ΤΕΙ Θεσσαλονίκης
Τμήμα ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

C++

ΕΓΧΕΙΡΙΔΙΟ και ΑΣΚΗΣΕΙΣ

τεύχος Β

Πασχάλης Ράπτης

2014

Βασικές αρχές της **C++** παραθέτονται στις σημειώσεις τεύχος Α με τίτλο "***C++ Εγχειρίδιο και Ασκήσεις, 1998***".

Αναφορές (References)
Αναφορές και συναρτήσεις
Τύποι δεδομένων χρήστη (User defined types)
Δομητές με μια παράμετρο
Αντιγραφή αντικειμένων
Δομητής αντιγράφου - Copy constructor
Υπερφορτωση τελεστή απονομής **=** (Overload assignment **operator=**)
new και **delete**
Αποδομητης (Destructor)
operator<< και **operator>>**
friend συναρτήσεις και **friend** ταξεις
const
const στις συναρτήσεις

Αρχεία (Files)
Πρότυπα (Templates)
    Γενικές Συναρτησεις
    Γενικές Τάξεις

typedef
namespace
using και **using namespace**

STL – Standard Template Library
- Containers
- Iterators (επαναλήπτες)
    Κατηγορίες Επαναληπτών
        Οι ***επαναλήπτες εισόδου*** (input iterators)
        Οι ***επαναλήπτες εξόδου*** (output iterators)
        Οι ***επαναλήπτες μιας κατεύθυνσης*** (forward iterators)
        Οι ***επαναλήπτες δύο κατευθύνσεων*** (bidirectional iterators)
        Οι ***επαναλήπτες τυχαίας πρόσβασης*** (random access iterators)
    Τυποι επαναληπτών
        iterator (επαναλήπτης).
        const_iterator (σταθερός επαναλήπτης).
        reverse_iterator (αντίστροφος επαναλήπτης).
        reverse_const_iterator (αντίστροφος σταθερός επαναλήπτης).
    Δήλωση επαναληπτών
- Function Objects
- Αλγόριθμοι
vector
list
deque
map
queue
string

Αναφορές (References)

-- Δήλωση αναφορών

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

int main()
{
    int  a  = 5;
    int& r1 = a; // Δήλωση αναφοράς r2
    cout << "Address a  " << &a  << "  timi " << a  << "\n"
         << "Address r1 " << &r1 << "  timi " << r1 << endl;

    int& r2(a); // Ιδιο με: int& r2(r1); // Δήλωση αναφοράς r2
    cout << "Address r2 " << &r2 << "  timi " << r2 << endl;

    system("pause");
    return 0;
}
// w15.cpp //
Address a  0x22ff3c  timi 5
Address r1 0x22ff3c  timi 5
Address r2 0x22ff3c  timi 5
```

Η μεταβλητή **a** και οι αναφορές **r1**, **r2** βρίσκονται στο ίδιο χώρο μνήμης και έχουν την αυτήν τιμή **5**.

Αναφορές και συναρτήσεις

Α) Αναφορά στις παραμέτρους των συναρτήσεων

1.
```
void Func(double& x,int y)
{
   //
}
// Κλήση
a = 3.14; b = 13;
Func(a,b);
```
Η μεταβλητή **a** περνάει στην συνάρτηση ως αναφορά **x** (call by reference). Τυχόν αλλαγές στην **x** επειρεάζουν την μεταβλητή **a**.
Η μεταβλητή **b** **αντιγράφεται** στην **y** (call by value). Τυχόν αλλαγές στην **y** **δεν** επειρεάζει την μεταβλητή **b**.

2.
```
char Func2(const int& y)
{
   char c;
   //
   return(c); // Η συνάρτηση επιστρέφει char
}
// Κλήση
int b = 17; char ch;
ch = Func2(b);
```
Η μεταβλητή **b** περνάει στην συνάρτηση ως **const** αναφορά **y** (call by reference). Δεν επιτρέπονται αλλαγες στην **y** λογω του const. Η μεταβλητή **c** **αντιγράφεται** στο **Func2(b)** και με την σειρά του περναει στην **ch**.

Β) Επιστροφή αναφοράς από συναρτηση (*ΤυποςΣυναρτησης*&). Το επιστρεφόμενο αντικειμενο πρεπει να είναι global αναφορά ή global μεταβλητή (και όχι τοπική αναφορά).

1. ```
   int& Func2(int& y)
   {
      //
      return(y); // Η συνάρτηση επιστρέφει αναφορά σε ακέραιο int&
   }
   // Κλήση
   int b = 17;
   int k = Func2(b);//Ιδιο με: int k = b; // Κλήση στο δεξιο μέρος (rvalue)
   Func2(b) = 22;    //Ιδιο με: b = 22;  // Κλήση στο αριστερό μέρος (lvalue)
   ```
   Η μεταβλητή **b** περνάει στην συνάρτηση ως **const** αναφορά **y** (call by reference) (Το **y** είναι συνώνυμο με το **b**. Το **y** **ΔΕΝ αντιγράφεται** στο **Func2(b)**.

2. ```
   int a;
   int& Func3()
   {
      //
      return(a); // Η συνάρτηση επιστρέφει αναφορά σε ακέραιο int&
   }
   // Κλήση
   a = 23;
   int k = Func3();//Ιδιο με: int k = a; // Κλήση στο δεξιο μέρος (rvalue)
   Func3() = 33;    //Ιδιο με: a = 33;  // Κλήση στο αριστερό μέρος (lvalue)
   ```

3. ΛΑΘΟΣ. Το επιστρεφόμενο αντιμείμενο πρεπει να είναι global και όχι τοπικό
   ```
   int& Func4(int& y)
   {
      int h = y;  // Τοπική μεταβλητη
       //
      return(h); // Η συνάρτηση επιστρέφει αναφορά σε ακέραιο int&
   }
   ```

Εάν αυτό που μας ενδιαφέρει είναι η **τιμή** που επιστρέφει μια συνάρτηση και δεν προκειται να χρησιμοποιηθεί η συναρτηση σε αριστερό μέλος ισότητας (lvalue), τότε δεν εχει σημασία πως επιστρέφει την τιμή αυτήν η συνάρτηση με *τύπο* ή με *τύπο***&** . Στα παραδείγματα που ακολουθούν μας ενδιαφέρει η τιμή **char** και η τιμή **double** που επιστρέφουν οι συναρτήσεις

α) **char Func()** ή **char& Func()**
```
 // κλήση
c = Func();
```

Συμπερισματικά μπορούμε να πούμε ότι όταν μια συναρτηση επιστρέφει αναφορα τότε μπορεί να χρησιμοποιηθεί και στο αριστερο (lvalue) και δεξιό (rvalue) μερος μιας ισοτητας.

-- Η συναρτηση **F** επιστρέφει μια αναφορά σε «γλομπαλ» μεταβλητη.
```
#include <iostream>
using namespace std;
```

```
int a = 5;

int& F()
{
    return a;
}

int main()
{
    int &ra = a;
    int& rb = ra; // Ιδιο με: int& rb(ra);
    ra = 2;
    cout << ra << endl;

    int &r = F();
    F() = 44;
    cout << F() << endl;

    cout << "Addres " << &a << "==" << &r << "==" << &(F()) << endl;
    cout << "Times  " <<  a << "==" <<  r << "==" <<   F()  << endl;

    system("pause");
    return 0;
}
// w33.cpp //
```

- Η συναρτηση **F** επιστρέφει μια αναφορά στην μεταβλήτη **a**.
- Η αναφορα **ra** χρησιμοποιείται για τον ορισμό της αναφοράς **rb**, δηλ. η μεταβλητή **a** και οι αναφορές **ra** και **rb** αναφερονται στον ιδιο χώρο μνήμης. Οπότε μπορώ να χρησιμοποιήσω την **ra** για ανακτηση και αποδοση τιμής στη **a**.
- Με την ίδια λογική, η αναφορά που επιστρέφει η **F** χρησιμοποιείται για τον ορισμό της αναφοράς **r**, δηλ. η μεταβλητή **a** και οι αναφορές **F()** και **r** αναφερονται στον ιδιο χώρο μνήμης. Οπότε μπορώ να χρησιμοποιήσω την συναρτηση **F()** για ανακτηση και αποδοση τιμής στη **a**.

-- Συνάρτηση *τυπου***&** με μια παραμετρό επιστρέφει γενική «γκλομπαλ» αναφορά.

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

int& F(int& a)
{
    return a;
}

int main()
{
    int a = 5;
    int &ra = a;
    int& rb = ra; // int& rb(ra);
    ra = 2;
    cout << ra << endl;

    int &r = F(a);
    F(a) = 44;
```

```
    cout << F(a) << endl;

    cout << "Addres " << &a << "==" << &r << "==" << &(F(a)) << endl;
    cout << "Times  " <<  a << "==" <<  r << "==" <<   F(a)  << endl;

    system("pause");
    return 0;
}
// w34.cpp //
```

Η συνάρτηση **Ε** *τύπου***&** καλείται με μια αναφορά **r1** (ή με μια μεταβλητή **a**), επεξεργάζεται την αναφορά **r** (**r=r1)** και επιστρέφει την αναφορά **r** (το αποτέλεσμα είναι σωστό).

////

Τύποι δεδομένων χρήστη (User defined types)

Ένας χρήστης – προγραμματιστής (user) μπορεί να ορίσει – δημιουργήσει νέους τύπους δεδομένων με τις δεσμευμένες λέξεις **class** ή **struct**. Οι νέοι αυτοί τύποι χρησιμοποιούν τους ενσωματωμένους βασικούς τύπους της C++ όπως **int**, **char**, **long**, κλπ., πίνακες, δείκτες, αναφορές.

Εξ' ορισμού τα μέλη σε μια **class** είναι ιδιωτικά, ενώ σε μια **struct** εξ' ορισμού τα μελή είναι δημόσια.

Ισοδύναμοι τύποι

`class NT {`
`  long g;`
`};`
	`struct NT {`
	`  private:`
	`  long g;`
	`};`

Ισοδύναμοι τύποι

`class GA {`
`  public:`
`  long h;`
`};`
	`struct GA {`
	`  long h;`
	`};`

-- Παραδείγματα ορισμού τύπων χρήστη

```
class A {
    public:
    int x;
};
```

Ορίζεται η ταξη **A** που εχει μια δημόσια μεταβλητή **x** τυπου **int**. Η μεταβλητή **x** ανήκει στην ταξη (είναι μελος της ταξης) **A**.

```
class B {
    public:
      int x;
      float y;
    private:
      char a[100];
      int n;
      float F1() { return (3.14) }
    public:
```

```
      void F2(A& ob,int k) { }
};
```
Το **public** και το **private** μπορούν να εναλλάσσονται πολλές φόρες.
Η τάξη **B** έχει δημόσιες και ιδιωτικές a) μεταβλητές (**x,y,n**), b) μεθόδους (**F1,F2**).

-- Δημιουργία αντικειμένων, πινάκων, δεικτών τύπου **A** (εκτελείται ο εξ' ορισμού (default) δομητής)

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

class A {
    //
};

int main()
{
    A a,       // Αντικείμενο a
      b[10],   // Πινακας b με 10 αντικείμενα τυπου A
      d[3][4],// Πινακας δύο διαστάσεων
      *p0,     // Ο δεικτης p0 δυναται να δειζει σε αντικεμενο τυπου A
      *par[5],// Πινακας από 5 δεικτες
      **pp,    // Δεικτης σε δεικτη
      *p1, *p2, *p3;
    A& c=a;    // Το c είναι αναφορά στο a
    p0 = &a;   // Ο p0 δειχνει στο a
    p0 = b;    // Ο p0 δειχνει στο πρώτο στοιχειο του πινακα b
    p0 = &b[0];
    p1 = new A; // Ο p1 δειχνει σε αντικειμενο που δημιουργηθηκε δυναμικά
    p2 = new A [10]; // Ο p1 δειχνει σε πινακα 10 στοιχείων
    A *p3(new A); // Ο p3 δειχνει σε ένα αντικειμενο τυπου A
    A *p4(new A[5]); // Ο p4 δειχνει σε πινακα 5 στοιχείων τύπου A
    A& h = *(new A); // Αναφορά h σε δυναμικο αντικείμενο

    A g(a); // Δημιουργία αντικειμένου g με τιμή από το a
    g = a;
    A();    // Δημιουργία προσωρινού (temporary) ανώνυμου αντικειμένου
    g = A();

    system("pause");
    return 0;
}
///
```

-- Δημιουργία δυναμικού πίνακα ακεραίων **n** στοιχείων. Ο εξ'ορισμού (default) δομητής εκτελείται **n** φορές.

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

static int x = 0;
class A {
      public:
      A(){cout << ++x <<"domit "; }
```

```
};

int main()
{
    int n = 7;
    A *pa = new A [n]; // Πίνακας ακεραίων με 7 στοιχεία

    system("pause");
    return 0;
}
// w17.cpp //
```
Αποτέλεσμα
```
1domit 2domit 3domit 4domit 5domit 6domit 7domit
```

explicit δομητής - δομητής με μια παράμετρο

Αντικείμενα τάξεων που διαθετουν δομητή με μια παραμετρο μπορούν να συμπεριφερονται ως απλά αντικείμενα δηλ. όπως οι μεταβλητές τύπου **int, char, float** όσο αφορά τον τελεστή απονομής τιμής **=**.

-- Δημιουργία ταξης **MC** με δομητή με μια παράμετρο
```
#include <iostream>
using namespace std;

class MC {
    int z;
    public:
    MC(int a){ z = a; cout << "Domitis\n"; }
    int get() { return(z); }
};

int main()
{
    MC ob(4);     //  Εκτέλεση δομητή
    cout << "Timi: " << ob.get() << endl;

    ob =  9;      // Εκτέλεση δομητή
    ob = 11;      // Εκτέλεση δομητή
    cout << "Timi: " << ob.get() << endl;

    A ox = 32;           // Εκτέλεση δομητή
    cout << "Timi: " << ob.get() << endl;
    ox = 66;             // Εκτέλεση δομητή
    cout << "Timi: " << ob.get() << endl;

    system("pause");
    return 0;
}
// w80.cpp //
Domitis
4
Domitis
13
```

```
Domitis
32
Domitis
66
```

- Ο δομητής (με μια παράμετρο) εκτελείται με τις εντολές α) **ob(4)**, β) **ob = 9**, γ) **ob = 11**, δ) **A ox = 32** , ε) **ox = 66**, δηλ. για το αντικείμενο **ob** ο δομητής με μια παράμετρο εκτελείται δηλ αρχικοποιείται **τρείς** φορές: με τιμη 4, με τιμή 9 και με τιμή 13 και για το αντικείμενο **ox** ο δομητής εκτελείται **δύο** φορές: με τιμη 32 και με τιμή 66.

-- Δημιουργία ταξης **A** με τρείς δομητές (ένας από τους δομητές εχει μια παράμετρο)

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

class A {
    int x;
    float y;
    char z;

    public:
    A(){ cout << "Default domitis" << endl; }

    A(int a)
    {
       cout << "Domitis me 1 param\n";
    }

    A(int a, float b)
    {
       cout << "Domitis me 2 param\n";
    }

    A(int a, float b, char c)
    {
       cout << "Domitis me 3 param\n";
    }
};

int main()
{
    A ob0,              // Default domitis
      ob1(44),          // Domitis me 1 param
      ob2(6,3.14),      // Domitis me 2 param
      ob3(11,55.5,'K'); // Domitis me 3 param

    ob1 = 13;           // Domitis me 1 param
    ob1 = 17;           // Domitis me 1 param

    system("pause");
    return 0;
}
///
```

- Ο δομητής με μια παραμετρο εκτελείται με τις εντολές α) **ob1(44)**, β) **ob1 = 13**, γ) **ob1 = 17**, δηλ. για το αντικείμενο **ob1** ο δομητής με μια παράμετρο εκτελείται **τρείς** φορές: με τιμη 44, με τιμή 13 και με τιμή 17.

-- Δημιουργία αντικειμένων τύπου **A** με δομητή που εχει μια παράμετρο.

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

class A {
    int z;
    public:
    A(int a) { cout << "Domitis me 1 param\n"; z = a; }
};

void F(A ob)
{
    //
}

int main()
{
    A a(9);     // Domitis me 1 param
    a = 17;     // Domitis me 1 param
    A t(a);
    a = t;
    A x = 33;  // Domitis me 1 param
    A(3);       // Domitis me 1 param
    x = A(23); // Domitis me 1 param
    F(13);      // Domitis me 1 param
    system("pause");
    return 0;
}
// w18.cpp //
```

- Για το αντικείμενο **a** ο δομητής εκτελείται δύο φορές μια φορά με τιμή 9 και μια φορά με τιμή 17.
- Για το αντικειμενο **x** ο δομητής εκτελείται με τιμή 33.
- Επίσης ο δομητής εκτελείται με τις εντολές **A(3)** και **A(23).**
- Τέλος ο δομητής εκτελείται στην κληση της **F** οπου το όρισμα σταθερά 13 περνά by value στην συναρτηση **F** (**A obj = 13**)

Εάν θελουμε να ειμαστε σιγουροι ότι ενας δομητής με μια παράμετρο θα εκτελείται μόνο με τον τρόπο που εκτελούνται ολοι οι άλλοι δομητές δηλ. a(9), A(3), A(23), τότε δομητής με μια παράμετρο πρεπει να δηλωθεί ως explicit.

-- Δημιουργία αντικειμένων τύπου **A** με **explicit** δομητή που εχει μια παράμετρο. Το πρόγραμμα είναι ίδιο με το προηγούμενο μόνο που ο δομητης εχει δηλωθεί ως **explicit**. Ελέγξετε τα αποτελέσματα αυτού του προγραμματος με τα αποτελέσματα του προηγούμενου προγράμματος.

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

class A {
```

```
    int z;
    public:
    explicit A(int a)
    {
       cout << "Domitis me 1 param\n";
       z=a;
    }
};

void F(A ob)
{
    //
}

int main()
{
    A a(9);        // Domitis me 1 param
    // a = 17;     // ΛΑΘΟΣ.
    A t(a);
    a = t;
    // A x = 33;   // ΛΑΘΟΣ.
    A(3);          // Domitis me 1 param
    A y = A(23);   // Domitis me 1 param

    // F(13);      // ΛΑΘΟΣ.

    system("pause");
    return 0;
}
// w19.cpp //
```

Οι εντολές **a=17** και **A x=33** δινουν λαθος στη μεταφραση (compilation) γιατι ο δομητής εχει δηλωθεί ως **explicit.**

Στο επόμενο πρόγραμμα **παρακάμπτεται μερικώς η δήλωση** **explicit** στον δομητή, με την υπερφορτωση του τελεστη **=** απονομής τιμής με την συνάρτηση **operator=** οπου ο προγραμματιστής γραφει τον αντιστοιχο κωδικα για απονομη τιμης ακεραίου σε αντικείμενο Στο συγκεκριμένο παραδειγμα γινεται απονομή ακεραιου σε αντικείμενο **(a = 17)** και ο κωδικας που κανει την αντιγραφή είναι η εντολή **z=b;.**

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

class A {
    public:
    int z;
    explicit A(int a) { cout << "Domitis me 1 param\n"; z=a; }
    A& operator=(int b) { z=b; }
};

int main()
{
    A a(9);   // Domitis me 1 param
    a = 17;   // OK! Idio me: a.operator(17);
```

```
    cout << a.z << endl;
    A t(a);
    a = t;
    // A x = 33;  // ΛΑΘΟΣ.
    A(3);         // Domitis me 1 param

    system("pause");
    return 0;
}
// w16.cpp //
```

Αντιγραφή αντικειμένων

Οι περιπτωσεις οπου εκτελείται bit-προς-bit αντιγραφή ακτιμειμενων είναι οι ακολουθες:

```
class A {
      //
};
```

α) με τον τελεστή απονομής τιμης

```
 A ob1, ob2;
ob1 = ob2;
```

// Το αντικειμενο **ob2** αντιγραφεται στο **ob1**

β) όταν περνα ένα αντικειμενο ως παραμετρος by value σε συνάρτηση

```
void F(A ob) // κληση με τιμή
{
      //
}
//
A obj;
F(obj);
```

// Το αντικειμενο **obj** αντιγραφεται στο **ob**.

γ) όταν ένα αντικείμενο χρησιμοποιηται στην δημιουργια ενός άλλου αντικειμένου

```
A ox;
A obj(ox);
```

// Το αντικείμενο **ox** αντιγραφεται στο **obj**.

δ) όταν μια συνάρτηση επιστρέφει ένα αντικειμενο

```
A F()
{
     A tmp;
      //
     return(tmp);
}
//
A ob;
F();
```

// Το αντικειμενο **tmp** αντιγραφεται στο **F()** (και μετά πχ. **ob = F();**).

Για την ταξη **A** : στην περιπτωση ***α*** εκτελειται ο εξ' όρισμου τελεστης **=** απονομής τιμής και στις περιπτώσεις ***β, γ*** και ***δ*** εκτελειται ο εξ' όρισμου δομητης αντιγράφου.

Η bit-προς-bit αντιγραφή είναι αξιοπιστη και δεν δημιουργεί προβληματα οσο τα αντικείμενα μιας ταξης δεν χρησιμοποιουν δυναμικά μνήμη (τα αντικείμενα δεν δεσμεύουν δυναμικα μνήμη με τον τελεστή **new** ή την συναρτηση **malloc** της **C**).

Στην περιπτωση που τα αντικειμενα μιας ταξης χρησιμοποιουν δυναμικα μνήμη τοτε επιβαλεται η υλοποιηση του **δομητή αντιγραφου** (**copy constructor**) και της συνάρτησης υπερφορτωσης του τελεστή**=** (assignment **operator=**). Επίσης επιβαλλεται η υλοποίηση του **αποδομητη** (**destructor**) ώστε όταν «τελειώνει» η ζωη ενός αντικειμένου να αποδεσμεύει την καταληφθείσα μνήμη.

Δομητής αντιγράφου - Copy constructor

-- Δημιουργια αντικειμένων τύπου **A** με δομητή αντιγραφου.

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

class A {
    int z;
    public:
    A() { cout << "Default domitis\n"; }
    A(const A& o) { cout << "Domitis antigrafou\n"; z=o.z; }
};

// void F(A& o) { // ΔΕΝ εκτελείται ο δομητης αντιγραφου στην κληση με αναφορά
void F(A ob) {    // Εκτελείται ο δομητης αντιγραφου στην κληση με τιμη
    cout << "Func F \n";
}

int main()
{
    A a;        // Default δομητης
    A t(a);     // α. Copy Constructor
    F(a);       // β. Copy Constructor.  3. Func F.

    system("pause");
    return 0;
}
```

Ο δομητης αντιγραφου καλείται, α) όταν το αντικείμενο **t** δημιουργείται (το **a** παιρναει ως **const** αναφορά στον δομητη αντιγραφου), β) στην κληση της συνάρτησης **F** το αντικείμενο **a** παιρναει ως αντιγραφο στο αντικειμενο **ob** (κληση με τιμη – call by value).

Και στις δύο περιπτώσεις (δημιουργια του **t** και του **ob**) εκτελειται ο δομητης αντιγραφου οπου ο προγραμματιστης καλείται να γραψει (στον δομητη αντιγράφου) τον κωδικα αντιγραφης από αντικειμενο σε αντικείμενο. Στο συγκεκριμενο παραδειγμα ο κωδικας αντιγραφης είναι η εντολη **z=o.z;**.

Εάν το πρωτοπτυπο της **F** αλλαξει από **void F(A ob)** σε **void F(A& o)** τοτε δεν καλειται ο δομητης αντιγραφου.

Υπερφορτωση τελεστή απονομής = (assignment operator=)

-- Περιπτωσεις οπου εκτελειται η συνάρτηση υπερφορτωσης τελεστή**=** (Ο δομητης αντιγραφου αν και είναι υλοποιημένος δεν εκτελειται)

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

class A {
    int z;
    public:
    A() { cout << "Default Domitis\n"; }
    A(const A& o) { cout << "Domitis antigrafoy\n"; z = o.z; }

    A& operator=(A& ox) {
       cout << "Telestis aponomis\n";
       z = ox.z;
       return(*this);
    }
};

A& F(A& ob) {  // DEN kalei ton copy constructor
    cout << "Func F \n";
    return( ob );
}

int main()
{
    A a, t;    // 1. 2. Default Domitis
    a = t;     // 3. Assignment operator
    a = F(t);  // 4. FuncF   5. Telestis aponomis

    system("pause");
    return 0;
}
```

Η συνάρτηση **operator=** εκτελείται όταν το περιεχόμενο ενος αντικειμένου αντιγραφεται σε ένα άλλο αντικείμενο, πχ. στην εντόλη **a = t** οπου γινεται η αντιγραφή του αντικειμένου **t** στο **a**. Το ίδιο συμβαίνει όταν το αντικείμενο που επιστρέφει η συνάρτηση **F** αντιγραφεται στο **a**. Ο προγραμματιστης καλείται να γραψει στην συνάρτηση **operator=** τον κωδικα αντιγραφης από αντικειμενο σε αντικείμενο. Στο συγκεκριμενο παραδειγμα ο κωδικας αντιγραφης είναι η εντολη **z=ox.z;**.

Εάν δεν υλοποιηθεί η συνάρτηση **operator=** τοτε γινεται bit προς bit αντιγραφή από το ένα αντικείμενο στο άλλο.

Εάν το πρότυπο της συνάρτησης **operator=** από **A& operator=(A& ox)** αλλαξει σε **A operator=(A& ox)**, τοτε μετα την κληση της **operator=** καλείται και ο δομητης αντιγραφου.

new και delete

-- Στο επομενο πρόγραμμα δημιουργειται πινακας ακεραιων **n** στοιχείων και διαγραφεται. Ο δεικτης **p** μετα την αποδεσμευση της μνημης εξακολουθει να δειχνει στον ιδιο χωρο μνήμης και η εσφαλμενη ή ηθελημένη χρηση του **p** για προσβαση στην μνημη δημιουργεί μη αξιοπιστα αποτελεσματα.

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

void prn(int *a,int n) // Εμφανιση πινακα
{
    for (int i=0; i<n; i++)
```

```
        cout << a[i] << "  ";
    cout << endl;
}

int main()
{
    int n = 5;
    int *p = new int [n];
    Int *t = p; // Οι δείκτες p και t δειχνουν στην δεσμευμένη μνήμη

    cout << "Prin p=" << p << endl;
    for(int i=0; i< n; i++)
        p[i] = i*i;
    cout << "Pinakas: "; prn(p,n);

    delete [] p;  // Ιδιο με: delete [] t;

    cout << "Meta p=" << p << endl;
    cout << "Pinakas: "; prn(p,n);

    system("pause");
    return 0;
} // w20.cpp //
```

Αποτελεσμα

```
Prin p=0x970fb0
Pinakas: 0  1  4  9  16
Meta p=0x970fb0
Pinakas: 9904672  9899848  4  9  16
```

Αποδομητης (Destructor)

Ο αποδομητης είναι μια συνάρτηση που εκτελείται όταν ένα αντικείμενο καταστρεφεται (η «ζωή» ενός αντικειμενου τελειώνει). Το ονομα του αποδομητη είναι ιδιο με το ονομα της ταξης (όπως και των δομητων) αλλα έπεται του **~** (tilde).

```
    ~ΟνομαΤαξης();

class A {
    public:
    ~A() {
        //
    }
};
```

-- Το παρακάτω πρόγραμμα επιδεικνύει την υλοποίηση και την κλήση του αποδομητη της τάξης **Arr**.

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

class Arr {
    int *pb;
    int *pe;
    public:
        // Domites
```

```
        Arr() { pb = 0; pe = 0; }
        // Apodomiths
        ~Arr() { cout << "Apodomitis" << '\n'; }
};

void Fx()
{
    Arr t;
    cout << "Func Fx...\n";
}

int main()
{
    {  // dummy block
       Arr o1;
       Fx();
    }

    system("pause");
    return 0;
}
// w02.cpp //
```

Ο αποδομητής εκτελείται το τέλος του block που εχει δηλωθεί α) το αντικειμενο **t** καταστρεφεται στο τέλος της συνάρτησης **Fx**, β) το αντικειμενο **o1** στο τελος του dummy block. Τα αντικείμενα καταστρεφονται με την αντιστροφη σειρά που δημιουργήθηκαν.

Ο αποδομητης δημιουργεί προβλημα όταν «ελευθερώνει» την μνημη που ένα αντικείμενο εχει δεσμεύσει δυναμικά.

-- Το παρακατω πρόγραμμα επιδικνύει το προβλημα που δημιουργει όταν ο αποδομητης αποδεσμεύει την μνήμη που ένα αντιικείμενο εχει δεσμεύσει δυναμικά.

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

class Arr {
    int *pb;
    int *pe;
    public:
        // Δομητές
        Arr() { pb = 0; pe = 0; }
        Arr(int a)
        {
            pb = new int [a];
            pe = pb + a;
        }

        // Αποδομητης
        ~Arr() { delete [] pb; }

        int& operator[](int i) { return pb[i]; }
};

int main()
```

```
{
    Arr a(10);  // Δυναμικός πινακας a 10 στοιχειων
    for(int i=0; i<10; ++i)
        a[i]=i; // Γεμισμα του πινακα με τιμές

    cout << "1 pinakas a: "; // Εμφανιση του a
    for(int i=0; i<10; ++i)
        cout << a[i] << " ";
    cout << endl;

    if (true) {
        Arr b(5); // Δυναμικός πινακας b 5 στοιχειων
        b = a;    // bit-προς-bit αντιγραφή του a στο b
        cout << " pinakas b: "; // Εμφανιση του b
        for(int i=0; i<10; ++i)
            cout << b[i] << " ";
        cout << endl;
    } // Κληση αποδομητη για το b. Αποδεσμευση μνημης

    cout << "2 pinakas a: "; // Εμφανιση του a
    for(int i=0; i<10; ++i)
        cout << a[i] << " ";
    cout << endl;

    system("pause");
    return 0;
}
// w03.cpp //
```

Αποτελεσμα

```
1 pinakas a: 0 1 2 3 4 5 6 7 8 9
  pinakas b: 0 1 2 3 4 5 6 7 8 9
2 pinakas a: 2953256 2953072 2 3 4 5 6 7 8 9
```

Μετά την bit-προς-bit αντιγραφή του **a** στο **b** το αντικείμενο **b** μοιραζεται τον ιδιο χώρο μνήμης με το αντικειμενο **a**. Όταν ο αποδομητης εκτελείται για το αντικειμενο **b** αποδεσμευει την μνημη κάνοντας το αντικείμενο **a** να δειχνει (ο δεικτης **pb** του **a**) σε μνημη που εχει ελευθερώθει με αποτελεσμα όταν εμφανιζεται ο πινακας **a** για δευτερη φορά να είναι μερικώς κατεστραμενος.

Το προβλημα αυτό λυνεται με την υλοποίηση του δομητή αντιγραφου και της υπερφορτωσης του (τελεστή) **operator=**.

-- Η ταξη **Arr** του προηγουμενου προγραμματος εμπλουτιζεται με έναν δομητή αντιγραφου και με μια συνάρτηση υπερφορτωσης του **operator=** για να λυθεί το προβλημα που περιγραφηκε στο προηγουμενο πρόγραμμα.

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

class Arr {
    int *pb;
    int *pe;
    public:
        // Δομητες
```

```
        Arr() { pb = 0; pe = 0; }
        Arr(int a)
        {
            if (a > 0) {
                pb = new int [a];
                pe = pb + a;
            }
            else {
                pb = 0; pe = 0;
            }
        }

        Arr(const Arr& ob) // Δομητης αντιγράφου
        {
            delete [] pb;
            int siz = ob.pe - ob.pb;
            pb = new int [siz];
            for(int i=0; i < siz; ++i)
                *(pb+i) = *(ob.pb+i);
            pe = pb + siz;
        }

        ~Arr() { delete [] pb; } // Αποδομητης

        Arr& operator=(const Arr& ox) // Τελεστης απονομης τιμής
        {
            int *t;
            delete [] pb;
            int siz = ox.pe - ox.pb;
            pb = t = new int [siz];
            for(int i=0; i < siz; ++i)
                *t++ = *(ox.pb+i);
            pe = pb + siz;
            return(*this);
        }

        int length() { return (int)(pe-pb); }

        int& operator[](int i) { return pb[i]; }
};

void print(Arr& oy)
{
    for(int i=0; i<oy.length(); ++i)
        cout << oy[i] << " ";
    cout << endl;
}

int main()
{
    Arr a(10);
    for(int i=0; i<a.length(); ++i)
        a[i]=i;
    cout << "1 pinakas a: ";
    print(a);
```

```
    if (true) {
        Arr b(0);
        b = a;
        cout << "  pinakas b: ";
        print(b);
    }

    cout << "2 pinakas a: ";
    print(a);

    system("pause");
    return 0;
}
// w04.cpp //
1 pinakas a: 0 1 2 3 4 5 6 7 8 9
  pinakas b: 0 1 2 3 4 5 6 7 8 9
2 pinakas a: 0 1 2 3 4 5 6 7 8 9
```

-- Στο προηγουμενο πρόγραμμα η ταξη **Arr** υποστηρίζει δεδομενα τυπου **int**. Στο επομενο πρόγραμμα η ταξη **Arr** μετατρεπεται σε γενική για να υποστηρίζει περισσοτερους τυπους δεδομενων όπως **char, double**, **int** και τυπους χρηστων (ταξη **Z**). Η ταξη **Z** εφοδιαζεται με τις καταλληλες μεθοδους .

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

template <typename T>
class Arr {
    T *pb;
    T *pe;
    public:
        // Δομητες
        Arr() { pb = 0; pe = 0; }
        Arr(int a)
        {
            if (a > 0) {
                pb = new T [a];
                pe = pb + a;
            }
            else {
                pb = 0; pe = 0;
            }
        }

        ~Arr() { delete [] pb; } // Αποδομητης

        Arr(const Arr& ob) // Δομητης αντιγραφου
        {
            delete [] pb;
            int siz = ob.pe - ob.pb;
            pb = new T [siz];
            for(int i=0; i < siz; ++i)
                *(pb+i) = *(ob.pb+i);
            pe = pb + siz;
```

```
        }

        Arr& operator=(const Arr& ox)// Τελεστης απονομης τιμής
        {
            T *t;
            delete [] pb;
            int siz = ox.pe - ox.pb;
            pb = t = new T [siz];
            for(int i=0; i < siz; ++i)
                *t++ = *(ox.pb+i);
            pe = pb + siz;
            return(*this);
        }

        int length() { return (int)(pe-pb); }

        T& operator[](int i) { return pb[i]; }
};

template<typename X>
void print(X& oy)
{
    for(int i=0; i<oy.length(); ++i)
      cout << oy[i] << " ";
    cout << endl;
}

class Z {
    int z;
    public:
    Z() {}
    Z(int a) { z = a; }
    void set(int a) { z = a; }
    int get() { return(z); }
    friend ostream& operator<<(ostream& os,Z& o);
};

ostream& operator<<(ostream& os,Z& o)
{
    os << o.get() << " ";  // OK. o.z
}

int main()
{
    Arr<Z> a(10);
    for(int i=0; i<a.length(); ++i)
        a[i] = i; // OK!
        // a[i].set(i); // OK!
        // a.operator[](i).set(i); // OK!
    cout << "pinakas a: ";
    print(a);

    Arr<double> b(12);
    for(int i=0; i<b.length(); ++i)
        b[i]=i+0.5;
```

```
    cout << "pinakas b: ";
    print(b);

    Arr<char> c(26);
    for(int i=0; i<c.length(); ++i)
        c[i]=i+'A';
    cout << "pinakas c: ";
    print(c);

    system("pause");
    return 0;
}
// w05.cpp //
```

Αποτελεσμα

```
pinakas a: 0  1  2  3  4  5  6  7  8  9
pinakas b: 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5
pinakas c: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

operator<< και operator>>

Ο τελεστης εξόδου **<<** υπερφορτωνεται με την συνάρτηση **operator<<** ώστε να είναι δυνατή εξοδος αντικειμένων (εμφανιση στην οθόνη ή αποθηκευση σε αρχείο).

```
class DC {
    //
};

DC obo;
cout << obo << endl;
```

Στην ταξη **DC** πρεπει να δηλωθει το πρότυπο της συνάρτησης ως **friend**.

```
      friend ostream& operator<<(ostream& os,DC& o);
```

Ο τελεστης εισόδου **>>** υπερφορτωνεται με την συνάρτηση **operator>>** ώστε να είναι δυνατή η εισοδος στοιχειων των αντικειμένων (από το πληκτρολογιο ή από αρχείο).

```
class DC {
    //
};

DC obi;
cin >> obi;
```

Στην ταξη **DC** πρεπει να δηλωθει το πρότυπο της συνάρτησης ως **friend**.

```
      friend istream& operator>>(istream& os,DC& o);
```

-- Το παρακατω πρόγραμμα επιδεικνύει την υλοποιηση των συναρτησεων εισοδου εξοδου σε μεταβλητες αντικειμενων.

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

class DC {
    int z;
    char nam[100];
```

```
    public:
    DC() : z(0) { nam[0]='\0'; }
    DC(int a,char *s) { z = a; strcpy(nam,s); }

    void set(int a,char s[]) { z = a; strcpy(nam,s); }
    int get() { return(z); }

    friend ostream& operator<<(ostream& os,DC& o);
    friend istream& operator>>(istream& is,DC& o);
};

ostream& operator<<(ostream& os,DC& o)
{
    os << o.z << " " << o.nam << endl;
    return os;
}

istream& operator>>(istream& is,DC& ob)
{
    int a; char s[100];
    cout << "Akeraio: ";
    is >> a;
    cout << "String: ";
    is >> s;
    ob = DC(a,s); // ob.set(a,s);
    return is;
}

int main()
{
    DC a(5,"Firebird");
    cout << "Emf a: " << a << endl;

    DC b;
    cout << "Times gia to b"<<'\n';
    cin >> b;
    cout << "Emf b: " << b << endl;

    system("pause");
    return 0;
}
// w06.cpp //
```

Η εκτολή **cin >> b** μπορεί επισης να γραφεί **operator>>(cin,b).**

-- Στην επομενη ιεραρχία τάξεων ο τελεστής **<<** υπερφορτώνεται με συναρτήσεις μελη και τάξη βάση και στην παραγόμενη τάξη.

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

class Bas {  // ταξη βαση
    int ib;
    public:
    Bas() { ib = 15; }
    ostream& operator<<(ostream& os);
```

```
};

class Par : public Bas { // παραγομενη τάξη
    int ip;
    public:
    Par() { ip = 39; }
    ostream& operator<<(ostream& os);
};

ostream& Bas::operator<<(ostream& os)
{
    os << "bas " << ib << endl;
    return os;
}

ostream& Par::operator<<(ostream& os)
{
    Bas::operator<<(os);  // Κληση της συναρτησης τελεστη<< της βασης
    os << "par " << ip << endl;
    return os;
}

int main()
{
    Par pa;
    pa.operator<<(cout);// Κλήση της συνάρτησης τελεστή<< της παραγόμενης
                        // η οποία με την σειρά της καλεί την συνάρτηση << της βάσης.
    Bas ba;
    ba.operator<<(cout);// Κλήση της συνάρτησης τελεστή<< της βασης

    system("pause");
    return 0;
}
// w21.cpp //
```
Αποτελεσματα
```
bas 15
par 29
```

-- Στην ιεραρχία τάξεων του προηγουμένου προγράμματος ο τελεστής **<<** υπερφορτώνεται με φιλες (**friend**) συναρτήσεις μελη και τάξη βάση και στην παραγόμενη τάξη.

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

class Bas { // Τάξη βάση
    int ib;
    public:
    Bas() { ib = 15; }
    friend ostream& operator<<(ostream& os, Bas& ob);
};

ostream& operator<<(ostream& os,Bas& ob)
{
    os << "bas " << ob.ib << endl;
```

```
    return os;
}

class Par : public Bas {  // Παραγόμενη τάξη
    int ip;
    public:
    Par() { ip = 39; }
    friend ostream& operator<<(ostream& os, Par& op);
};

ostream& operator<<(ostream& os,Par& op)
{
    Bas ba = (Bas)op; // Μετατροπή
    // OK!  os << ba;   // Idio me: operator<<(os,ba); // OK!
    os << ba << "par " << op.ip << endl;
    return os;
}

int main()
{
    Par pa;
    cout << pa; // Κλήση της friend συνάρτησης τελεστή<< της παραγόμενης
                // η οποία με την σειρά της καλεί την friend συνάρτηση << της βάσης.
    // pa = (Par) ba; // ΛΑΘΟΣ. Αδύνατη μετατροπή αντεικειμενου βασης
                      σε αντικειμενο παραγομενης
    ba = (Bas) pa;  // OK! μετατροπή αντεικειμενου παραγόμενης
                      σε αντικειμενο βασης
    Bas ba;
    cout << ba; // Κλήση της friend συνάρτησης τελεστή<< της βασης

    system("pause");
    return 0;
}
// w22.cpp //
```

`friend` συναρτήσεις και `friend` ταξεις

Οι φιλες (friend) συναρτησεις εχουν προσβαση σε δημοσια και ιδιωτικά μελη μιας ταξης. Συνηθως οι συναρτησεις οριζονται φιλες α) όταν χρειαζονται προσβαση σε δυο ταξεις, β) όταν δεν είναι η επιθυμητή η χρηση των αντικειμενων για την κληση των συναρτησεων μιας ταξης

α)
```
class N {
     public:
     friend N comp(N n, M m);
     //
  };

  class M {
     public:
     friend N comp(N n, M m);
     //
  };
```

```
  N comp(N n, M m) // Υλοποιηση της comp
  {
    //
  }
```

β)
```
class IT {
     public:
     int Fx(const IT& a) const;
     friend int Fy(const IT& a, const IT& b);
  } ;

  void f( IT& p, IT& q )
  {
      int x = p.Fx(q);
      int y = Fy(p,q);
  }
////
```

Επιτρέπεται σε μια τάξη να εχει προσβαση στα ιδιωτικά μελη μιας άλλης ταξης όταν στην δευτερη ταξη δηλωνεται η πρώτη ταξη ως φιλη.

```
      class Pr {
         //
       };
      class Dr {
         friend class Pr; // Δήλωση φιλης ταξης Pr.
       };
```

Μια ταξη **FC** επιτρεπεται να εχει προσβαση στα ιδιωτικά μελη μιας άλλης ταξης **A** όταν στην ταξη **A** δηλωνεται η ταξη **FC** ως φιλη (**friend class**).

```
class A {
    int dat;
    friend class FC; // Δήλωση φιλης ταξης FC.
    public:
};

class FC {
    A ob;
    public:
    FC() // Δομητης
    {
       ob.dat = 4; // Προσβαση σε ιδιωτικο μελος της A
    }
};
////
```

const

Δήλωση σταθερών με **const**.

```
const int  a=2; // Δήλωση σταθερας int a
a = 7; // ΛΑΘΟΣ. Το a δεν μπορει να αλλαξει
const int b(9); // Δήλωση σταθερας int b
```

const int c; // ΛΑΘΟΣ. Η σταθερά **c** πρεπει να παρει τιμή.

const στις συναρτήσεις

Μια συνάρτηση δυναται να επιστρέψει ένα αντικείμενο τύπου **const**

```
const int f() { int x = 5; return (x); }
```

Η συνάρτηση **f** επιστρέφει ένα αντικείμενο τύπου **const int**

Μια συνάρτηση μπορεί να έχει ως παράμετρο **const** αντικειμενο

1.
```
int Fun(const Myc& ob) { int x=7; return (x); }
```
Η συνάρτηση **Fun** εχει ως παραμετρο **const** αντικείμενο **ob** της ταξης **Myc**
Το αντικείμενο **ob** δεν μπορεί να αλλαξει μέσα στην συνάρτηση **Fun.**

2.
```
class TPC {
   public:
    void h(const TPC& ob) { }
};
```
Η συνάρτηση **h** δεν δύναται να αλλάξει το αντικείμενο **ob**.

Μια συναρτηση εχει το **const** στο τέλος του πρωτυπου των συναρτήσεων

1.
```
class Gen {
   public:
   Gen& g() const { return (*this); }
   //
 };
```
Η συνάρτηση **g** δεν δύναται να αλλάξει μέλη της τάξης **Gen**.

////

-- Επιστροφή από συνάρτηση **const** αντικειμενου.

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

const int w() { int x = 5; return (x); }

void F1(int x) { x=13; }

void F2(const int& x) { }

void F3(int& x) { }

int main()
{
    F1(w());
    F2(w());

    F3(w());// ΛΑΘΟΣ.

    system("pause");
    return 0;
}
```

Η συνάρτηση **w** επιστρέφει ένα **const** αντικειμενο τυπου **int** το οποίο στην συνεχεια περνάει ως παραμετρος στις συναρτησεις **F1, F2, F3**. Η **F1** δουλευει με αντιγραφο του **const** (το **const** αντικειμενο που επιστρεφει η **w** αντιγραφεται στο **x**). Στην κληση της **F2** υπαρχει συμφωνια τυπων (**const int**) στις παραμέτρους

κλησης και συνάρτησης. Η κληση της **F3** δημιουργεί σφαλμα καθως **F3** εχει την δυνατότητα να αλλαξει το καλών αντικειμενο που είναι **const**.
///

-- Παράδειγμα.

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

class A {
    public:
    A() { x = 0;} // Δομητής

    void F1(A& ob) { ob.x = 23; x=44; }

    void F2(const A& ob)
    {
       // ob.x = 17; // ΛΑΘΟΣ. Το αντικειμενο ob είναι const
       x = 23;
    }

    void F3(A& ob) const
    {
       ob.x = 88;
       // x = 2; // ΛΑΘΟΣ. Η συνάρτηση είναι const
    }

    void F4()
    {
        x = 96;
    }

    private:
    int x;
};

int main()
{
    A a,z;
    a.F1(z);
    a.F2(z);
    a.F3(z);
    a.F4();

    const A b;
    // b.F4(); // ΛΑΘΟΣ. Το αντικειμενο b είναι const

    system("pause");
    return 0;
}
```

Στην συνάρτηση **F1**, α) το αντικείμενο **ob** μπορεί να αλλαξει, β) μπορουν να αλλαξουν μελη της ταξης.
Στην συνάρτηση **F2**, α) το αντικείμενο **ob** είναι **const** και δεν μπορεί να αλλαξει, β) μπορουν να αλλαξουν μελη της ταξης.
Στην Συνάρτηση **F3**, α) το αντικείμενο **ob** μπορεί να αλλαξει, β) δεν μπορουν να αλλαξουν μελη της ταξης.
Στην Συνάρτηση **F4** μπορουν να αλλαξουν μελη της ταξης. Αλλα το **const** αντικειμενο **b** δεν μπορεί να την καλεσει **b.F4() // ΛΑΘΟΣ**.

Αρχεία (Files)

-- Το επομενο πρόγραμμα δημιουργεί ένα αρχειο κειμενου (με μια μονο γραμμη) για εισοδο και εξοδο (διαβασμα και γραψιμο).

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

int main()
{
    char fnam[] = "xxx.txt";
    fstream os;
    os.open(fnam,ios::in|ios::out);
    if (!os) {
        cout << "Lathos dimioyrgias arxeioy" << fnam << endl;
        return 1;
    }
    // καταχωρηση στο αρχειο (γραψιμο)
    os << "Thunderbird " << 13 << " " << 15.34 << endl; //

    os.seekg(0L); // Δεικτης get στην αρχή του αρχειου
    char x[100]; int a; float b;
    os >> x >> a >> b; // διαβασμα

    cout << x << " " << a << " " << b  << endl; // εμφανιση

    system("pause");
    return 0;
}
// w07.cpp //
```

Άσκηση: Επεκτεινετε το προηγούμενο πρόγραμμα ώστε να είναι δυνατη η δημιουργία πολλών γραμμών (στο αρχειο) με δεδομένα. Η εισαγωγή των στοιχείων μπορεί να γίνει από το πληκτρολόγιο ή με την χρήση της **rand()**.

-- Το επομενο πρόγραμμα εμφανιζει το περιεχόμενο ενός αρχειου κειμενου που περιεχει ακεραιους.

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

int main()
{
    ifstream ifile ("example_file.txt");
    int tmp;
    while (ifile >> tmp)
        cout << tmp << " ";
    cout << endl;

    system("pause");
    return 0;
}
// w08.cpp
/* αρχειο example_file.txt
```

```
1 2 3 4 5 6 7 8 9 10
*/
```

-- Το επομενο πρόγραμμα εμφανιζει το περιεχόμενο ενός αρχειου στην οθόνη χαρακτηρα-χαρακτήρα.

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

int main()
{
    ifstream ifil ("example_keim.txt");
    char ch;

    while (1) {
        ch = ifil.get();
        if( ifil.eof() ) break;
        cout << ch;
    }
    ifil.close();

    system("pause");
    return 0;
}
////
```

Το ιδιο ακριβώς αποτελεσμα θα παραχθεί αν αντικατασταθει η θηλεια **while** με τις παρακατω δύο εναλλακτικές λύσεις:

```
α)  while ( (ch = ifil.get()) != EOF)
        cout << ch;

β)  while ( ifil.get(ch) )
        cout << ch;
```

```
// w12.cpp //
```

-- Το επομενο πρόγραμμα εμφανιζει το περιεχόμενο ενός αρχειου στην οθόνη γραμμη-γραμμη. Η ροή εισοδου χρησιμοποιείται με το ιδιο τροπο όπως η καθιερωμενη ροή **cin**.

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

int main()
{
    ifstream ifile ("example_keim.txt");
    string tmp;  // char tmp[100]; // OK!
    while (ifile >> tmp) to
        cout << tmp << " "<< endl;

    system("pause");
    return 0;
}
// w10.cpp //
```

Η ροή **ifile** συμπεριφερεται όπως η **cin**, δηλ. σταματαει να διαβαζει όταν συναντηση το κενο διαστημα και να μη διαβαζει τον χαρακτηρα new line (την αλλαγή γραμμης).
////

-- Στο προηγουμενο πρόγραμμα η ροή **ifile** περνάει ως πρωτη παραμετρος στην Συνάρτηση **getline**, η δευτερη παραμετρος είναι τυπου **string**.

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

int main()
{
    ifstream ifile ("example_keim.txt");
    string tmp;
    while (getline(ifile,tmp)) // idio me to fgets or gets
        cout << tmp << " "<< endl;

    system("pause");
    return 0;
}
// w09.cpp //
```

Η συνάρτηση **getline** είναι συνάρτηση της βιβλιοθηκης, δηλ. δεν ανήκει σε καμία ταξη.

-- Το επομενο πρόγραμμα εμφανιζει το περιεχόμενο ενός αρχειου στην οθόνη γραμμη-γραμμη με την χρηση της Συνάρτησης **getline** της ταξης **ifstream**.

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

int main()
{
    ifstream ifile ("example_keim.txt");
    char tmp[30];
    while (ifile.getline(tmp,sizeof(tmp)))
        cout << tmp << " "<< endl;

    system("pause");
    return 0;
}
// w09.cpp //
```

Το πρόγραμμα αυτό «δουλεύει» καλα οσο το μηκος των γραμμων στο αρχειο είναι μικροτερο από 29+1 χαρακτηρες δηλ. μικροτερο από το μεγεθος του (buffer) **tmp**. Σε αρχεια οπου υπαρχουν γραμμες με μηκος μεγαλυτερο από ότι μπορεί να χωρέσει ο buffer το πρόγραμμα παρουσιαζει προβλημα. Για λυση δες το επομενο πρόγραμμα.

-- Το παρακατω πρόγραμμα εμφανιζει το περιεχομενο ενός αρχειου στην οθόνη, το διαβασμα του αρχειου και η εμφανιση γινεται γραμμη-γραμμη, ξεπερνά το προβλημα που περιγραφηκε στα σχόλια του προηγουμενου προγραμματος, δηλ. διαβαζει με επιτυχία αρχεια με οποιοδηποτε μηκος γραμμων.
// Λύση από τον *Ι. Πέτικα*

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char src[]="test.txt";
    cout << "Opening " << src << " for input\n";
    fstream fin(src, ifstream::in);
    if (!fin){
        cout << "Failed to open " << src << "\n";
        system("pause");
        return 0;
    }
    char buf[30];
    unsigned int count=0;
    while (1) {
        fin.getline(buf, sizeof(buf));
        if (fin.fail()) {
           if (fin.eof()) break;
           fin.clear();
           cout << buf;
        }
        else
           cout << buf << " (" << count++ << ")" << endl;
    }
    fin.close();
    cout << "Successfully read " << src << endl;

    system("pause");
    return 0;
}
// w13.cpp //
```

Πρότυπα (Templates)

Τα πρότυπα επιτρέπουν τον ορισμό συναρτήσεων (functions) και τάξεων (classes) που δέχονται ως παραμέτρους τους τα ονόματα των τύπων των δεδομένων.
Επιτρέπουν τον σχεδιασμό συναρτήσεων που χρησιμοποιούν διαφορετικούς τύπους παραμέτρων και τον ορισμό γενικών τάξεων όπου ο τύπος των μελών της τάξης δίνεται ως παράμετρο.

Χρήση προτύπων για δημιουργία αφηρημένων αλγορίθμων (algorithm abstraction).

Α. Γενικές Συναρτησεις

Μια συναρτηση είναι γενική α) εάν επιστρεφει (**return**) γενικο τυπο, β) εάν εχει μια παραμετρο γενικου τυπου, γ) εάν εχει το α) και το β).

1. Η γενική συνάρτηση επιστρεφει γενικό τυπο.

```
template<typename T>
T Func()  {  T t; return(t); }
//
```

Κληση της γενικής συναρτησης με επιστρεφόμενο τυπο – **int** και **double**

```
Func<int>();
```

```
Func<double>();
```

2. Η γενική συνάρτηση εχει μια παραμετρο γενικού τύπου.

```
template<typename X>
int Myf(X w)  {  int t; return(t); }
//
```

Κληση της γενικής συναρτησης με γενική παραμετρο **double**

```
int c = Myf(3.14);
```

3. Η γενική συνάρτηση επιστρεφει γενικό τυπο και εχει δύο παραμέτρους γενικού τύπου.

```
template<typename T,typename X, typename Y>
T Myf(X a,Y b)  {  T t; return(t); }
//
```

Κληση της γενικής συναρτησης **Myf** με παραμέτρους **double**, **int**. Η συναρτηση επιστρέφει τυπο **int**.

```
int c;
c = Myf<int>(3.14,44);
```

Β. Γενικες Τάξεις

1. Δήλωση γενικής ταξης **Gen** και αντικειμενων γενικης ταξης.

```
template < typename X >
class Gen {
    X m1;
};

class UserC { };
```

Δημιουργία αντικειμένων γενικής ταξης **Gen**

```
Gen<int> o1;      // X=int
Gen<char> o2;     // X=char
Gen<UserC > o3;   // X=UserC
```

2. Αρχικοι τύποι σε παραμετρους γενικής ταξης

```
template<typename X,typename Y=int,typename Z=UserC>
class Ginger {
  //
};

class UserC { };
```

Δημιουργία αντικειμένων γενικής ταξης

```
Ginger<int> o1;              // X=int,Y=int,Z=UserC
Ginger<float,UserC> o2;      // X=float,Y=UserC,Z=UserC
Ginger<int, float,char> o3;  // X=int,Y=float,Z=char
```

ΛΑΘΟΣ. δηλώσεις αντικειμενων γενικής ταξης

```
Ginger o4;   // Η Ginger είναι γενική ταξη
Ginger<> o5; // Η παραμετρος Χ πρεπει να παρει τιμη
```

3. Δηλωση «ελάχιστης» γενικής ταξης

```
Template<typename X=int>
class MyTe {
   X a;
```

```
        //
    }
```

Δημιουργια γενικών αντικειμένων

```
    MyTe<> a1;      // X = int
    MyTe<char> c1; // X = char
////
```

-- Δήλωση γενικής ταξης και δημιουργία αντικειμενων γενικης ταξης.

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

template<typename X> // Δήλωση γενικης ταξης
class Far {
    public:
       X w;
};

template<typename T=int> // Δήλωση γενικης ταξης
class G {
    public:
       T k;
};

int main()
{
    Far<int>    ox; // ο τυπος του ox.w είναι int
    Far<double> oy; // ο τυπος του ox.w είναι double
    Far<string> oz; // ο τυπος του ox.w είναι string

    G<> a;      // ο τυπος του a.k είναι int
    G<char> c;  // ο τυπος του a.k είναι char

    system("pause");
    return 0;
}
///
```

-- Η τάξη **Gabs** στο επομενο προγραμμα εχει μια γενική συναρτηση υπερφορτωσης του τελεστη **()**.

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

class Gabs {
    public:
    template<typename T>
    T operator()(T a) { return (a>0 ? a : -a); }
};

int main()
{
    Gabs absolut;

    cout << absolut(-15.6789) << endl; // double
    cout << absolut(-15) << endl;      // integer
```

```
    system("pause");
    return 0;
}
// w23.cpp //
```

-- Παράδειγμα υπερφοτωσης γενικής συναρτησης

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

template <typename T>
T F()                    // Γενική συνάρτηση τυπου T
{
    T t = (T)5;
    return t;
}

int F()                 // τυπου int
{
    int t = 99;
    return t;
}

template<int a>          // Γενική συνάρτηση με γενικό τυπο int
void F()
{
    cout << a << endl;
}

template<typename Z, int a>
void F()                // Γενική συνάρτηση με γενικους τυπους Z και int
{
    Z b = 3.14;
    cout << a+b << endl;
}

int main()
{
    cout << F<int>() << endl;  // 5
    cout << F() << endl;       // 99

    F<33>();                   // 33

    F<float,7>();              // 10.15

    system("pause");
    return (0);
}
// w38.cpp //
```

Αποτέλεσμα

```
5
99
33
10.14
```

typedef

Με την **typedef** οριζουμε συνώνυμα (συμβολικα ονοματα τυπων) σε ήδη γνωστούς τυπους και σημαινουν ακριβως το ιδιο πραγμα σε αυτό το οποίο αναφερονται.

```
A) typedef int* Int_pointer; // Το Int_pointer συνωμυμο του int*
   Int_pointer p;  // Ιδιο με: int* p;
   int* t;         // Δεν καταργείται ο γνωστός τυπος int*
   //
   typedef unsigned char UChar;
   UChar Delta = 'Δ'; // Ιδιο με: unsigned char Delta = 'Δ';

B) class Carwithalongname {
      //
   };
   //
   typedef Carwithalongname Car; // Ορισμός συμωνύμου
   Car ob; // Αντικείμενο τύπου Car

Γ) // ΛΑΘΟΣ. typedef WinnerOfBattle WOB;
   Το ονομα WinnerOfBattle δεν εχει ορισθει ως τυπος με class ή με
   typedef.
   Μια λύση
   typedef char* WinnerOfBattle;
   //
   typedef WinnerOfBattle WOB;
```

-- Παραδειγμα 1ο.

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

typedef unsigned char MyChar;

struct Ayti_h_taxi_exei_megalo_onoma {
    typedef int* Pint;
    //
};

typedef Ayti_h_taxi_exei_megalo_onoma Taxi;

int main()
{
    MyChar a = 'G'; // αντικειμενο a τυπου MyChar
    cout << a << endl;

    Taxi ob; // αντικειμενο ob τυπου Taxi
    Taxi::Pint pi; // αντικειμενο pi τυπου Taxi::Pint

    int b = 33;
    pi = &b;
    cout << *pi << endl;

    system("pause");
```

```
    return 0;
}
// w37.cpp //
```

-- Παραδειγμα 2ο.

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

typedef int Gint; // global

struct A {
    Gint a;
    typedef int saint;
};

struct B : A { // Η B κληρονομεί public την A
    typedef unsigned char NeoChar;
    saint k;
    NeoChar h;
    int p;
};

struct C {
    typedef A TaxiA;
};

int main()
{
    Gint x = 9;

    A::saint y = 22;
    A oba;
    oba.a = 2;

    B::NeoChar c = 'R';
    B::saint w = 6;
    B obb;
    obb.a = 16;
    obb.k = 26;
    obb.h = 'W';

    C::TaxiA it;
    it.a = 29;

    system("pause");
    return 0;
}
// w30.cpp //
```

namespace

Το **namespace** (ονοματοχώρος) ομαδοποιεί συγγενικές δηλώσεις ονοματων (τυπων, συναρτήσεων) για την αποφυγή «συγκρούσεων» στα ονόματα.
-- Παράδειγμα.

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

// Ορισμός του ονοματοχώρου μου
namespace MySp {
    int a;
    void Func(int x)
    {
        a += x;
        cout << "Func of MySp\n";
    }
    class A {
        public:
        int y;
    };
}

// «global» δηλώσεις
int a;

void Func(int x)
{
    a += x;
    cout << "Global Func "<< a << "\n";
}

class A {
    public:
    int y;
};

int main()
{
    // Προσβαση σε μεταβλητές και συναρτήσεις στο MySp
    MySp::a = 5;
    MySp::Func(4);
    cout << MySp::a << endl;
    MySp::A ob;
    ob.y=22;

    // Προσβαση σε global μεταβλητές και συναρτήσεις
    a = 2; // Ιδιο με: ::a = 2;
    ::Func(13); // Ιδιο με: Func(13);
    cout << a << endl;
    A ox; // Ιδιο με: ::A ox;
    ox.y=44;

    system("pause");
    return 0;
}
// w28.cpp //
```

using και using namespace

Η δηλωση **using** βοηθάει την προσβαση σε ονοματοχώρους. Για την αποφυγή επαναληψης του ονοματοχώρου **std** (για προσβαση σε γνωστά ονόματα όπως **std::cout, std::cin std::string, std::vector**) χρησιμοποιήται η οδηγια **using namespace.** Η **using namespace** είναι οδηγία στον μεταφραστή να «αναζητήσει στον ονοματοχώρο **std** ονόματα που δεν εχουν δηλωθεί».
-- Το προηγουμενο παράδειγμα με την χρηση του **using.**

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

// Ορισμός του ονοματοχώρου μου
namespace MySp {
    int a;
    void Func(int x)
    {
        a += x;
    }
    class A {
        public:
        int y;
    };
}

// «global» δηλώσεις
int a;

void Func(int x)
{
    a += x;
}

class A {
    public:
    int y;
};

int main()
{
    // Προσβαση σε μεταβλητές και συναρτήσεις στο MySp
    using MySp::Func;
    using MySp::a;
    a = 5;
    Func(4);
    cout << a << endl;

    using MySp::A;
    A ob;
    Ob.y = 22;

    // Προσβαση σε global μεταβλητές και συναρτήσεις
    ::a = 2;
    ::Func(13);
    cout << ::a << endl;
    ::A ox;
    ox.y = 44;
```

```
    system("pause");
    return 0;
}
// w29.cpp //
```

STL – Standard Template Library

Η STL της C++ καθιερώθηκε το 1999 και λύνει πολλά προβλήματα που αφορούν γνωστές δομές και αλγορίθμους. Αποτελείται από **πέντε τύπους αντικειμενών containers**, **αλγορίθμους**, **επαναλήπτες** (iterators), **συναρτήσεις ακτικειμένων**, (functors – function objects), **adaptors** (προσαρμογείς) και από **γενικές συναρτήσεις** (αλγορίθμους).

Κάποιοι αλγορίθμοι υλοποιούνται και στα containers ως συναρτήσεις μέλη και στην STL με το ίδιο όνομα. Τα containers διαθέτουν συναρτήσεις μέλη (member functions) όπως **`find`**, **`sort`**, **κ.ά.** Συναρτήσεις με το ίδιο όνομα και την ίδια λειτουργία υπάρχουν και στους αλγορίθμους της STL (συναρτήσεις που δεν είναι μέλη σε κάποια τάξη). Σε σύγκριση με τους αλγορίθμους, οι συναρτήσεις μέλη είναι σε ορισμένες περιπτώσεις πιο αποτελεσματικές όσο αφορά την ταχύτητα και «δένουν» καλύτερα με τα containers.

Containers

Γενικά, ο όρος container αναφέρεται στο πακετάρισμα (packaging) και στην ετικετοποίηση (labeling) προϊόντων για αποθήκευση και μεταφορά.

Στην **C++** ένα container είναι ένα αντικείμενο που εμπεριέχει μια συλλογή από άλλα αντικείμενα (στοιχεία - elements του container).

Η υλοποίηση τους γίνεται με τάξεις πρότυπα (class templates) που δίνουν την δυνατότητα να χρησιμοποιηθούν για οποιοδήποτε τύπο στοιχείων.

Ένα container **α)** διαχειρίζεται το αποθηκευτικό χώρο για τα στοιχεία του, **β)** παρέχει μεθόδους πρόσβασης στα στοιχεία του είτε απ' ευθείας είτε μέσω ειδικών δεικτών (smart pointers) που ονομάζονται iterators (iterators είναι αναφορές σε αντικείμενα με ιδιότητες ίδιες με τους δείκτες).

Τα containers ξαναδημιουργούν δομές που χρησιμοποιούνται συχνά στον προγραμματισμό όπως δυναμικούς πίνακες (**`vector`**), ουρές (**`queue`**), στοιβες (**`stack`**), συνδεδεμένες λιστες - linked lists - (**`list`**), δέντρα (**`set`**), συνδεδεμένους (συνδυασμένους) πίνακες - associative arrays - (**`map`**).

Οι σειριακοι (sequence) containers **`vector`**, **`list`**, και **`deque`** αποθηκευουν τα στοιχεια τους σε συνεχόμενες θεσεις το ένα διπλα στο άλλο.

Οι συγγενικοί (συνδεδεμένοι) associative containers **`set`** και **`map`** αποθηκευουν τα στοιχεια τους με τετοιον τροπο ώστε να είναι δυνατη η προσβαση τους με την χρήση *κλειδιών*.

Σειριακοί προσαρμογείς (adaptors, όπως stacks, queues και priority queues) είναι τύποι δεδομένων της STL που προσαρμόζουν ένα container για να παρεχει μια ειδική διεπαφή (interface). Για παραδειγμα το **stack** παρέχει διεπαφή στοίβας πανω σε ένα επιλεγμένο container που είναι συνήθως το **deque**.

Εξειδικευμένοι containers (**string**, **bitset**) προσφέρουν περιορισμένη αλλα αποτελεσματική επεξεργασία σε ειδικούς τύπους δεδομένων.

Οι πίνακες ανήκουν στον πυρήνα τους γλώσσας και είναι απλοί containers ενώ τα containers ανήκουν στην καθιερωμένη βιβλιοθήκη (standard library). Όπως οι πίνακες έτσι και τα containers περιέχουν ακολουθίες από αντικείμενα του ιδίου τύπου. Τα containers μπορούν να αυξήσουν ή να μειώσουν το μέγεθός τους με ευκολία ενώ οι πινακες είναι αδύνατο (στατική δήλωση πινακων) ή προγραμματιστικά χρονοβόρο.

Κάθε σειριακος container υποστηρίζει λειτουργίες που μας επιτρέπουν

α) να προσθέσουμε στοιχεία
β) να διαγραψουμε στοιχεία
γ) να βρουμε το μεγεθος
δ) να ανακτησουμε το πρωτο και το τελευταίο στοιχείο (εάν υπαρχουν)

Iterators (επαναλήπτες)

Οι επαναλήπτες (iterators) είναι αντικείμενα τα οποία δρούνε όπως οι δείκτες. Οι επαναλήπτες παρέχουν έναν ομοιόμορφο τρόπο πρόσβασης στα στοιχεία ενός container της STL καθώς και στα στοιχεία ενός πίνακα χωρίς να απαιτείται η λεπτομερής γνώση της εσωτερικής οργανωσης της δομής. Υπάρχουν iterators για σειριακή προσβαση με βηματισμό μπροστά και/ή πίσω, καθώς και iterators για τυχαία πρόσβαση. Οι iterators χρησιμοποιούν τους τελεστές **++**, **--**, και ***** που είναι γνωστοί στην χρήση των δεικτών.

Ένας δεικτης (pointer) είναι ένα είδος επαναλήπτη τυχαίας προσβασης.

```
int a[100], *p, n=32;
p = a + n-1; // οπου n>=1 και n<=100
```

Ο **p** δείχνει στο **n** (32ο) στοιχείο του πίνακα **a**, οπότε το **p+ακ** (**ακ**=ακέραιος) δείχνει στο **n+ακ** στοιχείο του πίνακα και **p-ακ** δείχνει στο **n-ακ** στοιχείο του πίνακα με την προυπόθεση ότι τα στοιχεία αυτά υπάρχουν δηλ. ο **ακ** βρίσκεται μέσα στα όρια του **n**.

```
vector<double> v; // Δήλωση πίνακα v
vector<double>::iterator itr; // Δήλωση iterator
// Η μεταβλητή itr είναι ένας iterator τυπου vector.
for(itr = v.begin(); itr != v.end(); ++itr) {
   // ...
}
```

Η θηλειά (loop) επιτρέπει την σειριακή πρόσβαση σε όλα τα στοιχεία του πίνακα.
Η συνάρτηση **end()** επιστρέφει έναν επαναλήπτη όπου το **end()-1** δείχνει στο τελευταίο στοιχείο του πίνακα **v**.

Κατηγορίες Επαναληπτών

Οι επεναλήπτες χωριζονται σε πεντε κατηγορίες ανάλογα με τις λειτουργίες που προσφέρουν

Οι ***επαναλήπτες εισόδου*** (input iterators) επιτρέπουν στον χρήστη να προχωρήση τον επαναλήπτη (με χρήση του τελεστή ++), να διαβάσει (ανακτήσει) την τιμή του στοιχείου που δείχνει ο δείκτης (με χρήση του τελεστή *).

Οι ***επαναλήπτες εξόδου*** (output iterators) επιτρέπουν στον χρήστη να προχωρήση τον επαναλήπτη (με χρήση του τελεστή ++), να δώσει (απονέμει) τιμή στο στοιχείο που δείχνει ο δείκτης (με χρήση του τελεστή *).

Οι ***επαναλήπτες μιας κατεύθυνσης*** (forward iterators) επιτρέπουν στον χρήστη να προχωρήση τον επαναλήπτη μόνο προς μια κατευθυνση, να διαβάσει (ανακτήσει) και να δώσει (απονέμει) τιμή στο στοιχείο που δείχνει ο δείκτης.

Οι ***επαναλήπτες δύο κατευθύνσεων*** (bidirectional iterators) επιτρέπουν στον χρήστη να προχωρήση τον επαναλήπτη και προς τις δύο κατευθύνσεις (χρήση των τελεστων **++** και **--**), να διαβάσει (ανακτήσει) και να δώσει (απονέμει) τιμή στο στοιχείο που δείχνει ο δείκτης.

Οι ***επαναλήπτες τυχαίας πρόσβασης*** (random access iterators) επιτρέπουν στον χρήστη να προχωρήση τον επαναλήπτη πολλαπλές θέσεις και όχι μόνο κατά μια θέση (χρήση αριθμητικής των δεικτών και των τελεστων **++** και **--**), να διαβάσει (ανακτήσει) και να δώσει (απονέμει) τιμή στο στοιχείο που δείχνει ο δείκτης. Είναι οι περισσότερο πλήρεις επαναλήπτες όσο αφορά την λειτουργικότητα τους.

Κατηγορίες των iterators	*Λειτουργίες*
Input iterators	**a=b, b(a),** **a==b, a!=b,** **=*a,** **a++, ++a, *a++**
Output iterators	**a=b, b(a),** **a++, ++a, *a++,** ***a=t**
Forward iterators	**X a, a=b, b(a),** **a==b, a!=b,** **=*a, a->m=,** **a++, ++a, *a++**
Bidirectional iterators	**X a, a=b, b(a),** **a==b, a!=b,** **=*a, a->m=,** **a++, ++a, *a++, a--, --a, *a--**
Random access	**X a, a=b,** **a==b, a!=b,** **=*a, a->m,** **a++, ++a, *a++, a--, --a, *a--.** **=a+n, a+n=, = a<b, a>b=, a+=n, a-=,** **= a <= b, a >= b=**

1. Επιτρέπεται η αποαναφοροποίηση (*****) όταν ο επαναλήπτης δεν είναι null.
2. **X a** (εξ'ορισμού δομητής)
3. Όπου **X** είναι ενας τυπος επαναλήπτη, τα **a** και **b** είναι αντικείμενα τυπου **X** και **t** είναι αντικείμενο του τυπου που δείχνεται από τον επαναλήπτη.

Τυποι επαναληπτών

iterator (επαναλήπτης). Μπορεί να ανακτήσει και/ή να αλλαξει το αντικειμενο που δειχνει ο επαναλήπτης.

const_iterator (σταθερός επαναλήπτης). Μπορεί να μονο ανακτήσει (δεν μπορεί να αλλαξει) το αντικειμενο που δειχνει ο επαναλήπτης.

reverse_iterator (αντίστροφος επαναλήπτης). Μπορεί να ανακτήσει και/ή να αλλαξει το αντικειμενο που δειχνει ο επαναλήπτης.

reverse_const_iterator (αντίστροφος σταθερός επαναλήπτης). Μπορεί να μονο ανακτήσει (δεν μπορεί να αλλαξει) το αντικειμενο που δειχνει ο επαναλήπτης.

Εκτός των παραπανω τύπων επαναληπτων που υποστηρίζονται από την καθιερωμένη βιβλιοθηκη STL ενας χρηστης μπορεί να ορίσει δικους του τύπους επαναληπτων (class *επαναλήπτη*) για τους δικους του τύπους δεδομένων ή να ορίσει επεναλήπτες που μημούνται την λειτουργία των επεναληπτών της STL.

Δήλωση επαναληπτών

Σε κάθε container τάξη δημιουργείται ένα **typedef** για κάθε τυπο επεναλήπτη. Ο γενικός τύπος δήλωσης επαναληπτών έχει την μορφή

ονομα_container<τυπος>::τυπος_επαναληπτη ονομα_επαναληπτη;

1. **vector<char>::reverse_iterator ritr;**

Ο **ritr** είναι επαναληπτης που μπορεί να χρησιμοποιηθεί σε **vector** container με αντικειμενα τυπου **char**.

2. **list<double>::const_iterator citr;**

Ο **citrA** είναι ενας **iterator** και μπορει να χρησιμοποιηθεί σε **list** container με αντικειμενα τυπου **double**.

3. **deque<string>::iterator itr;**

Ο **itrB** είναι ενας **iterator** και μπορει να χρησιμοποιηθεί σε **deque** container με αντικειμενα τυπου **string**.

4. **vector<MyClass>::iterator it;**

Ο **it** είναι επαναληπτης που μπορεί να χρησιμοποιηθεί σε **vector** container με αντικειμενα τυπου **MyClass**.

////

-- Παραδειγμα απλών επαναλήπτων σε πινακα ακεραίων.

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

int main()
{
    int a[] = { 3, 9, 34, 22, 62 };

    typedef int* iter;
    iter it = a, rit;  // επαναλήπτες
```

```
    // δηλωση δεικτων
    int* beg = a;
    int* end = a + 5;
    int* rbeg = a + 4;
    int* rend = a - 1;

    for( it = beg; it != end; ++it)
         cout << *it << " ";
    cout << endl;

    // εμφανιση πινακα αντιστροφα
    for( rit = rbeg; rit != rend; --rit)
         cout << *rit << " ";
    cout << endl;

    system("pause");
    return 0;
}
// w39.cpp //
```

Αποτελεσμα

```
3 9 34 22 62
62 22 34 9 3
```

-- Δημιουργία ταξη-επαναλήπτη.

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

class iter {
    int* p;
     public:
    iter() { p = 0; } ; // Δομητής

    // Υπερφορτωση τελεστών
    iter& operator=(int *x) { p = x; return *this; }
    iter& operator++() { ++p; return *this;}
    bool operator!=(const int* ptr) { return p!= ptr; }
    int& operator*() { return *p;}
};

int main()
{
    int a[] = { 3, 9, 34, 22, 62 };
    int* beg = a;
    int* end = a + 5;
    int* rbeg = a + 4;
    int* rend = a - 1;

    iter it; // Δήλωση επαναλήπτη it

    for( it = beg; it != end; ++it)
         cout << *it << " ";
    cout << endl;

    system("pause");
    return 0;
```

```
}
// w40.cpp //
```
Αποτελεσμα
```
3 9 34 22 62
```

-- Δήλωση ενός πίνακα χαρακτήρων με **vector<char>** και ενός **iterator**.
Γεμισμα του πινακα με χαρακτηρες και εμφανιση του πινακα στην οθονη.
```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // Δήλωση vector που κρατα 26 char αντικειμενα
    vector<char> v(26);

    // Δήλωση iterator τυπου vector<char>.
    vector<char>::iterator itr;

    // Γεμισμα του v.
    int i=0;
    for(itr = v.begin(); itr != v.end(); itr++)
            *itr = 'A'+i++;

    cout << "Emfanisi v: ";
    for(itr=v.begin(); itr != v.end(); ++itr)
        cout << *itr << " ";
    cout << "\n";

    system(pause);
    return(0);
}
// w11.cpp //
```
Αποτελεσμα
```
Emfanisi v: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Function Objects

Συναρτήσεις αντικείμενων (*function objects* ή *functors*) είναι τα αντικείμενα μιας ταξης που διαθέτει την συνάρτηση **operator()** – στην **class** υπερφορτώνεται ο τελεστής **operator()** – τα αντικείμενα αυτά αν και συμπεριφέρονται σαν συναρτήσεις στην πραγματικότητα είναι κανονικά αντικείμενα.

Η C++ εκτός από δείκτες σε συναρτήσεις διαθέτει συναρτήσεις αντικειμένων (function objects) που είναι μια καλύτερη επιλογή από τους δείκτες σε συναρτήσεις.

Δείκτες σε συναρτήσεις χρησιμοποιούνται για την υλοποίηση σύνδεσης του προγράμματος με συναρτήσεις (πχ. με συναρτήσεις της βιβλιοθήκης) στην C ονομάζονται ***callbacks***.

Στην STL χρησιμοποιούνται συναρτήσεις αντικειμένων αντί των δεικτών σε συναρτήσεις.

Υπάρχουν μερικά προκαθορισμένα αντικείμενα συναρτήσεων (functors) όπως **less(), greater(), plus(), minus()**, **multiplies(),** και **divides()**. Το πιο διαδεδομένο είναι το **less()** το οποίο καθορίζει πότε ένα αντικείμενο είναι μικρότερο από ένα άλλο.

Οι συναρτήσεις αντικειμένων που επιστρέφουν Boolean τιμή ονομαζονται ***predicates*.**

Τα αντικείμενα που συμπεριφερόνται ως συναρτήσεις αυξάνουν την αποτελεσματικότητα σε ορισμένους τύπους λειτουργιών και παρέχουν υποστηριξη σε διαφορες λειτουργίες οπου θα ήταν αδυνατες μόνο με την χρήση δεικτών σε συναρτήσεις.

Αν ο μεταφραστής (compiler) συναντήσει μια εντολή όπως πχ.

```
Lt(iv);
```

καταλαβαίνει ότι το **Lt** ειναι **ή** το ονομα μιας συνάρτησης **ή** ενας δεικτης σε συνάρτηση **ή** ένα ανώνυμο αντικείμενο μιας τάξης **ή** ένα αντικειμενο (function object) μιας ταξης οπου εχει ορισθεί ο τελεστης **()**.
Εάν το **Lt** είναι αντικειμενο συνάρτησης τοτε μεταφραζει την εντολη σε

```
Lt.operator()(iv);
```

Το αντικείμενο **Lt** θα μπορουσε να είναι ένα αντικείμενο μιας ταξης **LessThan** που δημιουργείται από

```
LessThan Lt(10);
```

Η ταξη **LessThan** εχει μια μεταβλητή **v** (που κρατα μια τιμη για συγκριση), έναν δομητη, μια συνάρτηση απονομης, μια συνάρτηση ανακτησης και μια συνάρτηση υπερφορτωσης του **operator()**

```
class LessThan {
    public:
        LessThan(int w) : v(w){}
        int getv() const { return v; }
        void setv(int nv){ v = nv; }
        bool operator()(int val) const { return (val < v);}
    private:
        int v;
};
```

-- Παραδειγμα δεικτη σε συνάρτηση - function pointer

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

void F(bool (*fn)())
{
   fn();
}

bool Fa()
{
    cout << "Hi there" << endl;
    return true;
}

bool Fb()
{
```

```
    cout << "Thrill has gone" << endl;
    return true;
}

int main()
{
    bool (*pf)();
    pf = Fa;  // Idio me: pf = &Fa;

    F(pf);  // Κληση της Fa
    pf = Fb;
    F(pf);  // Κληση της Fb
    F(Fa);

    system("pause");
    return 0;
}
```

Αποτελεσμα

```
Hi there
Thrill has gone
Hi there
```

Ο **pf** και **fn** είναι δεικτες σε **bool** συνάρτηση χωρίς παραμετρους.

```
////
```

-- Ανώνυμο αντικείμενο τύπου **A** περνάει ως παράμετρος σε συνάρτηση

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

class A {
    public:
    bool F()
    {
        cout << "Hi there" << endl;
        return true;
    }
};

void test(A tmp)
{
   tmp.F();
   return;
}

int main()
{
    test( A() ); // Ανώνυμο αντικειμενο τυπου A
    system("pause");
    return 0;
}
////
```

-- Η συνάρτηση test του προηγούμενου προγράμματος γινεται γενική.

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

```
class A {
    public:
    bool F()
    {
        cout << "Hi there" << endl;
        return true;
    }
};

class B {
    public:
    bool F()
    {
        cout << "Thrill has gone" << endl;
        return true;
    }
};

template <typename T>
void test(T tmp)
{
   tmp.F();
   return;
}

int main()
{
    test( A() );
    test( B() );
    system("pause");
    return 0;
}
```

Αποτελεσμα

```
Hi there
Thrill has gone
////
```

-- Παραδειγμα δημιουργιας συνάρτησης αντικειμένου (function object) .

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

class A {
   public:
      bool operator() ()
      {
          cout << "Hi there" << endl;
          return true;
      }
};

int main()
{
    A a;
    a();   // a.operator()();
    cout << a() << "\n";
```

```
    system("pause");
    return 0;
}
// w14.cpp //
```
Αποτελεσμα
```
Hi there
Hi there
1
```
Στην ταξη **A** εχει υπερφορτωθεί ο τελεστής **() -operator()**- με συνάρτηση τυπου **bool**. Το αντικείμενο **a** τυπου **A** συμπεριφέρεται σαν συναρτήση (είναι ενας functor).
////

-- Συνάρτηση αντικειμένου επιστρέφει την μεγαλύτερο τιμή από τις δυο **double** παραμέτρους.
```
#include <iostream>
using namespace std;

class Cmeg {
    public:
    double operator()(double a, double b) { return (a>b ? a : b); }
};

int main()
{
    Cmeg maxob; // δημιουργία αντικείμενου συναρτησης maxob

    // κλήση της operator()
    cout << maxob(-15.6789, 444.4)      << endl;
    cout << maxob.operator()(-9.6,-123) << endl;

    system("pause");
    return 0;
}
// w24.cpp //
```
Αποτέλεσμα
```
444.4
-9.6
////
```

-- Στο επόμενο πρόγραμμα μια συνάρτηση **metr_mikr_apo** με την βοηθεια ενός αντικειμένου **Lt** της ταξης **LessThan** μετρά σε έναν πίνακα ακεραίων ποσοι ακέραιοι αριθμοί είναι μικρότεροι από μια συγκεκριμένη τιμή.

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

class LessThan {
    public:
        LessThan(int w) : v(w){}
        int getv() const { return v; }
        void set_v(int nv){ v = nv; }
        bool operator()(int val) const ; // Υλοποιηση εκτος ταξης
    private:
        int v;
};
```

```
bool LessThan::operator()(int val) const
{
    return val < v;
}

int metr_mikr_apo(const int arr[], int n, int cmp)
{
    LessThan Lt(cmp); // cmp είναι η τιμή για συγκριση

    int cnt = 0;
    // Κάθε στοιχείο του πινακα συγκρινεται με την τιμη cmp
    for (int ix = 0; ix < n; ++ix)
        if (Lt(arr[ix]))
            ++cnt;
    return cnt;
}

int main()
{
    int a[10] = { 4, 2, 8, 5, 9, 10, 7, 3, 1, 6 };
    // Κάθε στοιχείο του πινακα a συγκρινρεται με την ακεραια τιμη 7
    cout << metr_mikr_apo(a,10,7) << endl;

    system("pause");
    return 0;
}
// w01.cpp //
```
Αποτέλεσμα
```
6
////
```

Προσαρμογείς (Adapters)

Υπάρχουν μερικές classes όπως **stack** και **queue** που δρούνε ως «wrappers» γύρω από άλλους containers και υλοποιούνται ως *container adaptors* (προσαρμογείς). Αυτό σημαίνει ότι παρέχουν την διεπαφή (interface) για μια στοιβα ή για μια ουρά αλλά στην ουσία δεν είναι containers. Χρησιμοποιούν κάποιο άλλο container (underlying container) όπως το **deque** ή **list** για να σώζουν τα δεδομένα. Η ταξη stack με πολύ λιγο κωδικα μεταφράζει την push και pop σε push_back και pop_back. Η ταξη queue κανει περίπου το ίδιο αλλά χρησιμοποιεί την push_front και την pop_back.

Ο προγραμματιστής μπορεί να ορίσει το τύπο δεδομένων, συνήθως είναι ένα container **deque** ή **list.**

Η απλή ουρά μπορεί να δημιουργηθεί με δύο τρόπους

```
#include <queue>
#include <deque>
```

αλλά η στοίβα μπορεί να δημιουργηθεί μονο με τον εξης τρόπο

```
#include <stack>
```

Η στοίβα (**stack** – απλή ουρά - single-ended FIFO) και η ουρά (**stack** – στοίβα LIFO) είναι containers υψηλοτερου επιπέδου από ότι η deque, η list, και το vector, το

οποίο σημαίνει ότι μπορεί να δημιουργηθεί μια στοίβα ή μια ουρά από containers χαμηλότερου επιπέδου.

```
stack<int, deque<int> > s;
queue<double, list<double> > q;
```

Δημιουργία μιας στοίβας **s** από ints με την χρήση μιας deque ως το underlying container και μιας ουράς **q** από doubles με την χρηση μιας list ως the underlying container.

Μπορούμε να θεωρήσοτμε την **s** ως μια restricted deque και την **q** as a restricted list

Αυτό που σημβαίνει είναι ότι τα container χαμηλότερου επιπέδου υλοποιούν τις μεθόδους που χρειαζονται από τα container υψηλότερου επιπέδου. Αυτές είναι οι **back()**, **push_back()** και **pop_back()** για το stack και οι **front()**, **back()**, **push_back()** και **pop_front()** για queue.

Οσο αφορά το container deque, είναι πολύ περισσότερο από μια ουρά στην οποία μπορείς εισάγεις στοιχεία και στα δυο ακρα. In particular, it has the random access operator[]. This makes it more like a vector, but a vector where you can insert and delete at the beginning with push_front() and pop_front().

Αλγόριθμοι

Πολλοί γνωστοί αλγόριθμοι έχουν υλοποιηθεί για τα στοιχεία των STL containers και των πινάκων (arrays) όπως ταξινόμησης (sorting), αναζήτησης (searching), εισόδου (inserting), διαγραφής (deleting), αντιστροφής (reversing), αντιγραφής (copying), μετακινησης (moving), μέτρισης (counting), μέγιστου (minimum), ελάχιστου (maximum), ανακατέματος (shuffling) κλπ.
Η καθιερώμενη βιβλιοθήκη περιλαμβάνει περίπου 60 αλγόριθμούς από τους οποίους άλλοι είναι γνωστοί και άλλοι λιγότερο γνωστοί.

Οι γενικοί αλγόριθμοι (generic algorithms) ορίζονται στο αρχειο κεφαλίδων

```
#include <algorithm>
```

binary_search (δυαδική αναζήτηση)

```
binary_search(b, e, t)
```

Επιστρεφει **bool** που δειχνει αν η τιμή **t** είναι στην ταξινομημένη ακολουθία που οριζεται από τους επαναλήπτες **b** και **e**.

copy (αντιγραφή)

```
copy(b, e, d)
```

Αντιγράφει τις τιμές από μια ακολουθία που οριζεται από επαναληπτες εισόδου (input iterators) **b** and **e** στον χώρο προορισμό που οριζεται από τον επαναλήπτη εξόδου (output iterator) **d**. Η συναρτηση υποθέτει ότι στο προορισμό υπαρχει ελευθερος χώρος να κρατήσει τις τιμές που θα αντιγραφούν. Η συνάρτηση επιστρεφει μια τιμή που δειχνει στην επόμενη θέση μετα το τέλος του τελευταίου στοιχείου στο χωρο προορισμό.

fill (γέμισμα)

```
fill(b, e, t)
```

Γεμίζει μια ακολουθία με τιμή **t** στην περιοχή που ορίζεται από τους επαναλήπτες εισόδου (input iterators) **b** και **e**.

find (εύρεση)

```
find(b, e, t)
find_if(b, e, p)
```

Επιστρέφει μια τιμή επαναλήπτη που δειχνει στο σημείο που βρισκεται (πρώτη φορά) η τιμή t ή για την τιμή που το predicate p είναι true αληθής στην περιοχή που ορίζεται από τους επαναλήπτες εισόδου input iterators b και e. Επιστρέφει e εάν δεν υπαρχει η τιμή t στην ακολουθία.

remove (διαγραφή)

```
remove(b, e, t)
remove_if(b, e, p)
```

search (αναζητηση)

```
search(b, e, b2, e2)
search(b, e, b2, e2, p)
```

Επιστρεφει έναν επαναλήπτη που δειχνει στο σημείο που βρισκεται (πρώτη φορά) η υποακολουθία που ορίζεται απο τους forward iterators b2 and e2 μέσα στην ακολουθία που οριζεται από τους forward iterators b and e. Χρησιμοποιείται το predicate p για τον ελεγχο ή ο τελεστης == όταν το p δεν διδεται.

sort (ταξινόμηση)

```
sort(b, e)
sort(b, e, p)
```

Ταξινομεί την ακολουθία που ορίζεται από τους επαναλήπτες τυχαίας προσβασης random-access iterators b and e. Χρησιμοποιείται το predicate p για τον ελεγχο ή ο τελεστης < όταν το p δεν διδεται.

-- Η συνάρτηση **sort** της βιβλιοθήκης ταξινομεί έναν πινακα.

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

int main()
{
    vector<int> v(10);  // Πινακας
    for(int i=0; i<v.size(); i++)
        v[i] = rand()%100 + 1; // Γεμισμα του πινακα με τυχαίους ακεραιους

    cout << "    Emfanisi: ";
    for(int i=0; i<v.size(); i++)
        cout << v[i] << ' ';
    cout << endl;

    sort(v.begin(),v.end()); // Ταξινόμηση

    cout << "Taxinomimeno: ";
    for(int i=0; i<v.size(); i++)
```

```
        cout << v[i] << ' ';
    cout << endl;

    system("pause");
    return 0;
}
// w31.cpp //
```
Αποτελεσμα
```
    Emfanisi: 42 68 35 1 70 25 79 59 63 65
Taxinomimeno: 1 25 35 42 59 63 65 68 70 79
```

-- Η συνάρτηση **find** της βιβλιοθήκης με την χρήση ενός iterator αναξητεί ένα συγκεκριμένο στοιχειο σε έναν πινακα.
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    //int a[] = { 5, 2, 8, 6, 9 };
    vector<int> v(10);

    for(int i=-3; i<7; i++)
        v[i+3] = i;

    cout << "    Emfanisi: ";
    for(int i=0; i<v.size(); i++)
        cout << v[i] << ' ';
    cout << endl;

    //sort(v.begin(),v.end());
    vector<int>::iterator it;
    it = find(v.begin(),v.end(),5);
    if (it != v.end())
        cout << "O aritmos 5 brethike ston pinaka\n";

    it = find(v.begin(),v.end(),-44);
    if (it == v.end())
        cout << "O aritmos 44 DEN brethike ston pinaka\n";

    system("pause");
    return 0;
}
// w32.cpp //
```

vector

Η ταξη **vector** δημιουργεί δυναμικούς πινακες οποιουδηποτε τυπου και υποστηρίζει τυχαία προσβαση στα στοιχεια του πινακα με την χρήση επαναληπτών τυχαίας προσβασης (random access iterators). Είναι το πιο χρησιμο container της καθιερωμένης βιβλιοθήκης.
```
    #include <vector>
```

Το αρχείο κεφαλίδων ορίζει την τάξη και οτιδήποτε χρειάζεται για να χρησιμοποιήσουμε τον container **vector**.

-- Παραδειγμα.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> vs;

    vs.push_back("King");
    vs.push_back("Queen");
    vs.push_back("Princess");
    cout << "Plithos stoixeiwn sto vector " << vs.size() << endl;

    cout << endl << "Xrisi deikti i:" << endl;
    for(int i=0; i < vs.size(); i++) {
        cout << vs[i] << ' ';
    }
    cout << endl;
    /*
    // Gemisma toy vector apo to plhktrologio
    for(ii=0; ii < vs.size(); ii++) {
        cin >> vs[ii];
    }
     */

    cout << endl << "Xrisi const epanalipti:" << endl;
    vector<string>::const_iterator ci;
    for(ci=vs.begin(); ci!=vs.end(); ci++) {
        cout << *ci << " ";
    }
    cout << endl;

    cout << endl << "Xrisi toy [] anti toy * :" << endl;
     for (int i=0; i<vs.size(); ++i) {
       cout << vs.begin()[i] << ' ';
    }
    cout << endl;

    cout << endl << "Xrisi anastrofoy epanalipti:" << endl;
    vector<string>::reverse_iterator ri;
    for(ri=vs.rbegin(); ri!=vs.rend(); ++ri) {
        cout << *ri << " ";
    }
    cout << endl;

    cout << endl << "3o stoixeio: ";
    cout << vs[2] << endl;
    cout << "Antallagi 1ou me to 3o" << endl;
    swap(vs[0], vs[2]);
```

```
    cout << "3o stoixeio: ";
    cout << vs[2] << endl;

    system("pause");
    return 0;
}
// w27.cpp //
```

Αποτέλεσμα

```
Plithos stoixeiwn sto vector 3

Xrisi deikti i:
King  Queen  Princess

Xrisi const epanalipti:
King  Queen  Princess

Xrisi toy [] anti toy * :
King  Queen  Princess

Xrisi anastrofoy epanalipti:
Princess  Queen  King

3o stoixeio: Princess
Antallagi 1ou me to 3o
3o stoixeio: King
```

-- Ο αλγόριθμος **sort()** παίρνει ως τρίτη παράμετρο ένα *predicate αντικείμενο* (predicate αντικείμενο είναι μια γενική συναρτηση αντικείμενου (function object) που επιστρέφει μια boolean τιμή). Στο επόμενο παραδειγμα χρησιμοποιούνται τα predicates **greater<>** ή **less<>** στην **sort()** για αυξουσα ή φθινουσα ταξινομηση των τιμών σε ένα **vector**.

```
#include <iostream>
#include <functional> // για το greater<> και το less<>
#include <algorithm>  // για το sort()
#include <vector>
using namespace std;

int main()
{
    vector <int> vi;
    vi.push_back(2);
    vi.push_back(5);
    vi.push_back(7);
    vi.push_back(1);

    //sort(vi.begin(), vi.end(), greater<int>() );//descending
    sort(vi.begin(), vi.end(), less<int>() ); //ascending

    vector<int>::iterator it;
    for(it = vi.begin(); it!= vi.end(); ++it)
        cout << *it << " ";
    cout << endl;

   system("pause");
   return 0;
}
```

```
// w26.cpp //
```
Αποτελεσμα
```
1 2 5 7
```

list

Η ταξη **list** δημιουργεί δυναμικές διπλά συνδεδεμένες λιστες οποιουδηποτε τυπου και υποστηρίζει προσβαση στα στοιχεια του πινακα με διπλή κατεύθυνση (από την αρχή της λιστας προς το τέλος και από το τέλος προς την αρχή) με την χρήση επαναληπτών διπλής κατεύθυνσης (bidirectional iterators).

#include <list>

Το αρχειο κεφαλίδων οριζει την ταξη και οτιδήποτε χρειαζεται για να χρησιμοποιήσουμε τον container **list**.

--- Παράδειγμα
```
#include <iostream>
#include <list>
using namespace std;

// Simple example uses type int

main()
{
   list<int> L;
   L.push_back(0);         // Τοποθετησε το 0 (new element) στο τέλος
   L.push_front(0);        // Τοποθετησε το 0 (new element) στην  αρχή
   L.insert(L.begin(),2);// Τοποθετησε το  2  πριν από το πρωτο στοιχείο

   L.push_back(5);
   L.push_back(6);

   list<int>::iterator it;
   for(it=L.begin(); it != L.end(); ++it)
      cout << *it << " ";
   cout << endl;

   return 0;
}
// w43.cpp //
```
Αποτέλεσμα
```
2 0 0 5 6
```

--- Παράδειγμα
```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    for (int i=1; i<=9; ++i) // εισαγωγη στοιχείων από το 1 ως το 9
```

```
        coll.push_back(i);

    list<int>::iterator pos = coll.begin();
    cout << *pos << endl; // εμφάνιση τρεχοντος στοιχείου

    advance (pos, 3); // προχώρα μπροστα κατά τρια στοιχεία
    cout << *pos << endl; // εμφάνιση τρεχοντος στοιχείου

    advance (pos, -1); // προχώρα προς τα πίσω κατά ένα στοιχείο
    cout << *pos << endl; // εμφάνιση τρεχοντος στοιχείου

    system("pause");
    return 0;
}
// w41.cpp //
```

Αποτέλεσμα

```
1
4
3
///
```

-- Δημιουργία λιστών με τεσσερις διαφορετικούς δομητές και εμφάνιση των στοιχείων τους.

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

typedef list<string> LISTSTR;

int main()
{
   LISTSTR::iterator i;
   LISTSTR Ls;                  // default δομητής (1)

   Ls.insert(Ls.end(), "one");
   Ls.insert(Ls.end(), "two");

   LISTSTR Ls2(Ls);             // Δημιουργία λιστας από άλλη λιστα (2)
   LISTSTR Ls3(3, "three"); // Δημιουργία και εισαγωγή 3 ιδιων στοιχείων (3)
   LISTSTR Ls4(++Ls3.begin(),Ls3.end()); // Δημιουργία λιστας και // //
                                // εισαγωγή καποιων στοιχείων από άλλη λιστα (4)

   // Εμφάνιση των στοιχειών των λιστων
   cout << "Ls :";
   for (i =  Ls.begin(); i != Ls.end(); ++i)// one two
        cout << " " << *i;
   cout << endl;

   cout << "Ls2:";
   for (i =  Ls2.begin(); i != Ls2.end(); ++i) // one two
        cout << " " << *i;
   cout << endl;
```

```
   cout << "Ls3:";
   for (i =  Ls3.begin(); i != Ls3.end(); ++i) // three three three
        cout << " " << *i;
   cout << endl;

   cout << "Ls4:";
   for (i =  Ls4.begin(); i != Ls4.end(); ++i) // three three
        cout << " " << *i;
   cout << endl;

   system("pause");
   return 0;
}
// w42.cpp //
```
Αποτέλεσμα
```
Ls : one two
Ls2: one two
Ls3: three three three
Ls4: three three
```

--- Παράδειγμα δημιουργίας και εισαγωγής στοιχείων σε λίστα
```
#include <iostream>
#include <list>
#include <vector>
using namespace std;

int main ()
{
  list<int> mylist;
  list<int>::iterator it; // o it είναι επαναληπτης

  // set some initial values:
  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;          // ο it δειχνει στον αριθμό 2            ^

  mylist.insert (it,10); // Παρεμβάλεται το 10 -> 1 10 2 3 4 5

           // ο "it" ακομη δειχνει στον αριθμό 2          ^
  mylist.insert (it,2,20);         // 1 10 20 20 2 3 4 5

  --it;       // ο it δειχνει τωρα στο δευτερο 20     ^

  // Δημιουργια vector και εισαγωση δύο αροθμών 30 και 30
  vector<int> myvector (2,30);
  // Παρεμβάλεται το myvector πριν από την θεση του it
  mylist.insert (it,myvector.begin(),myvector.end());
                                  // 1 10 20 30 30 20 2 3 4 5
                                  //               ^
  cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
     cout << ' ' << *it;
  cout << '\n';
```

```
  return 0;
}
// w44.cpp //
```
Αποτέλεσμα
```
mylist contains: 1 10 20 30 30 20 2 3 4 5
```

-- Δημιουργία λιστας ακεραίων και ταξινόμιση κατά αυξουσα και φθινουσα διαταξη με χρηση της συναρτησης-μελος **sort (list::sort)**

```
#include <iostream>
#include <list>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main()
{
      srand(time(NULL));
      list<int> mL;

      int x;
      for(int i = 0; i < 10; i++) {
            x = rand() % 100;
            mL.push_back(x);
      }
      list<int>::iterator it;
      for(it = mL.begin(); it != mL.end(); ++it)
         cout << *it << " " ;
      cout << endl;

    cout << "Sort in ascending order\n";
      mL.sort();   // OK. Âî ïñéóìïõ åêôåëåßôáé ôï  less()
      for(it = mL.begin(); it != mL.end(); ++it)
         cout << *it << " ";

    cout << "\nSort in descending order\n";
      mL.sort(greater<int>());
      for(it = mL.begin(); it != mL.end(); ++it)
         cout << *it << " ";
      cout << endl;
}
// w45.cpp //
```
Αποτέλεσμα
```
0 31 31 37 14 14 3 58 10 82
Sort in ascending order
0 3 10 14 14 31 31 37 58 82
Sort in descending order
82 58 37 31 31 14 14 10 3 0
```

-- Δημιουργία λιστας στρινγ, ταξινόμιση κατά αυξουσα διαταξη κανοντας χρήση της **sort** και του εξ' ορισμου **less()** και ταξινόμιση κατά αυξουσα διαταξη χωρις να λαμβάνεται υπ' οψιν αν είναι κεφαλαία ή πεζά τα γραμματα (χαρακτηρες) κανοντας

χρήση της **sort** και της συναρτησης **compare_nocase** που δεν ανήκει στην βιβλιοθήκη ου σσα διαταξη με χρηση της συναρτησης-μελος **sort (list::sort)**

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

// comparison, not case sensitive.
bool comp_nocase (const string& first, const string& second)
{
  unsigned int i=0;
  while ( (i < first.length()) && (i < second.length()) )  {
    if (tolower(first[i]) < tolower(second[i]))
          return true;
    else if (tolower(first[i]) > tolower(second[i]))
          return false;
    ++i;
  }
  return ( first.length() < second.length() );
}

int main ()
{
  list<string> mylist;
  list<string>::iterator it;
  mylist.push_back ("one");
  mylist.push_back ("two");
  mylist.push_back ("Three");

  cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
     cout << ' ' << *it;
  cout << '\n';

  mylist.sort(); // sort

  cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
     cout << ' ' << *it;
  cout << '\n';

  mylist.sort(comp_nocase); // σορτ

  cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
     cout << ' ' << *it;
  cout << '\n';

  return 0;
}
```

deque

Το container **deque** (deQueue) δημιουργεί μια γραμμική λιστα από στοιχεία. Τα στοιχεία εισάγονται/διαγράφονται και στην αρχή και στο τέλος της λιστας αλλά όχι στην μέση.

--- Δημιουργία deque και χρήση των μεθόδων **assign, swap, begin, end**

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

typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
   CHARDEQUE  a(3,'A'); // Δημιουργία του deque a με 3 A.
   CHARDEQUE  b(4,'B'); // Δημιουργία του deque a με 4 B.

   cout << "Emfanisi periexomenwn\n";
   print_contents (a,"a");
   print_contents (b,"b");

   a.swap(b); //swap a and b
   cout << "Emfanisi meta apo swap\n" ;
   print_contents (a,"a");
   print_contents (b,"b");

   a.swap(b); //swap it back
   cout << "Emfanisi meta apo 2o swap\n";
   print_contents (a,"a");
   print_contents (b,"b");

   cout << "Antigrafi b sto a. Emfanisi tou a\n";
   a.assign(b.begin(),b.end()); // a = b
   print_contents (a,"a");

   cout << "Antigrafi merous tou b sto a. Emfanisi tou a\n";
   a.assign(b.begin(),b.begin()+2); // αντιγραφή των 2 πρωτων στοιχείων
   print_contents (a,"a");

   cout << "Aponomi newn timwn sto a\n";
   a.assign(3,'Z'); //assign 3 'Z's to a
   print_contents (a,"a");

   system("pause");
   return 0;
}

// Συναρτηση που εμφανίζει το περιεχόμενο του deque
void print_contents (CHARDEQUE  deque, char *namedeq)
{
   CHARDEQUE::iterator pd;

   cout <<"deque "<< namedeq <<" : ";
```

```
   for(pd = deque.begin(); pd != deque.end(); pd++)
        cout << *pd <<" " ;
   cout<<endl;
}
// deque01.cpp //
```

Αποτέλεσμα

```
Emfanisi periexomenwn
deque a : A A A
deque b : B B B B

Emfanisi meta apo swap
deque a : B B B B
deque b : A A A

Emfanisi meta apo 2o swap
deque a : A A A
deque b : B B B B

Antigrafi b sto a. Emfanisi tou a
deque a : B B B B

Antigrafi merous tou b sto a. Emfanisi tou a
deque a : B B

Aponomi newn timwn sto a
deque a : Z Z Z
```

-- Δημιουργία deque και χρηση **push_back**, **front**, **back**, και **deque reference**.

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

int main( )
{

   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );

   deque<int>::const_reference  k = c1.front( );
   deque<int>::reference  m = c1.back( );
   cout << "The first  element is " << k << endl;
   cout << "The second element is " << m << endl;

   const int &i = c1.front( );
   int &j = c1.back( );
   cout << "The first  element is " << i << endl;
   cout << "The second element is " << j << endl;

   system("pause");
   return 0;
}
// deque_reference.cpp //
```

Αποτέλεσμα

```
The first  element is 10
The second element is 20

The first  element is 10
The second element is 20
```

-- Παράδειγμα.

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

typedef deque<char >  CHARDEQUE;
void print_contents (CHARDEQUE  deque, char*);

int main()
{
   // Δημιουργία του deque  a  me times A, B, C και D
   CHARDEQUE  a;
   a.push_back('A');
   a.push_back('B');
   a.push_back('C');
   a.push_back('D');

   print_contents (a,"a");// εμφάνιση περιεχομένου
   cout <<"To proto   stoixeio tou a is " <<a.front() <<endl;
   cout <<"To deytero stroixei tou a is " <<a.back() <<endl;

   // 'Αλλαξε το πρώτο και το τελευταίο στοιχειο με την χρηση του reference, front, και back
   CHARDEQUE::reference reffront=a.front();
   CHARDEQUE::reference refback=a.back();

   reffront='X';
   refback='Y';
   print_contents (a,"a"); // εμφάνιση περιεχομένου

   system("pause");
   return 0;
}

// Συναρτηση που εμφανίζει το περιεχόμενο του deque
void print_contents (CHARDEQUE  deque, char *namedeq)
{
   CHARDEQUE::iterator pd;
   cout <<"deque "<< namedeq <<" : ";
   for(pd = deque.begin(); pd != deque.end(); pd++)
        cout << " " << *pd ;
   cout<<endl;
}
// deque02.cpp //
```

Αποτέλεσμα

```
deque a :  A B C D
To proto   stoixeio tou a is A
To deytero stroixei tou a is D
deque a :  X B C Y
```

map

Η ταξη **map** δημιουργεί ταξηνομημένους δυναμικούς σηριακούς (συγγενικούς) πίνακες (*associative arrays*) οποιουδηποτε τυπου. Τα στοιχεία ενός map είναι ζευγάρια (κλειδί, τιμή) οπου μπορεί καποιος να αναζήτηση μια τιμή με βάση το κλείδι. Υποστηρίζει προσβαση στα στοιχεια του πινακα με διπλή κατεύθυνση (από την αρχή προς το τέλος και από το τέλος προς την αρχή) με την χρήση επαναληπτών διπλής κατεύθυνσης (bidirectional iterators).
Το container **unordered_map** είναι ένα βελτιστοποιημένο **map** για κλειδια που είναι strings. Δομές δεδομένων παρόμοιες με **map** και **unordered_map** είναι γνωστές ως *associative arrays, hash tables* και *red-black trees.*

#include <map>
Το αρχειο κεφαλίδων οριζει την ταξη και οτιδήποτε χρειαζεται για να χρησιμοποιήσουμε το container **map**.

-- Παράδειγμα
```
#include <string>
#include <map>
using namespace std;

int main()
{
    map<int,string> data;

    // αρχικές τιμές στο container map
    data[10] = "Ginger";
    data[45] = "Eva";
    data[34] = "Adam";
    data[22] = "Billie"; // Καθώς τα κλειδια είναι μοναδικά
    data[22] = "Elli";   // η Billie υπερκαλήπτεται από την Elli

    map<int,string>::const_iterator cit;
    for (cit = data.begin(); cit != data.end(); ++cit) {
       cout << "Who(key = first): " << cit->first;
       cout << " weight(value = second): " << cit->second << '\n';
    }

    system("pause");
    return 0;
}
// map02c.cpp //
```
Αποτέλεσμα
```
(key = first): 10 (value = second): Ginger
(key = first): 22 (value = second): Elli
(key = first): 34 (value = second): Adam
(key = first): 45 (value = second): Eva
////
```

-- Παράδειγμα
```
#include <iostream>
#include <string>
```

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

int main()
{
    typedef map<int,string> MyMap;
    MyMap data;

    // αρχικές τιμές στο container map
    data["Ginger"] = 10;
    data["Eva"] = 15;
    data["Adam"] = 34;
    data["Billie"] = 22; // Καθώς τα κλειδια είναι μοναδικά
    data["Billie"] = 23; // το 22 υπερκαλήπτεται από το 23

    MyMap::const_iterator cit;
    for (cit = data.begin(); cit != data.end(); ++cit) {
        cout << "(key = first): " << cit->first;
        cout << " (value = second): " << cit->second << '\n';
    }

    system("pause");
    return 0;
}
// map02a.cpp //
```

Αποτέλεσμα

```
(key = first): Adam   (value = second): 34
(key = first): Billie (value = second): 23
(key = first): Eva    (value = second): 15
(key = first): Ginger (value = second): 10
```

string

Η ταξη **string** δημιουργεί μεταβλητου μήκους ακολουθίες χαρακτήρων (δυναμικούς πινακες χαρακτήρων) και υποστηρίζει τυχαία προσβαση στα στοιχεια του πινακα με την χρήση επαναληπτών τυχαίας προσβασης (random access iterators).

#include <string>

Το αρχειο κεφαλίδων οριζει την ταξη και οτιδήποτε χρειαζεται για να χρησιμοποιήσουμε το container **string**.

queue

Οι ουρές είναι ενας τύπος container adaptor, ειδικά σχεδιασμένος να λειτουργεί ως FIFO (first-in first-out) ουρά όπου τα στοιχεία εισάγονται στο ένα ακρο και εξάγονται από το άλλο ακρο του container.

Οι ουρές υλοποιούνται ως προσαρμοστικοί containers οι οποίες είναι classes που χρησιμοποιούν ένα ενφωλιασμένο αντικείμενο ενός ορισμένου container ως υποκείμενο container του παρέχοντας ένα συνολο από συναρτήσεις μέλη για

προσβαση στα στοιχεία του. Τα στοιχεία εισάγονται στο «τέλος» (back) του ορισμένου container και εξάγονται από την «αρχή» (front).

Το υποκείμενο container μπορεί να είναι ένα από τα καθιερωμένα container ή άλλο ειδικά σχεδιασμένο container. Η μονη απαίτηση είναι ότι πρέπει να υποστηρίζει τις λειτουργίες: **front()**, **pop_front()**, **back()**, **push_back()**. Τα καθιερωμένα containers **deque** και **list** μπορούν να χρησιμοποιηθούν ως καθιερωμένα containers. Εάν δεν ορισθεί κανένα καθιερωμένο container τότε χρησιμοποιείται εξ' ορισμου (default) η **deque**.

-- Δημιουργία ουράς με **queue** και χρήση των μεθόδων **push()**, **pop()**, **empty()**, **back()**, **front()**, **size()**. Αν ενεργοποιησουμε την γραμμή **(1)** τοτε χρησιμοποιείται εξ' ορισμού το καθιερωμένο container **deque**. Αν ενεργοποιήσουμε την **(2)** τοτε δηλωνουμε ότι θα χρησιμοποιείσουμε το καθιερωμένο container **deque**.

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

// typedef queue<char*> CHARQUEUE; // (1)
// typedef queue< char* , deque<char*> > CHARQUEUE; // (2)

int main(void)
{
    size_t size_q;
    CHARQUEUE p;

    // Insert items in the queue(uses deque)
    p.push("cat");
    p.push("ape");
    p.push("dog");
    p.push("mouse");
    p.push("horse");

    // Output the item inserted last using back()
    cout << p.back() << endl;

    size_q = p.size();// Output the size of queue
    cout << "size of p is:" << size_q << endl;

    // Output items in queue using front() and use pop() to get to next item until queue is empty
    while (!p.empty())
    {
        cout << p.front() << endl;
        p.pop();
    }

    system("pause");
    return 0;
}
// queue.cpp //
```

Αποτέλεσμα

```
horse
size of p is:5
```

```
cat
ape
dog
mouse
horse
```

-- Δημιουργία ουράς ακεραίων με **queue**, με καθιερωμένο container **list** και χρήση των μεθόδων **push()**, **pop()**, **empty()**, **back()**, **front()**, **size()**.

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

// Using queue with list
typedef queue< int, list<int> > INTQUEUE;

int main(void)
{
    size_t size_q;
    INTQUEUE q;

    // Insert items in the queue(uses list)
    q.push(42);
    q.push(100);
    q.push(49);
    q.push(201);

    // Output the size of queue
    size_q = q.size();
    cout << "size of q is: " << size_q << endl;

    // Output items in queue using front()
    // and use pop() to get to next item until
    // queue is empty
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }

    system("pause");
    return 0;
}
// queue_list.cpp //
```

Αποτέλεσμα

```
size of q is: 4
42
100
49
201
```