

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

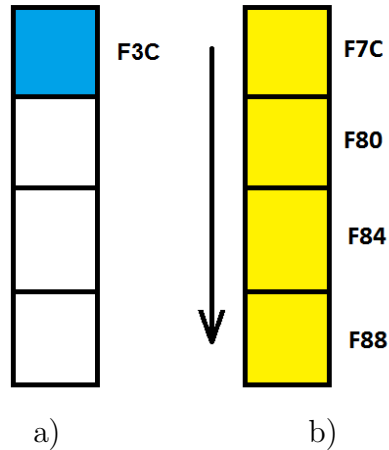


Fig. 3.1. Memory allocation for 1D array:
a) declaration of pointer using `int *iA;`
b) creation of 1D array using `iA=new int[4].`
Arrow visualizes allocation process using operator `new` .

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.
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	Output of 2D array: 0 1 2 ##### 0 # 7 7 8 1 # 8 6 6 Press any key to continue ...

```
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:

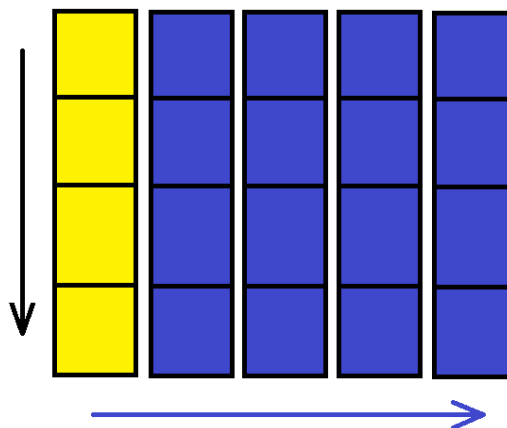


Fig. 3.2. Memory allocation for 2D array.

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.

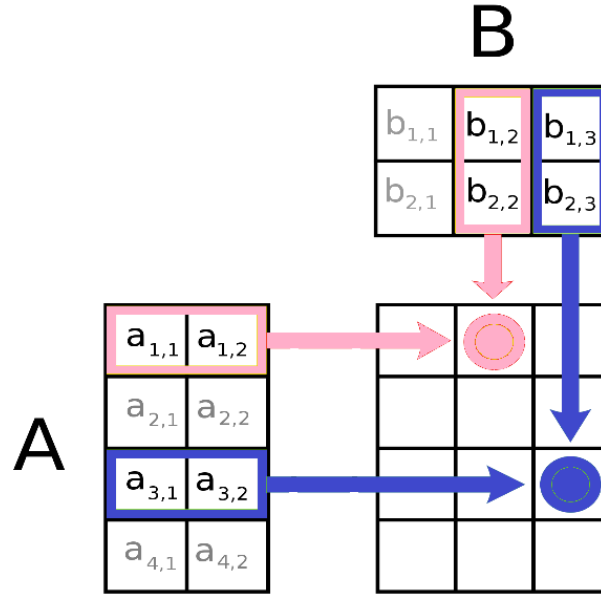


Fig. 3.5. Multiplication scheme.

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 technigue. 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.

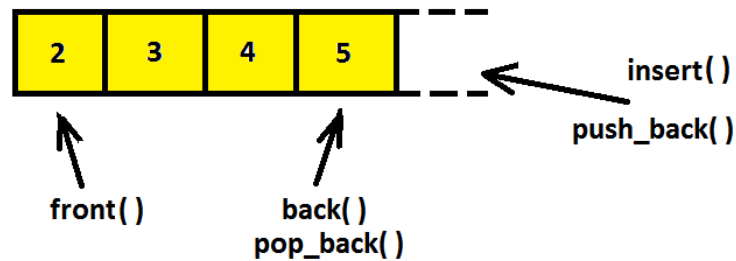


Fig. 3.7. Container vector. Data organization scheme.

```
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.

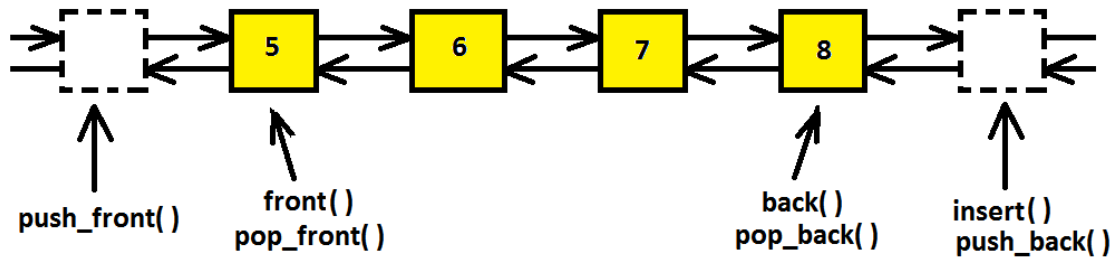


Fig. 3.8. Container `list`. Data organization scheme.

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);

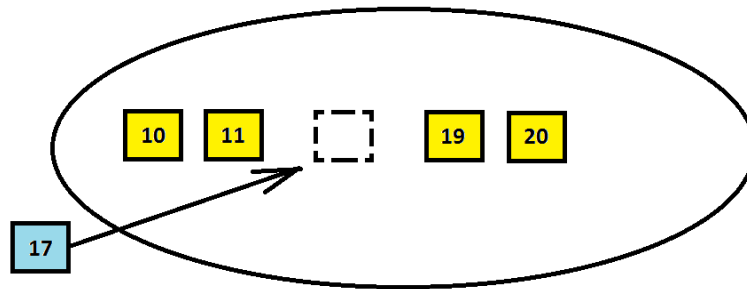
```

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.
<pre> #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; } </pre>	

```

// 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	Charlie B. <== 6290
	Dave Q. <== 6211
7220 ==> Charlie B.	Hank Q. <== 6250
7224 ==> Harry D.	Harry D. <== 6222
7228 ==> David R.	Kent Z. <== 6271
7400 ==> John S.	Larry N. <== 6005
7401 ==> Mike Q.	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!

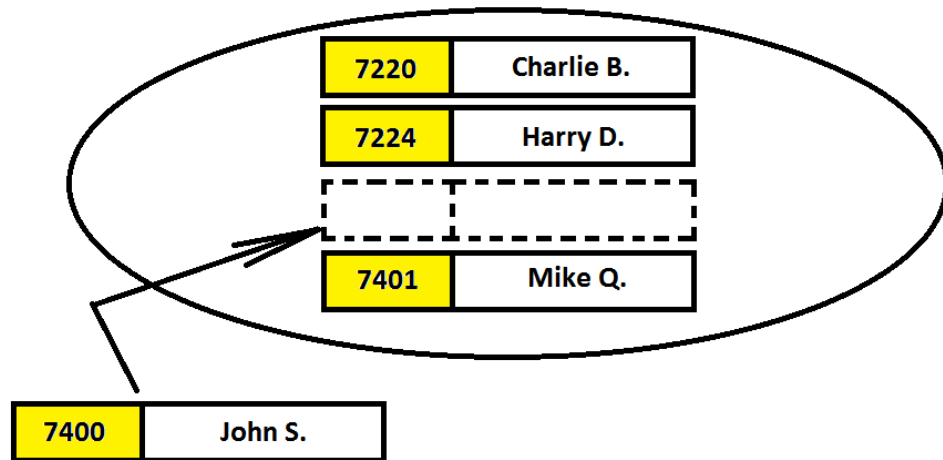


Fig. 3.10. Container map. Data organization scheme.

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.

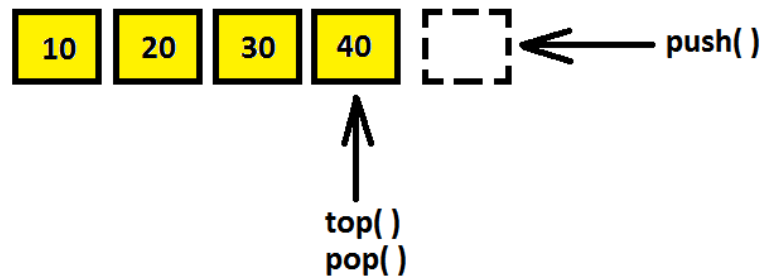


Fig. 3.11. Container `stack`. Data organization scheme.

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.

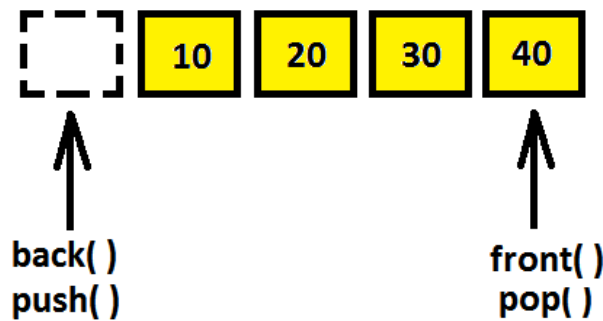


Fig. 3.12. Container queue. Data organization scheme.

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

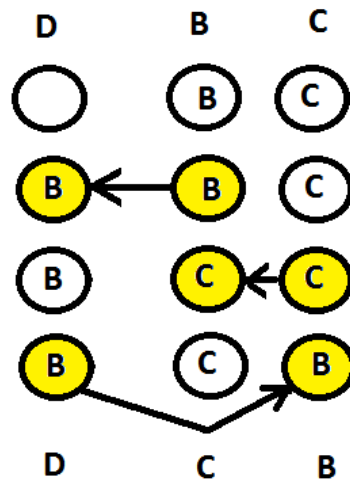


Fig. 3.13. Swapping operation.

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) swaps 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

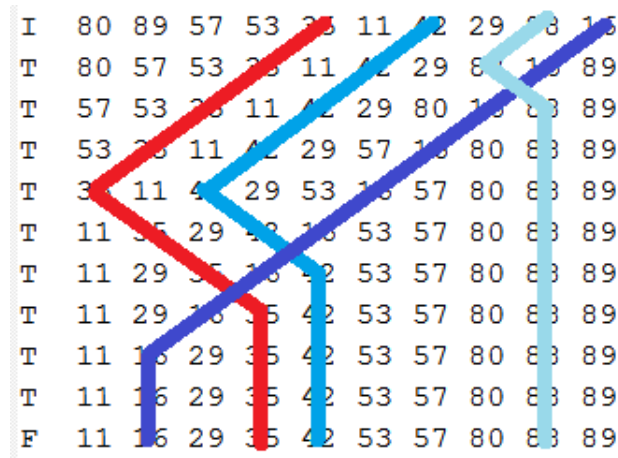


Fig. 3.14a. Visualization of dynamics. Output file of *Burple sort* (Table 3.40).

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.

```

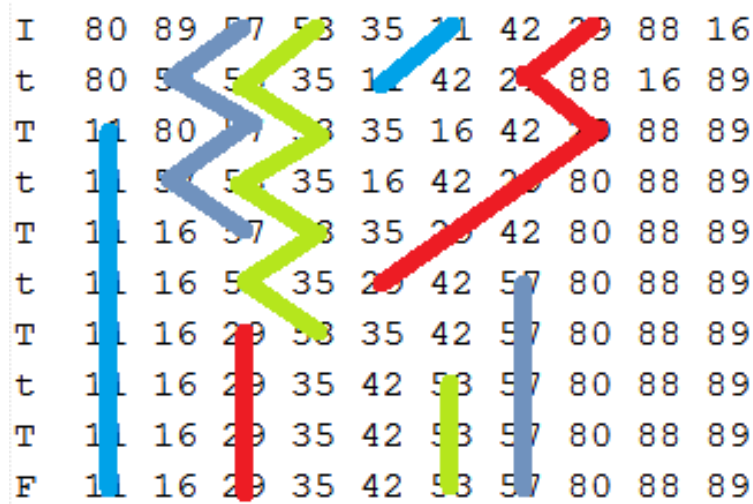
bAction=false;
for (int i=(m-2); i>=0; i--)
{
    // test whether the two elements are in the wrong order
    if (AA[i]>AA[i+1])
    {
        swap(AA[i],AA[i+1]);
        bAction=true;
    };
};

```

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

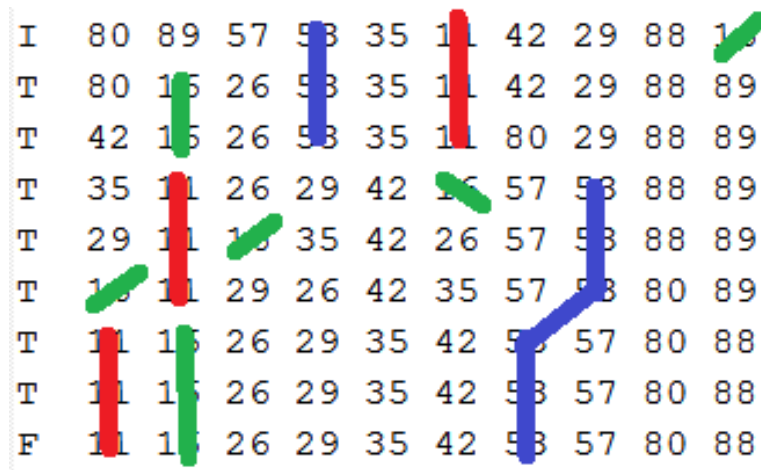


Fig. 3.15. Visualization of dynamics. Output file of *Comb-sort* (Table 3.44).

```

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.

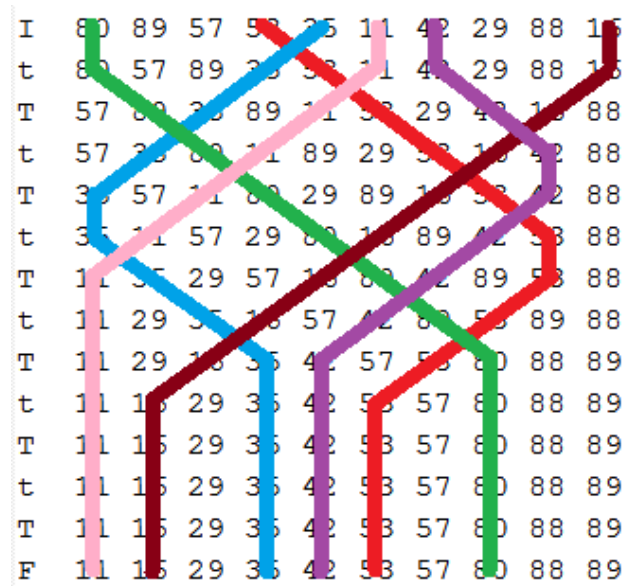
Table 3.45. Odd_even_sort.

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.

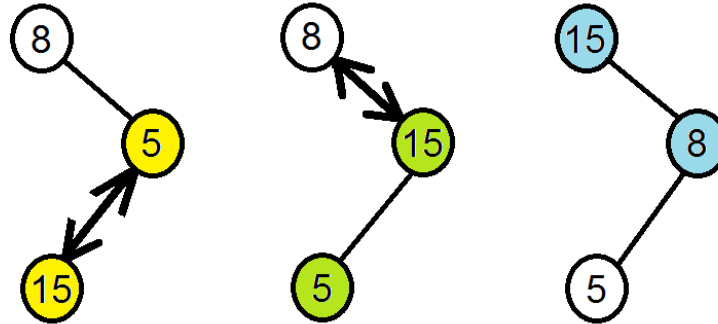


Fig. 3.17. Ordering in heap. Two swap operations.

```

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;

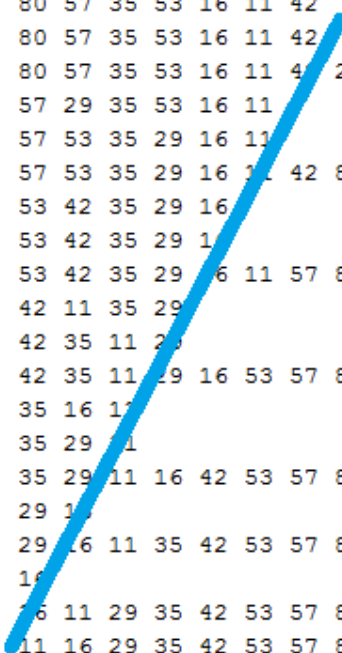
```

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.

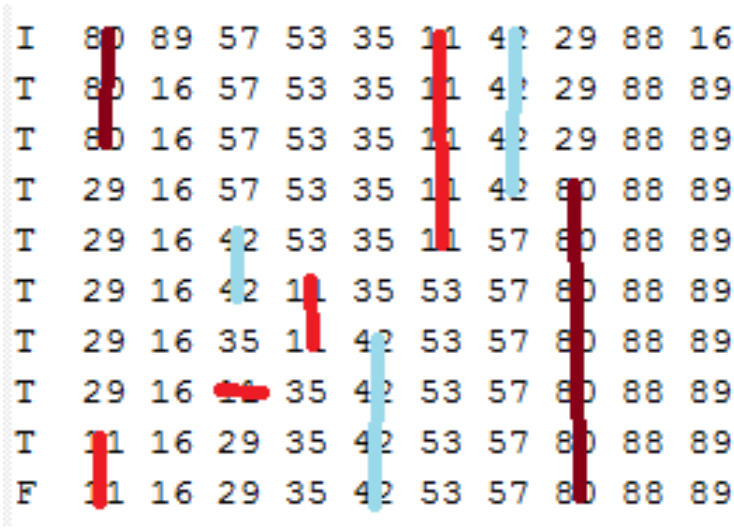


Fig. 3.19. Visualization of dynamics. Output file of *Selection-sort* (as presented in Table 3.50).

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.

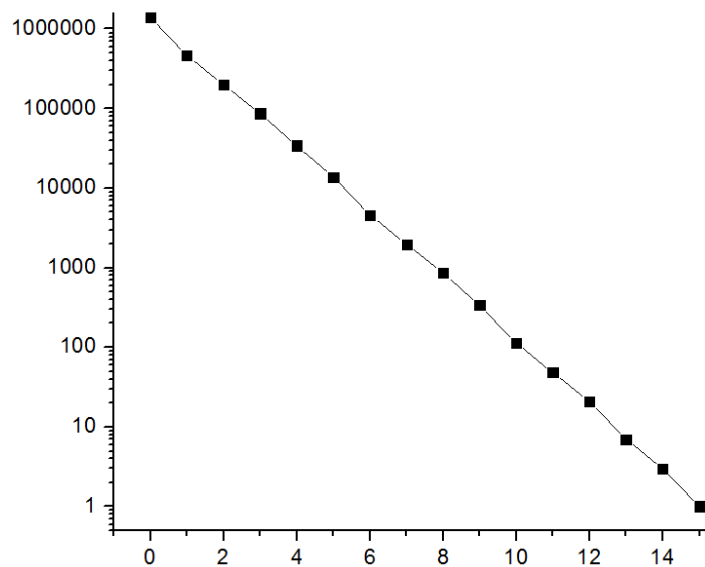


Fig. 3.20. Array of cols in lognormal distribution.

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.

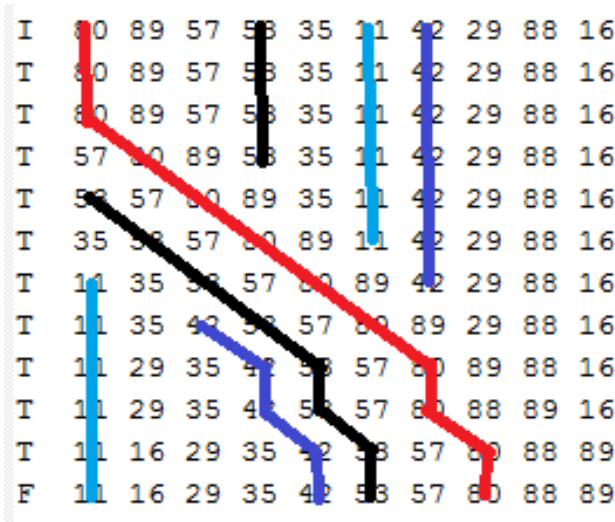


Fig. 3.21. Visualization of dynamics.
Output file of *Selection-sort* (Table 3.58).

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

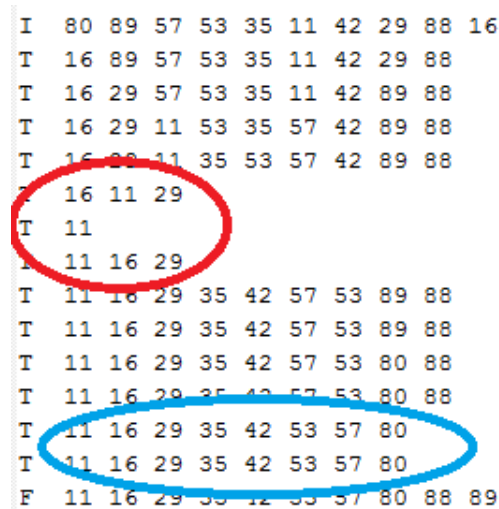


Fig. 3.22. Visualization of dynamics. Output file of *Quick-sort* (Table 3.60).

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).

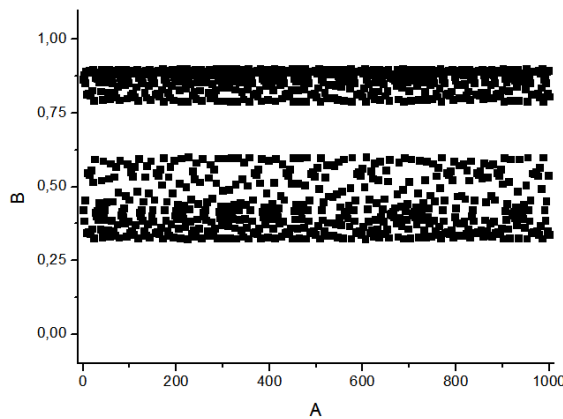
0	0.96	500	0.854761
1	0.1536	501	0.496578
2	0.520028	502	0.999953
3	0.998395	503	0.000187301
4	0.00640774	504	0.000749064
5	0.0254667	505	0.00299401
...		...	
100	0.495893	990	0.340463
101	0.999933	991	0.898191
102	0.000269858	992	0.365774
103	0.00107914	993	0.927934
104	0.0043119	994	0.26749
105	0.0171732	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) $r = 0.40$.

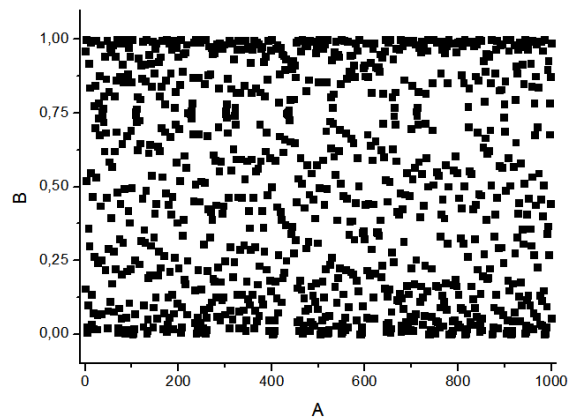
Probable numbers are presented as a variables from loop, continuously increased in interval $(1 \div 100)$. For case a), the big hole in the functional space is present. It means that some part is quite predictable. For case b), a quasi-uniform distribution is present. It means, that RND-1 algorithm 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.

```

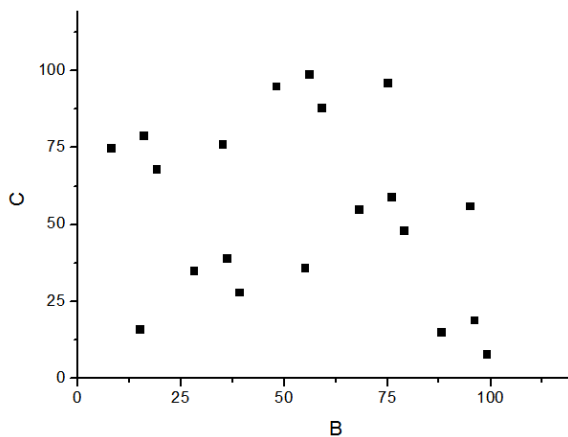
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

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;
    };

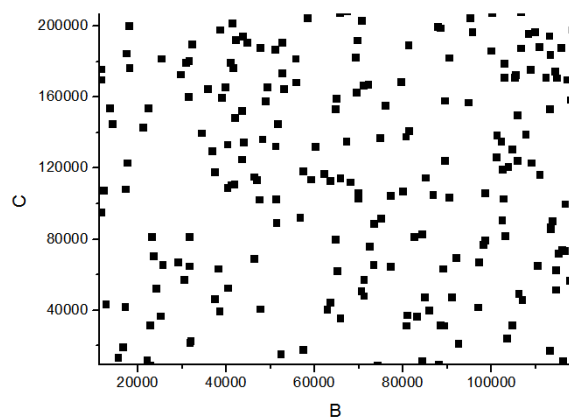
    fout1.close();

    system("PAUSE");
    return EXIT_SUCCESS;
}

```



a)



b)

Fig. 3.26. Random number distributions, generated by RGN-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:

$$\text{Ciphertext} = \text{Plaintext} \oplus \text{Key} \quad (3.11)$$

$$\text{Plaintext} = \text{Ciphertext} \oplus \text{Key} \quad (3.12)$$

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.