Προγραμματισμός Υπολογιστών C++

Κλάσεις και Αντικείμενα
Υπερφόρτωση Τελεστών
Κληρονομικότητα

Κλάση

- Ο τύπος κλάση (class) είναι παρόμοιος με τον τύπο της δομής
- Όπως και μία δομή, μία κλάση είναι ένας **τύπος δεδομένων** που ορίζεται από τον προγραμματιστή (user-defined type), για να προσδιορίσει τα χαρακτηριστικά και τις λειτουργίες μίας αφηρημένης οντότητας
- Μία κλάση περιέχει ένα σύνολο δεδομένων το οποίο περιγράφει μία οντότητα σύμφωνα με τις ανάγκες της εφαρμογής και μπορεί επίσης να περιέχει ένα σύνολο συναρτήσεων οι οποίες να εφαρμόζονται στα δεδομένα
- Αυτή η συνύπαρξη δεδομένων και συναρτήσεων στην κλάση αναφέρεται σαν **ενθυλάκωση** δεδομένων (data encapsulation)
- Γενικά, μπορούμε να χρησιμοποιήσουμε κλάσεις για να αναπαραστήσουμε όποια οντότητα του πραγματικού κόσμου επιθυμούμε και να γράψουμε προγράμματα που να βασίζονται σε αυτές
- Ουσιαστικά, η κλάση είναι η **καρδιά** του αντικειμενοστρεφούς προγραμματισμού

Δήλωση Κλάσης

- Όπως είπαμε, η κλάση είναι το βασικό δομικό στοιχείο για να αναπαραστήσουμε μία οντότητα στο πρόγραμμα. Για παράδειγμα, για να αναπαραστήσουμε μία τηλεφωνική κλήση θα μπορούσαμε να ορίσουμε μία κλάση, η οποία να περιέχει πληροφορίες για αυτήν, όπως τον αριθμό του καλούντα συνδρομητή, τον αριθμό του καλούμενου, τη διάρκεια της κλήσης και συναρτήσεις για την εγκατάσταση, την αποδέσμευση και χρέωση της κλήσης
- Γενικά, ο προγραμματιστής αποφασίζει το τι θα περιέχει μία κλάση ανάλογα με τον τρόπο που σκοπεύει να χρησιμοποιηθεί η κλάση
- Η γενική μορφή δήλωσης μίας κλάσης είναι:

```
class όνομα_κλάσης
{
        Προσβασιμότητα: Δηλώσεις;
        ...
        Προσβασιμότητα: Δηλώσεις;
};
```

- Η κοινή πρακτική είναι η δήλωση μίας κλάσης να τοποθετείται σε ένα αρχείο επικεφαλίδας, που συνήθως έχει το ίδιο όνομα με το όνομα της κλάσης, ενώ η υλοποίηση των συναρτήσεων της κλάσης σε ένα ή περισσότερα αρχεία κώδικα τα οποία κάνουν `#include` το αρχείο επικεφαλίδας. Με τον διαχωρισμό της δήλωσης της κλάσης από την υλοποίηση της λειτουργικότητας της, ο κώδικας γίνεται πιο εύκολο να γραφεί και να συντηρηθεί

Αντικείμενα Κλάσης

- Παρόμοια με μία ονομαστική δομή, αφού δηλωθεί μία κλάση, μπορούμε να δηλώσουμε μεταβλητές αυτού του τύπου
- Αυτές οι μεταβλητές ονομάζονται **αντικείμενα** (objects) ή **στιγμιότυπα** (instances) της κλάσης και δημιουργούνται με βάση τις προδιαγραφές που ορίζει η κλάση
- Κάθε αντικείμενο αποτελεί μία υπόσταση μίας **συγκεκριμένης** κλάσης και η πληροφορία που περιέχει είναι αποθηκευμένη στα μέλη του
- Η κλάση απλά **περιγράφει** την οντότητα, είναι μία **γενική** έννοια που δεν αναφέρεται σε κάποιο συγκεκριμένο αντικείμενο. Να το ξεκαθαρίσουμε λοιπόν, η **δήλωση μίας κλάσης**, όπως και η **δήλωση μίας δομής**, **δεν δημιουργεί** κάποιο αντικείμενο
- Για παράδειγμα, ένα συγκεκριμένο μοντέλο αυτοκινήτου αποτελεί αντικείμενο της κλάσης που περιγράφει την οντότητα αυτοκίνητο. Μπορούμε να δημιουργήσουμε και να χειριστούμε όσα τέτοια αντικείμενα επιθυμούμε και καθένα από αυτά θα έχει τις δικές του ιδιότητες

Παράδειγμα

- Όπως και με τις δομές, μπορούμε να μεταβιβάσουμε ένα αντικείμενο σε μία συνάρτηση, να ορίσουμε μία συνάρτηση που να επιστρέφει αντικείμενο και να εκχωρήσουμε ένα αντικείμενο σε ένα άλλο της ίδιας κλάσης
- Για παράδειγμα, το παρακάτω πρόγραμμα δηλώνει την κλάση **Student**, διαβάζει τα στοιχεία ενός φοιτητή, τα αποθηκεύει στα αντίστοιχα μέλη ενός αντικειμένου και τα εμφανίζει στην οθόνη

```
// student.h
#ifndef STUDENT_H
#define STUDENT_H

#include <string>

class Student
{
private:
        int code;
public:
        std::string name;
        float grd;

        void set(int c) {code = c;}
        void show();
};

#endif
```

Παράδειγμα

```
#include <iostream>
#include "student.h"
using std::cin;
using std::cout;

void Student::show()
{
        cout << "N:" << name << " C:" << code << " G:" << grd << '\n';
}

int main()
{
        int code;
        class Student s; /* Το s είναι αντικείμενο της κλάσης Student. Η λέξη
class δεν είναι απαραίτητη. */

        cout << "Name: ";
        getline(cin, s.name);

        cout << "Grade: ";
        cin >> s.grd;

        cout << "Code: ";
        cin >> code;
        s.set(code);
        s.show();
        return 0;
}
```

Παράδειγμα

- Όπως και με τα ονόματα δομών, το πρώτο γράμμα της κλάσης συνηθίζεται να είναι κεφάλαιο
- Όταν δηλώνεται ένα αντικείμενο η λέξη `class` δεν χρειάζεται, γιατί το όνομα της κλάσης αποτελεί όνομα τύπου. Για παράδειγμα, το όνομα `Student` αποτελεί το όνομα ενός νέου τύπου το οποίο ορίστηκε από τον προγραμματιστή
- Μία κλάση μπορεί να περιέχει και αντικείμενα άλλων κλάσεων που είτε εμείς έχουμε δημιουργήσει είτε άλλοι προγραμματιστές. Για παράδειγμα, το πεδίο `name` είναι αντικείμενο της κλάσης βιβλιοθήκης `string`. Άρα, είμαστε χρήστες της κλάσης `string` και έχουμε πρόσβαση, όπως θα δούμε στη συνέχεια, στο τμήμα που οι προγραμματιστές της `string` κλάσης αποφάσισαν να κάνουν δημόσιο
- Κάθε φορά που δημιουργείται ένα νέο αντικείμενο, ο μεταγλωττιστής δεσμεύει μνήμη για τις μεταβλητές του. Για παράδειγμα, αν δηλώσουμε τα αντικείμενα `s1` και `s2`, η μνήμη που έχει δεσμευτεί για τα `s1.grd` και `s2.grd` είναι διαφορετική
- Αντίθετα, η μνήμη που δεσμεύεται για τις συναρτήσεις είναι κοινή. Για παράδειγμα, ο κώδικας της `show()` βρίσκεται αποθηκευμένος μόνο σε ένα σημείο, δεν δημιουργούνται αντίγραφά του κάθε φορά που δηλώνεται ένα νέο αντικείμενο. Και αυτό είναι λογικό, αφού ο κώδικας των συναρτήσεων είναι κοινός για όλα τα αντικείμενα. Δηλαδή, οι εντολές `s1.show()` και `s2.show()` εκτελούν τον ίδιο κώδικα και αυτός ο κώδικας εφαρμόζεται στα δεδομένα του αντικειμένου που κάνει την κλήση

Συναρτήσεις Μέλη (1)

- Συναρτήσεις που ανήκουν σε μία κλάση ονομάζονται συναρτήσεις μέλη (member functions)
- Μία συνάρτηση μπορεί να οριστεί **μέσα ή έξω** από την κλάση. Συνήθως, οι συναρτήσεις ορίζονται έξω από την κλάση σε ξεχωριστό(ά) αρχείο(α) κώδικα. Αυτή η διάκριση κάνει ξεκάθαρο το **τι κάνει** η κλάση (δηλώσεις συναρτήσεων) από το **πώς το κάνει** (ορισμοί συναρτήσεων)
- Όπως και με τις δομές, για να προσπελάσουμε τα μέλη ενός αντικειμένου χρησιμοποιούμε τους τελεστές τελεία (.) και **->**. Δηλαδή, προσπελαύνουμε τις συναρτήσεις με τον ίδιο τρόπο όπως και τα απλά πεδία
- Επειδή διαφορετικές κλάσεις μπορεί να έχουν συναρτήσεις με το ίδιο όνομα, για να την ορίσουμε χρησιμοποιούμε τον τελεστή **επίλυσης εμβέλειας** **::** (scope resolution operator) και το όνομα της κλάσης στην οποία ανήκει. Έτσι, προσδιορίζεται σε ποια κλάση ανήκει η συνάρτηση και ο μεταγλωττιστής μπορεί να μεταγλωττίσει τον κώδικα

Συναρτήσεις Μέλη (2)

- Κάθε συνάρτηση που ορίζεται μέσα στην κλάση θεωρείται **εμβόλιμη** (inline). Έτσι, δεν χρειάζεται να προστεθεί η λέξη `inline` στον ορισμό της. Δηλαδή, η `set()` στο παράδειγμα είναι εμβόλιμη
- Γενικά, η συνήθης πρακτική είναι να ορίζονται μέσα στην κλάση συναρτήσεις που είναι μικρές
- Σημειώστε ότι μία συνάρτηση που έχει οριστεί έξω από την κλάση μπορεί και αυτή να γίνει εμβόλιμη προσθέτοντας τη λέξη `inline` στον ορισμό της. Για παράδειγμα:

```
inline void Student::show()
```

Εμβέλεια Κλάσης

- Κάθε κλάση ορίζει μέσα στα `{}` τη δική της **εμβέλεια** (class scope)
- Η εμβέλεια των μελών μίας κλάσης περιορίζεται στην κλάση στην οποία ανήκουν. Επομένως, τα ίδια ονόματα μπορούν να χρησιμοποιηθούν και σε άλλες κλάσεις. Για παράδειγμα, μία κλάση `C` μπορεί να περιέχει τη δική της `show()`:

  ```
  int C::show()
  ```

- Επειδή ακριβώς τα ονόματα των μελών δεν είναι ορατά έξω από την κλάση πρέπει να χρησιμοποιούμε το όνομα της κλάσης. Με τη χρήση του ονόματος της κλάσης και του τελεστή `::` ο ορισμός της κάθε `show()` συσχετίζεται με την αντίστοιχη κλάση
- Όταν καλείται μία συνάρτηση, τα μέλη αναφέρονται στο αντικείμενο που κάλεσε την συνάρτηση. Για παράδειγμα, όταν καλείται η `show()`, εμφανίζονται οι τιμές των μελών του αντικειμένου που κάλεσε την `show()`

Έλεγχος Πρόσβασης (1)

- Τα μέλη μίας κλάσης μπορεί να είναι δημόσια (`public`), ιδιωτικά (`private`) ή προστατευμένα (`protected`)
- Τα **ιδιωτικά** μέλη είναι προσβάσιμα **μόνο** από συναρτήσεις της **ίδιας** κλάσης. Δηλαδή, **δεν επιτρέπεται** να προσπελαστούν από συνάρτηση ή αντικείμενο έξω από την κλάση
- Για παράδειγμα, δεν επιτρέπεται να γράψουμε `s.code = 1`. Αντίθετα η `code = c;` μέσα στη `set()` επιτρέπεται, γιατί μία συνάρτηση μέλος μπορεί να προσπελάσει **οποιοδήποτε** μέλος της ίδιας κλάσης, ανεξάρτητα από το προσδιοριστικό του
- Όσον αφορά τα **δημόσια** μέλη, δεν υπάρχει περιορισμός στην προσπέλασή τους, το οποίο σημαίνει ότι είναι προσπελάσιμα από **οπουδήποτε** μέσα ή έξω από την κλάση
- Τα **προστατευμένα** μέλη συμπεριφέρονται όπως τα ιδιωτικά μέλη με την διαφορά ότι μπορεί να προσπελαστούν από συναρτήσεις παραγώγων κλάσεων, όπως θα δούμε στο Κ.20
- Τα μέλη μίας κλάσης είναι **εξ'ορισμού** ιδιωτικά. Δηλαδή, αν στο παράδειγμά μας έλειπε η λέξη `private` πριν από τη δήλωση του `code`, το `code` θα ήταν και πάλι ιδιωτικό

Έλεγχος Πρόσβασης (2)

- Βλέπουμε λοιπόν ότι μία κλάση όχι μόνο επιτρέπει την αναπαράσταση μίας λογικής οντότητας με την **ενθυλάκωση δεδομένων** σε έναν τύπο (data encapsulation), αλλά παρέχει και τη δυνατότητα στον προγραμματιστή να καθορίσει ποια δεδομένα να είναι **προσβάσιμα** από το εξωτερικό περιβάλλον και ποια να **αποκρύβονται** (data hiding)
- Όπως θα δούμε αργότερα, η απόκρυψη δεδομένων είναι πολύ σημαντική για την **ακεραιότητα** του αντικειμένου
- Με τον όρο εξωτερικό περιβάλλον, εννοούμε τους χρήστες της κλάσης, δηλαδή, προγραμματιστές που την χρησιμοποιούν
- Για παράδειγμα, στο παρακάτω πρόγραμμα, θεωρήστε ότι την κλάση `Test` την γράψατε εσείς και την δώσατε σε κάποιον άλλο προγραμματιστή, ο οποίος την χρησιμοποιεί στο πρόγραμμά του για να δημιουργήσει `Test` αντικείμενα. Βρείτε τα λάθη:

Παράδειγμα

```
#include <iostream>
class Test
{
private: /* Επειδή εξ'ορισμού τα μέλη μίας κλάσης είναι ιδιωτικά, μπορούμε να παραλείψουμε
τη λέξη private. */
          int a;
          void f1();
protected:
          int b;
          void f2();
public:
          int c;
          void f3();
};
void Test::f1()
{
          std::cout << a << ' ' << b << ' ' << c << ' ' << '\n';
}
void Test::f2()
{
          a = 100;
}
void Test::f3()
{
          f1();
}
int main()
{
          Test t;
          t.a = 1;
          t.b = 2;
          t.c = 3;
          t.f1();
          t.f2();
          t.f3();
          return 0;
}
```

Παράδειγμα

- Αφού τα μέλη `a` και `f1()` έχουν δηλωθεί ως `private` δεν επιτρέπεται η πρόσβαση σε αυτά από το εξωτερικό περιβάλλον. Επομένως, οι εντολές `t.a = 1;` και `t.f1();` δεν είναι αποδεκτές. Το ίδιο ισχύει και για τα `protected` μέλη. Άρα, οι εντολές `t.b = 2;` και `t.f2();` δεν είναι αποδεκτές. Αφού η πρόσβαση στα `public` μέλη επιτρέπεται, οι εντολές `t.c = 2;` και `t.f3();` είναι αποδεκτές. Τέλος, αφού οι συναρτήσεις μίας κλάσης έχουν πρόσβαση στα μέλη της, οι `f1()`, `f2()` και `f3()` μεταγλωττίζονται κανονικά
- Να το πούμε και διαφορετικά, η `f1()` μπορεί να χρησιμοποιηθεί άμεσα από τον προγραμματιστή που έγραψε την κλάση, αλλά όχι από τον προγραμματιστή που χρησιμοποιεί την κλάση
- Θυμόμαστε λοιπόν ότι η πρόσβαση στα ιδιωτικά και προστατευμένα μέλη μίας κλάσης επιτρέπεται από συναρτήσεις της κλάσης και, όπως θα δούμε στη συνέχεια, από φιλικές συναρτήσεις

Παρατηρήσεις

- Γενικά, όταν σχεδιάζουμε μία κλάση, τα μέλη και τις συναρτήσεις που δεν θέλουμε να είναι άμεσα προσβάσιμα τα δηλώνουμε **ιδιωτικά** (π.χ. συναρτήσεις που σχετίζονται με την υλοποίηση της κλάσης), ενώ τα μέλη και τις συναρτήσεις που θέλουμε να αποτελούν τη διεπαφή (interface) με το εξωτερικό περιβάλλον τα δηλώνουμε **δημόσια**
- Όπως θα δούμε στο Κ.20, η απόφαση για να κάνουμε ένα μέλος **προστατευμένο** βασίζεται στο αν θέλουμε να επιτρέψουμε την απευθείας πρόσβαση σε αυτό από μία παράγωγη κλάση. Αν ναι, το κάνουμε προστατευμένο, αν όχι, το κάνουμε ιδιωτικό
- Ουσιαστικά, ο χρήστης μίας κλάσης επικοινωνεί με την κλάση μέσω των **δημοσίων** μελών της, τα οποία αποτελούν την διεπαφή της κλάσης
- Έχουμε ήδη συναντήσει παραδείγματα στο Κ.10, όπου σαν χρήστες της κλάσης `string` χρησιμοποιήσαμε δημόσιες συναρτήσεις (π.χ. `size()`) για να την διαχειριστούμε. Το ίδιο κάναμε και σε παραδείγματα με την κλάση `vector`

const Συναρτήσεις Κλάσης

- Αν δεν θέλουμε μία συνάρτηση να μπορεί να αλλάξει τις τιμές του αντικειμένου που την καλεί προσθέτουμε τη λέξη const στο τέλος της δήλωσής της
- Η λέξη const πρέπει να προστεθεί και στον ορισμό της. Για παράδειγμα, η δήλωση της show() γίνεται:

  void show() const; και ο ορισμός της:

```
void Student::show() const
{
        grd = 2; // Λάθος.
}
```

  Αν η show() επιχειρήσει να αλλάξει την τιμή κάποιου πεδίου της κλάσης, όπως το grd, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για μη επιτρεπτή ενέργεια

- Για καλύτερο έλεγχο της λειτουργίας της κλάσης, τις συναρτήσεις που δεν αλλάζουν τις τιμές των μελών της να τις δηλώνετε const

Φιλικές Συναρτήσεις

- Όπως μάθαμε, μία συνάρτηση μέλος μπορεί να έχει πρόσβαση στα ιδιωτικά μέλη μίας κλάσης. Υπάρχει τρόπος μία συνάρτηση που δεν είναι μέλος να έχει και αυτή πρόσβαση στα ιδιωτικά μέλη;
- Ναι, αρκεί να έχει δηλωθεί σαν **φιλική** (friend) της κλάσης. Συγκεκριμένα, μία φιλική συνάρτηση είναι μία συνάρτηση **μη-μέλος** η οποία μπορεί να έχει πρόσβαση στα μέλη μίας κλάσης ανεξάρτητα από το τμήμα στο οποίο δηλώνονται
- Δηλαδή, μία φιλική συνάρτηση έχει τα **ίδια δικαιώματα πρόσβασης** με μία συνάρτηση μέλος της κλάσης
- Για να γίνει μία συνάρτηση φιλική πρέπει να δηλωθεί μέσα στην κλάση με το πρόθεμα `friend`. Η θέση δεν παίζει ρόλο, μπορεί να δηλωθεί στο ιδιωτικό, προστατευμένο ή δημόσιο τμήμα

Παράδειγμα

```
#include <iostream>

class T
{
private:
        int a;
public:
        int show() {std::cout << a << '\n';}
        friend void f(T& t);
};

void f(T& t) // Η λέξη friend δεν προστίθεται στον ορισμό.
{
        t.a = 10; // Είναι λάθος να γράψουμε a = 10.
}

int main()
{
        T t;

        f(t);
        t.show();
        return 0;
}
```

Παράδειγμα

- Επειδή, όπως είπαμε, μία φιλική συνάρτηση δεν αποτελεί μέλος της κλάσης δεν προσθέτουμε το `T::` στον ορισμό της `f()`
- Για να γίνει ακόμα πιο ξεκάθαρο, αν είχαμε δηλώσει το αντικείμενο `t1` και γράφαμε `t1.f(t)`, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους παρόμοιο με «f is not a member of T»
- Για τον ίδιο λόγο, επειδή η `f()` δεν είναι μέλος της `T` δεν μπορούμε να προσπελάσουμε τα μέλη της γράφοντας μόνο τα ονόματά τους
- Σημειώστε ότι η λέξη `friend` δεν προστίθεται στον ορισμό της, εκτός αν δηλωθεί μέσα στην κλάση. Όπως είπαμε, αν και η `f()` δεν είναι συνάρτηση μέλος έχει τα ίδια δικαιώματα πρόσβασης με μία συνάρτηση μέλος
- Έτσι, επιτρέπεται η πρόσβαση στο ιδιωτικό πεδίο `a` και το πρόγραμμα εμφανίζει `10`

Κανόνες Πρόσβασης

- Τώρα που μιλήσαμε για τις φιλικές συναρτήσεις, ας συνοψίσουμε τους κανόνες που ισχύουν για την πρόσβαση στα μέλη μίας κλάσης:

  α) Τα ιδιωτικά μέλη είναι προσβάσιμα μόνο από συναρτήσεις-μέλη της κλάσης και φιλικές συναρτήσεις της κλάσης

  β) Τα προστατευμένα μέλη είναι προσβάσιμα μόνο από συναρτήσεις-μέλη της κλάσης και φιλικές της συναρτήσεις, καθώς και από συναρτήσεις-μέλη και τις φιλικές συναρτήσεις των κλάσεων που παράγονται από αυτή

  γ) Τα δημόσια μέλη είναι προσβάσιμα από οποιαδήποτε συνάρτηση μέσα ή έξω από την κλάση

Κατασκευαστής

- Ο **κατασκευαστής** ή **συνάρτηση δόμησης** (constructor) είναι μία **ειδική** συνάρτηση της κλάσης που καλείται **κάθε φορά που δημιουργείται** ένα αντικείμενό της
- Η δήλωση του κατασκευαστή πρέπει να ακολουθεί κάποιους **κανόνες**
- Συγκεκριμένα, πρέπει να έχει το ίδιο όνομα με την κλάση, δεν έχει τύπο επιστροφής, δεν επιτρέπεται να δηλωθεί σαν `const`, μπορεί να δεχτεί παραμέτρους και μπορεί να υπερφορτωθεί, δηλαδή, μία κλάση μπορεί να έχει πολλούς κατασκευαστές
- Όπως και με τις απλές υπερφορτωμένες συναρτήσεις, οι εκδόσεις των κατασκευαστών πρέπει να διαφέρουν στον αριθμό ή τον τύπο των παραμέτρων τους
- Συνήθως, ένας κατασκευαστής χρησιμοποιείται για την απόδοση αρχικών τιμών στα πεδία του αντικείμενου ή για τη δέσμευση μνήμης

Αποδομητής

- Ο **αποδομητής** (destructor) είναι μία **ειδική** συνάρτηση της κλάσης που καλείται αυτόματα όταν **καταστρέφεται** ένα αντικείμενό της (π.χ. όταν βγει εκτός εμβέλειας)
- Πρέπει να έχει το ίδιο όνομα με την κλάση με τη διαφορά ότι προστίθεται ο χαρακτήρας ~ πριν από αυτό, δεν έχει τύπο επιστροφής, δεν δέχεται παραμέτρους και επομένως δεν μπορεί να υπερφορτωθεί. Δηλαδή, μία κλάση μπορεί να έχει έναν μόνο αποδομητή
- Ο λόγος για να ορίσουμε έναν αποδομητή είναι όταν θέλουμε να κάνουμε κάποιες ενέργειες όταν ένα αντικείμενο καταστρέφεται
- Αν δεν χρειάζεται να κάνουμε κάτι, τότε ο εξ'ορισμού αποδομητής, όπως θα δούμε αργότερα, είναι αρκετός. Για παράδειγμα, η πιο συνηθισμένη χρήση του αποδομητή είναι για την απελευθέρωση μνήμης που μπορεί να έχει δεσμεύσει ένα αντικείμενο με δυναμικό τρόπο

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

class Test
{
public:
        int a, b;

        Test();
        Test(int i, int j);
        ~Test();
        void check();
};
Test::Test()
{
        std::cout << "First\n";
}
Test::Test(int i, int j)
{
        std::cout << "Second\n";
        a = i;
        b = j;
}
Test::~Test()
{
        std::cout << "Out\n";
}

void Test::check()
{
        Test t;
}

int main()
{
        Test t1, t2(5, 10);

        std::cout << t1.a << ' ' <<
t1.b << '\n';
        std::cout << t2.a << ' ' <<
t2.b << '\n';
        t1.check();
        return 0;
}
```

Παράδειγμα

- Όπως είπαμε, οι δηλώσεις του κατασκευαστή και του αποδομητή δεν έχουν τύπο επιστροφής
- Επειδή υπάρχουν δύο υπερφορτωμένοι κατασκευαστές, ο μεταγλωττιστής επιλέγει τον κατάλληλο ανάλογα με τον τρόπο που δημιουργείται το κάθε αντικείμενο. Όπως θα δούμε στη συνέχεια, ένας κατασκευαστής που δεν δέχεται παραμέτρους ονομάζεται εξ'ορισμού κατασκευαστής (default constructor)
- Όταν δημιουργείται το αντικείμενο `t1` καλείται ο πρώτος κατασκευαστής και το πρόγραμμα εμφανίζει `First`. Οι μεταβλητές `a` και `b` του `t1` αρχικοποιούνται με τυχαίες τιμές
- Γενικά, μην ξεχνάτε να αρχικοποιείτε τα πεδία μίας κλάσης, γιατί, αν τα χρησιμοποιήσετε, απροσδόκητα προβλήματα μπορεί να προκύψουν

Παράδειγμα

- Οι κατασκευαστές υπακούουν στους ίδιους κανόνες υπερφόρτωσης με τις συνηθισμένες συναρτήσεις
- Ο μεταγλωττιστής ελέγχει τους τύπους των ορισμάτων για να επιλέξει τον κατάλληλο. Επομένως, όταν δημιουργείται το `t2`, καλείται ο δεύτερος κατασκευαστής, μεταβιβάζονται οι τιμές `5` και `10` και το πρόγραμμα εμφανίζει `Second`
- Στη συνέχεια, το πρόγραμμα εμφανίζει τις τυχαίες τιμές των `a` και `b` του αντικειμένου `t1` και τις τιμές `5` και `10` για τις αντίστοιχες του `t2`
- Όταν καλείται η `check()` δημιουργείται το αντικείμενο `t` και καλείται ο πρώτος κατασκευαστής
- Αμέσως μετά η `check()` τερματίζεται και το `t` παύει να υπάρχει. Επομένως, καλείται αυτόματα ο αποδομητής του `t`, ο οποίος εμφανίζει το μήνυμα `Out`

Παράδειγμα

- Όταν τερματίζει η εκτέλεση του προγράμματος, καλούνται οι αποδομητές των `t1` και `t2` και το πρόγραμμα εμφανίζει άλλες δύο φορές το μήνυμα `Out`
- Ας δούμε με ποια σειρά καλούνται οι αποδομητές. Επειδή η μνήμη για τις αυτόματες μεταβλητές `t1` και `t2` δεσμεύεται στη στοίβα, το αντικείμενο που δημιουργήθηκε τελευταίο είναι το πρώτο που θα καταστραφεί. Άρα, πρώτα θα κληθεί ο αποδομητής του `t2` και μετά του `t1`
- Αν έχουμε έναν πίνακα αντικειμένων, για παράδειγμα, `Test t[10];` η σειρά καταστροφής θα είναι από το τελευταίο προς το πρώτο, δηλαδή, `t[9]...t[0]`.

Εξ'ορισμού Κατασκευαστής και Αποδομητής

- Η ύπαρξη κατασκευαστή είναι απαραίτητη για τη δημιουργία των αντικειμένων
- Τότε όμως, θα μπορούσε να ρωτήσει κάποιος, πώς δημιουργήθηκαν `Student` αντικείμενα στο πρώτο μας παράδειγμα, αφού δεν ορίσαμε κάποιον κατασκευαστή;
- Η απάντηση είναι ότι αν μία κλάση δεν περιέχει κατασκευαστές, τότε ο μεταγλωττιστής παρέχει **αυτόματα** τον **εξ'ορισμού κατασκευαστή** (default constructor), ο οποίος δεν έχει σώμα. Αν δεν υπήρχε αυτός ο κατασκευαστής, δεν θα μπορούσαμε να δημιουργήσουμε `Student` αντικείμενα. Κάθε φορά που δηλώνεται ένα `Student` αντικείμενο, ο μεταγλωττιστής τον καλεί για τη δημιουργία του αντικειμένου
- Ο εξ'ορισμού κατασκευαστής καλεί τους εξ'ορισμού κατασκευαστές των αντίστοιχων κλάσεων για μέλη που είναι αντικείμενα
- Παρόμοια, αν δεν ορίζεται ο αποδομητής, ο μεταγλωττιστής παρέχει τον **εξ'ορισμού αποδομητή** (default destructor) που επίσης δεν έχει σώμα. Για παράδειγμα, ο εξ'ορισμού κατασκευαστής και αποδομητής για την κλάση Student είναι:

  `Student::Student() {}` και `Student::~Student() {}`

  Όπως βλέπετε, και οι δύο δεν δέχονται ορίσματα και δεν κάνουν τίποτα

Παρατηρήσεις

- Ο μεταγλωττιστής παρέχει αυτόματα τον εξ'ορισμού κατασκευαστή **μόνο αν** δεν έχει δηλωθεί άλλος κατασκευαστής
- Για παράδειγμα, αν δηλώσετε τον κατασκευαστή:

  `Student(const string& n, int c, float g);` αποτελεί δική σας ευθύνη να δηλώσετε την δική σας έκδοση του εξ'ορισμού κατασκευαστή, αν βέβαια το επιθυμείτε. Αν δεν το κάνετε, τότε μία δήλωση όπως:

  ```
  Student s;
  ```

  είναι **μη αποδεκτή** και ο μεταγλωττιστής θα εμφανίσει ένα μήνυμα λάθους παρόμοιο με «No default constructor available». Αν βέβαια γράψετε:

  ```
  Student s("P.Lew", 100, 5.5);
  ```

  δεν υπάρχει πρόβλημα, αφού θα κληθεί ο ορισμένος κατασκευαστής

Προκαθορισμένες Συναρτήσεις

- Οι συναρτήσεις που παρέχει εξ'ορισμού ο μεταγλωττιστής για μία κλάση `T` είναι:

  α) Ο προκαθορισμένος κατασκευαστής: `T()`

  β) Μία προκαθορισμένη συνάρτηση αντιγραφής τιμής (copy assignment operator) που δηλώνεται ως: `T& operator=(const T&)`. Για παράδειγμα, όπως θα δούμε στο Κ.18, όταν γράφουμε `t1 = t2;` καλείται η συνάρτηση `operator=`, η οποία αντιγράφει τις τιμές των μη στατικών μελών του `t2` στο `t1`. Αν το μέλος είναι πίνακας γίνεται αντιγραφή των στοιχείων του. Η συνάρτηση επιστρέφει αναφορά στον αριστερό τελεστέο

  γ) Ο προκαθορισμένος αποδομητής: `~T()`

- Αυτές είναι οι συναρτήσεις που εξ'ορισμού θα δημιουργήσει και θα καλέσει ο μεταγλωττιστής, εκτός αν ο προγραμματιστής έχει ορίσει τις δικές του. Στο Κ.19 θα δείτε και άλλες προκαθορισμένες συναρτήσεις
- Σημειώστε ότι υπάρχουν κάποιες ειδικές περιπτώσεις, όπου ο μεταγλωττιστής δεν δημιουργεί τις προκαθορισμένες συναρτήσεις. Για παράδειγμα, αν η κλάση περιέχει αναφορά ή `const` μέλος, ο μεταγλωττιστής δεν δημιουργεί τη συνάρτηση αντιγραφής τιμής

Αποδομητής και Αποδέσμευση Μνήμης

- Ας δούμε ένα παράδειγμα που επιδεικνύει τη συνηθέστερη χρήση ενός αποδομητή, η οποία είναι η αποδέσμευση μνήμης που έχει δεσμευτεί από πεδία-δείκτες του αντικειμένου

```
#include <iostream>

class T
{
private:
        int *data;
public:
        T(int num);
        ~T();
};
T::T(int num)
{
        data = new int[num];
}
T::~T()
{
        delete[] data;
}
int main()
{
        T t(10);
        return 0;
}
```

Ο Δείκτης this

- Κάθε συνάρτηση μίας κλάσης, εκτός από τις στατικές συναρτήσεις όπως θα δούμε στη συνέχεια, έχει πρόσβαση σε έναν κρυμμένο δείκτη που ονομάζεται this
- Όταν καλείται μία μη-στατική συνάρτηση μέλος, ο this δείχνει στο αντικείμενο που κάλεσε τη συνάρτηση. Για παράδειγμα, θεωρήστε ότι η κλάση T περιέχει τις συναρτήσεις f() και g(), ένα ακέραιο μέλος με το όνομα mem και το t είναι ένα αντικείμενο αυτής της κλάσης. Αν γράψουμε:

  ```
  t.f();
  ```

  ο μεταγλωττιστής μεταβιβάζει την διεύθυνση του t αντικειμένου στην f() και ο this δείκτης στην f() αρχικοποιείται με αυτή την διεύθυνση
- Μέσα στη συνάρτηση κάθε απευθείας πρόσβαση σε ένα μέλος της κλάσης θεωρείται ότι είναι μία έμμεση αναφορά στο μέλος μέσω του δείκτη this. Για παράδειγμα:

  ```
  T::f()
  {
      mem = 10; // Είναι σαν να έχουμε γράψει this->mem = 10;
      g(); // Είναι σαν να έχουμε γράψει this->g();
  }
  ```

  Όταν προσπελαύνεται η mem, ο μεταγλωττιστής προσπελαύνει την mem του αντικείμενου στο οποίο δείχνει ο this. Δηλαδή, ο μεταγλωττιστής, στην πραγματικότητα, μεταφράζει την πρόσβαση σε this->mem

Πίνακας Αντικειμένων

- Ένας πίνακας αντικειμένων είναι ένας πίνακας, όπου το κάθε του στοιχείο είναι αντικείμενο. Για παράδειγμα:

```
#include <iostream>

class Test
{
public:
        int a, b;
        Test() {a = b = 1;}
        Test(int i, int j) {a = i; b = j;}
};
int main()
{
        int i;
        Test t[10] = {Test(1, 2), Test(), Test(5, 6)};

        for(i = 0; i < 10; i++)
                std::cout << t[i].a << ' ' << t[i].b << '\n';
        return 0;
}
```

Για τα t[0] και t[2] καλείται ο δεύτερος κατασκευαστής, ενώ για το t[1] ο εξ'ορισμού κατασκευαστής. Επειδή η λίστα αρχικοποίησης είναι μικρότερη από το πλήθος των στοιχείων, για τα υπόλοιπα επτά αντικείμενα καλείται ο εξ'ορισμού κατασκευαστής. Αν αυτός έλειπε, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους. Το πρόγραμμα εμφανίζει τις τιμές των πεδίων του κάθε αντικειμένου

Στατικά Μέλη Κλάσης

- Όταν σε μία κλάση θέλουμε να δηλώσουμε μία μεταβλητή που να τη **μοιράζονται όλα** τα αντικείμενα της κλάσης τη δηλώνουμε `static`. Μία `static` μεταβλητή μπορούμε να τη δηλώσουμε σε όποιο τμήμα πρόσβασης επιθυμούμε και είναι προσβάσιμη από **οποιοδήποτε** αντικείμενο. Ανήκει στην κλάση ως σύνολο, και όχι σε ένα συγκεκριμένο αντικείμενο. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
        int v;
        double d;
public:
        static inline int num = 5; // Δήλωση της στατικής μεταβλητής (C++17).
};
int main()
{
        T t[10];

        t[2].num = 20; // Εναλλακτικά, T::num = 20;
        std::cout << t[5].num << '\n';
        return 0;
}
```

Ο μεταγλωττιστής δεσμεύει μνήμη για ένα μόνο αντίγραφο της `static` μεταβλητής `num`, ανεξάρτητα από τον αριθμό των αντικειμένων που θα δημιουργηθούν. Επειδή δεσμεύεται μνήμη για μία μόνο `num`, η `num` είναι κοινή για όλα τα αντικείμενα και το πρόγραμμα εμφανίζει την τιμή 20

Στατικές Συναρτήσεις Κλάσης

- Μία συνάρτηση που είναι μέλος μίας κλάσης μπορεί να δηλωθεί στατική χρησιμοποιώντας τη λέξη `static`. Μία στατική συνάρτηση μέλος δεν συνδέεται με κάποιο αντικείμενο της κλάσης. Η λέξη `static` πρέπει να υπάρχει στη δήλωση της συνάρτησης αλλά όχι στον ορισμό της, αν αυτή ορίζεται έξω από την κλάση. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
          int code;
public:
          T(int c);
          static inline int cnt = 0;
          static int get_objs();
};
T::T(int c)
{
          code = c;
          cnt++;
}
int T::get_objs() // Δεν προσθέτουμε τη λέξη static.
{
          return cnt;
}
int main()
{
          T t1(5), t2(6);
   std::cout << t1.get_objs() << ' ' << T::get_objs() << '\n'; /* Αν η get_objs() είχε δηλωθε
στο private τμήμα, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους για μη επιτρεπτή πρόσβαση. */
          return 0;
}
```

Στατικές Συναρτήσεις Κλάσης

- Όταν δημιουργείται ένα αντικείμενο η στατική μεταβλητή `cnt` αυξάνεται. Η στατική συνάρτηση `get_objs()` επιστρέφει αυτή την τιμή. Άρα, αφού δημιουργούνται δύο αντικείμενα το πρόγραμμα εμφανίζει `2`
- Όπως φαίνεται, μία στατική συνάρτηση μέλος δεν χρειάζεται να κληθεί από κάποιο αντικείμενο
- Όπως και μία στατική μεταβλητή μέλος, μία στατική συνάρτηση μέλος μπορεί να κληθεί ακόμα και αν δεν έχει δημιουργηθεί κάποιο αντικείμενο της κλάσης (π.χ. `T::get_objs()`)
- Βέβαια, μπορείτε να χρησιμοποιήσετε ένα αντικείμενο για να την καλέσετε (π.χ. `t1`)

Άσκηση

- Προσθέστε στην κλάση T όποια συνάρτηση θεωρείτε σκόπιμο ώστε να λειτουργεί το παρακάτω πρόγραμμα

```
class T
{
private:
        char *s;
        ...
};
int main()
{
        char str[100];

        cout << "Enter text: ";
        cin.getline(str, sizeof(str)); /* Θεωρήστε ότι ο χρήστης θα εισάγει αλφαριθμητικό με λιγότερους από 100 χαρακτήρες. */
        T t1(str); // Το αλφαριθμητικό να αντιγράφεται στο πεδίο s.
        T t2;
        t1.show(); /* Η show() να εμφανίζει το αλφαριθμητικό που εισήγαγε ο χρήστης με αντίστροφη σειρά. Για παράδειγμα, αν ο χρήστης εισάγει abcd το πρόγραμμα να εμφανίζει dcba. Επίσης, υπάρχει ο περιορισμός η μόνη μεταβλητή που επιτρέπεται να δηλώσετε στη show() να είναι δείκτης. */
        return 0;
}
```

Άσκηση

```
#include <iostream>
#include <cstring>
using std::cout;
using std::cin;

class T
{
private:
          char *s;
public:
          T(char str[]);
          T();
          ~T();
          void show() const;
};

T::T()
{
          s = nullptr; /* Αρχικοποιούμε τον δείκτη, ώστε να μην υπάρχει πρόβλημα με την
αποδέσμευση της μνήμης. Για παράδειγμα, για το t2 δεν έχει δεσμευτεί μνήμη. */
}

T::T(char str[])
{
          s = new char[strlen(str)+1];
          strcpy(s, str);
}
T::~T()
{
          delete[] s;
}
void T::show() const
{
          for(char *p = s+strlen(s)-1; p >= s; p--)
                    cout << *p;
}
```

Άσκηση

- Σχόλια: Ας δούμε με ποια λογική προστέθηκαν οι παραπάνω συναρτήσεις. Αφού στη `main()` βλέπουμε ότι τα αντικείμενα δημιουργούνται με δύο τρόπους, επιλέγουμε να ορίσουμε δύο κατασκευαστές. Αφού το πεδίο `s` είναι δείκτης, ο κατασκευαστής πρέπει να δεσμεύσει μνήμη για την αντιγραφή του αλφαριθμητικού. Άρα, πρέπει να προστεθεί αποδομητής για την αποδέσμευση της μνήμης.

Υπερφόρτωση Τελεστών

- Όπως μία συνάρτηση μπορεί να υπερφορτωθεί, ένας τελεστής μπορεί επίσης να υπερφορτωθεί (operator overloading) και να εκτελεί διαφορετικές λειτουργίες
- Για να υπερφορτώσουμε έναν τελεστή δηλώνουμε μία συνάρτηση υπερφόρτωσης χρησιμοποιώντας τη λέξη `operator`, ως εξής:

```
τύπος_επιστροφής operatorop(παράμετροι)
```

- Ο `op` μπορεί να είναι οποιοσδήποτε από τους τελεστές της C++ (π.χ. +, -, ..) με κάποιες εξαιρέσεις
- Όπως κάθε συνηθισμένη συνάρτηση, έχει σώμα, τύπο επιστροφής και μπορεί να δέχεται παραμέτρους
- Ο αριθμός των παραμέτρων είναι ίδιος με τον αριθμό των τελεστέων που έχει τελεστής. Δηλαδή, η συνάρτηση υπερφόρτωσης ενός δυαδικού τελεστή δέχεται δύο παραμέτρους, ενώ ενός μοναδιαίου τελεστή δέχεται μία

Παράδειγμα

- Για παράδειγμα, στο παρακάτω πρόγραμμα ορίζεται η κλάση Rect, η οποία περιέχει μία συνάρτηση που υπερφορτώνει τον τελεστή +:

```
#include <iostream>

class Rect
{
private:
        float length;
        float height;
public:
        Rect(float l = 0, float h = 0); /*
Εξ'ορισμού κατασκευαστής με προκαθορισμένες
αρχικές τιμές. */
        Rect operator+(const Rect& r) const;
        float area() const;
        void show() const;
};

Rect::Rect(float l, float h)
{
        length = l;
        height = h;
}

Rect Rect::operator+(const Rect& r) const
{
        Rect tmp;

        tmp.length = length + r.length;
        tmp.height = height + r.height;
        return tmp;
}
```

```
float Rect::area() const
{
        return length * height;
}

void Rect::show() const
{
        std::cout << "L:" << length <<
" H:" << height << '\n';
}
int main()
{
    Rect r1(10, 20), r2(30, 40), r3;

        r3 = r1+r2; /* Ισοδύναμο με
r3 = r1.operator+(r2); */
        r3.show();
        return 0;
}
```

Παράδειγμα

- Όταν ο μεταγλωττιστής συναντήσει την εντολή `r1+r2` εξετάζει τους τύπους των τελεστέων και αντιλαμβάνεται ότι πρέπει να καλέσει την αντίστοιχη `operator+` συνάρτηση
- Συγκεκριμένα, η εντολή `r1+r2` ερμηνεύεται σαν `r1.operator+(r2)`, άρα καλείται η συνάρτηση `operator+` του αντικειμένου `r1` με όρισμα το `r2`
- Γενικά, όταν η συνάρτηση υπερφόρτωσης ενός δυαδικού τελεστή δηλώνεται στην κλάση, ο αριστερός τελεστέος (π.χ. `r1`) είναι το αντικείμενο που την καλεί, ενώ ο δεξιός τελεστέος (π.χ. `r2`) μεταβιβάζεται σαν όρισμα στη συνάρτηση
- Αν δεν είχε δηλωθεί η συνάρτηση `operator+`, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους για μη αποδεκτή ενέργεια, αφού δεν είναι δυνατή η πρόσθεση αντικειμένων
- Εναλλακτικά, θα μπορούσαμε να καλέσουμε ρητά τη συνάρτηση και να γράψουμε `r1.operator+(r2)`
- Απλά, η χρήση του τελεστή είναι μία πιο εύκολη, για να διαβάσουμε και να γράψουμε, συντομογραφία της ρητής κλήσης
- Η συνάρτηση επιστρέφει ένα νέο αντικείμενο με διαστάσεις ίσες με το άθροισμα των αντιστοίχων διαστάσεων των δύο αντικειμένων. Άρα, το πρόγραμμα εμφανίζει `40` και `60`

Παράδειγμα

- Μήπως παρατηρήσατε τίποτα παράξενο με τον αριθμό των παραμέτρων;
- Είπαμε, ότι ο αριθμός των παραμέτρων είναι ίδιος με τον αριθμό των τελεστέων που έχει τελεστής. Όμως, βλέπουμε ότι η `operator+` δέχεται μία παράμετρο, ενώ ο τελεστής `+` δέχεται δύο τελεστέους
- Τι συμβαίνει; Αν η συνάρτηση υπερφόρτωσης είναι μέλος, η διεύθυνση μνήμης του πρώτου τελεστέου (π.χ. `r1`), η οποία ουσιαστικά είναι η τιμή του δείκτη `this`, μεταβιβάζεται έμμεσα σαν ένα «κρυμμένο» όρισμα, όπως είδαμε στο Κ.17
- Αυτός είναι και ο λόγος που οι συναρτήσεις υπερφόρτωσης που είναι μέλη έχουν μία λιγότερη παράμετρο από τον αριθμό των τελεστέων
- Για παράδειγμα, αν η κλάση υπερφορτώνει τον τελεστή `!` που είναι μοναδιαίος γράφουμε `operator!()` χωρίς παραμέτρους
- Αν η συνάρτηση υπερφόρτωσης δεν είναι μέλος κλάσης, τότε δέχεται τον ίδιο αριθμό παραμέτρων με τους τελεστέους του

Αντιγραφή Αντικειμένων που Περιέχουν Δείκτες

- Ας δούμε τώρα την περίπτωση της αντιγραφής αντικειμένων που περιέχουν δείκτη(ες). Δώστε ιδιαίτερη **προσοχή**, ποιο είναι το **λάθος** στο παρακάτω πρόγραμμα;

```
#include <iostream>
#include <cstring>

class T
{
private:
          char *s;
public:
          T(const char str[]);
          ~T();
          void show() const {std::cout << s << '\n';}
};
T::T(const char str[])
{
          s = new char[strlen(str)+1];
          strcpy(s, str);
}
T::~T()
{
           delete[] s;
}
int main()
{
          T t1("Peter"), t2("Mike");
          t2 = t1;
          t2.show();
          return 0;
}
```

Παράδειγμα

- Η εντολή `t2 = t1;` προκαλεί πρόβλημα στη λειτουργία του προγράμματος. Ας δούμε γιατί:
- Με την αντιγραφή των μελών, ο δείκτης `t2.s` δείχνει στην ίδια μνήμη που δείχνει και ο δείκτης `t1.s`. Η μνήμη που είχε δεσμευτεί για τον `t2.s` δεν αποδεσμεύεται
- Όταν το πρόγραμμα τερματίζει καλείται ο αποδομητής του αντικειμένου `t2`, ο οποίος και απελευθερώνει την μνήμη στην οποία τώρα δείχνει ο `t2.s`. Όμως, την ίδια μνήμη επιδιώκει να απελευθερώσει και ο αποδομητής του `t1`, αφού οι δείκτες `t2.s` και `t1.s` δείχνουν στην ίδια μνήμη
- Όπως ξέρουμε, το αποτέλεσμα αυτής της ενέργειας είναι απροσδιόριστο (π.χ. το πρόγραμμα μπορεί να καταρρεύσει)
- Αν η κλάση περιέχει δείκτη(ες) σε μνήμη που έχει δεσμευτεί δυναμικά, το πιθανότερο είναι ότι θα πρέπει να ορίσετε τη δική σας συνάρτηση αντιγραφής, ώστε να αποφύγετε προβλήματα με τη διαχείριση της μνήμης

Αντιγραφή Αντικειμένων που Περιέχουν Δείκτες

- Σε μία τέτοια περίπτωση η δυνατότητα υπερφόρτωσης τελεστών αποδεικνύεται ιδιαίτερα χρήσιμη. Συγκεκριμένα, για να αποφύγουμε ανεπιθύμητες καταστάσεις μπορούμε να υπερφορτώσουμε τον τελεστή =, ώστε να καθορίσουμε τον τρόπο με τον οποίο θα γίνει η αντιγραφή. Για παράδειγμα:

```
class T
{
public:
            ...
            T& operator=(const T& t);
};

T& T::operator=(const T& t)
{
            if(this == &t) /* Ελέγχουμε για αυτοεκχώρηση, όπως με μία εντολή t1 = t1; */
                        return *this;

            delete[] s;
            s = new char[strlen(t.s)+1];
            strcpy(s, t.s);
            return *this;
}
```

- Τώρα, με την εντολή `t2 = t1;` η οποία είναι ισοδύναμη με `t2.operator=(t1);` πρώτα αποδεσμεύεται η μνήμη που είχε δεσμευτεί για το `t2.s` και μετά δεσμεύεται νέα μνήμη για να αντιγραφεί σε αυτήν το αλφαριθμητικό στο οποίο δείχνει ο `t1.s`. Η συνάρτηση επιστρέφει το ίδιο το αντικείμενο, δηλαδή, το `t2`. Άρα, τώρα το πρόγραμμα λειτουργεί κανονικά και εμφανίζει `Peter`

Περιορισμοί Υπερφόρτωσης (1)

- Ένας τουλάχιστον τελεστέος του υπερφορτωμένου τελεστή πρέπει να είναι τύπος δηλωμένος από τον προγραμματιστή. Αυτός ο κανόνας εμποδίζει την υπερφόρτωση τελεστών για τους βασικούς τύπους δεδομένων. Για παράδειγμα, δεν μπορείτε να υπερφορτώσετε τον τελεστή + για να επιστρέψετε τη διαφορά δύο ακεραίων αντί για το άθροισμα
- Δεν επιτρέπεται να αλλάξετε τον τρόπο σύνταξης του τελεστή. Για παράδειγμα, δεν επιτρέπεται να χρησιμοποιήσετε τον τελεστή + με έναν τελεστέο
- Δεν μπορείτε να αλλάξετε την προτεραιότητα και συσχέτιση του τελεστή. Για παράδειγμα, σε μία παράσταση `a*b+c` όπου τα `a`, `b` και `c` είναι αντικείμενα και οι τελεστές `*` και `+` έχουν υπερφορτωθεί πρώτα εκτελείται ο πολ/σμός και μετά η πρόσθεση

Περιορισμοί Υπερφόρτωσης (2)

- Δεν επιτρέπεται να δημιουργήσετε νέους τελεστές και στη συνέχεια να τους υπερφορτώσετε. Για παράδειγμα, δεν επιτρέπεται να δηλώσετε τη συνάρτηση `operator^^` για να δημιουργήσετε ύψωση σε δύναμη, αφού δεν υπάρχει ο τελεστής `^^`
- Δεν επιτρέπεται η υπερφόρτωση των παρακάτω τελεστών:

  `.` `.*` `::` `?:` `sizeof` `typeid` `alignas` `noexcept`
- Οι περισσότεροι τελεστές μπορούν να υπερφορτωθούν είτε με τη χρήση συναρτήσεων μελών είτε με συναρτήσεις που δεν είναι μέλη κάποιας κλάσης. Ωστόσο, οι παρακάτω τελεστές μπορούν να υπερφορτωθούν μόνο με μη στατικές συναρτήσεις μέλη:

  `=` `()` `[]` `->`
- Χωρίς αυτό να αποτελεί περιορισμό, όταν υπερφορτώνεται ένας τελεστής είναι ωφέλιμο για την καλύτερη κατανόηση και τον έλεγχο του προγράμματος να επιτελεί μία λειτουργία παρόμοια με τον αρχικό του σκοπό. Για παράδειγμα, μία συνάρτηση υπερφόρτωσης του τελεστή `+` να επιτελεί την πρόσθεση και όχι κάποια άλλη πράξη

Υπερφόρτωση των Τελεστών ++ και --

- Στο παράδειγμα με την κλάση, ας υπερφορτώσουμε τον τελεστή ++, παρόμοια υπερφορτώνεται και ο --, ώστε να αυξάνει τις διαστάσεις του αντικειμένου που τον καλεί κατά ένα
- Όπως γνωρίζουμε, ο τελεστής ++ μπορεί να εφαρμοστεί με δύο τρόπους. Επομένως, θα υλοποιήσουμε δύο εκδόσεις της συνάρτησης, μία για την επιθεματική και μία για την προθεματική αύξηση
- Για παράδειγμα, με την εντολή `++r;` η συνάρτηση να επιστρέφει το αντικείμενο `r` μετά την αύξηση των διαστάσεών του, ενώ με την `r++;` να επιστρέφει το `r` πριν την αύξηση
- Υπάρχει όμως ένα πρόβλημα. Αφού και οι δύο συναρτήσεις έχουν το ίδιο όνομα `operator++` και δεν δέχονται παραμέτρους με ποιο τρόπο ο μεταγλωττιστής θα καταλάβει ποια να καλέσει;
- Για να λυθεί αυτό το πρόβλημα, στην επιθεματική έκδοση προστίθεται μία ακέραια παράμετρος. Είναι μόνο ένα ψευδο-όρισμα που χρησιμοποιεί ο μεταγλωττιστής για να ξεχωρίσει ποια έκδοση να καλέσει
- Επίσης, η προθεματική έκδοση συνήθως επιστρέφει μία αναφορά στο αντικείμενο, ενώ η επιθεματική δημιουργεί και επιστρέφει ένα αντίγραφο του αρχικού αντικειμένου, όχι αναφορά

Υπερφόρτωση των Τελεστών ++ και --

```
Rect& Rect::operator++() /* Προθεματική έκδοση (π.χ. ++r). Η δήλωση
προστίθεται στην κλάση. */
{
        length++;
        height++;
        return *this;
}
Rect Rect::operator++(int) /* Επιθεματική έκδοση (π.χ. r++). Η δήλωση
προστίθεται στην κλάση. */
{
        Rect tmp;

        tmp = *this;
        length++;
        height++;
        return tmp;
}
int main()
{
        Rect r1(10, 20), r2, r3;

        r2 = ++r1; // Ισοδύναμη με r2 = r1.operator++();
        r1.show(); // Το πρόγραμμα εμφανίζει 11 21
        r2.show(); // Το πρόγραμμα εμφανίζει 11 21

        r3 = r1++; // Ισοδύναμη με r3 = r1.operator++(0);
        r1.show(); // Το πρόγραμμα εμφανίζει 12 22
        r3.show(); // Το πρόγραμμα εμφανίζει 11 21
        return 0;
}
```

Παράδειγμα

- Να προσθέσετε στην κλάση `Rect` μία συνάρτηση υπερφόρτωσης του τελεστή `==`, η οποία να δέχεται σαν παράμετρο ένα `Rect` αντικείμενο (π.χ. `r`) και να επιστρέφει `1` αν και οι δύο διαστάσεις του αντικειμένου που την καλεί είναι ίδιες με του `r`, `2` αν κάποια είναι ίδια, αλλιώς `3`. Τροποποιήστε τη `main()`, ώστε να ελέγξετε τη λειτουργία της συνάρτησης

Παράδειγμα

```
class Rect
{
        ...
public:
        int operator==(const Rect& r) const;
        ...
};
int Rect::operator==(const Rect& r) const
{
        if(length == r.length && height == r.height)
                return 1;
        else if(length == r.length || height == r.height)
                return 2;
        else
                return 3;
}
int main()
{
        int a;
        Rect r1(10, 20), r2(10, 0);

        a = (r1 == r2); // Ισοδύναμο με a = (r1.operator==(r2));
        if(a == 1)
                std::cout << "Both the same\n";
        else if(a == 2)
                std::cout << "One is the same\n";
        else
                std::cout << "Unequal\n";
        return 0;
}
```

Παράδειγμα

- Σχόλια. Όπως μάθαμε, η έκφραση (`r1 == r2`) δεν εκτελεί την συνηθισμένη σύγκριση, αλλά προκαλεί την κλήση της συνάρτησης `operator==` του αντικειμένου `r1` με όρισμα το `r2`. Μετά, μπορούμε να ελέγξουμε την τιμή επιστροφής της όπως κάνουμε με κάθε συνάρτηση. Οι παρενθέσεις δεν είναι απαραίτητες, απλά τις προσθέτουμε για να φαίνεται ξεκάθαρα ότι πρώτα θα εκτελεστεί η συνάρτηση

Παράδειγμα

- Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream>

class A
{
        int t;
public:
        A(int i) {t = i;}
        void operator+(int i) {t += i;}
};
class B
{
        int t;
public:
        B(int i) {t = i;}
};
int main()
{
        A a1(10), a2;
        B b1(a1.t);

        a2 = a1+5;
        a1+5;
        b1+5;
        5+a1;
        return 0;
}
```

Παράδειγμα

- Αφού στην κλάση A δηλώνεται ένας μη εξ'ορισμού κατασκευαστής, η δήλωση του `a2` είναι λάθος, γιατί δεν έχει δηλωθεί εξ'ορισμού κατασκευαστής
- Η δήλωση `b1(a1.t)` δεν είναι σωστή, γιατί δεν επιτρέπεται η πρόσβαση στο ιδιωτικό πεδίο `t`. Θυμηθείτε ότι η ετικέτα `private` δεν χρειάζεται, αφού τα πεδία μίας κλάσης είναι ιδιωτικά εξ'ορισμού
- Η εντολή `a2 = a1+5;` δεν είναι αποδεκτή, γιατί η συνάρτηση υπερφόρτωσης του τελεστή `+` δεν επιστρέφει κάτι
- Η εντολή `a1+5;` είναι αποδεκτή και προσθέτει το `5` στην μεταβλητή `t` του αντικειμένου `a1`
- Η εντολή `b1+5;` δεν είναι αποδεκτή, γιατί δεν ορίζεται συνάρτηση υπερφόρτωσης του τελεστή `+` στην κλάση `B`
- Τέλος, η εντολή `5+a1;` δεν είναι αποδεκτή, γιατί αριστερά του τελεστή πρέπει να βρίσκεται αντικείμενο της κλάσης `A`. Και για να το καταλάβετε καλύτερα, η `5+a1;` μεταφράζεται σε `5.operator(a1);` η οποία δεν είναι αποδεκτή έκφραση. Θα δούμε πώς να χειριστούμε αυτή την περίπτωση στην επόμενη ενότητα

Υπερφόρτωση με Συναρτήσεις μη Μέλη

- Όπως είπαμε, οι περισσότεροι τελεστές μπορούν να υπερφορτωθούν και με χρήση συναρτήσεων που δεν είναι μέλη κλάσης
- Μάλιστα, υπάρχουν περιπτώσεις που η χρήση τους μπορεί να είναι απαραίτητη. Για παράδειγμα, θεωρήστε την κλάση `A` της προηγούμενης άσκησης. Όπως είδαμε, η εντολή `a1+5;` είναι αποδεκτή, όχι όμως και η `5+a1;`
- Για να επιτύχουμε κάτι τέτοιο πρέπει να δηλώσουμε μία συνάρτηση που να μην είναι μέλος. Μία τέτοια συνάρτηση δεν μπορεί να κληθεί από κάποιο αντικείμενο. Συγκεκριμένα, οι τιμές των τελεστέων μεταβιβάζονται σαν ορίσματα στη συνάρτηση
- Η απαίτηση είναι ένας τουλάχιστον από τους τελεστέους να έχει τύπο κλάσης. Στο παράδειγμά μας, το πρωτότυπο της συνάρτησης είναι:

  ```
  void operator+(int i, A& a);
  ```

  Τώρα, η έκφραση `5+a1;` μεταφράζεται σε `operator+(5, a1);`
- Ας θυμόμαστε λοιπόν, όταν θέλουμε να υπερφορτώσουμε έναν τελεστή, όπου ο πρώτος τελεστέος να είναι κάποιος βασικός τύπος (π.χ. `int`), η συνάρτηση υπερφόρτωσης δεν μπορεί να είναι συνάρτηση μέλος της κλάσης. Αν θέλουμε η συνάρτηση να έχει πρόσβαση στα ιδιωτικά μέλη της κλάσης, την δηλώνουμε φιλική

Υπερφόρτωση με Συναρτήσεις μη Μέλη

```
#include <iostream>

class A
{
        int t;
public:
        A(int i) {t = i;}
        void operator+(int i) {t += i;}
        friend void operator+(int i, A& a);
        void show() const {std::cout << t << '\n';}
};
void operator+(int i, A& a)
{
        a.t += i;
}
int main()
{
        A a(10);

        a+5; // Μεταφράζεται σε a.operator+(5).
        a.show(); // Το πρόγραμμα εμφανίζει 15.
        5+a; // Μεταφράζεται σε operator+(5, a).
        a.show(); // Το πρόγραμμα εμφανίζει 20.
        return 0;
}
```

Υπερφόρτωση του << Τελεστή

- Τώρα που είδατε παραδείγματα υπερφόρτωσης τελεστών μπορείτε να καταλάβετε αυτό που διαβάσατε στο Κ.3, ότι οι τελεστές ολίσθησης << και >> έχουν υπερφορτωθεί για την εμφάνιση και εισαγωγή δεδομένων, αντίστοιχα
- Για παράδειγμα, το cout είναι αντικείμενο της κλάσης ostream, η οποία περιέχει υπερφορτωμένες συναρτήσεις operator<< για κάθε βασικό τύπο
- Ας δούμε πώς μπορούμε να χρησιμοποιήσουμε το cout με έναν δικό μας τύπο, όπως τον Rect. Ο συνηθισμένος τρόπος για να εμφανίσουμε τις τιμές των πεδίων μίας κλάσης είναι να ορίσουμε μία δημόσια συνάρτηση show() και να την καλέσουμε. Η ερώτηση είναι, θα μπορούσαμε, αντί της show(), να γράψουμε κάτι τέτοιο και να εμφανίζουμε τις τιμές τους;

```
Rect r(10, 20);
cout << r;
```

- Βεβαίως, αρκεί να υπερφορτώσουμε τον τελεστή <<. Πράγματι, είναι πολύ συνηθισμένη η υπερφόρτωση του τελεστή << και η χρήση του μαζί με το cout για την εμφάνιση των τιμών του αντικειμένου
- Δείτε πώς λειτουργεί. Αφού ο αριστερός τελεστέος είναι αντικείμενο της κλάσης ostream και ο δεξιός αντικείμενο της κλάσης Rect πρέπει να ορίσουμε μία συνάρτηση που να δέχεται σαν ορίσματα δύο τέτοιους τύπους. Να είναι η συνάρτηση φιλική στην κλάση Rect ή όχι; Αφού η συνάρτηση πρέπει να έχει πρόσβαση στα ιδιωτικά μέλη για να τα εμφανίσει, ναι πρέπει να είναι φιλική. Ας δούμε το πρόγραμμα:

Υπερφόρτωση του << Τελεστή

```
class Rect
{
        ...
public:
   friend ostream& operator<<(ostream& out, const Rect& r);
        ...
};
ostream& operator<<(ostream& out, const Rect& r)
{
        out << r.length << ' ' << r.height << '\n';
        return out;
}
int main()
{
        Rect r(10, 20);
        cout << r; // Μεταφράζεται σε operator<<(cout, r).
        return 0;
}
```

Άσκηση

- Να ορίσετε την κλάση Time με ιδιωτικά πεδία τα ώρες (π.χ. hrs), λεπτά (π.χ. mins) και δευτερόλεπτα (π.χ. secs). Να προσθέσετε κατάλληλες συναρτήσεις ώστε να λειτουργεί το παρακάτω πρόγραμμα.

```
int main()
{
	Time t1, t2(23, 59, 59); /* Υπάρχει ο περιορισμός η κλάση να έχει έναν κατασκευαστή. Τα πεδία του t1 να αρχικοποιούνται με 0. Επίσης, υπάρχει ο περιορισμός οι παράμετροι του κατασκευαστή να έχουν τα ίδια ονόματα με τα ιδιωτικά πεδία. */
	++t2; /* Αυξάνεται ο αριθμός των δευτερολέπτων κατά ένα. Αν η ώρα είναι 23:59:59, όπως στην περίπτωση του t2, η νέα ώρα να γίνει 0:0:0. */
	cout << (t1 == t2) << '\n'; /* Να συγκρίνονται οι δύο ώρες και αν είναι ίδιες, το πρόγραμμα να εμφανίζει 1 αλλιώς 0. Για παράδειγμα, με αυτές τις τιμές των t1 και t2 το πρόγραμμα εμφανίζει 1. */
	cout << t1; /* Να εμφανίζεται η ώρα στη μορφή h:m:s. Για παράδειγμα, 0:0:0. */
	return 0;
}
```

Άσκηση

```
#include <iostream>
using std::cout;
using std::ostream;

class Time
{
private:
        int hrs;
        int mins;
        int secs;
public:
        Time(int hrs=0, int mins=0, int secs=0);
        bool operator==(const Time& t) const;
        void operator++();
        friend ostream& operator<<(ostream& out, const Time& t);
};
Time::Time(int hrs, int mins, int secs)
{
        this->hrs = hrs; /* Επειδή υπάρχει ο περιορισμός για ίδια ονόματα
χρησιμοποιούμε τον δείκτη this. */
        this->mins = mins;
        this->secs = secs;
}
bool Time::operator==(const Time& t) const
{
        if((hrs == t.hrs) && (mins == t.mins) && (secs == t.secs))
                return true;
        else
                return false;
}
```

Άσκηση

```
void Time::operator++()
{
        secs++;
        if(secs == 60)
        {
                secs = 0;
                mins++;
                if(mins == 60)
                {
                        mins = 0;
                        hrs++;
                        if(hrs == 24)
                                hrs = 0;
                }
        }
}
ostream& operator<<(ostream& out, const Time& t)
{
        out << t.hrs << ':' << t.mins << ':' << t.secs << '\n';
        return out;
}
```

Περισσότερα για Κλάσεις

Λίστα Αρχικοποίησης Μελών

- Όπως ξέρουμε, ένας τρόπος για να αρχικοποιήσουμε τα μέλη μίας κλάσης είναι μέσα στο σώμα του κατασκευαστή. Για παράδειγμα:

```
class T
{
private:
        int a;
public:
        double b;
        string s;
        T(int arg1, double arg2);
};

T::T(int arg1, double arg2)
{
        a = arg1;
        b = arg2;
        s = "Text";
}
```

- Εναλλακτικά, τα μέλη μίας κλάσης μπορούν να αρχικοποιηθούν με μία ειδική σύνταξη όταν καλείται ο κατασκευαστής της κλάσης. Για παράδειγμα:

```
T::T(int arg1, double arg2) : s("Text"), a(arg1), b(arg2) /* Λίστα
αρχικοποίησης μελών. */
{
}
```

Λίστα Αρχικοποίησης Μελών

- Αυτή η σύνταξη αρχικοποίησης μελών (member initializer list) μπορεί να χρησιμοποιηθεί μόνο με κατασκευαστές, και όχι με συναρτήσεις μέλη της κλάσης
- Όπως φαίνεται, προστίθεται ένα `:` και οι αρχικοποιήσεις των μελών χωρίζονται με κόμμα. Πρώτα γράφουμε το όνομα του μέλους και μετά την αρχική τιμή μέσα σε παρενθέσεις
- Έτσι, το `Text` αποθηκεύεται στο `s`, το `a` γίνεται ίσο με `arg1` και το `b` ίσο με `arg2`
- Αν και το λογικό είναι να νομίζετε ότι οι αρχικοποιήσεις γίνονται από αριστερά προς τα δεξιά, δηλαδή, πρώτα το `s` αρχικοπόιείται και μετά τα υπόλοιπα, η πραγματικότητα είναι ότι τα μέλη αρχικοποιούνται με την σειρά με την οποία εμφανίζονται στη δήλωση της κλάσης, δηλαδή, πρώτα το `a`, μετά το `b`, και τελευταίο το `s`

Κατασκευαστής Αντιγράφου (1)

- Ο κατασκευαστής αντιγράφου (copy constructor) είναι μία ειδική συνάρτηση που καλείται **αυτόματα** όταν δημιουργείται ένα νέο αντικείμενο και αρχικοποιείται με ένα υπάρχον. Για παράδειγμα:

```
#include <iostream>

class T
{
private:
        int a;
public:
        int b;
        T(int i, int j) {a=i; b=j;}
        void show() const {std::cout << a << ' ' << b << '\n';}
};
int main()
{
        T t1(1, 2);
        T t2(t1); // Καλείται ο εξ'ορισμού κατασκευαστής αντιγράφου.
        T t3 = t1; /* Καλείται ο εξ'ορισμού κατασκευαστής
αντιγράφου. */
        t2.show();
        t3.show();
        return 0;
}
```

Κατασκευαστής Αντιγράφου (2)

- Όπως ο μεταγλωττιστής παρέχει τον εξ'ορισμού κατασκευαστή αν δεν έχει οριστεί κατασκευαστής, τον εξ'ορισμού αποδομητή αν δεν έχει οριστεί αποδομητής και την εξ'ορισμού συνάρτηση αντιγραφής, παρέχει και τον εξ'ορισμού κατασκευαστή αντιγράφου αν δεν έχει οριστεί κάποιος
- Έτσι, για τη **δημιουργία** των `t2` και `t3` αντικειμένων καλείται ο **εξ'ορισμού** κατασκευαστής αντιγράφου
- Και τι κάνει αυτός ο κατασκευαστής; Ότι λέει το όνομά του, δηλαδή, **αντιγράφει** ένα-προς-ένα τα **μη στατικά** μέλη του **υπάρχοντος** αντικειμένου στο **νέο**. Για παράδειγμα, με την εντολή: `T t2(t1);` ο μεταγλωττιστής δημιουργεί το `t2` αντικείμενο και **καλεί** τον κατασκευαστή αντιγράφου για να **αντιγράψει** τα πεδία του `t1` στο `t2`
- Αυτή η αντιγραφή ονομάζεται και **ρηχή αντιγραφή** (shallow copy). Το πώς θα αντιγραφεί το κάθε μέλος εξαρτάται από τον τύπο του. Οι βασικοί τύποι αντιγράφονται απευθείας, αν είναι πίνακας γίνεται αντιγραφή των στοιχείων του και αν είναι αντικείμενο καλείται ο κατασκευαστής αντιγράφου της κλάσης του για να γίνει η αντιγραφή

Κατασκευαστής Αντιγράφου (3)

- Σημειώστε ότι σε μία συνηθισμένη εντολή εκχώρησης, όπως `t2 = t1`, δεν καλείται ο κατασκευαστής αντιγράφου, αλλά καλείται η εξ'ορισμού συνάρτηση αντιγραφής operator=, η οποία αντιγράφει τα πεδία του `t1` στο `t2`
- Για να είναι σαφές, ο κατασκευαστής αντιγράφου καλείται όταν το αντικείμενο **κατασκευάζεται** (π.χ. `T t2 = t1;`), ενώ όταν το αντικείμενο υπάρχει (π.χ. `t2 = t1;`) καλείται η συνάρτηση αντιγραφής
- Αν η κλάση περιέχει στατικά μέλη αυτά δεν επηρεάζονται αφού ανήκουν στην κλάση και όχι σε κάποιο αντικείμενο
- Το πρόγραμμα εμφανίζει δύο φορές `1 2`

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

- Αν η κλάση περιέχει μέλη που είναι δείκτες, προβλήματα μπορεί να προκύψουν αν χρησιμοποιήσουμε τον εξ'ορισμού κατασκευαστή αντιγράφου (και την εξ'ορισμού συνάρτηση αντιγραφής, όπως είδαμε στο Κ.18). Για παράδειγμα, δείτε το παρακάτω πρόγραμμα και προσπαθήστε να βρείτε ποιο είναι το πρόβλημα

```
#include <iostream>
#include <cstring>
class T
{
public:
        char *s;
        T(const char str[]);
        ~T();
};
T::T(const char str[])
{
        s = new char[strlen(str)+1];
        strcpy(s, str);
}
T::~T()
{
         delete[] s;
}
int main()
{
        T t1("text");
        T t2 = t1; // Καλείται ο εξ'ορισμού κατασκευαστής αντιγράφου.
        t2.s[0] = 'a';
        std::cout << t1.s << '\n';
        return 0;
```

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

- Το πρόβλημα οφείλεται στην κλήση του εξ'ορισμού κατασκευαστή αντιγράφου όταν δημιουργείται το `t2`
- Συγκεκριμένα, το `t2.s` θα γίνει ίσο με το `t1.s`, δηλαδή, και οι δύο δείκτες θα δείχνουν στην ίδια μνήμη. Άρα, όταν αλλάζει το `t2.s[0]` αλλάζει και το `t1.s[0]`. Έτσι, το πρόγραμμα εμφανίζει `aext`
- Εκτός από αυτό, το ότι και οι δύο δείκτες δείχνουν στην ίδια μνήμη προκαλεί και άλλο λάθος. Ειδικότερα, όταν καταστραφεί το `t2` καλείται ο αποδομητής του και αποδεσμεύεται η μνήμη στην οποία δείχνει ο `t2.s`, η οποία, όπως είπαμε, είναι αυτή που δείχνει και ο `t1.s`. Άρα, όταν καταστραφεί το `t1` ο αποδομητής του επιχειρεί να απελευθερώσει την ίδια μνήμη που είναι λανθασμένη ενέργεια. Και πώς διορθώνονται αυτά τα προβλήματα;
- Η λύση είναι να ορίσουμε τον δικό μας κατασκευαστή αντιγράφου. Δηλαδή, προσθέτουμε στην κλάση την παρακάτω συνάρτηση:

```
class T
{
public:
        ...
        T(const T& t);
};
T::T(const T& t)
{
        s = new char[strlen(t.s)+1];
        strcpy(s, t.s);
}
```

Κατασκευαστής Αντιγράφου και Μέλη Δείκτες

- Τώρα, όταν δημιουργείται το `t2` καλείται αυτός ο κατασκευαστής αντιγράφου και το `t1` μεταβιβάζεται σαν όρισμα
- Δηλαδή, η αναφορά `t` αναφέρεται στο `t1` αντικείμενο. Στη συνέχεια, δεσμεύεται μνήμη για το `t2.s` και αντιγράφεται σε αυτήν το `t1.s`
- Αυτή η αντιγραφή ονομάζεται και **βαθιά αντιγραφή** (deep copy) με την έννοια ότι για κάθε πεδίο-δείκτη του αντικειμένου που δημιουργείται, πρώτα δεσμεύεται μνήμη και μετά γίνεται η αντιγραφή σε αυτήν
- Έτσι, αυτή τη φορά το πρόγραμμα εμφανίζει `text` και όχι `aext` όπως προηγουμένως
- Επίσης, αφού η μνήμη που δείχνει ο `t1.s` είναι διαφορετική από τη μνήμη που δείχνει ο `t2.s`, δεν θα δημιουργηθεί πρόβλημα όταν στο τέλος του προγράμματος κληθεί ο αποδομητής του κάθε αντικειμένου
- Αν η κλάση **περιέχει** δείκτη(ες), θα **πρέπει** να ορίσετε τον **δικό** σας **κατασκευαστή αντιγράφου** και να δεσμεύσετε μνήμη για να αντιγράψετε τα πεδία του πρωτότυπου αντικειμένου
- Για τον ίδιο λόγο, όπως είδαμε σε παρόμοιο παράδειγμα στο προηγούμενο κεφάλαιο, θα πρέπει να ορίσετε και τη **δική** σας **συνάρτηση αντιγραφής** με την υπερφόρτωση του τελεστή `=`