (memdc, hbit);
```

Function `GetStockObject()` retrieves a handle to one of the stock pens, brushes, fonts, or palettes (in that case - brush). `SelectObject()` function selects an object into the specified device context (according to the descriptor `hbrush`). Function `PatBlt()` paints the specified rectangle using the brush that is currently selected into the specified device context.

It means that the screen part will be coloured in white, or in other words so called screen initialization will be done. Screen part is limited by rectangle (left top angle at (0,0) position and right bottom angle at (iXmax, iYmax) position.

```
hbrush = (HBRUSH)GetStockObject (WHITE_BRUSH);
SelectObject (memdc, hbrush);
PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
```

Now is necessary to prepare the tools. All pens (`hRedpen`, `hGreenpen`,...), previously declared in global section, must be associated to the colour parameters. Function `CreatePen()` creates a logical pen that has the specified style (`PS_SOLID`), width(1..6), and colour (macros of RGB). The pen can subsequently be selected into a device context and used to draw lines and curves. Table 4.5f represents coding schema for several RGB colours: three pure and three mixed. Value `0xff` in hexadecimal format corresponds to number 255 in usual decimal format. Positioning of tree components in values from interval (0÷255) allows to receive full palette pure and mixed colours.

```
hRedpen      = CreatePen(PS_SOLID, 1, RGB(0xff, 0, 0));
hGreenpen    = CreatePen(PS_SOLID, 2, RGB( 0, 0xff, 0));
hBluepen     = CreatePen(PS_SOLID, 3, RGB( 0, 0, 0xff));
hYellowpen   = CreatePen(PS_SOLID, 4, RGB(0xff, 0xff, 0));
hCyanopen    = CreatePen(PS_SOLID, 5, RGB( 0, 0xff, 0xff));
hMargentapen = CreatePen(PS_SOLID, 6, RGB(0xff, 0, 0xff));
```

After selecting the new tools it is always necessary to keep old tools. In that case, `hOldpen` is associated with the yellow pen. `SelectObject()` function selects an object into the specified device context (according to the descriptor `hYellowpen`). `ReleaseDC()` function releases a device context (according to DC), freeing it for use by other applications. It frees only common and window DCs.

```
// too keep default pen
hOldpen = (HPEN)SelectObject (memdc, hYellowpen);
SelectObject (memdc, hOldpen);
ReleaseDC (hwnd, hdc);
```

Table 4.5g.

Program win03_destroy.h.

```

DeleteObject (hRedpen);
DeleteObject (hGreenpen);
DeleteObject (hBluepen);
DeleteObject (hYellowpen);
DeleteObject (hCyanopen);
DeleteObject (hMargentapen);
DeleteDC (memdc);
PostQuitMessage (0);

```

WM_DESTROY. Code block, devoted for case `WM_DESTROY`, is realized in file "win03_destroy.h" - Table 4.5g. All previously created objects from descriptors `hRedpen`, `hGreenpen`, `hBluepen`, `hYellowpen`, `hCyanopen`, `hMargentapen`, `memdc` must be destroyed, using function `DeleteObject()`. Function `PostQuitMessage(0)` indicates to the OS that a thread has made a request to terminate (quit). It is typically used in response to a `WM_DESTROY` message.

WM_PAINT. Code block, devoted for case `WM_PAINT` is aimed to redraw window - see Table 4.5c. Three functions are used.

`BeginPaint()` function prepares the specified window for painting and fills a `PAINTSTRUCT` structure with information about the painting. This function requires two input parameters: (`hwnd`, handle to the window to be repainted; `&paintstruct`, pointer to the `PAINTSTRUCT` structure that will receive painting information). If the function succeeds, the return value is the handle (DC) to a display device context for the specified window.

```
hdc = BeginPaint (hwnd, &paintstruct);    // to get DC
```

`BitBlt()` function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context. Function requires nine input parameters.

The first parameter - `hwnd`, handle to the destination device context.

The second parameter - 0, x -coordinate of the upper-left corner of the destination rectangle.

The third parameter - 0, y -coordinate of the upper-left corner of the destination rectangle.

The fourth parameter - `iXmax`, width of the source and destination rectangles.

The fifth parameter - `iYmax`, height of the source and the destination rectangles.

The sixth parameter - `memdc`, handle to the source device context.

The seventh parameter - 0, x -coordinate of the upper-left corner of the source rectangle.

The eighth parameter - 0, y -coordinate, in logical units, of the upper-left corner of the source rectangle.

The ninth parameter - `SRCCOPY`, a raster-operation code. These codes define how the colour data for the source rectangle will be combined with the colour data for the destination rectangle to achieve the final colour.

Table 4.5h.

Program win03_command_01.h.

```

MessageBox (hwnd,"","Pixels",MB_OK);
for (int i=0; i<=200; i++)
{
    SetPixel (
        memdc,    // handle to the device context
        40,       // x-coord. of the point to be set
        i,        // y-coord. of the point to be set
        RGB( 0,    0, 0xff)); //blue

    SetPixel (memdc, 50, i, RGB( 0, 0xff, 0)); //green
    SetPixel (memdc, 60, i, RGB( 0, 0xff, 0xff)); //cyano
};
InvalidateRect (hwnd, NULL, 1);

```

```

BitBlt (hdc, 0, 0, iXmax, iYmax, memdc, 0, 0, SRCCOPY);

```

EndPaint() function marks the end of painting in the specified window. This function is required for each call to the **BeginPaint()** function, but only after painting is completed. This function requires two input parameters: **hwnd**, handle to the window to be repainted; **&paintstruct**, pointer to the **PAINTSTRUCT** structure that will receive painting information.

```

EndPaint (hwnd, &paintstruct);    // to free DC

```

IDM_01 - PIXEL. **case WM_COMMAND** part represents user-created programing code, which could be used by selecting several **case** statements. Code block, devoted for **case IDM_01**, is realized in file "win03_command_01.h" (Table 4.5h). Pixeling operation is provided in loop **for()** in order to receive *y*-coordinate of the point to be set in interval (0÷200).

```

for (int i=0; i<=200; i++)
{
    SetPixel (
        memdc,    // handle to the device context
        40,       // x-coordinate of the point to be set
        i,        // y-coordinate of the point to be set
        RGB(0,0,0xff)); //blue
};

```

The **SetPixel()** function sets the pixel at the specified coordinates (x,y) to the specified colour. Function requires four input parameters: **memdc**, handle to the device context; **40**, *x*-coordinate of the point to be set; **i**, *y*-coordinate of the point to be set; the colour to be used to paint the point. In order to create a colour value, **RGB** macro was used. Colour request according to RGB value is presented in the Table 4.5f.

If the function succeeds, the return value is the RGB value that the function sets the pixel to. This value may differ from the color specified by user; that occurs when an exact match for the specified colour cannot be found. If the function fails, the return value is equal to (-1).

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. `1`, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Pixels* vertical lines formed from pixels are presented on the left.

IDM_02 - LINE. Code block, devoted for `case IDM_02`, is implemented in file "`win03.command_02.h`" - Table 4.5i. Line drawing (not using pixel context) is realized as a sequence of three operations: i) selecting of tool (pen), generally is possible to use default pen, (function `SelectObject()`; ii) establishing the initial position (function `MoveToEx()`) and iii) lining to final position (function `LineTo()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hGreenpen`).

```
SelectObject (memdc, hGreenpen);
```

Function `MoveToEx()` updates the current position to the specified point and optionally returns the previous position. This function requires three parameters: `memdc`, handle to a device context; `200`, x-coordinate of the new position; `200`, y-coordinate of the new position; `NULL`, pointer to a `POINT` structure; for `NULL` pointer, the previous position is not returned.

```
MoveToEx (memdc, 200, 200, NULL);
```

Function `LineTo()` draws a line from the current position up to but not including the specified point. This function requires three parameters: `memdc`, handle to a device context; `520`, x-coordinate of the line's ending point; `220`, y-coordinate of the line's ending point;

```
LineTo      (memdc, 520, 220);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Lines*, corresponding graphical primitives are presented: six lines as a horizontal lines at the center are drawn on different width and colour according to scheme presented in case `case WM_CREATE`.

IDM_03 - ELLIPSE. Code block, devoted for `case IDM_03`, is implemented in file "`win03.command_03.h`" - Table 4.5j. Ellipse and pie as a geometrical figures will be drawn using as a sequence of two operations:

- i) selecting of tool (pen) or default pen; (function `SelectObject ()`);
- ii) drawing ellipse (function `Ellipse ()`) or drawing cut of ellipse (pie) (function `Pie ()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hBluepen`).

```
SelectObject (memdc, hBluepen);
```

Table 4.5i.

Program win03_command_02.h.

```

MessageBox (hwnd, "", "Lines", MB_OK);
MoveToEx (memdc, 200, 200, NULL);

LineTo (
    memdc, // handle to the device context
    500,   // x-coord. of the line's ending point
    200); // y-coord. of the line's ending point
MoveToEx (
    memdc, // handle to the device context
    220,   // x-coord. of the new position
    220,   // y-coord. of the new position
    NULL); // previous position is not returned

//green
SelectObject (memdc, hGreenpen);
LineTo      (memdc, 520, 220);
MoveToEx    (memdc, 240, 240, NULL);
//red
SelectObject (memdc, hRedpen);
LineTo      (memdc, 540, 240);
MoveToEx    (memdc, 260, 260, NULL);
//blue
SelectObject (memdc, hBluepen);
LineTo      (memdc, 560, 260);
MoveToEx    (memdc, 280, 280, NULL);
//yellow
SelectObject (memdc, hYellowpen);
LineTo      (memdc, 580, 280);
MoveToEx    (memdc, 300, 300, NULL);
//cyano
SelectObject (memdc, hCyanopen);
LineTo      (memdc, 600, 300);
MoveToEx    (memdc, 320, 320, NULL);
//magenta
SelectObject (memdc, hMargentapen);
LineTo      (memdc, 620, 320);
MoveToEx    (memdc, 340, 340, NULL);

InvalidateRect (hwnd, NULL, 1);

```

`Ellipse()` function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle. The ellipse is outlined by using the current pen and is filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 10, x-coordinate of upper-left corner; 200, y-coordinate of upper-left corner; 200, x-coordinate of lower-right corner; 350, y-coordinate of lower-right corner.

Table 4.5j.

Program win03_command_03.h.

```

MessageBox (hwnd, "", "Ellipses", MB_OK);

SelectObject (memdc, hBluepen);
Ellipse (
    memdc, // handle to the device context
    10,    // x-coord., upper-left corner
    200,   // y-coord., upper-left corner
    200,   // x-coord., lower-right corner
    350);  // y-coord., lower-right corner

SelectObject (memdc, hBluepen);
Pie (
    memdc, // handle to the device context
    550,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    700,   // x-coord., lower-right corner
    250,   // y-coord., lower-right corner
    600,   // x-coord., endpoint of first radial
    15,    // y-coord., endpoint of first radial
    650,   // x-coord., endpoint of second radial
    150);  // y-coord., endpoint of second radial

SelectObject (memdc, hOldpen);
InvalidateRect (hwnd, NULL, 1);

```

```

Ellipse (
    memdc, // handle to the device context
    10,    // x-coordinate of upper-left corner
    200,   // y-coordinate of upper-left corner
    200,   // x-coordinate of lower-right corner
    350);  // y-coordinate of lower-right corner

```

`Pie()` function draws a pie-shaped wedge bounded by the intersection of an ellipse and two radials. The pie is outlined by using the current pen and filled by using the current brush.

```

Pie (
    memdc, // handle to the device context
    550,   // x-coordinate of upper-left corner
    10,    // y-coordinate of upper-left corner
    700,   // x-coordinate of lower-right corner
    250,   // y-coordinate of lower-right corner
    600,   // x-coordinate of endpoint of first radial
    15,    // y-coordinate of endpoint of first radial
    650,   // x-coordinate of endpoint of second radial
    150);  // y-coordinate of endpoint of second radial

```

`Pie()` function requires nine parameters: `memdc`, handle to the device context; 550, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 700, x -coordinate of lower-right corner; 250, y -coordinate of lower-right corner; 600, x -coordinate of endpoint of first radial; 15, y -coordinate of endpoint of first radial; 650, x -coordinate of endpoint of second radial; 150, y -coordinate of endpoint of second radial.

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. Function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. 1, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of `win03`. After pushing the menu button *Ellipses*, corresponding graphical primitives are presented: ellipse and pie at left bottom and right bottom respectively.

IDM_04 - RECTANGLE. Code block, devoted for case `IDM_04`, is implemented in file "`win03_command_04.h`" - Table 4.5k. Rectangle as a geometrical figure will be drawn using a sequence of two operations:

- i) selecting of tool (pen) or default pen, (function `SelectObject()`);
- ii) draw geometrically not-damaged rectangle (function `Rectangle ()`) or draw rectangle with rounded corners (function `RoundRect ()`).

`SelectObject()` function selects an object into the specified device context (according to the descriptor `hRedpen`).

```
SelectObject (memdc, hRedpen);
```

`Rectangle()` function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 100, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 150, x -coordinate of lower-right corner; 150, y -coordinate of lower-right corner.

```
Rectangle (
    memdc, // handle to the device context
    100,   // x-coordinate of upper-left corner
    10,    // y-coordinate of upper-left corner
    150,   // x-coordinate of lower-right corner
    150);  // y-coordinate of lower-right corner
```

`RoundRect()` function draws a geometrically damaged figure - rectangle with rounded corners. The rectangle is outlined by using the current pen and filled by using the current brush. This function requires five parameters: `memdc`, handle to the device context; 350, x -coordinate of upper-left corner; 10, y -coordinate of upper-left corner; 180, x -coordinate of lower-right corner; 180, y -coordinate of lower-right corner; 15, width of ellipse to draw the rounded corners; 13, height of ellipse to draw the rounded corners.

Table 4.5k.

Program win03_command_04.h.

```

MessageBox (hwnd, "", "Rectangles", MB_OK);

SelectObject (memdc, hRedpen);
Rectangle (
    memdc, // handle to the device context
    100,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    150,   // x-coord., lower-right corner
    150);  // y-coord., lower-right corner

SelectObject (memdc, hCyanopen);
RoundRect (
    memdc, // handle to the device context
    350,   // x-coord., upper-left corner
    10,    // y-coord., upper-left corner
    180,   // x-coord., lower-right corner
    180,   // y-coord., lower-right corner
    15,    // width of ellipse to draw the rounded corners
    13);   // height of ellipse to draw the rounded corners

SelectObject (memdc, hBluepen);
Rectangle (memdc, 500, 50, 390, 130);

SelectObject (memdc, hOldpen);
InvalidateRect (hwnd, NULL, 1);

```

```

RoundRect (
    memdc, // handle to the device context
    350,   // x-coordinate of upper-left corner
    10,    // y-coordinate of upper-left corner
    180,   // x-coordinate of lower-right corner
    180,   // y-coordinate of lower-right corner
    15,    // width of ellipse to draw the rounded corners
    13);   // height of ellipse to draw the rounded corners

```

`InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. `1`, background is erased when the `BeginPaint()` function is called.

```
InvalidateRect (hwnd, NULL, 1);
```

Fig. 4.7 represents output of win03. After pushing of menu button *Rectangles* corresponding graphical primitives are presented: three rectangles in the center.

IDM_05 - CLEAR. Code block, devoted for `case IDM_05` is aimed for clearing operations. Function `PatBlt()` paints the specified rectangle using the brush that is currently selected into the specified device context. It means that screen part starting from left top angle (0,0) until right bottom angle (iXmax, iYmax) will be coloured by white (so called screen incialization). `InvalidateRect()` function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn. That function requires three parameters: `hwnd`, handle to the window whose update region has changed; `NULL`, the entire client area is added to the update region. 1, background is erased when the `BeginPaint()` function is called.

```
MessageBox (hwnd, "", "Clear", MB_OK);
PatBlt (memdc, 0, 0, iXmax, iYmax, PATCOPY);
InvalidateRect (hwnd, NULL, 1);
```

IDM_HELP - HELP. Code block, devoted for `case IDM_HELP` traditionally is aimed for user-required information. String of *C* type contains information about used graphical primitives and welcome sentence, which will be presented in message box.

```
char *cc;
cc="Pixels, Lines, Ellipses, Rectanges.\n\nEnjoy Window graphics! ";
MessageBox (hwnd, cc, "Help", MB_OK);
```

Fig. 4.8 represents output of win03. After pushing of menu buttons *Clear* and *Help*, message box with the welcome sentence is presented.

HEADER FILE. Table 4.6 represents header file win03.h which must be included into win03.cpp and win03.rc files. The content of header file represents an association of `case` identifiers in `WM_COMMAND` expression only to the integer numbers through operator `define`. Integer number could be chosen freely, but it must be unique for each identifier.

```
#define IDM_01      100
```

RC FILE. Table 4.7 represents resource script (RC) file win03.rc which is responsible for program management organization using menu buttons. First of all, this file contains two `include` statements: library `<windows.h>` and header file `"win03.h"`.

Table 4.6. Header file.	Program win03.h.
<pre>// win03.h // #define IDM_01 100 #define IDM_02 101 #define IDM_03 102 #define IDM_04 103 #define IDM_05 104 #define IDM_HELP 110</pre>	

Table 4.7. RC file.

Program win03.rc.

```

//win03.rc, resource script.
#include "windows.h"
#include "win03.h"

// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

// Menu
MYMENU_003 MENU DISCARDABLE
BEGIN
    MENUITEM "&PIXELS",      IDM_01
    MENUITEM "&LINES",       IDM_02
    MENUITEM "&ELLIPSES",    IDM_03
    MENUITEM "&RECTANGLES",  IDM_04
    MENUITEM "&CLEAR",       IDM_05
    MENUITEM "&HELP",        IDM_HELP
END

#endif // English (U.S.) resources

```

```

#include "windows.h"
#include "win03.h"

```

The second part of resource script (RC) file contains the description of menu buttons. Statement containing menu name MYMENU_003 is responsible for menu recognition in WinMain() function.

```
MYMENU_003 MENU DISCARDABLE
```

After that, distribution of menu items is presented. Each menu item must be associated with the identifier, previously described in header file. After that, this menu item could be used in call-back function.

```

MENUITEM "&PIXELS",      IDM_01
MENUITEM "&LINES",       IDM_02
MENUITEM "&ELLIPSES",    IDM_03
MENUITEM "&RECTANGLES",  IDM_04
MENUITEM "&CLEAR",       IDM_05
MENUITEM "&HELP",        IDM_HELP

```

Figs. 4.6-4.8 represent the output of win03. Window and menu are present according to the task requirements.

In Fig. 4.7, pushing of menu buttons *Pixels*, *Lines*, *Ellipses*, *Rectangles* was provided. Corresponding graphical primitives occur on screen: vertical lines formed using function `pixel()` on the left; horizontal lines formed using function `pixel()` in the center; ellipses and cut of ellipses (pie) formed using functions `ellipse()` and `pie()` on the left bottom and on the right bottom respectively; rectangles formed using functions `rectangles()` in the center.

In Fig. 4.8, pushing of menu buttons *Clear* and *Help* was provided. Geometrical primitives are absent, message box with the welcome sentence is present. Enjoy window graphics!

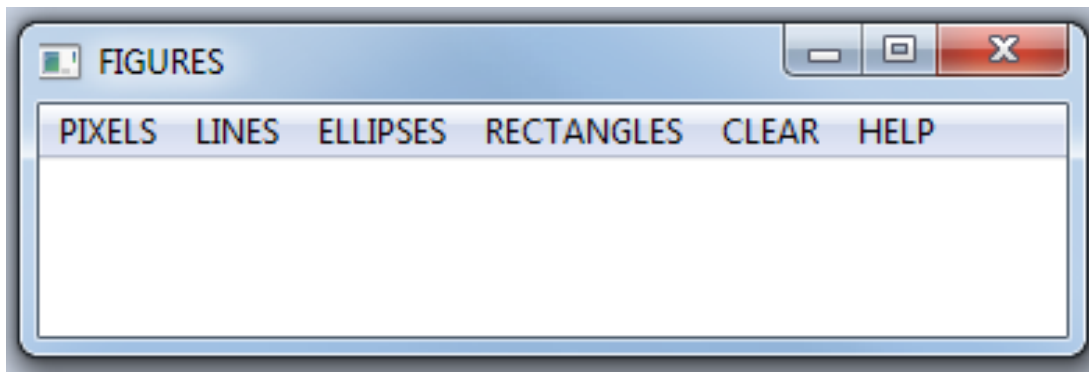


Fig. 4.6. Output of win03. Window and menu.

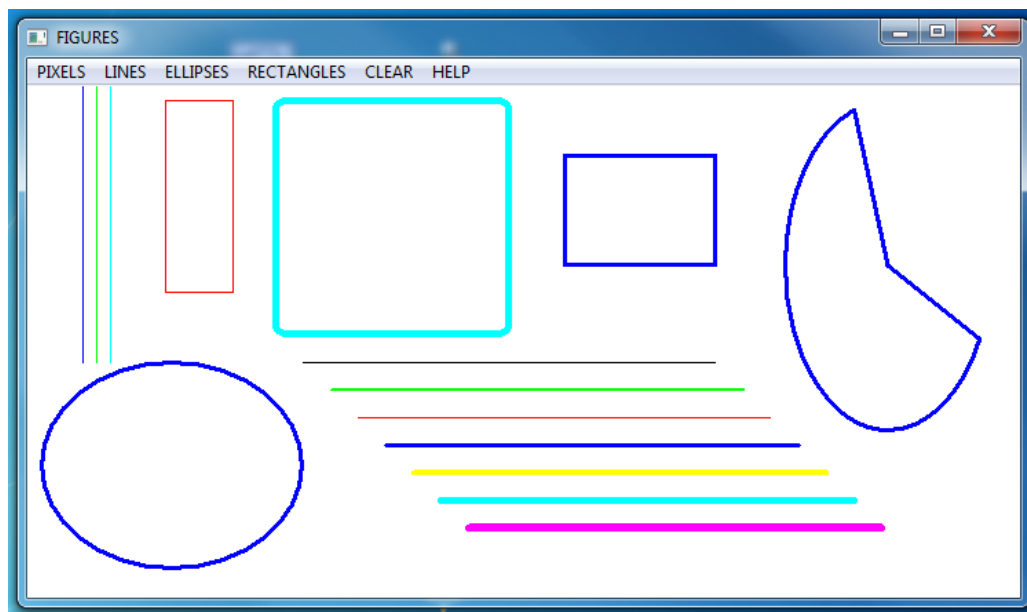


Fig. 4.7. Output of win03. Several graphical primitives.

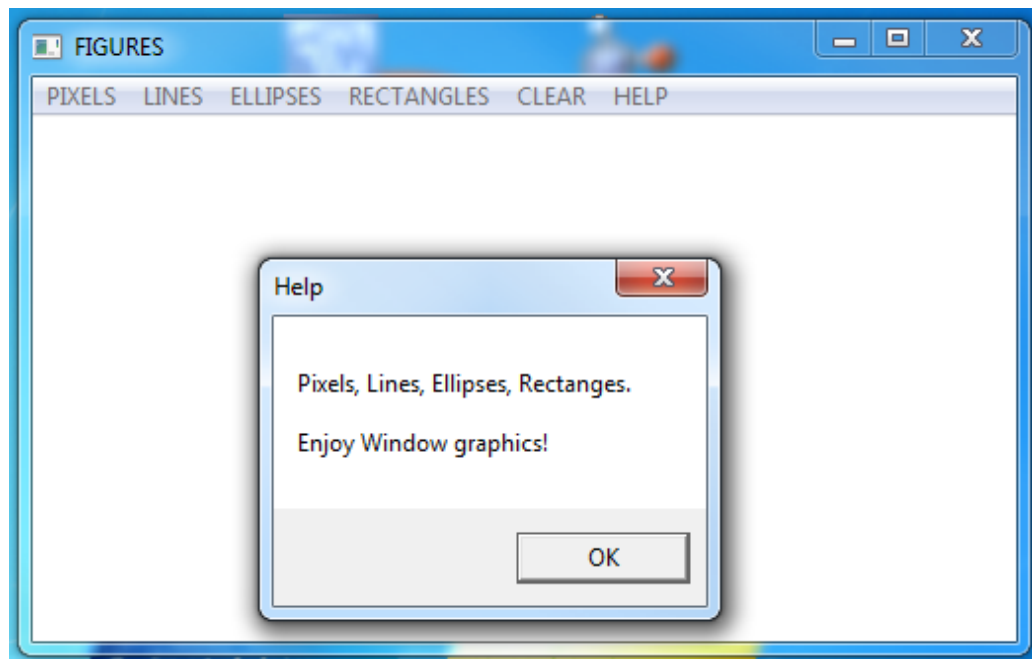


Fig. 4.8. Output of win03. Message box for help.

Chapter 5

Graphical programming using *wxWidgets*

C++ is a horrible language.
It's made more horrible
by the fact that a lot
of substandard programmers use it.

Linus Torvalds ^a

5.1 Graphic objects. Operating by events

WxDev-C++ uses wide and flexible library of typical objects. Data determines the object properties. For example, in a graphic object data determines size, position, colour, inscriptions, if it is visible or not and other properties. Some data about the object could be changed by users. In order to change the data we use *properties*. Object functions are called *methods*. Various operations can be performed with objects properties using methods. For example, form object has methods which make it possible to hide and show it, change the size, renew etc. Most *wxDev-C++* objects conform to the Windows graphic objects: windows, buttons and others.

We will not create any new class, but only apply typical objects for our requirements by changing their properties. We will change them while preparing and designing a program and also while executing it. That is why our objects during the performances could change the colour, disappear, move and etc.

WxDev-C++ graphic objects are forms and controls. Forms are windows where controls: labels, buttons, text fields, etc. are placed. As we will see there can be invisible controls. The program could have a lot of forms part of which could be invisible during the operating time.

The object use syntax similar to structure syntax. Objects and methods are separated using -> signs.

^aLinus Benedict Torvalds (born 1969) - a Finnish software engineer and hacker, who was the principal force behind the development of the *Linux* kernel. He later became the chief architect of the *Linux* kernel, and now acts as the project coordinator.

Operating by events. The action of a graphical program is supported by *events*. This is opposite to the console program operating where the execution starts with the first line then comes different loops, branches, the results are printed and the work of the program is finished. Programmer in this case is able to predict the work of the system, considering himself the master of the situation.

Operating by events the program execution depends on the user's actions and the state of the program. For example, working with the text editor, users action order and the work of the program can be very different, and hard to be predicted. Events are created not only by the user but also by the operating system, by the program itself and even by some separate program elements.

The alive nature also works according to this event oriented paradigm. While going somewhere we do not plan how many steps we are going to do forward, how many after turning right and if on our way we face an event "a closed door", we will execute the function open, if we come across an obstacle, we will overcome it.

An event is caused both by a user (for example, clicking a mouse or using a key on a keyboard) and a program (for example, if there is a mistake or if there is a program time signal). The objects (forms and regulators) reacting to the event implement some part of the program, called the event function. It is written by a programmer and the events recognition is made automatically, because the C++ system is responsible for it. If there is no event to which the object knows how to react, the program will wait for the following event. For example, when we click the mouse while the cursor is on the button with the inscription "Finish the program", the button click event appears. Such event function will have one line which will finish the program execution.

Operating by events the objects become like independent figures to which are determined how to react to this or that event as presented in Fig. 5.1. So the whole performance script stays very difficult for prediction. Because of great amount of objects, many different kinds of events and a great variety of users actions the traditional program creation becomes very complicated. Meanwhile, the use of events makes complex software product development much easier. The program code is divided into several separate event function creation.

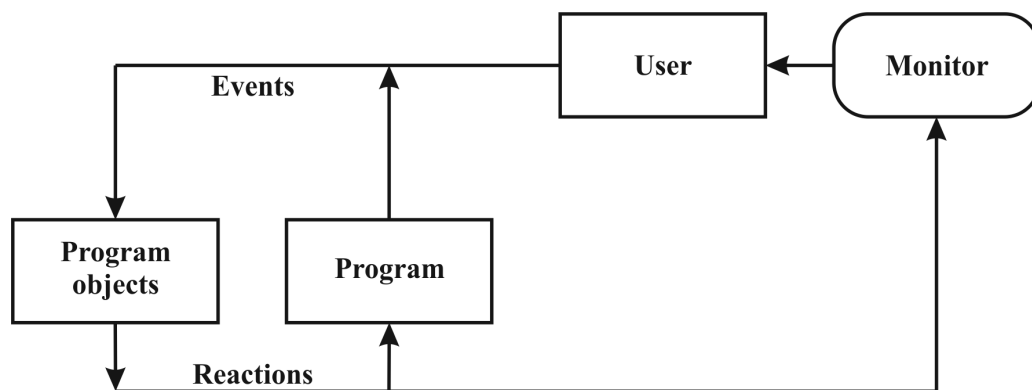


Fig. 5.1. The scheme of operating by events.

5.2 Creation of graphical project

We use the set of commands *File\New\Project* to create a new project in *wxDev-C++*. We can use *File\Open Project* or *File* to open an existing project. While creating a new project the dialogue window appears, where one must choose the type of the project and give it the name. We will use *wxWidgets* projects to create graphical programs. *WxDev-C++* contains three *wxWidgets* (GUI) project types:

- i) empty *wxWidgets* project a project that initially contains no source files but has the correct compiler and linker settings to build *wxWidgets* source code.
- ii) *wxWidgets Dialog* a project that contains skeleton source files for a *wxDialog*.
- iii) *wxWidgets Frame* a project that contains skeleton source files for a *wxFrame*.

We will use the second type of project in most of our programs. Some programs with *MDI* interface will be created using the third project type.

Selecting the second and the third project types, the dialogue window appears (as presented in Fig. 5.2). It is possible to set file and class names, project saving path, title, window style and other parameters in this window.

The view of the environment changes a bit using *wxWidgets* projects. Let us examine the parts of this IDE environment in more details (Fig. 5.3).

In the centre of the screen one can see the window named *Project1*. This is the form designing window where the regulators will be placed during the design process.

The form designer is the acting field for the design. Buttons and labels, pictures and other things can be situated on the form. If we create a complex program there is no need to limit ourselves by the only one form (Fig. 5.4).

The form by itself is one of the standard *wxDev-C++* objects. The buttons, scroll bars, drawings that will be situated in the form are also standard. So everything is made from standard objects. Don't you think that the program created with the *wxDev-C++* tools will be monotonous, as if it is a block of flats?

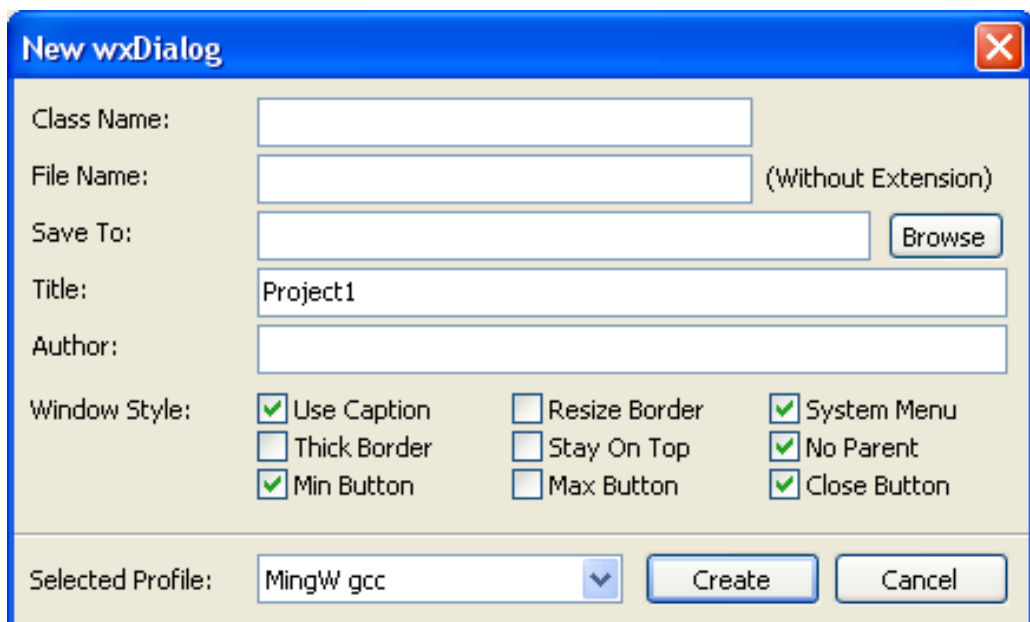


Fig. 5.2. Project *wxWidgets*: properties.

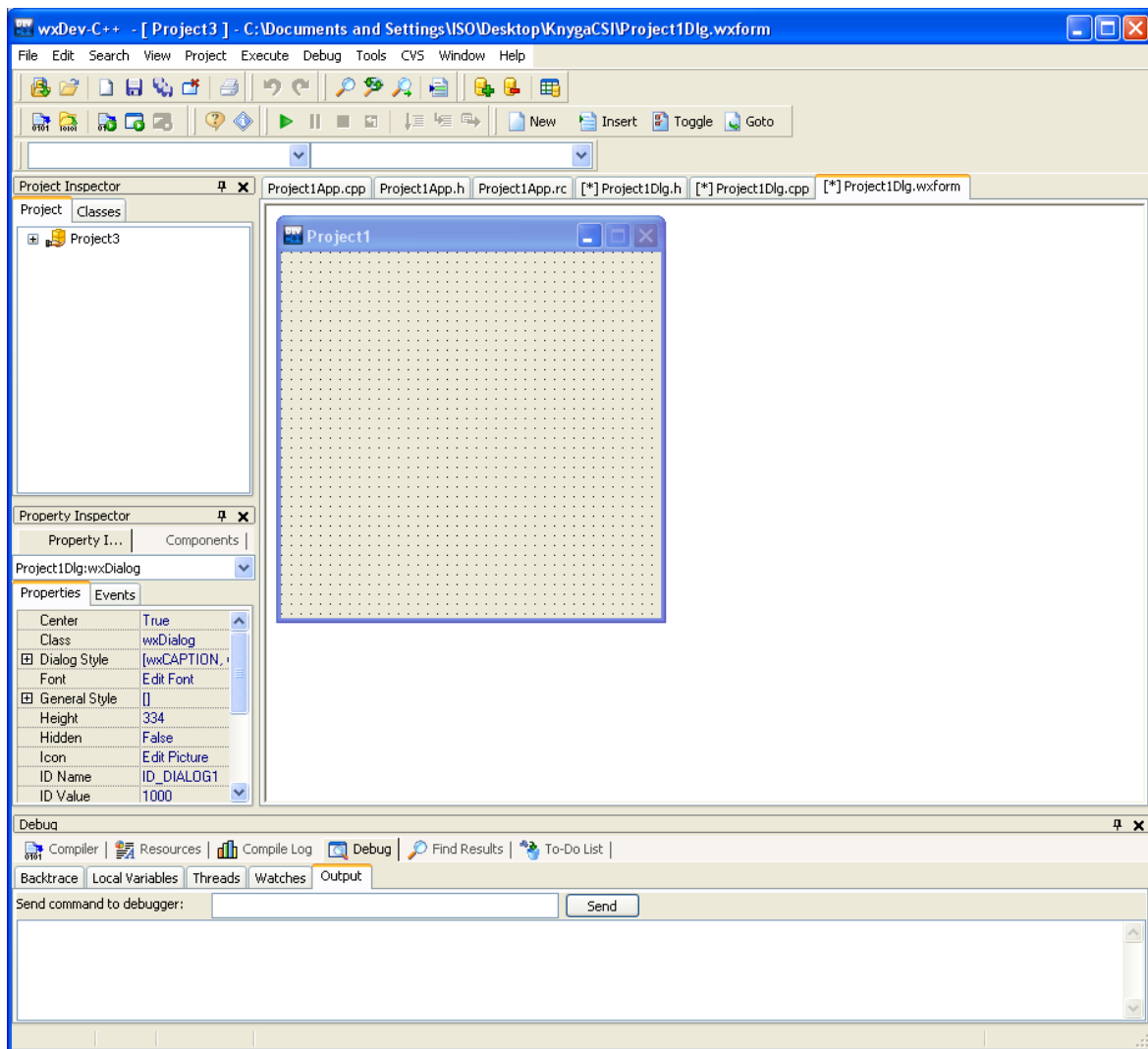


Fig. 5.3. The general view of the *wxWidgets**wxDev-C++* IDE environment.

This might happen if we didn't have a great possibility to change objects. Objects have a lot of changeable properties and it makes possible to create different programs and design graphic images of many various shapes.

We will use two concepts: *design* and *run*. While preparing a *wxDev-C++* program a design takes place when the program design is implemented, the property values are set and the event functions are written. When the program is started the project runs. These two stages sometimes are mixed because during the design the internal view of the program is almost the same as if it runs.

Working with *wxWidgets* project, *Property inspector* and *Components* panels are activated. The pictograms of the objects that can be used are seen in the *Components* toolbox which is usually on the right side of the screen (Fig. 5.5).

It is very easy to "take" an object from the toolbox and to "place" it into a necessary place in the form. It is enough to click the mouse on a desired object and then place it where you need by clicking the mouse on selected place. It is also easy to change the size of the object using the mouse. It is obvious that created objects can be copied, pasted and even deleted

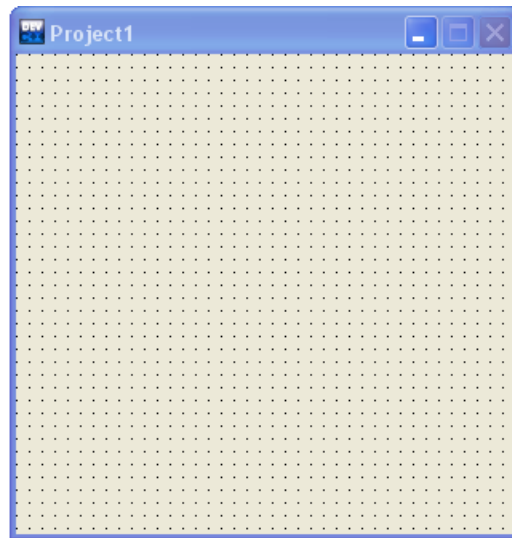


Fig. 5.4. Form design.

using the common operations of the *Windows* environment.

The example of the simplest object which pictogram *A* is in the toolbox is a static text. It is used exactly for the text writing.

The properties of the chosen object, in our case *wxBUTTON1*, are named in the window of *Property Inspector* (Fig. 5.6). Any property during the design period could be changed. All the objects have the property *Name*, most of the objects have the property *Label* or *Text*.

If it is possible to use only the fixed values of property, it is very convenient to choose the value from the list which appears by clicking with the mouse on a small button with a triangle at the right side of the editing line.

For the choice of the values of some properties it is possible to use the dialogue, for example, the dialogue of the choice of the colour, dialogue of the choice of font and its properties etc. Dialogue window is opened by clicking on appropriate button with the three-point.

Each object has its own list of properties. Window of *Project Inspector* shows the program modules and classes.

The program window appears only when we start to write event functions. For example, if we open the *Property Inspector's Events* area we can pick an event and choose *Add New Function*. We will see a program code window with our new function prepared. If we want, we can write a text in this function layout.

The most often used events are: *Click*, *Paint*, *MouseMotion*, *KeyDown*, *KeyUp*, etc.

The object may have a lot of events and also event processing functions. The list of all the regulator events can be seen in the *Event* area of *Property Inspector*. We can see that the form, for example, reacts to the movement of the mouse (*MouseMotion*) and other events.

There are a lot of different windows that are used in the designing process and not all of them are used at the same time. Almost all the time is used the form design window, often the *Property Inspector* and *Components window*. It is necessary to learn how to manipulate the windows and organize them in a convenient order for the use. If one has lost some window, it is necessary to appeal to the menu command *View\Show Panels* or *View\Show Plugin*.

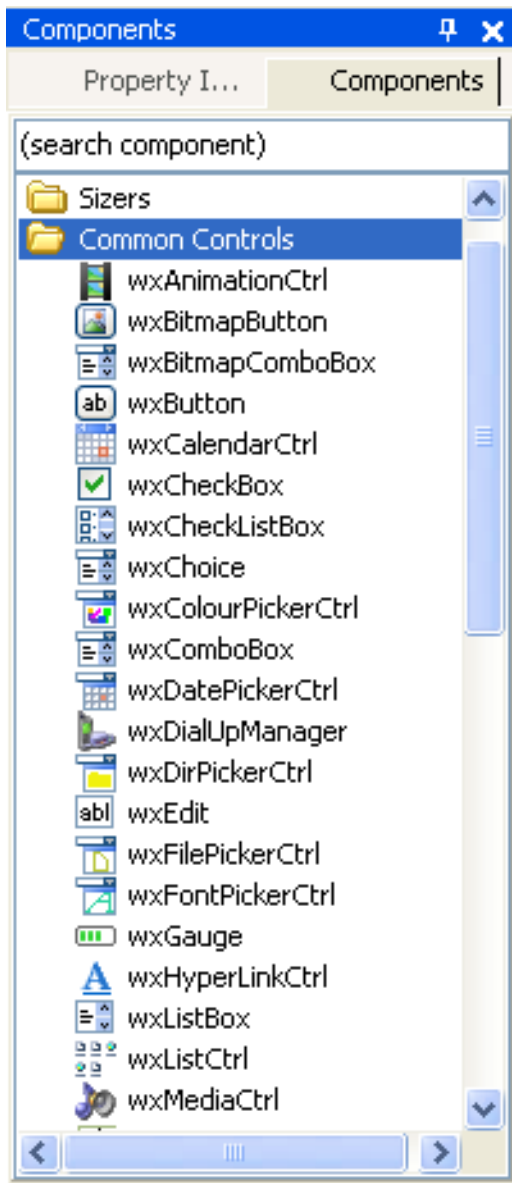


Fig. 5.5. Components toolbox.

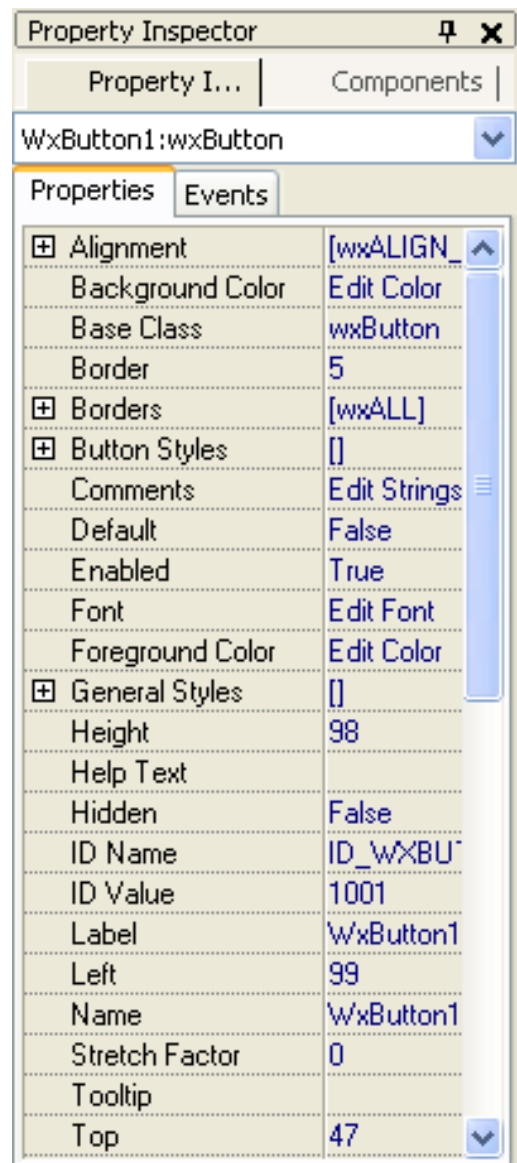


Fig. 5.6. Property Inspector.

5.3 Creation of the first program

The creation of the program could be divided into three main steps:

- i) creation of the user interface;
- ii) setting of the properties;
- iii) writing of the program text.

But this does not mean that in practice these steps or their fulfilment is always followed strictly. For example, it is possible to return back to the interface, to the setting of the properties because some new ideas could appear when the program is tested.

Table 5.1. The changes of the object properties.

Object	Property	Value of property
<i>Project1Dlg</i>	Title	My first program
<i>WxEdit1</i>	Text	Empty
<i>WxButton1</i>	Label	Click me

We will illustrate the steps of the creation of a program with some simple examples. At the beginning in the program window should be an empty text box (*wxEdit*) and a button (*wxButton*). If we click the button, "Hello world!!!" will appear in the text box.

First of all we will place our objects in the form using the *Component box*. Open *Common Controls* folder and click an appropriate pictogram in the *Component box*, then click the mouse on the wished place and change the object size using the mouse. The image of project interface is presented in Fig. 5.7.

The properties will be changed with the use of the *Property Inspector*. Let us make changes of the values of the objects properties presented in Table 5.1.

We write the text of the program in the program text window which is opened when we chose *Add New Function* in the *Event* area of *Property Inspector*. Such event function runs only when the particular event connected to the object **happens**. In our case it is event *Click*:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxEdit1 -> SetValue("Hello world!!!");
}
```

The program changes the text field *WxEdit1* property *Text* value from empty into "Hello world!!!". The value is changed bay using *SetValue* method. Fig. 5.8 illustrates the ouput of program, after the *Click me* button has been pressed.

If we want to save our project, we should not forget that it consists not of one file only. So the best way to save the projects is to create a separate catalogue for each particular project.

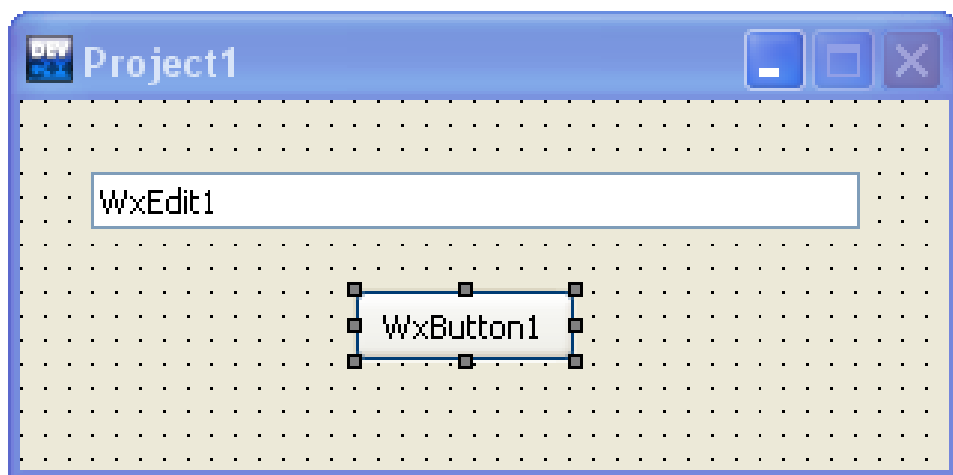


Fig. 5.7. The project interface.

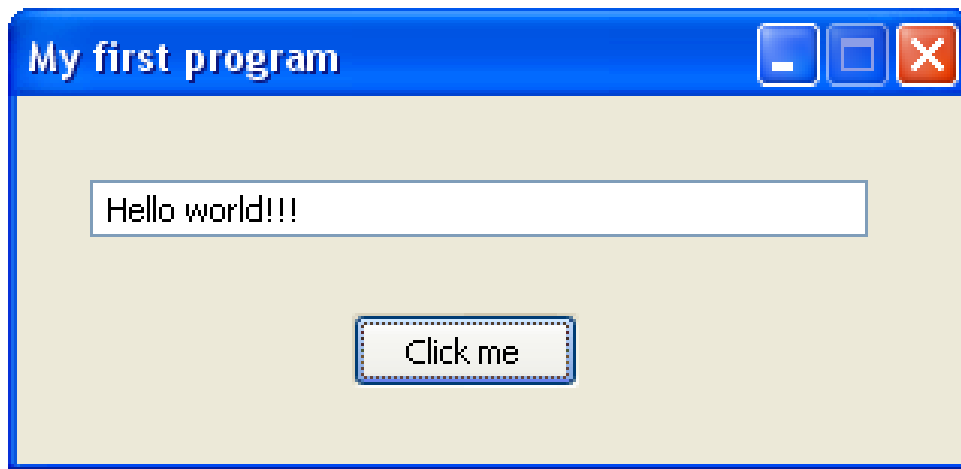


Fig. 5.8. Output of program: after button click.

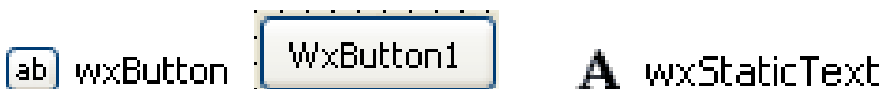
Fig. 5.9a. Pictogram of a button among the *Common Controls* image of the button in the program.

Fig. 5.9b. The image of the textbox and static text.

5.4 Basic control. Input/output of text

We can find the button among the *Common Controls* in the *Components box*. The pictogram of a button and its image in the program are presented in Fig. 5.9a.

The click of the button causes the event *Click*. That is why the part of the program we want to run with the click of this button we will write into the button *Click* event function. Very often the button is used to finish the program. That button *Click* event function will be the following:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    exit(0);
}
```

In graphical programs text boxes are usually used for text input, they are also used for the text output. But otherwise for text output there is one more rather simple and convenient regulator *Static Text*. Pictograms of these regulators are presented in Fig.5.9b.

The text of the *Static Text* regulator (the property *Label*) is set mostly during the design time. This is one of the ways to output text to the form.

For example, the font of the *Static Text* label "Jonas" (italic or normal) could be changed with the buttons during the running time.

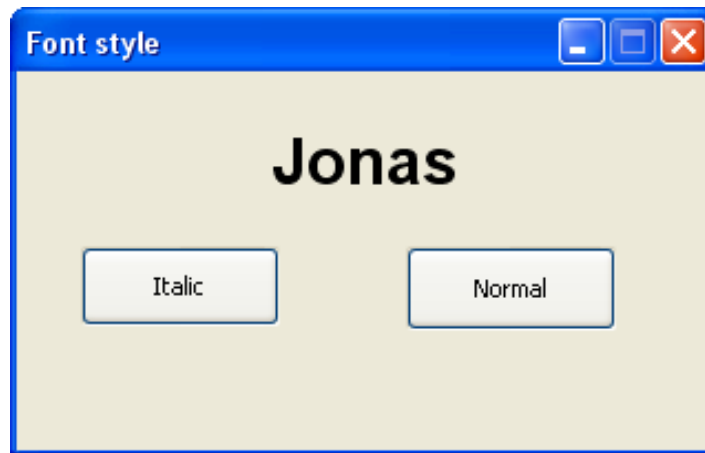


Fig. 5.10. Changing of the font.

Figure below presents the image of the form in which we can see the *WxStaticText1* with the inscription "Jonas" (the property *Label*) and two buttons *WxButton1* and *WxButton2* with the inscriptions "Italic" and "Normal" (see Fig. 5.10).

It is necessary to write the *Click* event function for the buttons (their first and last lines are always presented). The *SetFont* method is used to change the font of *StaticText*:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxStaticText1 -> SetFont(wxFont(22, wxSWISS, wxITALIC, wxBOLD, false));
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    WxStaticText1 -> SetFont(wxFont(22, wxSWISS, wxNORMAL, wxBOLD, false));
}
```

A font is an object which determines the appearance of the text, primarily when drawing text to a window or device context. A font is determined by the following parameters (not all of them have to be specified, of course):

<i>Point size</i>	standard way of referring to text size;
<i>Family</i>	supported families are: <code>wxDEFAULT</code> , <code>wxDECORATIVE</code> , <code>wxROMAN</code> , <code>wxSCRIPT</code> , <code>wxSWISS</code> , <code>wxMODERN</code> ;
<i>Style</i>	value can be <code>wxNORMAL</code> , <code>wxSLANT</code> or <code>wxITALIC</code> ;
<i>Weight</i>	value can be <code>wxNORMAL</code> , <code>wxLIGHT</code> or <code>wxBOLD</code> ;
<i>Underlining</i>	value can be <code>TRUE</code> or <code>FALSE</code> ;
<i>Face name</i>	an optional string specifying the actual typeface to be used. If <code>NULL</code> , a default typeface will chosen based on the family;
<i>Encoding</i>	font encoding.

Specifying a family, rather than a specific typeface name, ensures a degree of portability across platforms because a suitable font will be chosen for the given font family.

The text boxes are usually used for the text input, but also they can be used for the output of it. The most important thing is their property *Text* in which we can find the entered text. The property *Edit Style* `wxTE_MULTILINE` makes it possible to create a text box that consists of a lot of lines (if the value of the `wxTE_MULTILINE` is equal to `TRUE`). The property `MaxLength` specifies the maximum amount of symbols that can be entered into text box.

To enter the password, we shall use the text box property *Edit Style* - `wxTE_PASSWORD`. We use the event *Updated*, when some changes happen in the text.

Text box methods. Some text box methods are presented below.

`AppendText(const wxString& text)` - appends the text to the end of the text box. The parameter `text` specifies the text to write to the text control. After the text is appended, the insertion point will be at the end of the text control. If this behaviour is not desired, the programmer should use `GetInsertionPoint()` and `SetInsertionPoint()`.

`Clear()` - clears the text in the control. Note, that this function will generate a text updated event.

`Copy()` - copies the selected text to the clipboard.

`Cut()` - copies the selected text to the clipboard and removes the selection.

`GetInsertionPoint()` - returns the insertion point. This is defined as the zero based index of the character position to the right of the insertion point. For example, if the insertion point is at the end of the text control, it is equal to both `GetValue().Length()` and `GetLastPosition()`.

`GetLastPosition()` - returns the zero based index of the last position in the text control, which is equal to the number of characters in the control.

`GetLineLength(long lineNo)` - gets the length of the specified line, not including any trailing newline character(s). The parameter `lineNo` is the line number (starting from zero). The length of this line is returned, or (-1) if `lineNo` was invalid.

`GetLineText(long lineNo)` - returns the content of a given line in the text control, not including any trailing newline character(s). The parameter `lineNo` is the line number (starting from zero). The content of the specified line is returned.

`GetStringSelection()` - gets the text currently selected in the control. If there is no selection, the returned string is empty.

`GetValue()` - gets the contents of the control. Note, that for a multiline text control, the lines will be separated by (*UNIX*-style) `\n` characters, even under *Windows* where they are separated by a `\r\n` sequence in the native control.

`IsEditable()` - returns `TRUE` if the controls contents may be edited by user (note that it always can be changed by the program), i.e. if the control has not been put in read-only mode by a previous call to `SetEditable`.

`IsEmpty()` - returns `TRUE` if the control is currently empty. This is the same as `GetValue().empty()` but can be much more efficient for the multiline controls containing big amounts of text.

`IsMultiLine()` - returns `TRUE` if this is a multi line edit control and `FALSE` otherwise.

`IsSingleLine()` - returns `TRUE` if this is a single line edit control and `FALSE` otherwise.

`LoadFile(const wxString& filename, int fileType=wxTEXT_TYPE_ANY)` - loads and displays the named file, if it exists. The parameter `filename` specifies the filename of the file to load. The parameter `fileType` specifies the type of file to load.

`Paste()` - pastes text from the clipboard to the text item.

`Remove(long from, long to)` - removes the text starting at the first given position (parameter `from`) up to (but not including) the character at the last position (parameter `to`).

`Replace(long from, long to, const wxString& value)` - replaces the text starting at the first position (parameter `from`) up to (but not including) the character at the last position (parameter `to`) with the given text (parameter `value`).

`SaveFile(const wxString& filename, int fileType=wxTEXT_TYPE_ANY)` - saves the contents of the control in a text file specified by the `filename` parameter. The parameter `fileType` specifies the type of file to save.

`SetEditable(const bool editable)` - makes the text item editable or read-only (Boolean parameter `editable`). If the parameter value is `TRUE`, the control is editable. If `FALSE`, the control is read-only.

`SetInsertionPoint(long pos)` - sets the insertion point at the given position (parameter `pos`).

`SetInsertionPointEnd()` - sets the insertion point at the end of the text control. This is equivalent to `SetInsertionPoint(GetLastPosition())`.

`SetMaxLength(unsigned long len)` - this function sets the maximum number of characters the user can enter into the control. In other words, it allows to limit the text value length to `len` not counting the terminating `NULL` character. If `len` is equal to 0, the previously set maximal length limit, if any, is discarded and the user may enter as much text as the underlying native text control widget supports (typically at least 32Kb). If the user tries to enter more characters into the text control when it already is filled up to the maximal length, a `wxEVT_COMMAND_TEXT_MAXLEN` event is sent to notify the program about it (giving it the possibility to show an explanatory message, for example) and the extra input is discarded.

`SetSelection(long from, long to)` - selects the text starting at the first position (parameter `from`) up to (but not including) the character at the last position (parameter `to`). If both parameters are equal to (-1) all text in the control is selected.

`SetValue(const wxString& value)` - sets the text value and marks the control as not-modified (which means that `IsModified` would return `FALSE` immediately after the call to `SetValue`).

`ChangeValue(const wxString& value)` - sets the text value and marks the control as not-modified (which means that `IsModified` would return `FALSE` immediately after the call to `SetValue`). This function is new since *wxWidgets* version 2.7.1.

`ShowPosition(long pos)` - makes the line containing the given position (parameter `pos`) visible.

`WriteText(const wxString& text)` - writes the text (parameter `text`) into the text control at the current insertion position. After the write operation, the insertion point will be at the end of the inserted text, so subsequent write operations will be appended. To append text after the user may have interacted with the control, call `wxTextCtrl::SetInsertionPointEnd` before writing.

For example, let us create the input field that will not allow starting the text with any wrong word.

```
// WxEdit1Updated
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    if (WxEdit1 -> GetValue() == "bad")
        WxEdit1 -> SetValue ("This is a wrong word!");
}
```

For example, let us create a simple calculator which multiplies two numbers and shows the result as soon as the data is entered. Implementing this task, we shall create two text boxes *WxEdit1* and *WxEdit2* for the variables and the *StaticText* for the results. Than we shall write the following program code:

```
// WxEdit1Updated
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    wxString answer;
    answer << wxAtof (WxEdit1 -> GetValue()) *
        wxAtof (WxEdit2 -> GetValue());
    WxStaticText1 -> SetLabel(answer);
}
```

The same we shall repeat for the *WxEdit2*. Note, that we enter text data into text boxes, so we must convert the text into number (*wxAtof()* function), and then we must convert the result back to the string format.

5.5 Properties

5.5.1 The geometrical properties

Every *wxDev-C++* object contains a great list of properties: most properties in the objects are analogical. Almost all objects have their position and size. These properties are called the geometrical properties. The position coordinates are determined by the properties: *Left* in horizontal axis and *Top* in vertical axis. The size is determined by the properties: *Width* and *Height* (see Fig. 5.11).

The initial values of the geometrical properties are determined automatically during the designing time changing with the mouse the size and the position of the objects. During the program working time it is possible to move regulators, change their size by changing these properties.

The coordinates system begins form the upper left corner of the form or other container. It is, so called, a container coordinates system, which does not change, if it is moved, because the container and its regulators move together. The form can enclose other containers in which there can be their own regulators. The coordinates system beginning of those regulators is the upper left corner of their container.

The unit of measurement of geometrical properties is usually a pixel.

The example. There are four buttons and a static text in the form. The position of the static text is changed by the buttons (Fig. 5.12).

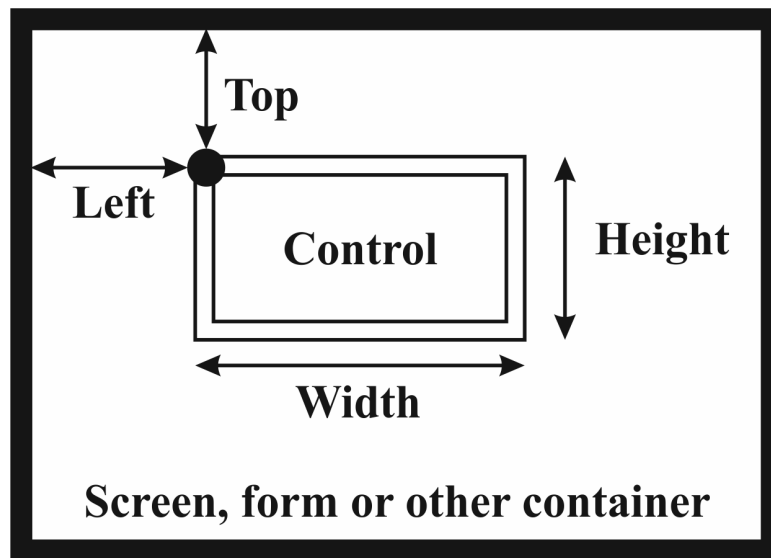


Fig. 5.11. The object properties that determine the position and size.

The property of the position of the label consists of two coordinates. During the execution of the program it is possible to get the values of control position (`GetPosition()` method) and it is also possible to set the new value of the coordinates (`Move()` method). So, let us write the following functions for the buttons:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x,y-5);
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x,y+5);
}

void Project1Dlg::WxButton3Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x-5,y);
}

void Project1Dlg::WxButton4Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x+5,y);
}
```

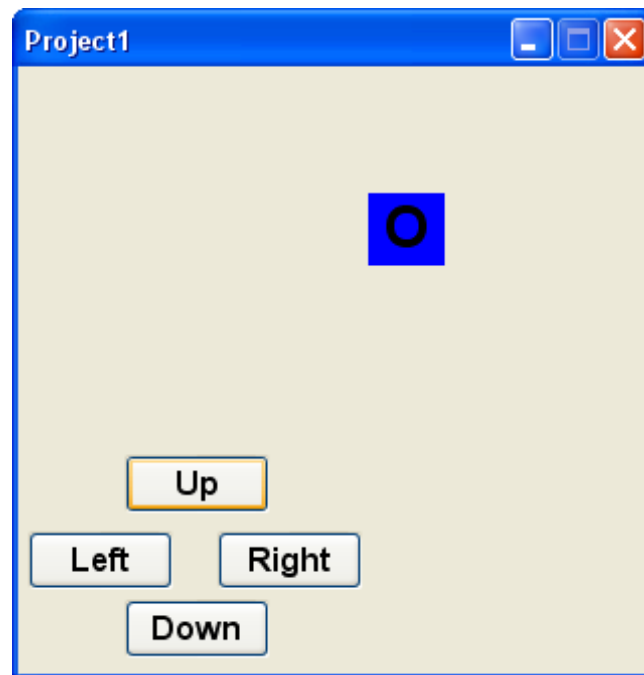


Fig. 5.12. Moving object program.

5.5.2 The properties of colours

Most of the objects have colour properties: *Background Color* as the colour of the background; *Foreground Color* as the colour of the text and drawings. It is easy to set the property of colour during the designing stage because we set it up in the special window (see Fig. 5.13). There are five possible ways to set the colour:

- i) it is possible to use default colour;
- ii) we can pick the colour from standard colours;
- iii) we can use system colours;
- iv) it is possible to define custom colour (using RGB format);
- v) it is possible to pick a colour from the palette (*Choose button*).

We can use `SetBackgroundColor()` and `SetForegroundColor()` methods to change control's colour properties during program run. The parameter of both methods is `wxColour`.

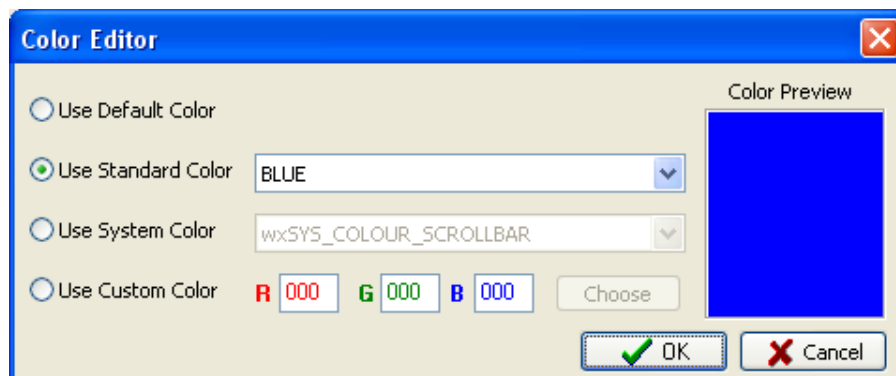


Fig. 5.13. Colour setting.



Fig. 5.14. Colour mixing.

Using RGB colour setting the meanings of the arguments can change from 0 (no colour) till 255 (the most intensive colour) (see Fig. 5.13).

The example. There are three text boxes in the form which corresponds three main colours (RGB). If we change the meaning of the intensity of the colour the static text will colour into a given tint (Fig 5.14).

Let us use the event *Updated* that happens just as we change the value of the text box. We shall change the text into numbers using the already known method `wxAtoi()`.

```
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    WxStaticText1->SetBackgroundColour(wxColour(wxAtoi (WxEdit1 -> GetValue()),
                                                    wxAtoi (WxEdit2 -> GetValue()),
                                                    wxAtoi (WxEdit3 -> GetValue())));
    WxStaticText1->Refresh();
}
```

The same code is copied and for the other text boxes. Note, that it is necessary to refresh the static text to see the changed colour.

5.5.3 Some most often used properties

The change of values of the logical property *Hidden* is very often. This property enables the programmer to hide or show the object. Method `Show()` can be used to hide or show the object (belonging on input parameter: `TRUE` show, `FALSE` hide).

The value of the logical property *Enabled* is usually equal to `TRUE`. The meaning `FALSE` is used if we want to make any of the objects out of work, because its work at that moment is not necessary. If the form is out of work, all the regulators are out of work too.

For example, the button that changes font into italic is not necessary, if that font is already in use at the moment.

The property *Font* determines the font style. It is the object itself with its own properties. Its properties:

- a) size (whole numbers);
- b) font family;
- c) font style;
- d) font weight;
- e) logical Underline;
- f) name (*Arial*, *Symbol* and etc.).

During the program design the font properties are determined through the dialogue.

```

wxFont(           int    pointSize,
                  wxFontFamily family,
                  int    style,
                  wxFontWeight weight,
                  const bool underline = false,
const wxString&   faceName  = "",
                  wxFontEncoding encoding = wxFONTENCODING\_DEFAULT)

```

Lets starts to describe all significant parameters.

<i>pointSize</i>	Size in points.
<i>family</i>	font family, a generic way of referring to fonts without specifying actual facename. One of: wxFONTFAMILY_DEFAULT , chooses a default font; wxFONTFAMILY_DECORATIVE , a decorative font; wxFONTFAMILY_ROMAN , a formal, serif font; wxFONTFAMILY_SCRIPT , a handwriting font; wxFONTFAMILY_SWISS , a sans-serif font; wxFONTFAMILY_MODERN , a fixed pitch font; wxFONTFAMILY_TELETYPE , a teletype font;
<i>style</i>	one of : wxFONTSTYLE_NORMAL ; wxFONTSTYLE_SLANT ; wxFONTSTYLE_ITALIC ;
<i>weight</i>	Font weight, sometimes also referred to as font boldness. One of;; wxFONTWEIGHT_NORMAL , normal font; wxFONTWEIGHT_LIGHT , light font; wxFONTWEIGHT_BOLD , old font;
<i>underline</i>	The value can be TRUE or FALSE ;
<i>faceName</i>	An optional string specifying the actual typeface to be used. If it is an empty string, a default typeface will be chosen based on the family;
<i>encoding</i>	An encoding which may be one of: wxFONTENCODING_SYSTEM , default system encoding; wxFONTENCODING_DEFAULT , default application encoding: this is the encoding set by calls to SetDefaultEncoding and which may be set to, say, KOI8 to create all fonts by default with KOI8 encoding. Initially, the default application encoding is the same as default system encoding; wxFONTENCODING_ISO8859_1...15 , ISO8859 encodings; wxFONTENCODING_KOI8 , standard Russian encoding for Internet; wxFONTENCODING_CP1250...1252 , <i>Windows</i> encodings similar to ISO8859 (but not identical).

While starting the program the first form is loaded and shown and the function **CreateGUIControls()** is run.

At the beginning the values of the numerical variables are usually equal to zero that is why it is necessary to describe the determination of the starting meanings in the `CreateGUIControls()` function.

The local variable values are accessed only in their own function. They are also declared in that function.

The class level variable values are accessed in all the functions of this class.

For example. Let us create an event function, that will proportionally change the position of the regulators in the form (for example, the position of the button in the form) while changing the size of the form (see Fig. 5.15).

Implementing this program, we should create two global variables to store the relative position of the button. We can do it in class description (*Project1Dlg.h* file). Two public real variables (`rx` and `ry`) are declared.

```
class Project1Dlg : public wxDialog
{
private:
    DECLARE_EVENT_TABLE();
public:
    Project1Dlg(        wxWindow    *parent,
                      wxWindowID    id    = 1,
                      const wxString &title = wxT("Project1"),
                      const wxPoint  &pos  = wxDefaultPosition,
                      const wxSize    &size = wxDefaultSize,
                      long            style = Project1Dlg_STYLE);
    virtual ~Project1Dlg();
    float rx, ry;
    void Project1DlgSize(wxSizeEvent& event);
    ...
}
```

Then, at the beginning of the program (`CreateGUIControls()` function in *Project1Dlg.cpp* file) it is necessary to calculate the relative position of the button in the form and to remember it. After the change of the size of the form (form event *Size*) recalculate the coordinates of the position of the button (Table 5.2).

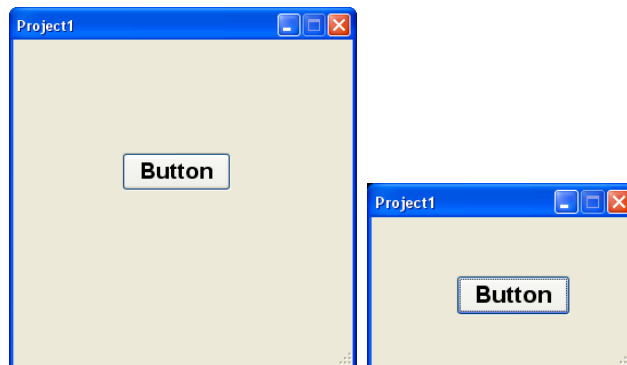


Fig. 5.15. Relative position.

Table 5.2. Relative position.

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    // //GUI Items Creation Start

    WxButton1 = new wxButton(this, ID_WXBUTTON1, _("Button"),
                             wxPoint(100, 130),
                             wxSize(100, 35), 0,
                             wxDefaultValidator, _("WxButton1"));
    WxButton1 -> SetFont(wxFont(16, wxSWISS, wxNORMAL, wxBOLD, false));

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End
    int x1, y1, x2, y2;
    GetSize(&x1, &y1); // The size of the form
    WxButton1 -> Move(100, 130); // Button position
    WxButton1 -> GetPosition(&x2, &y2); // Position values
    rx = (float) x2/x1; // Relative horizontal position
    ry = (float) y2/(y1-55); // Relative vertical position
                                // 55 - height of form header
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

void Project1Dlg::Project1DlgSize(wxSizeEvent& event)
{
    int x, y;
    GetSize(&x, &y); // New size of the form
    WxButton1 -> Move((int)(rx*x), (int)(ry*y)); // New position of button
}

```

5.6 Timer and scrollbars

Timer control is invisible during program execution, but it enables program to measure time and to perform different actions connected with time intervals.

Fig. 5.16. *wxTimer* view.

In *wxDev-C++* we have **wxTimer** function (Components folder) (Fig. 5.16). The timer component has no geometrical properties. The main purpose of this control is to execute code at specified intervals (*Timer* events are initiated). Time interval is set using *Interval* property. The timer can be enabled or disabled using methods **Start()** and **Stop()**.

For example, let us create a form, which slowly moves to the right. We will need timer to implement this program (*Interval* value 100, *AutoStart()* value **TRUE**).

Table 5.3a. Blinking rectangle.

```
void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    // //GUI Items Creation Start

    WxStaticText1 = new wxStaticText(this,
                                     ID_WXSTATICTEXT1, _("WxStaticText1"),
                                     wxPoint(105, 190), wxDefaultSize,
                                     0, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,255,0));

    WxTimer4 = new wxTimer(); WxTimer4 -> SetOwner (this, ID_WXTIMER4);
    WxTimer3 = new wxTimer(); WxTimer3 -> SetOwner (this, ID_WXTIMER3);
    WxTimer2 = new wxTimer(); WxTimer2 -> SetOwner (this, ID_WXTIMER2);
    WxTimer1 = new wxTimer(); WxTimer1 -> SetOwner (this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    // //GUI Items Creation End
    WxTimer1 -> Start(2000,wxTIMER_ONE_SHOT);
}
```

Table 5.3b. Blinking rectangle.

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    WxStaticText1->Hide();          WxTimer2 -> Start(1000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer2Timer(wxTimerEvent& event)
{
    WxStaticText1->Show();          WxTimer3 -> Start(3000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer3Timer(wxTimerEvent& event)
{
    WxStaticText1->Hide();          WxTimer4 -> Start(2000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer4Timer(wxTimerEvent& event)
{
    WxStaticText1->Show();          WxTimer1 -> Start(2000,wxTIMER_ONE_SHOT);
}

```

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    int x, y;
    GetPosition(&x, &y);
    Move(x+5,y);
}

```

We can use several timers to perform more complex tasks.

For example, let us create a program, where a red rectangle will blink according to some complex rhythm (2 seconds - visible, 1 second - disappeared, 3 seconds - visible, 2 seconds disappeared and then the rhythm is repeated). We will need four timers and a static text. Timer functions are shown in Tables 5.3a and 5.3b.

Start method has two parameters: timer interval and shooting mode:

`wxTIMER_CONTINUOUS` - start a normal, continuously running, timer;
`wxTIMER_ONE_SHOT` - start a one shot timer.

We can refuse three timers using global variables (Tables 5.4a, 5.4b, 5.4c).

Timer program. Let us create a time measuring program (it will measure time between *Start* and *Stop* button presses). We will measure time in milliseconds using `wxTimer` control (Time 1) and using time functions (Time 2), local time value will be also shown in `wxEdit3` (Local time) after button *Show time* click (see Fig. 5.17). Time 2 will be counted using stop watch class. We must define the public `wxStopWatch` object `sw` and `long t` in project class declaration. The program code is presented in Table 5.5.

Table 5.4a. Blinking rectangle with one timer.

```

class Project1Dlg : public wxDialog
{
    private:
        DECLARE_EVENT_TABLE();

    public:
        Project1Dlg(        wxWindow    *parent,
                           wxWindowID    id = 1,
                           const wxString &title = wxT("Project1"),
                           const wxPoint& pos = wxDefaultPosition,
                           const wxSize& size = wxDefaultSize,
                           long style = Project1Dlg_STYLE);
        virtual ~Project1Dlg();
        int interval, step; // Two public variables are declared
        void Project1DlgSize (    wxSizeEvent& event);
        void WxButton1Click (wxCommandEvent& event);
        void WxTimer1Timer ( wxTimerEvent& event);
        ...
}

```

Table 5.4b. Blinking rectangle with one timer.

```

void Project1Dlg::CreateGUIControls()
{
    WxStaticText1 = new wxStaticText(this,
                                     ID_WXSTATICTEXT1, _("WxStaticText1"),
                                     wxPoint(105, 190), wxDefaultSize,
                                     0, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,255,0));

    WxTimer1 = new wxTimer();
    WxTimer1 -> SetOwner(this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End

    step=0;
    interval=2000;
    WxTimer1->Start(interval,wxTIMER_ONE_SHOT);
}

```

Table 5.4c. Blinking rectangle with one timer.

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    step++;
    if (step>4) step=1;
    switch (step)
    {
        case 1:
            WxStaticText1 -> Hide();    interval=1000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 2:
            WxStaticText1 -> Show();    interval=3000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 3:
            WxStaticText1 -> Hide();    interval=2000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 4:
            WxStaticText1 -> Show();    interval=2000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
    }
}

```

Timer control and stopwatch time measurements will differ because of timer precision. Its precision is platform-dependent, but, in general, will not be better than 1 ms or worse than 1 s. So, it is better to use `wxStopWatch` object to measure time (especially in milliseconds).

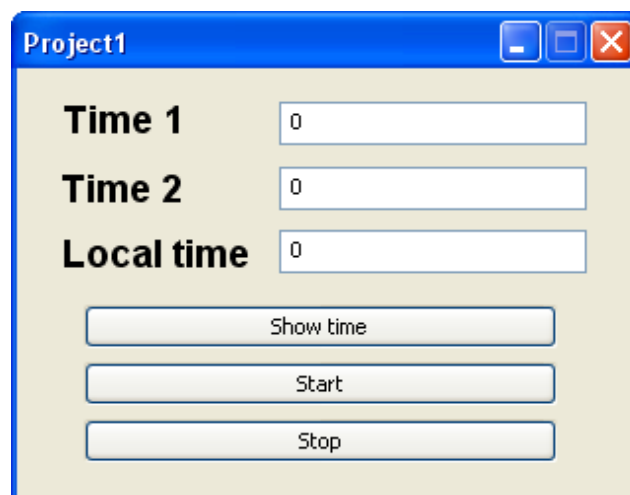


Fig. 5.17. Timer program interface.

Table 5.5.	Timer program code.
	<pre> void Project1Dlg::WxButton1Click(wxCommandEvent& event) { WxEdit3->SetValue(wxNow()); } void Project1Dlg::WxButton2Click(wxCommandEvent& event) { t=0; WxTimer1 -> Start(1,wxTIMER_CONTINUOUS); // Timer is started sw.Start(); WxEdit1 -> SetValue("0"); WxEdit2 -> SetValue("0"); } void Project1Dlg::WxTimer1Timer(wxTimerEvent& event) { t++; } void Project1Dlg::WxButton3Click(wxCommandEvent& event) { WxTimer1->Stop(); sw.Pause(); wxString ss; ss << sw.Time(); WxEdit2 -> SetValue(ss); ss = ""; ss << t; WxEdit1 -> SetValue(ss); } </pre>

Moving bars program. In this program we will create two moving bars (Static text controls). Each bar will move with its own speed and will reflect from form's borders (see Fig. 5.18). Program code is presented in Tables 5.6a, 5.6b.

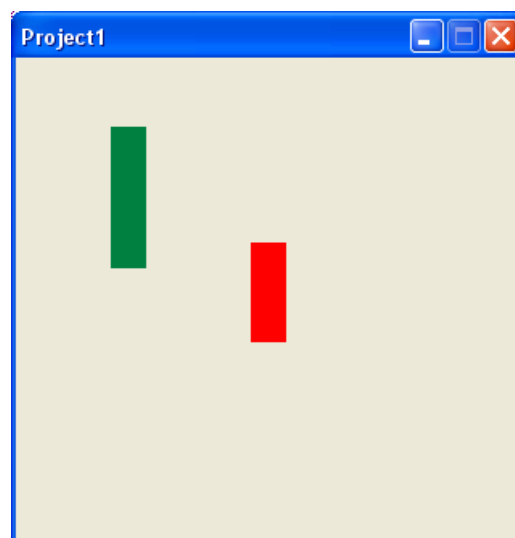


Fig. 5.18. Interface of Moving bars program.

Table 5.6a. Moving bars program code.

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    //// GUI Items Creation Start

    WxStaticText2 = new wxStaticText(this, ID_WXSTATICTEXT2, _(""),
                                     wxPoint(146, 115), wxSize(22, 62),
                                     wxST_NO_AUTORESIZE, _("WxStaticText2"));
    WxStaticText2 -> SetBackgroundColour (wxColour(255,0,0));

    WxStaticText1 = new wxStaticText(this, ID_WXSTATICTEXT1, _(""),
                                     wxPoint(59, 43), wxSize(22, 88),
                                     wxST_NO_AUTORESIZE, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,128,64));

    WxTimer2 = new wxTimer(); WxTimer2 -> SetOwner(this, ID_WXTIMER2);
    WxTimer1 = new wxTimer(); WxTimer1 -> SetOwner(this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End

    WxTimer1 -> Start(100, wxTIMER_CONTINUOUS);
    WxTimer2 -> Start(100, wxTIMER_CONTINUOUS);
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

```

In order to enter the parameter values it is possible to use not only text box (`wxEdit`). Sometimes it is more convenient to use scroll bars or sliders (`wxScrollBar` or `wxSlider`). These are rather similar controls which allow to set the parameter value from fixed interval. Both of these controls can be either horizontal or vertical (Orientation property: `wxSB_HORIZONTAL` or `wxSB_VERTICAL`). The main event for both of them is `Scroll`.

`WxScrollBar` is a control that represents a horizontal or vertical scrollbar. A scrollbar has the following main attributes: range, thumb size, page size, and position. The range is the total number of units associated with the view represented by the scrollbar.

Table 5.6b.

Moving bars program code.

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    static int x, y, h, w, hb, wb, dx=5, dy=7;
    // dx is horizontal speed, and dy is vertical speed of bar
    WxStaticText1 -> GetPosition(&x,&y);
    GetSize(&w,&h);
    WxStaticText1 -> GetSize(&wb,&hb);
    if (((x<abs(dx)) && (dx<0)) || ((x>(w-wb-dx)) && (dx>0))) dx=-dx;
    if (((y<abs(dy)) && (dy<0)) || ((y>(h-hb-dy)) && (dy>0))) dy=-dy;
    WxStaticText1 -> Move(x+dx,y+dy);
}

void Project1Dlg::WxTimer2Timer(wxTimerEvent& event)
{
    static int x, y, h, w, hb, wb, dx=-7, dy=-3;
    // dx is horizontal speed, and dy is vertical speed of bar
    WxStaticText2 -> GetPosition(&x,&y);
    GetSize(&w,&h);
    WxStaticText2 -> GetSize(&wb,&hb);
    if (((x<abs(dx)) && (dx<0)) || ((x>(w-wb-dx)) && (dx>0))) dx=-dx;
    if (((y<abs(dy)) && (dy<0)) || ((y>(h-hb-dy)) && (dy>0))) dy=-dy;
    WxStaticText2 -> Move(x+dx,y+dy);
}

```

For a table with 15 columns, the range would be 15. The thumb size is the number of units that are currently visible.

For the table example, the window might be sized so that only 5 columns are currently visible, in which case the application would set the thumb size to 5. When the thumb size becomes the same as or greater than the range, the scrollbar will be automatically hidden on most platforms. The page size is the number of units that the scrollbar should scroll by, when 'paging' through the data. This value is normally the same as the thumb size length, because it is natural to assume that the visible window size defines a page. The scrollbar position is the current thumb position.

Some scroll bar methods are presented below.

`GetRange()` - returns the length of the scrollbar.

`GetPageSize()` - returns the page size of the scrollbar. This is the number of scroll units that will be scrolled when the user pages up or down. Often it is the same as the thumb size.

`GetThumbPosition()` - returns the current position of the scrollbar thumb.

`GetThumbSize()` - returns the thumb or 'view' size.

`SetThumbPosition(int viewStart)` - sets the position of the scrollbar. The parameter `viewStart` represents the position of the scrollbar thumb.

`SetScrollbar(int position, int thumbSize, int range, int pageSize, const bool refresh = TRUE)` - sets the scrollbar properties. Parameters are

presented below.

position -	the position of the scrollbar in scroll units.
thumbSize -	the size of the thumb, or visible portion of the scrollbar, in scroll units.
range -	the maximum position of the scrollbar.
pageSize -	the size of the page size in scroll units. This is the number of units the scrollbar will scroll when it is paged up or down. Often it is the same as the thumb size.
refresh -	TRUE to redraw the scrollbar, FALSE otherwise.

Slider. A slider is a control with a handle which can be pulled back and forth to change the value. Slider events are handled in the same way as a scrollbar. Some useful methods:

GetMax() - gets the maximum slider value.

GetMin() - gets the minimum slider value.

GetPageSize() - returns the page size.

GetValue() - gets the current slider value.

SetLineSize(int lineSize) - sets the line size for the slider. The parameter **lineSize** specifies the number of steps the slider moves when the user moves it up or down a line.

SetPageSize(int pageSize) - sets the page size for the slider. The parameter **pageSize** specifies the number of steps the slider moves when the user pages up or down.

SetRange(int minValue, int maxValue) - sets the minimum and maximum slider values.

SetValue(int value) - sets the slider position. The parameter **value** specifies the slider position.

For example, this is a simple program with scroll bar, slider and gauge. We can set scroll bar position using slider, and we can change slider position using scroll bar.

The values are presented using gauge and text box (see Fig. 5.19). The function code is presented in Tables 5.7a, 5.7b.

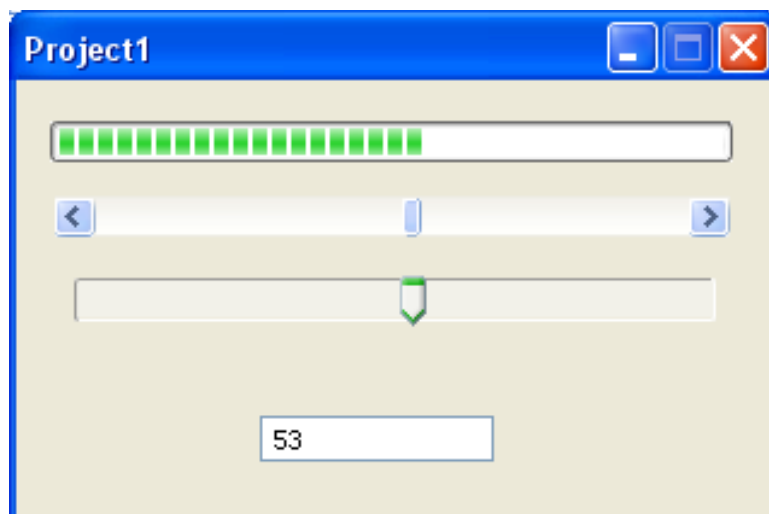


Fig. 5.19. Slider and scrollbar sample program.

Table 5.7a.	Slider and scrollbar sample program event function code.
-------------	---

```

void Project1Dlg::CreateGUIControls()
{
    WxGauge1 = new wxGauge(this, ID_WXGAUGE1, 100,
                           wxPoint(14, 17), wxSize(283, 17),
                           wxGA_HORIZONTAL,
                           wxDefaultValidator, _("WxGauge1"));
    WxGauge1 -> SetRange(100);
    WxGauge1 -> SetValue(0);

    WxEdit1 = new wxTextCtrl(this, ID_WXEDIT1, _(""),
                             wxPoint(101, 139), wxSize(97, 19),
                             0,
                             wxDefaultValidator, _("WxEdit1"));

    WxSlider1 = new wxSlider(this, ID_WXSLIDER1, 0, 0, 10,
                             wxPoint(16, 78), wxSize(282, 45),
                             wxSL_HORIZONTAL | wxSL_SELRANGE ,
                             wxDefaultValidator, _("WxSlider1"));
    WxSlider1 -> SetRange(0,10);
    WxSlider1 -> SetValue(0);

    WxScrollBar1 = new wxScrollBar(this, ID_WXSCROLLBAR1,
                                    wxPoint(16, 48), wxSize(280, 17),
                                    wxSB_HORIZONTAL,
                                    wxDefaultValidator, _("WxScrollBar1"));

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,212);
    Center();

    //// GUI Items Creation End
    WxScrollBar1 -> SetScrollbar(0, 1, 100, 10, true);
    WxSlider1 -> SetRange(0,100);
    WxSlider1 -> SetValue(0);
}

```

Colour mixing program. Let us modify previously created colour mixing program using sliders (see Fig. 5.20). We need three sliders with minimum value 0 and maximum value 255 each. Slider1 scroll event function code is presented in Table 5.8. The same code should be copied to WxSlider2 and WxSlider3 scroll event functions.

Table 5.7b.

Slider and scrollbar sample program
event function code.

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::Project1DlgActivate(wxActivateEvent& event)
{
    // insert
    // your
    // code
    // here
}
void Project1Dlg::WxScrollBar1Scroll(wxScrollEvent& event)
{
    WxSlider1 -> SetValue (WxScrollBar1 -> GetThumbPosition());
    WxGauge1  -> SetValue (WxScrollBar1 -> GetThumbPosition());
    wxString ss;
    ss << WxScrollBar1 -> GetThumbPosition();
    WxEdit1 -> SetValue(ss);
}
void Project1Dlg::WxSlider1Scroll(wxScrollEvent& event)
{
    WxScrollBar1 -> SetThumbPosition (WxSlider1 -> GetValue());
    WxGauge1      -> SetValue          (WxSlider1 -> GetValue());
    wxString ss;
    ss << WxSlider1 -> GetValue();
    WxEdit1 -> SetValue(ss);
}

```

Table 5.8.

Colour mixing program
scroll event function code.

```

void Project1Dlg::WxSlider1Scroll(wxScrollEvent& event)
{
    WxStaticText4 -> SetBackgroundColour (wxColour (WxSlider1 -> GetValue(),
                                                    WxSlider2 -> GetValue(),
                                                    WxSlider3 -> GetValue()));
    WxStaticText4 -> Refresh();
}

```

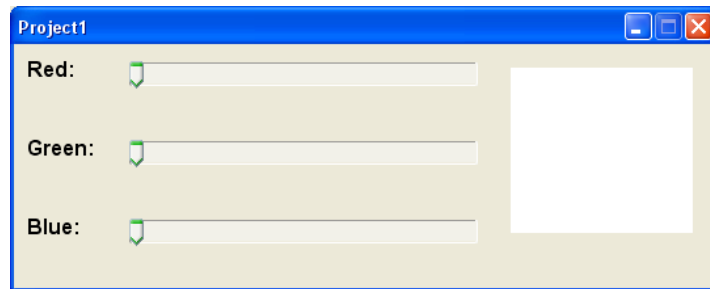


Fig. 5.20. Colour mixing program interface.

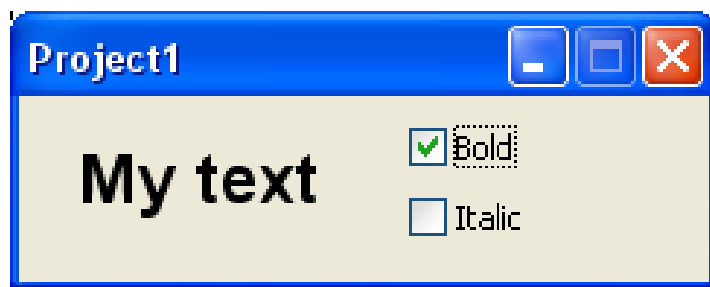


Fig. 5.21. Font style changing program interface.

5.7 Selection controls

It is convenient to use check boxes (`wxCheckBox`) or radio buttons (`wxRadioButton`) while working with binary data. If there are several check boxes in the form, they work independently from each other. If there are several radio buttons they compound a group, where only one radio button from the group can be checked. If we want to have several groups of radio buttons, we should use radio box control (`wxRadioBox`).

We can set check box or radio button value by using `SetValue()` method. The argument of this method is boolean constant (`TRUE` or `FALSE`). In order to get the value of check box or radio button we use `GetValue()` method. We get a boolean value which indicates, whether the component is checked or not.

For example, let us use check boxes to change the font stile (bold and italic). The program interface is shown in Fig. 5.21. `CheckBox1` click event function code is presented in Table 5.9. The same code should be copied into `CheckBox2` click event function.

We can use list box (`wxListBox`) or combo box (`wxComboBox`), if we want to select one or several options from a fixed list. Combo box enables us not only to select a value from a list, but also to enter a new value.

It is possible to fill in a list box using `Strings` property during the program design. We can also edit the content of a list box during the program execution. We can use `Insert()` or `Append()` methods to add an item into a list box or a combo box and `Delete()` method to remove an item. Method `Clear()` removes all items from the control. `GetSelections()` method allows to determine, which items are selected in a list box. `GetValue()` method is used to determine combo box value. `GetCount()` method returns the number of items in the control.

Table 5.9. Font style changing program code.

```

void Project1Dlg::WxCheckBox1Click(wxCommandEvent& event)
{
    if      ((WxCheckBox1 -> GetValue()) && (WxCheckBox2 -> GetValue()))
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxITALIC, wxBOLD,
                                         false));

    else if (WxCheckBox1 -> GetValue())
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxNORMAL, wxBOLD,
                                         false));

    else if (WxCheckBox2 -> GetValue())
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxITALIC, wxNORMAL,
                                         false));

    else
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxNORMAL, wxNORMAL,
                                         false));
}

```

For example, let us create a program with list box. It should be possible to add new items into a list box, to remove selected ones (Fig. 5.22). Function code is presented in Table 5.10.

5.8 Keyboard events

Keyboard is a common and very convenient input device. **WxWidgets** programs are able to respond to keyboard events. There are three keyboard events, that are identified by **WxWidgets** program: **KeyDown**, **KeyUp** and **Char**.

The difference between the first two is clear – the first corresponds to a key press and the second to a key release – otherwise they are identical. Just note, that if the key is maintained in a pressed state you will typically get a lot of (automatically generated) down events but only one up. So, it is wrong to assume, that there is one up event corresponding to each down one. **Char** event does not distinguish, whether the key is up or down. It reacts to key press and carries the translated key code.

Both key events provide untranslated key codes, while the **char** event carries the translated one. The untranslated code for alphanumeric keys is always an upper case value. For the other keys it is one of **WXK_XXX** values from the keycodes table. The translated key is, in general, the character, the user expects to appear as the result of the key combination when typing the text into a text entry zone, for example.

A few examples to clarify this (all assume that **CAPS LOCK** is unpressed and the standard US keyboard is used).

When the 'A' key is pressed, the key down event key code is equal to ASCII **A == 65**. But the **char** event key code is ASCII **a == 97**. On the other hand, if you press both *SHIFT* and 'A' keys simultaneously, the key code in key down event will still be just 'A' while the **char** event key code parameter will now be 'A' as well.

Table 5.10.

List box sample program event function code.

```

void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    if (WxEdit1 -> GetValue() != "")
    {
        int count;
        wxString ss;
        count = wxAtoi(WxEdit2 -> GetValue());
        WxListBox1 -> Append(WxEdit1 -> GetValue());
        count++;
        ss << count;
        WxEdit2 -> SetValue(ss);
        WxEdit1 -> SetValue("");
    }
}

void Project1Dlg::WxListBox1Selected(wxCommandEvent& event)
{
    WxEdit3 -> SetValue (WxListBox1 ->
                        GetString (WxListBox1 -> GetSelection()));
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    if (WxListBox1 -> GetSelection() != wxNOT_FOUND)
    {
        WxListBox1 -> Delete(WxListBox1 -> GetSelection());
        wxString ss;
        ss << WxListBox1 -> GetCount();
        WxEdit2 -> SetValue(ss);
        WxEdit3 -> SetValue("");
    }
}

```

Although in this simple case it is clear that the correct key code could be found in the key down event handler by checking the value returned by `ShiftDown`, in general, you should use `EVT_CHAR` for these as for non alphanumeric keys or non-US keyboard layouts the translation is keyboard-layout dependent and can only be done properly by the system itself.

Another kind of translation is done, when the control key is pressed.

For example, for *CTRL-A* key press the key down event still carries the same key code 'A', as usual, but the char event will have key code of 1, the ASCII value of this key combination.

Key codes are presented in Table 5.11.

Summary of most common methods is presented below.

<code>AltDown()</code> -	returns <code>TRUE</code> if the <i>Alt</i> key was down at the time of the event.
<code>ControlDown()</code> -	returns <code>TRUE</code> if the <i>Control</i> key was down at the time of the event.
<code>GetKeyCode()</code> -	returns the virtual key code.

Table 5.11.	Key codes.	
WXK_NUMPAD_SPACE	WXK_BACK	WXK_WINDOWS_LEFT
WXK_NUMPAD_TAB	WXK_EXECUTE	WXK_WINDOWS_RIGHT
WXK_NUMPAD_ENTER	WXK_SNAPSHOT	WXK_WINDOWS_MENU
WXK_NUMPAD_HOME	WXK_TAB	WXK_RBUTTON
WXK_NUMPAD_END	WXK_RETURN	WXK_LBUTTON
WXK_NUMPAD_PAGEUP	WXK_INSERT	WXK_MBUTTON
WXK_NUMPAD_PAGEDOWN	WXK_CLEAR	WXK_PAUSE
WXK_NUMPAD_NEXT	WXK_CONTROL	WXK_CAPITAL
WXK_NUMPAD_PRIOR	WXK_ALT	WXK_MENU
WXK_NUMPAD_BEGIN	WXK_ESCAPE	WXK_HELP
WXK_NUMPAD_RIGHT	WXK_SHIFT	WXK_CANCEL
WXK_NUMPAD_LEFT	WXK_MULTIPLY	WXK_SEPARATOR
WXK_NUMPAD_UP	WXK_SUBTRACT	WXK_PRIOR
WXK_NUMPAD_DOWN	WXK_ADD	WXK_NEXT
WXK_NUMPAD_INSERT	WXK_DIVIDE	WXK_END
WXK_NUMPAD_DELETE	WXK_NUMLOCK	WXK_HOME
WXK_NUMPAD_EQUAL	WXK_DECIMAL	WXK_LEFT
WXK_NUMPAD_MULTIPLY	WXK_SCROLL	WXK_RIGHT
WXK_NUMPAD_DIVIDE	WXK_PAGEUP	WXK_DOWN
WXK_NUMPAD_ADD	WXK_PAGEDOWN	WXK_UP
WXK_NUMPAD_SUBTRACT	WXK_SELECT	WXK_SPACE
WXK_NUMPAD_SEPARATOR	WXK_DELETE	WXK_PRINT
WXK_NUMPAD_DECIMAL		
WXK_NUMPAD_F1 ...	WXK_NUMPAD_F4	
WXK_SPECIAL1 ...	WXK_SPECIAL20	
WXK_NUMPAD0 ...	WXK_NUMPAD9	

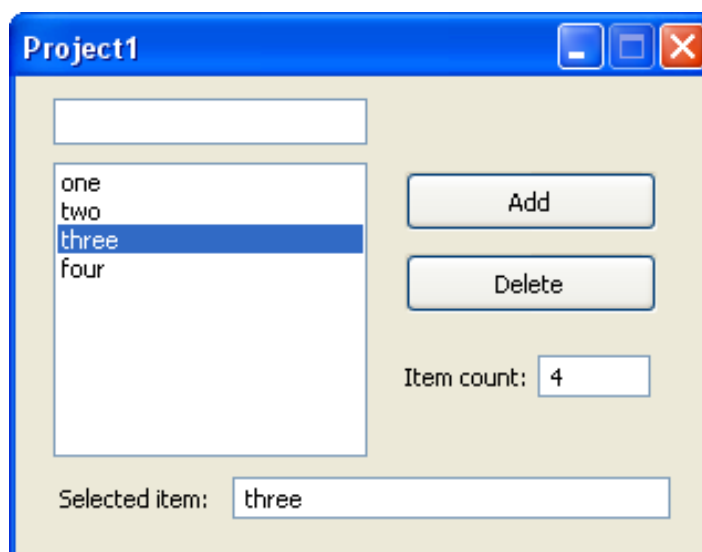


Fig. 5.22. List box sample program interface.

Table 5.12. *KeyDown* event function code.

```

void Project1Dlg::Project1DlgKeyDown(wxKeyEvent& event)
{
    wxString ss;
    ss << " KeyCode=" << event.GetKeyCode()
        << " Alt=" << event.AltDown();
    ss << " Control=" << event.ControlDown()
        << " Shift=" << event.ShiftDown();
    ss << " Key character=" << (char)event.GetKeyCode();
    WxStaticText1 -> SetLabel(ss);
}

```

HasModifiers() - returns TRUE if either *CTRL* or *ALT* keys was down at the time of the key event.

ShiftDown() - returns TRUE if the *Shift* key was down at the time of the event.

Let us create a simple program demonstrating keyboard event handling (see Fig. 5.23). We must set *General Style* - **wxWANTS_CHARS** property value TRUE to get all key codes. The event function code is presented in Table 5.12.

5.9 Mouse events

WxWidgets program can react to different mouse events: **LeftDown**, **LeftUp**, **LeftDclick** (left mouse button), **MiddleDown**, **MiddleUp**, **MiddleDclick** (middle mouse button), **RightDown**, **RightUp**, **RightDclick** (right mouse button), **MouseMotion**, **EnterWindow**, **LeaveWindow**, **MouseWheel** and **MouseEvents** . The last one processes all the mouse events.

Handling mouse events we will need some additional methods.

AltDown() - returns TRUE if the *Alt* key was down at the time of the event.

Button(button) - returns TRUE if the identified mouse button is changing state.

Valid values of **button** are presented below:

wxMOUSE_BTN_LEFT -	check if left button was pressed;
wxMOUSE_BTN_MIDDLE -	check if middle button was pressed;
wxMOUSE_BTN_RIGHT -	check if right button was pressed;
wxMOUSE_BTN_ANY -	check if any button was pressed.

ButtonDClick(button) - returns TRUE if the event was a mouse double click event. The argument specifies, which double click event was generated.

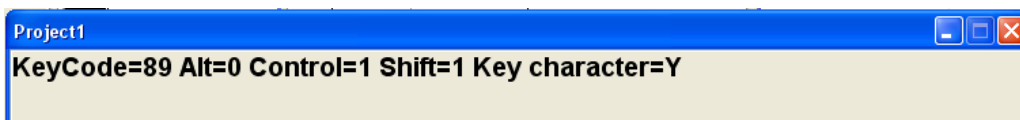


Fig. 5.23. Keyboard events handling program.

`ButtonDown(button)` - returns `TRUE` if the event was a mouse button down event. The argument specifies, which button-down event was generated.

`ButtonUp(button)` - returns `TRUE` if the event was a mouse button up event. The argument specifies, which button-up event was generated.

`ControlDown()` - returns `TRUE` if the control key was down at the time of the event.

`Dragging()` - returns `TRUE` if this was a dragging event (motion, while a button is depressed).

`Entering()` - returns `TRUE` if the mouse was entering the window.

`GetButton()` - returns the mouse button, which generated this event or `wxMOUSE_BTN_NONE` if no button is involved. For mouse move, enter or leave event, for example. Otherwise, `wxMOUSE_BTN_LEFT` is returned for the left button down, up and double click events, `wxMOUSE_BTN_MIDDLE` and `wxMOUSE_BTN_RIGHT` for the same events for the middle and the right buttons respectively.

`GetPosition(long* x, long* y)` - sets `*x` and `*y` to the position, at which the event occurred on the window. Returns the physical mouse position on the window in pixels.

`GetX()` - returns the *X* coordinate of the physical mouse event position on the window.

`GetY()` - returns the *Y* coordinate of the physical mouse event position on the window.

`IsButton()` - returns `TRUE` if the event was a mouse button event (not necessarily a button down event - that may be tested using *ButtonDown*).

`Leaving()` - returns `TRUE` if the mouse was leaving the window.

`LeftDClick()`, `MiddleDClick()`, `RightDClick()` - returns `TRUE` if the event was a left or middle or right double click.

`LeftDown()`, `MiddleDown()`, `RightDown()` - returns `TRUE` if the left or middle or right mouse button changed to down.

`LeftIsDown()`, `MiddleIsDown()`, `RightIsDown()` - returns `TRUE` if the left or middle or right mouse button is currently down, independent of the current event type.

`LeftUp()`, `MiddleUp()`, `RightUp()` - returns `TRUE` if the left or middle or right mouse button changed to up.

`Moving()` - returns `TRUE` if this was a motion event and no mouse buttons were pressed. If any mouse button is held pressed, this method returns `FALSE` and `Dragging()` returns `TRUE`.

`ShiftDown()` - returns `TRUE` if the shift key was down at the time of the event.

Let us create a sample mouse event illustrating program. The position of the green rectangle in the program form will be set using mouse click (see Fig. 5.24). Event function code is presented in Table 5.13.

Now we will improve previous program. The green rectangle will go after the mouse pointer (see Table 5.14).

Table 5.13.	Mouse event function code.
-------------	----------------------------

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    int x, y;
    event.GetPosition(&x, &y);
    WxStaticText1 -> Move(x,y);
}
```

Table 5.14.

MouseMotion event function code.

```
void Project1Dlg::Project1DlgMouseMotion(wxMouseEvent& event)
{
    // variable indicating which part of the distance
    // the rectangle will cover
    float ko = 0.05;
    int x1, y1, x2, y2, dx, dy;

    // the position of static text control
    // (our rectangle)
    WxStaticText1 ->GetPosition(&x1, &y1);

    // the position of mouse pointer
    event.GetPosition(&x2, &y2);
    dx = x2 - x1;
    dy = y2 - y1;
    WxStaticText1 -> Move(x1+(int)(dx*ko),y1+(int)(dy*ko));
                                // new rectangle position
}
```

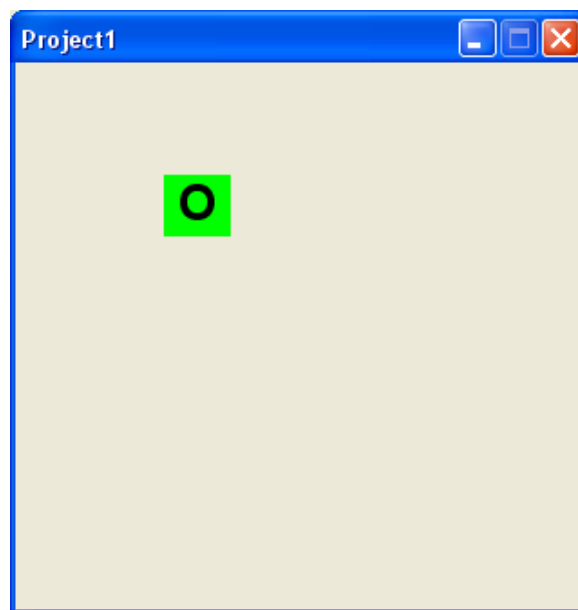


Fig. 5.24. Sample mouse event illustrating program interface.

5.10 Graphical controls and methods

It is possible to operate with images and graphics in `wxWidgets` programs. We can insert image into `wxWidgets` window using `wxStaticBitmap` control. A static bitmap control displays a bitmap. We can assign the image to show using *Picture* property.

Some useful methods are presented below.

`GetBitmap()` - returns the bitmap currently used in the control.

`SetBitmap(wxBitmap)` - sets the bitmap.

To set static bitmap picture during the program run, we must use `wxBitmap` object and its methods.

`LoadFile(name, wxBitmap type)` - loads a bitmap from a file or resource. Function parameters are as follows:

i) **name** - either a filename or a Windows resource name. The meaning of name is determined by the type parameter.

ii) **type** - one of the following values:

<code>wxBITMAP_TYPE_BMP</code> -	load a Windows bitmap file;
<code>wxBITMAP_TYPE_BMP_RESOURCE</code> -	load a Windows bitmap resource from the executable;
<code>wxBITMAP_TYPE_PICT_RESOURCE</code> -	load a PICT image resource from the executable. <i>Mac</i> OS only;
<code>wxBITMAP_TYPE_GIF</code> -	load a GIF bitmap file;
<code>wxBITMAP_TYPE_XBM</code> -	load an X bitmap file;
<code>wxBITMAP_TYPE_XPM</code> -	load an XPM bitmap file.

In addition, `wxBitmap` can read all formats that `wxImage` can such as following: `wxBITMAP_TYPE_JPEG`, `wxBITMAP_TYPE_PNG`, `wxBITMAP_TYPE_GIF`, `wxBITMAP_TYPE_PCX`, `wxBITMAP_TYPE_PNM`. (Of course you must have `wxImage` handlers loaded.)

`SaveFile(name, wxBitmap type, wxPalette* palette = NULL)` - saves a bitmap in the named file. Function parameters are follows:

i) **name** - a filename. The meaning of name is determined by the type parameter;

ii) **type** - one of the following values:

<code>wxBITMAP_TYPE_BMP</code> -	save a Windows bitmap file;
<code>wxBITMAP_TYPE_GIF</code> -	save a GIF bitmap file;
<code>wxBITMAP_TYPE_XBM</code> -	save an X bitmap file;
<code>wxBITMAP_TYPE_XPM</code> -	save an XPM bitmap file.

In addition, `wxBitmap` can save all formats that `wxImage` can such as following: (`wxBITMAP_TYPE_JPEG`, `wxBITMAP_TYPE_PNG`). (Of course you must have `wxImage` handlers loaded.)

iii) **palette** - an optional palette used for saving the bitmap.

Simple animation. Let us create simple animation program: a butterfly flies from the bottom left corner of the program window to the upper right corner after *Start* button is pressed (see Fig. 5.25). Animation effect occurs due to moving `wxStaticBitmap1` control and changing pictures in this bitmap. Function code is presented in Tables 5.15a, 5.15b.

We can use graphic methods to draw on the program window. A `wxDC` object is a device context onto which graphics and text can be drawn.

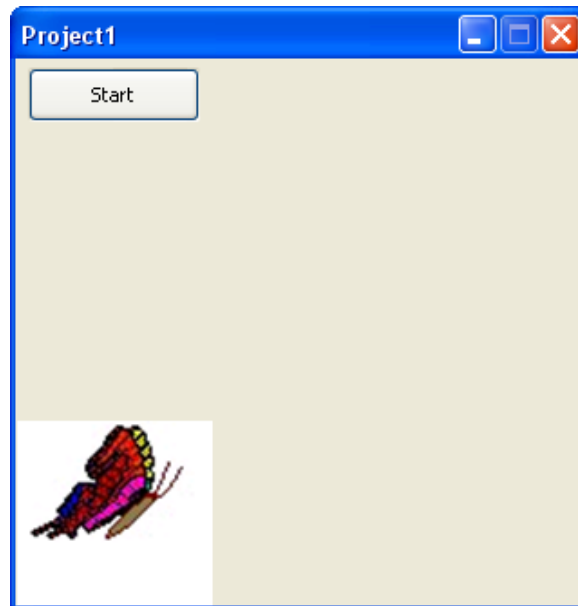


Fig. 5.25. Interface of animation program.

The device context is intended to represent a number of output devices in a generic way, with the same API being used throughout.

Some device contexts are created temporarily in order to draw on a window. This is TRUE of `wxScreenDC`, `wxClientDC`, `wxPaintDC`, and `wxWindowDC`. The following describes the difference between these device contexts and situation when you should use them.

`wxScreenDC` - it is used to paint on the screen, as opposed to an individual window.

`wxClientDC` - it is used to paint on the client area of window (the part without borders and other decorations), but do not use it from within an `wxPaintEvent`.

`wxPaintDC` - it is used to paint on the client area of a window, but only from within a `wxPaintEvent`.

`wxWindowDC` - it is used to paint on the whole area of a window, including decorations. This may not be available on non-Windows platforms. To use a client, paint or window device context, create an object on the stack with the window as argument.

We can draw different shapes and print out text using following methods.

`Clear()` - clears the device context using the current background brush.

`DrawArc(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2, wxCoord xc, wxCoord yc)` - draws an arc of a circle, centred on (xc, yc) , with starting point $(x1, y1)$ and ending at $(x2, y2)$. The current pen is used for the outline and the current brush for filling the shape. The arc is drawn in an anticlockwise direction from the start point to the end point.

`DrawCircle(wxCoord x, wxCoord y, wxCoord radius)` or

`DrawCircle(const wxPoint& pt, wxCoord radius)` - draws a circle with the given centre and radius.

`DrawEllipse(const wxRect& rect)` or

`DrawEllipse(const wxPoint& pt, const wxSize& size)` or

`DrawEllipse(wxCoord x, wxCoord y, wxCoord width, wxCoord height)` - draws an ellipse contained in the rectangle specified either with the given top left corner and the given size or directly. The current pen is used for the outline and the current brush for filling the shape.

Table 5.15a.

Animation program code.

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    ////GUI Items Creation Start

    wxInitAllImageHandlers();    //Initialize graphic format handlers

    WxButton1 = new wxButton(this, ID_WXBUTTON1, _("Start"), wxPoint(7, 5),
                             wxSize(95, 30), 0, wxDefaultValidator, _("WxButton1"));

    WxTimer1 = new wxTimer();
    WxTimer1->SetOwner(this, ID_WXTIMER1);

    WxStaticBitmap1 = new wxStaticBitmap(this, ID_WXSTATICBITMAP1,
                                         wxNullBitmap, wxPoint(1, 188), wxSize(111, 111) );

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End
    WxStaticBitmap1 -> Move(1, 200);
    wxBitmap pic;
    pic.LoadFile("Butterfly1.bmp", wxBITMAP_TYPE_BMP);
    WxStaticBitmap1 -> SetBitmap(pic);
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

```

`DrawEllipticArc(wxCoord x, wxCoord y, wxCoord width, wxCoord height, double start, double end)` - draws an arc of an ellipse. The current pen is used for drawing the arc and the current brush is used for drawing the pie. `x` and `y` specify the coordinates of the upper-left corner of the rectangle that contains the ellipse. `width` and `height` specify the width and height of the rectangle that contains the ellipse, `start` and `end` specify the start and end of the arc relative to the three-o'clock position from the center of the rectangle. Angles are specified in degrees (360 is a complete circle). Positive values mean counter-clockwise motion. If `start` is equal to `end`, a complete ellipse will be drawn.

`DrawLine(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2)` - draws a line from the first point to the second. The current pen is used for drawing the line.

Table 5.15b.

Animation program code.

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    static int i=1, j=1;    int x, y;
    WxStaticBitmap1 -> GetPosition(&x, &y);
    if (i==1)
    {
        wxBitmap pic;
        pic.LoadFile("Butterfly1.bmp", wxBITMAP_TYPE_BMP);
        WxStaticBitmap1 -> SetBitmap(pic);
        i=2; j++;
    }
    else
    {
        wxBitmap pic;
        pic.LoadFile("Butterfly2.bmp", wxBITMAP_TYPE_BMP);
        WxStaticBitmap1 -> SetBitmap(pic);
        i=1;
    };
    WxStaticBitmap1 -> Move(x+5, y-5);
    if (j==20) WxTimer1 -> Stop();
}

void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxStaticBitmap1 -> Move(1, 200);
    WxTimer1         -> Start(100);
}

```

Note that the point (x2, y2) is not part of the line and is not drawn by this function (this is consistent with the behaviour of many other toolkits).

`DrawLines(int n, wxPoint points[], wxCoord xoffset=0, wxCoord yoffset=0)` or
`DrawLines(wxList *points, wxCoord xoffset=0, wxCoord yoffset=0)` - draws lines using an array of points of size `n`, or list of pointers to points, adding the optional offset coordinate. The current pen is used for drawing the lines. The programmer is responsible for deleting the list of points.

`DrawPolygon(int n, wxPoint points[], wxCoord xoffset=0, wxCoord yoffset=0, int fill_style = wxODDEVEN_RULE)` or

`DrawPolygon(wxList *points, wxCoord xoffset=0, wxCoord yoffset=0, int fill_style = wxODDEVEN_RULE)` - draws a filled polygon using an array of points of size `n`, or list of pointers to points, adding the optional offset coordinate. The last argument specifies the fill rule: `wxODDEVEN_RULE` (the default) or `wxWINDING_RULE`. The current pen is used for drawing the outline. The current brush is used for filling the shape. The use of a transparent brush suppresses filling. The programmer is responsible for deleting the list of points.

`DrawPolyPolygon(int n, int count[], wxPoint points[], wxCoord xoffset = 0, wxCoord yoffset = 0, int fill_style = wxODDEVEN_RULE)` - draws two or more filled poly-

gons using an array of points, adding the optional offset coordinates. Notice, that for the platforms providing a native implementation of this function (*Windows* and *PostScript*-based `wxDC` currently), this is more efficient than using `DrawPolygon` in a loop. The number of polygons to draw is specified by parameter `n`, the array count of size `n` specifies the number of points in each of the polygons in the points array. The last argument specifies the fill rule: `wxODDEVEN_RULE` (the default) or `wxWINDING_RULE`. The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling. The polygons may be disjoint or overlapping. Each polygon specified in a call to `DrawPolyPolygon` must be closed. Unlike polygons created by the `DrawPolygon` member function, the polygons created by `DrawPolyPolygon` are not closed automatically.

`DrawPoint(wxCoord x, wxCoord y)` - draws a point using the colour of the current pen. Note, that the other properties of the pen, such as width etc. are not used.

`DrawRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height)` - draws a rectangle with the given top left corner, and with the given size. The current pen is used for the outline and the current brush for filling the shape.

`DrawRotatedText(const wxString& text, wxCoord x, wxCoord y, double angle)` - draws the text rotated by angle degrees. Note, that under *Windows9x* only *TrueType* fonts can be drawn by this function. In particular, a font different from `wxNORMAL_FONT` should be used as the latter is not a textitTrueType font. `wxSWISS_FONT` is an example of suitable font.

`DrawRoundedRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height, double radius)` - draws a rectangle with the given top left corner and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current brush for filling the shape. If the radius is positive, the value is assumed to be the radius of the rounded corner. If radius is negative, the absolute value is assumed to be the proportion of the smallest dimension of the rectangle. This means that the corner can be a sensible size relative to the size of the rectangle and avoids the strange effects X produces when the corners are too big for the rectangle.

`DrawText(const wxString& text, wxCoord x, wxCoord y)` - draws a text string at the specified point, using the current text font, the current text foreground and background colours. The coordinates refer to the top-left corner of the rectangle bounding the string.

`FloodFill(wxCoord x, wxCoord y, const wxColour& colour, int style=wxFLOOD_SURFACE)` - flood fills the device context starting from the given point, using the current brush colour and using a style: `wxFLOOD_SURFACE` - the flooding occurs until a colour other than the given colour is encountered. `wxFLOOD_BORDER` - the area to be flooded is bounded by the given colour.

`GetBackground()` - gets the brush used for painting the background.

`GetBackgroundMode()` - returns the current background mode: `wxSOLID` or `wxTRANSPARENT`.

`GetBrush()` - gets the current brush.

`GetFont()` - gets the current font. Notice, that even although each device context object has some default font after creation, this method would return a `wxNullFont` initially and only after calling `wxDC::SetFont` a valid font is returned.

`GetPen()` - gets the current pen.

`GetPixel(wxCoord x, wxCoord y, wxColour *colour)` - gets the colour at the specified location.

`GetTextBackground()` - gets the current text background colour.

`GetTextForeground()` - gets the current text foreground colour.

`GradientFillConcentric(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour)` or

`GradientFillConcentric(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour, const wxPoint& circleCenter)` - fill the area specified by `rect` with a radial gradient, starting from `initialColour` at the centre of the circle and fading to `destColour` on the circle outside. `circleCenter` are the relative coordinates of centre of the circle in the specified `rect`. If not specified, the circle is placed at the centre of `rect`.

`GradientFillLinear(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour, wxDirection nDirection = wxEAST)` - fill the area specified by `rect` with a linear gradient, starting from `initialColour` and eventually fading to `destColour`.

The `nDirection` specifies the direction of the colour change, default is to use `initialColour` on the left part of the rectangle and `destColour` on the right one.

`SetBackground(const wxBrush& brush)` - sets the current background brush for the DC. `SetBackgroundMode(int mode)` - mode may be one of `wxSOLID` and `wxTRANSPARENT`. This setting determines whether text will be drawn with a background colour or not.

`SetBrush(const wxBrush& brush)` - sets the current brush for the DC. `SetFont(const wxFont& font)` - sets the current font for the DC. It must be a valid font, in particular you should not pass `wxNullFont` to this method.

`SetPen(const wxPen& pen)` - sets the current pen for the DC.

`SetTextBackground(const wxColour& colour)` - sets the current text background colour for the DC.

`SetTextForeground(const wxColour& colour)` - sets the current text foreground colour for the DC.

Graphic methods draw directly on program window or other DC. Graphic methods are executed during program run. So, we see the result of these methods only during program execution.

Let us create a simple program drawing a circle in that place, where mouse left button is clicked. Mouse event function code is presented in Table 5.16.

We can copy the code into `MouseMotion` event function. In this case circles will appear continuously while moving mouse cursor on top of the program window (see Fig. 5.26).

It is also possible to draw a line using mouse events (see Table 5.17, Fig. 5.27). We can add circle in to click point (see Table 5.18).

Table 5.16.

Mouse event function code.

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    dc.DrawCircle(event.GetPosition(), 3);
    dc.SetPen(wxNullPen);
}
```

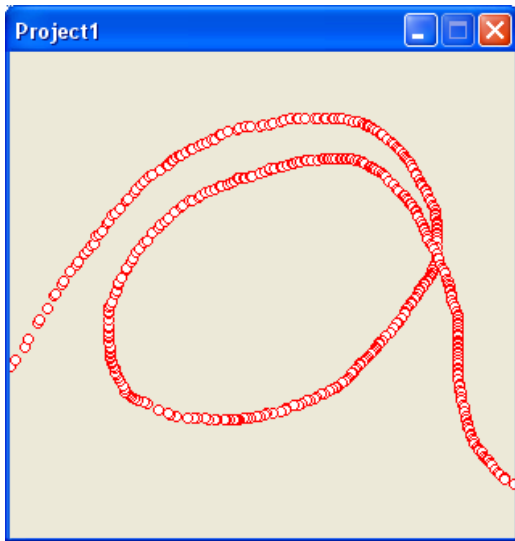


Fig. 5.26. Drawing program.

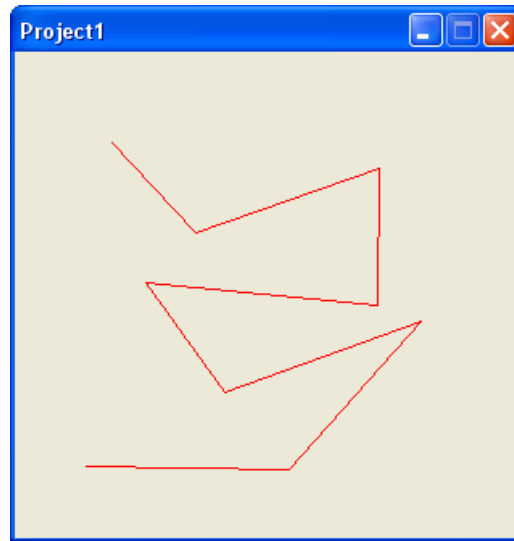


Fig. 5.27. Line drawing program.

Table 5.17. Line drawing code.

```

void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    static int x1=0, y1=0;
    int x2, y2;
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    event.GetPosition(&x2, &y2);
    if ((x1==0)&&(y1==0)) dc.DrawLine(x2, y2, x2, y2);
    else                  dc.DrawLine(x1, y1, x2, y2);
    x1=x2; y1=y2;
    dc.SetPen(wxNullPen);
}

```

5.11 Menu, dialog boxes

It is very convenient to use menus, while implementing a large program. We can not use menu bar component while working with *wxWidgetsDialog* project type. This component will be observed later. It is possible to use popup menu while working with *wxWidgetsDialog* project. We must add *wxPopupMenu* component to the form to use this type of menu. We use *Menu Items* property to fill in the menu. Menu items are managed using special editor (see Fig. 5.28).

Table 5.18.

Line drawing with circles.

```

void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    static int x1=0, y1=0;
    int x2, y2;
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    event.GetPosition(&x2, &y2);
    dc.DrawCircle(x2,y2,3);
    if ((x1==0)&&(y1==0))    dc.DrawLine(x2, y2, x2, y2);
    else                    dc.DrawLine(x1, y1, x2, y2);
    x1=x2; y1=y2;
    dc.SetPen(wxNullPen);
}

```

Menu items field of the editor presents already created items. It is possible to add a new menu item using *Add Item* button. We can set a type of the item (*Menu Item*, *Separator*, *Check Item*, *Radio Item* or *File History*) in the *Type* field. We can also set other parameters (such as *Caption*, *ID Name*, *Enabled*, *Hint* and others) and set event processing functions in the *Events* area. We should press *Apply* to save the created item. It is possible to edit or to delete earlier created items, or to create submenu using this editor (see Fig. 5.28). We can close the editor window pressing *OK* or *Cancel* button. We can show the popup menu using `PopupMenu()` method during the execution of the program.

It is possible to use different dialogues to improve the program behaviour. *WxWidgets* offer wide variety of dialogues: *wxColourDialog*, *wxDirDialog*, *wxFindReplaceDialog*, *wxFontDialog*, *wxMessageDialog*, *wxMultiChoiceDialog*, *wxOpenFileDialog*, *wxPageSetupDialog*, *wxPasswordEntryDialog*, *wxPrintDialog*, *wxProgressDialog*, *wxSaveFileDialog*, *wxSingleChoiceDialog* and *wxTextEntryDialog* (see Fig. 5.29).

Some methods are suitable for all dialogues:

`Centre(int direction = wxBOTH)` - centres the dialog box on the display. Parameter *direction* may be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

`Show(const bool show)` - hides or shows the dialog. If the parameter *show* value is `TRUE`, the dialog box is shown and brought to the front; otherwise the box is hidden. If the parameter *show* value is `FALSE` and the dialog is modal, control is returned to the calling program.

`ShowModal()` - shows a modal dialog. Program flow does not return until the dialog has been dismissed with `EndModal()`. The purpose of dialogues is clear from their names. Let us briefly review most common properties of some dialogues.

`wxMessageDialog` - this class represents a dialog that shows a single or multi-line message, with a choice of *OK*, *Yes*, *No* and *Cancel* buttons. This class supports the following styles.

- `wxOK` - puts an *OK* button in the message box. May be combined with `wxCANCEL`.
- `wxCANCEL` - puts a *Cancel* button in the message box. Must be combined with either `wxOK` or `wxYES_NO`.

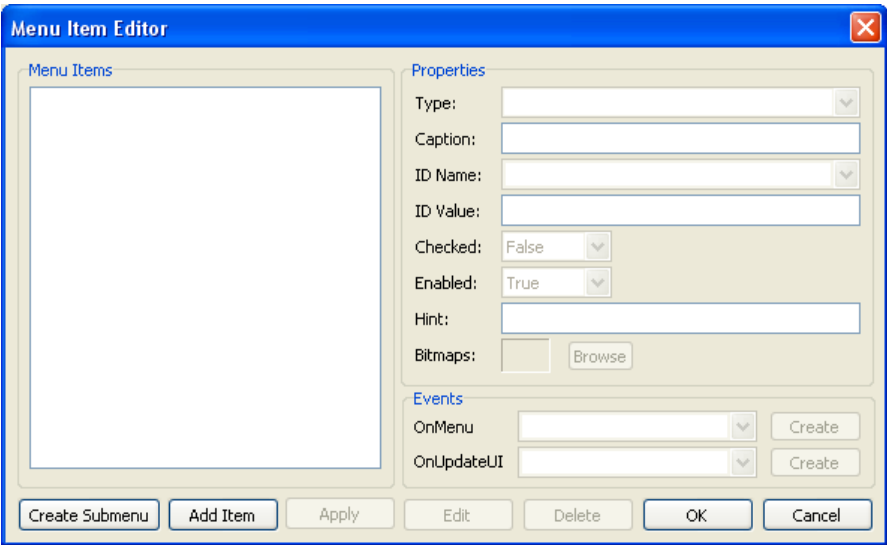


Fig. 5.28. Menu item editor.

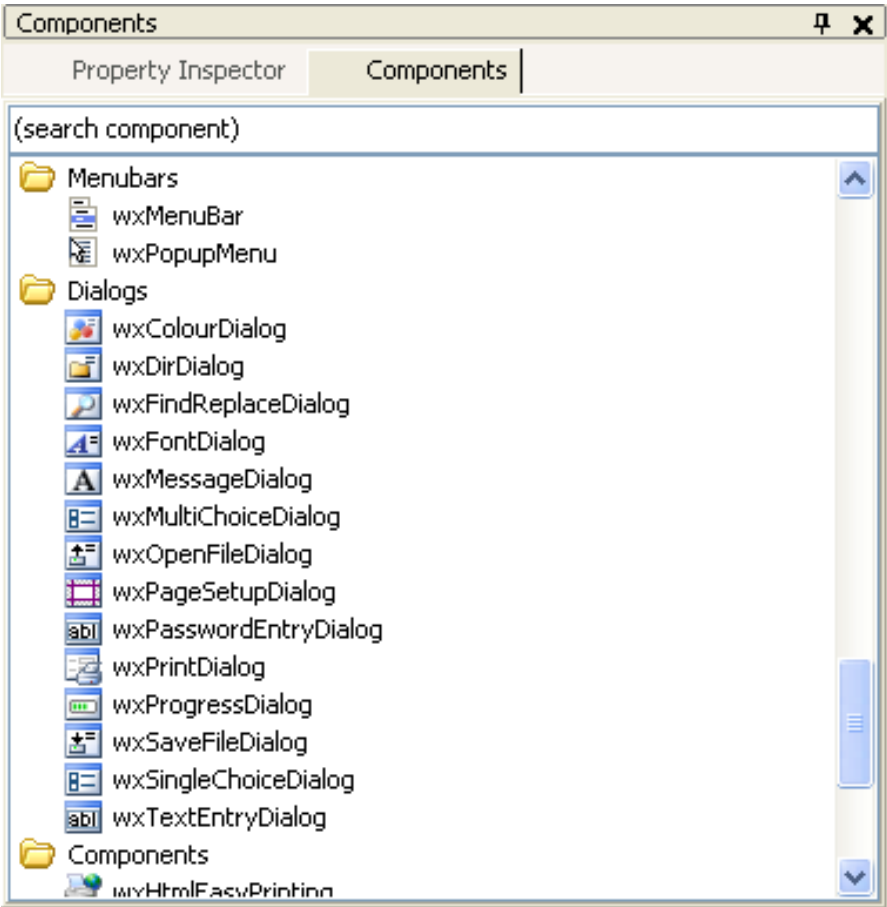


Fig. 5.29. Dialogue components.

<code>wxYES_NO</code> -	puts <i>Yes</i> and <i>No</i> buttons in the message box. It is recommended to use <code>wxCANCEL</code> always with this style. Otherwise the message box won't have a close button under <code>wxMSW</code> and the user will be forced to answer it.
<code>wxHELP</code> -	puts a <i>Help</i> button to the message box. This button can have special appearance or be specially positioned if its label is not changed from the default one.
<code>wxNO_DEFAULT</code> -	makes the <i>No</i> button default, can only be used with <code>wxYES_NO</code> .
<code>wxCANCEL_DEFAULT</code> -	makes the <i>Cancel</i> button default, can only be used with <code>wxCANCEL</code> .
<code>wxYES_DEFAULT</code> -	makes the <i>Yes</i> button default, this is the default behaviour and this flag exists solely for symmetry with <code>wxNO_DEFAULT</code> .
<code>wxOK_DEFAULT</code> -	makes the <i>OK</i> button default, this is the default behaviour and this flag exists solely for symmetry with <code>wxCANCEL_DEFAULT</code> .
<code>wxICON_NONE</code> -	displays no icon in the dialog if possible (an icon might still be displayed if the current platform mandates its use). This style may be used to prevent the dialog from using the default icon based on <code>wxYES_NO</code> presence as explained in <code>wxICON_QUESTION</code> and <code>wxICON_INFORMATION</code> documentation below.
<code>wxICON_EXCLAMATION</code> -	displays exclamation or warning icon in the dialog.
<code>wxICON_ERROR</code> -	displays an error icon in the dialog.
<code>wxICON_HAND</code> -	displays an error symbol, this is a MSW-inspired synonym for <code>wxICON_ERROR</code> .
<code>wxICON_QUESTION</code> -	displays a question mark symbol. This icon is automatically used with <code>wxYES_NO</code> so it is usually unnecessary to specify it explicitly. This style is not supported for message dialogs under <code>wxMSW</code> when a task dialog is used to implement them (i.e. when running under <i>Windows Vista</i> or later) because <i>Microsoft</i> guidelines indicate that no icon should be used for routine confirmations. If it is specified, no icon will be displayed.
<code>wxICON_INFORMATION</code> -	displays an information symbol. This icon is used by default if <code>wxYES_NO</code> is not given so it is usually unnecessary to specify it explicitly.
<code>wxSTAY_ON_TOP</code> -	makes the message box stay on the top of all other windows and not only just its parent (currently implemented only under MSW and GTK).
<code>wxCENTRE</code> -	centres the message box on its parent or on the screen if parent is not specified. Setting this style under MSW makes no differences as the dialog is always centered on the parent.

Some useful methods are presented below.

`SetExtendedMessage (const wxString &extendedMessage)` - sets the extended message for the dialog. This message is usually an extension of the short message specified in the constructor or set with `SetMessage()`.

`wxTextEntryDialog` - this class represents the dialog that requests one-line text string from the user. `wxPasswordEntryDialog` is very similar to `wxTextEntryDialog`, but the input characters are masked. Some useful methods:

`GetMessage ()` - gets the message from the dialog.

`SetMessage (const wxString &message)` - sets the message shown by the dialog.

`GetValue()` - returns the text that the user has entered if the user has pressed *OK*, or the original value if the user has pressed *Cancel*.

`SetValue(const wxString& value)` - sets the default text value.

`wxSingleChoiceDialog` and `wxMultiChoiceDialog` - enable to use dialogues with radio buttons and to check boxes respectively. We can get the selected items using `GetSelections()` and `GetSelection()` methods.

`wxColourDialog` - represents the colour chooser dialog. We can get the colour value using `GetColourData()` method.

`wxDirDialog` - represents the directory chooser dialog. Some useful methods:

`GetPath()` - returns the default or user-selected path.

`GetMessage()` - returns the message that will be displayed on the dialog.

`SetMessage(const wxString& message)` - sets the message that will be displayed on the dialog.

`SetPath(const wxString& path)` - sets the default path.

`wxFontDialog` - represents the font chooser dialog. We can get the font properties using `GetFontData()` method.

`wxOpenFileDialog` and `wxSaveFileDialog` - enables to choose the file. Some useful methods:

`GetDirectory()` - returns the default directory.

`GetFilename()` - returns the default filename.

`GetFilenames (wxArrayString &filenames)` - fills the array filenames with the names of the files chosen.

`GetMessage ()` - returns the message that will be displayed on the dialog.

`GetPath ()` - returns the full path (directory and filename) of the selected file.

`GetPaths (wxArrayString &paths)` - fills the array paths with the full paths of the files chosen.

`SetDirectory (const wxString &directory)` - sets the default directory.

`SetFilename (const wxString &setfilename)` - sets the default filename.

`SetMessage (const wxString &message)` - sets the message that will be displayed on the dialog.

`SetPath (const wxString &path)` - sets the path (the combined directory and filename that will be returned when the dialog is dismissed).

`SetWildcard (const wxString &wildCard)` - sets the wildcard, which can contain multiple file types, for example: "BMP files (*.bmp)—*.bmp—GIF files (*.gif)—*.gif".

Let us create a sample program illustrating popup menu and different dialogues. The popup menu will appear after click of left mouse button on the form. The view of the program window during design stage is shown in Fig. 5.30.

Popup menu editor with menu items is shown in Fig. 5.31. Program window is shown in Fig. 5.32. Program code is presented in Table 5.19a, 5.19b, 5.19c, 5.19d, 5.19e, 5.19f.

Menu bar. Menu bar is a very convenient and frequently used control. We can use menu bar control while working with `wxWidgetsFrame` project type.

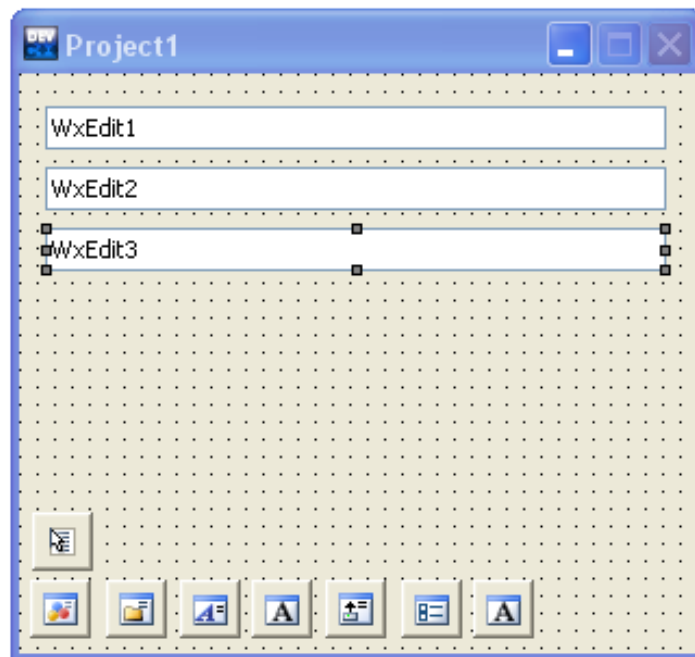


Fig. 5.30. Sample program design.

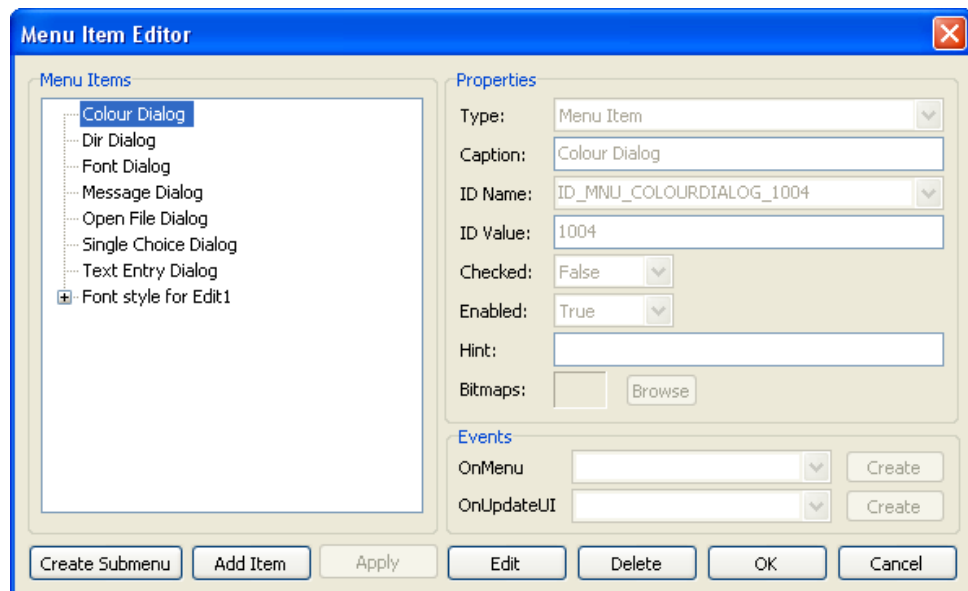


Fig. 5.31. Menu item editor with items.

A menu is a pull down (or popup while using popup menu) list of items, one of which may be selected before the menu goes away (clicking elsewhere dismisses the menu). Menus may be used to construct either menu bars or popup menus.

Table 5.19a.	Program code.
	<pre> /// @file Project1Dlg.cpp ///----- #include "Project1Dlg.h" //----- // Project1Dlg //----- // Add Custom Events only in the appropriate block. // Code added in other places will be removed by wxDev-C++ //// Event Table Start BEGIN_EVENT_TABLE(Project1Dlg,wxDialog) ///Manual Code Start Menu bar///Manual Code End EVT_CLOSE(Project1Dlg::OnClose) EVT_LEFT_DOWN(Project1Dlg::Project1DlgLeftDown) EVT_MENU(ID_MNU_COLOURDIALOG_1004 , Project1Dlg::Mnucolourdialog1004Click) EVT_MENU(ID_MNU_DIRDIALOG_1005 , Project1Dlg::Mnudirdialog1005Click) EVT_MENU(ID_MNU_FONTDIALOG_1006 , Project1Dlg::Mnufontdialog1006Click) EVT_MENU(ID_MNU_MESSAGEDIALOG_1007 , Project1Dlg::Mnumessagedialog1007Click) EVT_MENU(ID_MNU_OPENFILEDIALOG_1008 , Project1Dlg::Mnuopenfiledialog1008Click) EVT_MENU(ID_MNU_SINGLECHOICEDIALOG_1009 , Project1Dlg::Mnusinglechoicedialog1009Click) EVT_MENU(ID_MNU_TEXTENTRYDIALOG_1010 , Project1Dlg::Mnutextentrydialog1010Click) EVT_MENU(ID_MNU_IALIC_1012 , Project1Dlg::Mnuialic1012Click) EVT_MENU(ID_MNU_BOLD_1014 , Project1Dlg::Mnubold1014Click) END_EVENT_TABLE() //// Event Table End </pre>

Table 5.19b.	Program code.
	<pre> void Project1Dlg::CreateGUIControls() { //// GUI Items Creation Start WxTextEntryDialog1 = new wxTextEntryDialog(this, _(""), _("Please enter text"), _("") ,wxOK wxCANCEL wxCENTRE); wxString arrayStringFor_WxSingleChoiceDialog1; arrayStringFor_WxSingleChoiceDialog1.Add(_("Winter")); arrayStringFor_WxSingleChoiceDialog1.Add(_("Spring")); arrayStringFor_WxSingleChoiceDialog1.Add(_("Summer")); arrayStringFor_WxSingleChoiceDialog1.Add(_("Autumn")); WxSingleChoiceDialog1 = new wxSingleChoiceDialog(this, _(""), _("Single Choice List"), arrayStringFor_WxSingleChoiceDialog1, NULL); WxOpenFileDialog1 = new wxFileDialog(this, _("Choose a file"), _(""), _(""), _("*.*)", wxFD_OPEN); WxMessageDialog1 = new wxMessageDialog(this, _(""), _("Message box")); WxFontDialog1 = new wxFontDialog(this); WxDirDialog1 = new wxDirDialog(this, _("Choose a directory"), _("")); WxPopupMenu1 = new wxMenu(_("")); WxPopupMenu1 -> Append (ID_MNU_COLOURDIALOG_1004, _("Colour Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_DIRDIALOG_1005, _("Dir Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_FONTDIALOG_1006, _("Font Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_MESSAGEDIALOG_1007, _("Message Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_OPENFILEDIALOG_1008, _("Open File Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_SINGLECHOICEDIALOG_1009, _("Single Choice Dialog"), _(""), wxITEM_NORMAL); WxPopupMenu1 -> Append (ID_MNU_TEXTENTRYDIALOG_1010, _("Text Entry Dialog"), _(""), wxITEM_NORMAL); // continued to Table 5.19c </pre>

Table 5.19c.	Program code.
	<pre> // continued from Table 5.19b wxMenu *ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj = new wxMenu(); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -> Append (ID_MNU_IALIC_1012, _("Italic"), _(""), wxITEM_CHECK); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -> Check (ID_MNU_IALIC_1012, false); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -> Append (ID_MNU_BOLD_1014, _("Bold"), _(""), wxITEM_CHECK); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -> Check (ID_MNU_BOLD_1014, false); WxPopupMenu1 -> Append(ID_MNU_FONTSTYLEFOREEDIT1_1011, _("Font style for Edit1"), ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj); WxColourDialog1 = new wxColourDialog(this); WxEdit3 = new wxTextCtrl(this, ID_WXEDIT3, _("WxEdit3"), wxPoint(12, 71), wxSize(286, 20),0, wxDefaultValidator, _("WxEdit3")); WxEdit2 = new wxTextCtrl(this, ID_WXEDIT2, _("WxEdit2"), wxPoint(12, 43), wxSize(286, 20),0, wxDefaultValidator, _("WxEdit2")); WxEdit1 = new wxTextCtrl(this, ID_WXEDIT1, _("WxEdit1"), wxPoint(12, 15), wxSize(286, 20),0, wxDefaultValidator, _("WxEdit1")); SetTitle(_("Project1")); SetIcon(wxNullIcon); SetSize(8,8,320,301); Center(); /////GUI Items Creation End } </pre>

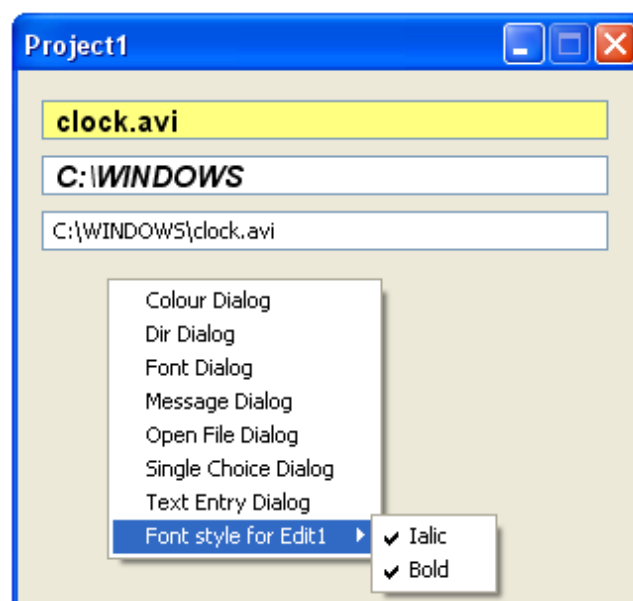


Fig. 5.32. Sample program during the execution stage.

Table 5.19d

Program code.

```

Project1Dlg::Project1Dlg(wxWindow *parent,
                        wxWindowID id,
                        const wxString &title,
                        const wxPoint &position,
                        const wxSize &size,
                        long style)
    : wxDialog( parent, id, title, position, size, style)
{
    CreateGUIControls();
}
Project1Dlg::~Project1Dlg()
{
}
void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::Mnucolourdialog1004Click(wxCommandEvent& event)
{
    WxColourDialog1 -> ShowModal();
    WxEdit1 -> SetBackgroundColour(WxColourDialog1 ->
                                GetColourData().GetColour());
    WxEdit1 -> Refresh();
}
void Project1Dlg::Mnudirdialog1005Click(wxCommandEvent& event)
{
    WxDirDialog1 -> ShowModal();
    WxEdit1 -> SetValue(WxDirDialog1 -> GetPath());
}
void Project1Dlg::Mnufontdialog1006Click(wxCommandEvent& event)
{
    WxFontDialog1 -> ShowModal();
    WxEdit1 -> SetFont(WxFontDialog1 -> GetFontData().GetChosenFont());
}
void Project1Dlg::Mnumessagedialog1007Click(wxCommandEvent& event)
{
    WxMessageDialog1 -> SetMessage("This is a message dialogue.");
    WxMessageDialog1 -> ShowModal();
}
void Project1Dlg::Mnuopenfiledialog1008Click(wxCommandEvent& event)
{
    WxOpenFileDialog1 -> ShowModal();
    WxEdit1 -> SetValue(WxOpenFileDialog1 -> GetFilename());
    WxEdit2 -> SetValue(WxOpenFileDialog1 -> GetDirectory());
    WxEdit3 -> SetValue(WxOpenFileDialog1 -> GetPath());
}

```

Table 5.19e	Program code.
	<pre> void Project1Dlg::Mnusinglechoicedialog1009Click(wxCommandEvent& event) { wxString ss; WxSingleChoiceDialog1->ShowModal(); ss << "It is " << WxSingleChoiceDialog1 -> GetStringSelection() << " now"; WxEdit1 -> SetValue(ss); ss << "The number of selected option is: " << WxSingleChoiceDialog1 -> GetSelection(); WxEdit2 -> SetValue(ss); } void Project1Dlg::Mnutextentrydialog1010Click(wxCommandEvent& event) { wxString ss; WxTextEntryDialog1 -> ShowModal(); ss << "Hi, " << WxTextEntryDialog1 -> GetValue(); WxEdit1 -> SetValue(ss); } void Project1Dlg::Mnuialic1012Click(wxCommandEvent& event) { if ((WxPopupMenu1 -> IsChecked(1012))&&(WxPopupMenu1 -> IsChecked(1014))) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxBOLD, false)); else if (WxPopupMenu1 -> IsChecked(1014)) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxBOLD, false)); else if (WxPopupMenu1 -> IsChecked(1012)) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxNORMAL, false)); else WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxNORMAL, false)); } </pre>

A menu item has an integer ID associated with it, which can be used to identify the selection, or to change the menu item in some way. A menu item with a special identifier -1 is a separator item and doesn't have an associated command but just makes a separator line appear in the menu. Menu items may be either normal items, check items or radio items.

Normal items do not have any special properties, while the check items have a boolean flag associated to them and they show a checkmark in the menu when the flag is set. `wxWidgets` automatically toggles the flag value when the item is clicked and its value may be retrieved using either `IsChecked` method of `wxMenu` or `wxMenuBar` itself or by using `wxEvent::IsChecked` when you get the menu notification for the item in question.

The radio items are similar to the check items except that all the other items in the same radio group are unchecked when one radio item is checked. The radio group is formed by a contiguous range of radio items, i.e. it starts at the first item of this kind and ends with the first item of a different kind (or the end of the menu).

Table 5.19f.	Program code.
	<pre> void Project1Dlg::Mnubold1014Click(wxCommandEvent& event) { if ((WxPopupMenu1 -> IsChecked(1012))&&(WxPopupMenu1 -> IsChecked(1014))) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxBOLD, false)); else if (WxPopupMenu1 -> IsChecked(1014)) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxBOLD, false)); else if (WxPopupMenu1 -> IsChecked(1012)) WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxNORMAL, false)); else WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxNORMAL, false)); } void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event) { PopupMenu(WxPopupMenu1); } </pre>

Notice, that because the radio groups are defined in terms of the item positions inserting or removing the items in the menu containing the radio items risks not to work correctly.

All menus, except the popup ones, must be created on the heap. All menus, attached to a menu bar or to another menu, will be deleted by their parent when it is deleted. As the frame menu bar is deleted by the frame itself, it means, that normally all menus used are deleted automatically.

We can use menu item editor (see Fig. 5.31) to create and manage menu bar items. Some useful menu bar methods:

`wxMenuItem* Append(int id, const wxString& item = "", const wxString& helpString = "", wxItemKind kind = wxITEM_NORMAL)` - adds a string item to the end of the menu.

`wxMenuItem* Append(int id, const wxString& item, wxMenu *subMenu, const wxString& helpString = "")` - adds a pull-right submenu to the end of the menu. Append the submenu to the parent menu after you have added your menu items, or accelerators may not be registered properly.

`wxMenuItem* Append(wxMenuItem* menuItem)` - adds a menu item object. This is the most generic variant of `Append()` method because it may be used for both items (including separators) and submenus. You can also specify various extra properties of a menu item in this way, such as bitmaps and fonts. Parameters are presented below:

<code>id</code> -	the menu command identifier;
<code>item</code> -	the string to appear on the menu item;
<code>menu</code> -	pull-right submenu;
<code>kind</code> -	may be <code>wxITEM_SEPARATOR</code> , <code>wxITEM_NORMAL</code> , <code>wxITEM_CHECK</code> or <code>wxITEM_RADIO</code> ;
<code>helpString</code> -	an optional help string associated with the item. By default, the handler for the <code>wxEVT_MENU_HIGHLIGHT</code> event displays this string in the status line;

menuItem - a menuItem object. It will be owned by the wxMenu object after this function is called, so do not delete it yourself.

This command can be used after the menu has been shown, as well as on initial creation of a menu or menu bar. The item string for the normal menu items (not submenus or separators) may include the accelerator which can be used to activate the menu item from the keyboard. The accelerator string follows the item label and is separated from it by a TAB character ('\t'). Its general syntax is any combination of "CTRL", "ALT" and "SHIFT" strings (case doesn't matter) separated by either '-' or '+' characters and followed by the accelerator itself.

The accelerator may be any alphanumeric character, any function key (from F1 to F12) or one of the special characters listed in the table below (again, case doesn't matter):

DEL or DELETE	<i>Delete key</i>
INS or INSERT	<i>Insert key</i>
ENTER or RETURN	<i>Enter key</i>
PGUP	<i>PageUp key</i>
PGDN	<i>PageDown key</i>
LEFT	<i>Left cursor arrow key</i>
RIGHT	<i>Right cursor arrow key</i>
UP	<i>Up cursor arrow key</i>
DOWN	<i>Down cursor arrow key</i>
HOME	<i>Home key</i>
END	<i>End key</i>
SPACE	<i>Space</i>
TAB	<i>Tab key</i>
ESC or ESCAPE	<i>Escape key (Windows only)</i>

`wxMenuItem* AppendCheckItem(int id, const wxString& item, const wxString& helpString = "")` - adds a checkable item to the end of the menu.

`wxMenuItem* AppendRadioItem(int id, const wxString& item, const wxString& helpString = "")` - adds a radio item to the end of the menu. All consequent radio items form a group and when an item in the group is checked, all the others are automatically unchecked.

`wxMenuItem* AppendSeparator()` - adds a separator to the end of the menu.

`wxMenuItem* AppendSubMenu(wxMenu *submenu, const wxString& text, const wxString& help = wxEmptyString)` - adds the given submenu to this menu. Parameter `text` is the text, shown in the menu for it. Parameter `help` is the help string, shown in the status bar when the submenu item is selected.

`Break()` - inserts a break in a menu, causing the next appended item to appear in a new column.

`Check(int id, const bool check)` - checks or unchecks the menu item. Parameter `id` is the menu item identifier. If parameter `check` value is `TRUE`, the item will be checked, otherwise it will be unchecked.

`Delete(int id)` or `Delete(wxMenuItem *item)` - deletes the menu item from the menu. If the item is a submenu, it will not be deleted. Use `Destroy` if you want to delete a submenu. Parameter `id` is the id of the menu item to be deleted. Parameter `item` is the menu item to be deleted.

`Destroy(int id)` or `Destroy(wxMenuItem *item)` - deletes the menu item from the menu. If the item is a submenu, it will be deleted. Use *Remove* if you want to keep the submenu (for example, to reuse it later). Parameter `id` is the id of the menu item to be deleted. Parameter `item` is the menu item to be deleted.

`Enable(int id, const bool enable)` - enables or disables (greys out) a menu item. Parameter `id` is the menu item identifier. Parameter `enable` value should be `TRUE` to enable the menu item, `FALSE` to disable it.

`wxString GetHelpString(int id)` - returns the help string associated with a menu item. Parameter `id` is the menu item identifier.

`size_t GetMenuItemCount()` - returns the number of items in the menu.

`wxMenuItemList& GetMenuItems()` - returns the list of items in the menu. `wxMenuItemList` is a pseudo-template list class containing `wxMenuItem` pointers, see `wxList`.

`wxMenuItem* Insert(size_t pos, wxMenuItem *item)` or

`wxMenuItem* Insert(size_t pos, int id, const wxString& item = "", const wxString& helpString = "", wxItemKind kind = wxITEM_NORMAL)` - inserts the given item before the position `pos`. Inserting the item at position `GetMenuItemCount` is the same as appending it.

`wxMenuItem* InsertCheckItem(size_t pos, int id, const wxString& item, const wxString& helpString = "")` - inserts a checkable item at the given position.

`wxMenuItem* InsertRadioItem(size_t pos, int id, const wxString& item, const wxString& helpString = "")` - inserts a radio item at the given position.

`wxMenuItem* InsertSeparator(size_t pos)` - inserts a separator at the given position.

`bool IsChecked(int id)` - determines whether a menu item is checked. Parameter `id` is the menu item identifier.

`bool IsEnabled(int id)` - determines whether a menu item is enabled. Parameter `id` is the menu item identifier.

`wxMenuItem * Remove(int id)` or `wxMenuItem * Remove(wxMenuItem *item)` - removes the menu item from the menu but doesn't delete the associated C++ object. This allows to reuse the same item later by adding it back to the menu (especially useful with submenus). Parameter `id` is the identifier of the menu item to remove. Parameter `item` is the menu item to remove.

`void SetHelpString(int id, const wxString& helpString)` - sets an item's help string. Parameter `id` is the menu item identifier.

Parameter `helpString` is the help string to set.

Index

Index

asm, 31

bool, 21

break, 37

Char, 264

char, 20

cin, 24

const, 22

container

- list, 122, 128
- map, 132, 138
- pair, 132, 134, 138
- queue, 143, 146
- set, 132, 135
- stack, 143, 144
- vector, 122, 124

continue, 38

cout, 19, 23

define, 22, 30

delete, 87

do-while, 35

double, 21

endl, 24

exit, 39

float, 20

for, 35

friend, 71

if, 32

int, 20

KeyDown, 264

KeyUp, 264

long, 21

macros, 30

main, 18

menu

- MENUITEM, 212, 230
- POPUP, 212

mod, 196

MouseEvents, 267

namespace, 18

new, 87

precision, 24

primes, 181

printf, 26

resource script, 212, 230

RGB, 222, 248

RNG, 186, 187

scanf, 26

short, 21

sizeof, 88

sort

- bubble-sort, 151
- cocktail-sort, 153
- comb-sort, 156
- heap-sort, 161
- insertion-sort, 174
- merge-sort, 179
- odd-even-sort, 158
- quick-sort, 177
- selection-sort, 166
- shell-sort, 170, 171

static, 49

strcpy, 53

swap, 150

switch, 33

system, 24

template, 150

this, 115

Vernam, 191

virtual, 77

while, 33

width, 24

windows.h

- BeginPaint, 223

- callback, 203, 209, 219

- CreatePen, 222

- CreateWindow, 206, 218

- CreateWindowEx, 202

- DeleteObject, 223

- Ellipse, 225

- EndPaint, 223

- GetStockObject, 205, 218

- header, 211, 230

- InvalidateRect, 224, 230

- LineTo, 225

- LoadAccelerators, 209

- MessageBox, 210

- MoveToEx, 225

- PatBlt, 230

- Pie, 225

- Rectangle, 228

- RoundRect, 228

- SetPixel, 224

- WinMain, 200, 205, 215

wxButton, 242

wxCheckBox, 263

wxComboBox, 263

wxDC, 270

wxDialog, 277, 280

wxEdit, 243

wxFont, 243, 249

wxListBox, 263

wxMenu, 287

wxPopupMenu, 276, 280

wxRadioButton, 263

wxScrollBar, 258

wxSlider, 258

wxStaticBitmap, 270

wxStaticText, 242

wxStopWatch, 254, 256

wxTimer, 252

XOR, 196

xor, 191, 196

Bibliography

- [1] *Dev-C*. http://sourceforge.net/projects/wxdsgn/files/wxDev-C%2B%2B/Version%207/wxdevcpp_7.4.2_full_setup.exe/download, retrieved Aug20, 2012.
- [2] Andrei Alexandrescu. *Modern C++ Desing*. Boston: Addison-Wesley, 2002.
- [3] Herb Sutter; Andrei Alexandrescu. *C++ coding standards. 101 Rules, Guidelines, and best practices*. Boston: Addison-Wesley, 2005.
- [4] Bruce Eckel; Chuck Allison. *Thinking in C++, 2nd Edition, Volume 2*. Prentice Hall, 2003.
- [5] Matt Austern. *Generic programming and the STL*. Addison-Wesley, 1998.
- [6] Jon Bertly. *Programming pearls. Second edition*. Addison Wesley, 2000.
- [7] Harry H. Cheng. *C for Engineers and Scientists. An Interpretive Approach*. NewYork: McGraw-Hill, 2010.
- [8] III Chris H. Pappas; William H. Murray. *Debugging C++. Troubleshooting for programmers*. Osborne: McGraw-Hill, 2000.
- [9] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.
- [10] H. M. Deitel; P. J. Deitel. *How to program C. Second Edition*. Prentice Hall International, Inc, 2000.
- [11] H. Demuth. *Electronic Data Sorting. PhD thesis*. Stanford University, 1956.
- [12] Bruce Eckel. *Thinking in C++, 2nd Edition, Volume 1, (3rd Edition to appear)*. Prentice Hall, 2000.
- [13] Krzysztof Czarnecki; Ulrich Eisencecker. *Generative programming*. Addison-Wesley, 2000.
- [14] Abraham Silberschatz; Peter B. Galvin. *Operating system concepts*. Addison: Wesley Publishing Co, 1994.
- [15] Pablo Halpern. *The C++ Standard library from scratch. // In: Jesse libertys from scratch programming series*. QUE, Indianapolis, 2000.
- [16] Tan Kiat Shi; Willi-Hans Steeb; Yorick Hardy. *Symbolic C++: an introduction to computer algebra using object-oriented programming*. Springer, London, 2000.

- [17] David Vandervoorde; Nicolai M. Josuttis. *C++ Templates, The Complete Guide*. Addison-Wesley, 2002.
- [18] Nicolai Josuttis. *The C++ Standard Library*. Addison-Wesley, 1999.
- [19] Allen B. Tucker; Robert D. Cupper; W. James Bradley; Richard G. Epstein; Charles F. Keleman. Chapter 7. operating systems and software design. In *Fundamentals of computing II. Abstraction, Data Structures and Large Software design. C++ edition*. New York: McGrawHill int., 1995.
- [20] D. E. Knuth. Volume 3: Sorting and searching. In *The Art of Computer programming*. Addison-Wesley, 1968.
- [21] Stanley B. Lippman; Josee Lajoie. *C++ primer. Third edition*. Addison-Wesley, 1998.
- [22] Andre LaMothe. *The Black Art of Video Game Console Design*. Indianapolis: SAMS, 2003.
- [23] Andre LaMothe. *Tricks of the 3D Game Programming Gurus-Advanced 3D Graphics and Rasterization (Other Sams)*. Indianapolis: SAMS, 2005.
- [24] Jesse Liberty. *C++ unleashed*. Indianapolis, SAMS, 1999.
- [25] Jesse Liberty. *Teach Yourself C++ in 21 Days. Third Edition*. Indianapolis, SAMS, 1999.
- [26] Stan Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.
- [27] Stan Lippman. *Essential C++*. Addison-Wesley, 1999.
- [28] Dov Bulka; David Mayhew. *Efficient C++*. Addison-Wesley, 1999.
- [29] R. E. Berry; B. A. E. Meekings. *A book on C*. MACMILLAN, London, 1984.
- [30] Scott Meyers. *More effective C++. 35 new ways to improve your programs and designs*. Addison Wesley, 1996.
- [31] Scott Meyers. *Effective C++, 2nd Edition*. Addison-Wesley, 1997.
- [32] Scott Meyers. *Effective STL*. Addison-Wesley, 2001.
- [33] Andrew Koenig; Barbara E. Moo. *Accelerated C++*. Boston: Addison-Wesley, 2000.
- [34] David R. Musser. *STL Tutorial and Reference Guide, 2nd Edition*. Addison-Wesley, 2001.
- [35] P.J. Plauger. *The C++ Standard Template Library*. Prentice Hall, 2000.
- [36] Stephen Prata. *C++ Primer Plus. Fourth Edition*. Indianapolis: SAMS, year = 2002.
- [37] Steve Qualline. *How not to program in C++*. San Francisco: NO STACH press, 2003.
- [38] Brian W. Kernighan; Dennis M. Ritchie. *The C programming Language (1st ed.)*. Englewood Cliffs, NJ: Prentice Hall., 1978.

- [39] Brian W. Kernighan; Dennis M. Ritchie. *The C programming Language (2nd ed.)*. Prentice Hall, 1998.
- [40] Herbert Schildt. *Schildts expert C++*. Osborne: McGraw-Hill, 1995.
- [41] Herbert Schildt. *Schildts Windows 95. programming in C and C++*. Osborne: McGraw-Hill, 1995.
- [42] Herbert Schildt. *Teach yourself C++*. Osborne: McGraw-Hill, 1998.
- [43] Herbert Schildt. *C++: The Complete Reference. Four Edition*. Osborne: McGraw-Hill, 2003.
- [44] Robert Sedgewick. *Algorithms in C++*. Addison: Wesley, 2009.
- [45] Juan Soulie. *C++ Language tutorial*. <http://www.cplusplus.com>, 2007.
- [46] Bjarne Stroustrup. *The design and evolution of C++*. Addison Wesley, 1994.
- [47] Bjarne Stroustrup. *The C++ programming Language, Special Third Edition*. Addison-Wesley, 2000.
- [48] Herb Sutter. *Exceptional C++*. Addison-Wesley, 2000.
- [49] Herb Sutter. *More Exceptional C++*. Addison-Wesley, 2001.
- [50] William Ford; William Topp. Chapter 4: The vector class. In *Data Structures with C++ and STL, Second Edition*. Prentice Hall, 2002.
- [51] William H. Press; Brian P. Flannery; Saul A. Teukolsky; William T. Vetterling. *Numerical Recipes in C*. Cambridge university press, 1988.
- [52] Al Williams. *Windows 2000. Systems programming. Black book*. Coriolis, 2000.
- [53] Daoqui Yang. *C++ and object-oriented numeric computing for scientist and engineers*. Springer, 2001.

Alytis Gruodis, Sergėjus Ivanikovas

C++ Programming from Basics

Vilnius: Vilniaus verslo kolegija, 2012.

Language editors: Giedrė Būdienė, Jelena Ivanikova

Typesetter - Alytis Gruodis

Cover design & printing - UAB *ART SET*, Vilnius