Αντικειμενοστρεφής Προγραμματισμός

Διαφάνειες του μαθήματος

Απόστολος Ζάρρας
Πάνος Βασιλειάδης
Παναγιώτης Τσαπάρας

www.cs.uoi.gr/~pvassil/courses/oop/

Εισαγωγή στον Αντικειμενοστρεφή Προγραμματισμό (Object Oriented Programming)

Εισαγωγή

- Ως σχεδιαστές/προγραμματιστές λογισμικού σκοπό έχουμε να υλοποιούμε λογισμικό για την επίλυση προβλημάτων με βάση κάποιες δοσμένες απαιτήσεις.

Εισαγωγή

- Η ανάπτυξη ενός λογισμικού αποτελείται από μια ακολουθία από διακριτές φάσεις.
  - **Ανάλυση απαιτήσεων**
    - Τι ζητάει το πρόβλημα
  - **Σχεδίαση**
    - Από ποια βασικά δομικά στοιχεία θα αποτελείται το λογισμικό.
      - Πχ. Δομές δεδομένων, συναρτήσεις, κλάσεις
  - **Υλοποίηση / Κατασκευή**
    - Κωδικοποίηση σε C, C++, Java, Fortran …
  - **Έλεγχος**
    - αποσφαλμάτωση
  - **Εγκατάσταση**
  - **Συντήρηση**

Εισαγωγή

- Στην όλη διαδικασία ανάπτυξης ενός λογισμικού εμπλέκονται διάφοροι χαρακτήρες (actors)
  - Ο πελάτης που καθορίζει τις απαιτήσεις
  - Η ομάδα σχεδίασης
  - Η ομάδα υλοποίησης
  - Η ομάδα ελέγχου
  - Η ομάδα εγκατάστασης
  - Η ομάδα συντήρησης

Εισαγωγή

- Όσο **μεγαλύτερης κλίμακας** είναι το λογισμικό που πρόκειται να αναπτύξουμε **τόσο πιο χρήσιμος είναι ο αντικειμενοστρεφής τρόπος σκέψης** και υλοποίησης.
- Σε τέτοιες περιπτώσεις **το μείζον πρόβλημα δεν είναι πλέον η υλοποίηση ενός μόνο αλγορίθμου ή η μετατροπή ενός αλγορίθμου σε κώδικα**
  - υπάρχουν διάφοροι actors καθένας από τους οποίους έχει τη δική του αρμοδιότητα στην επίλυση του συνολικού προβλήματος

Εισαγωγή

- **Το μείζον πρόβλημα είναι η βέλτιστη σχεδίαση και υλοποίηση του κώδικα** με στόχους
  - Επιτάχυνση της φάσης της υλοποίησης
  - Διευκόλυνση της φάσης του ελέγχου
  - Διευκόλυνση της συντήρησης
    - Ειδικότερα η φάση της συντήρησης είναι εξαιρετικά σημαντική σε εφαρμογές μεγάλης κλίμακας και το κόστος της είναι συνήθως μεγαλύτερο από το συνολικό κόστος όλων των υπόλοιπων φάσεων
    - Τι είναι συντήρηση?
      - επιδιόρθωση σφαλμάτων που προκύπτουν μετά την εγκατάσταση και χρήση του λογισμικού
      - προσθήκη νέων λειτουργιών λόγω της μεταβολής των αρχικών απαιτήσεων των χρηστών του λογισμικού

Η γενική ιδέα…..

- **Για να πετύχουμε τους στόχους αυτούς….**
  - Ομαδοποίηση των δεδομένων που διαχειρίζεται το λογισμικό μας και του κώδικα που χρησιμοποιείται για την διαχείριση αυτών των δεδομένων σε κλάσεις αντικειμένων.

Η γενική ιδέα…..

- … σε πρώτη φάση, φανταστείτε μια κλάση σαν ένα struct X
  - το οποίο πέρα από τα πεδία του ορίζει και
  - ένα σύνολο από συναρτήσεις μέσω των οποίων το υπόλοιπο λογισμικό επεξεργάζεται τα δεδομένα που αποθηκεύονται στα πεδία μεταβλητών τύπου X

Η γενική ιδέα…..

```
struct Human {
   int height;
   int age;
};
void isborn(struct Human *aHuman)
void ages(struct Human *aHuman);

main(){
 struct Human peter;
 isborn(&peter)
 ages(&peter);
}

void isborn(struct Human *aHuman){
  aHuman->height = 0;
  aHuman->age = 0;
}
void ages(struct Human *aHuman){
  aHuman->age += 1;
  aHuman->height += 20;
}
```

Η γενική ιδέα…..

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

class Human {
private:
   int height;
   int age;
public:
   void ages();
   void isborn();
};

void Human::ages(){
 age += 1; // απλούστερος κώδικας !!!
}
void Human::isborn(){
 printf("A human is born");
……
}
int main(){
  Human peter;
  peter.isborn();
  peter.ages(); // απλούστερος κώδικας !!!
}
```

10

Η γενική ιδέα…..

- Λίγη ορολογία ….
  - Τα πεδία τα λέμε χαρακτηριστικά της κλάσης
  - Οι συναρτήσεις ονομάζονται μέθοδοι της κλάσης
  - Οι μεταβλητές τύπου Χ ονομάζονται αντικείμενα της κλάσης Χ
  - Οι τιμές των πεδίων ενός αντικειμένου κατά τη διάρκεια εκτέλεσης του λογισμικού ονομάζεται κατάσταση του αντικειμένου
  - Οι μέθοδοι της κλάσης ορίζουν τη συμπεριφορά των αντικειμένων της
    - Δηλ τον τρόπο με τον οποίο μεταβάλλεται η κατάσταση των αντικειμένων της κλάσης όταν καλεστεί μια μέθοδος….

11

Η γενική ιδέα…..

- Διευκόλυνση της φάσης του ελέγχου
- Επιτάχυνση της φάσης της υλοποίησης
- Διευκόλυνση της συντήρησης

- Ενθυλάκωση (encapsulation)
- Συνάθροιση αντικειμένων (aggregation)
- Σύνθεση αντικειμένων (composition)
- Κληρονομικότητα (inheritance) & Πολυμορφισμός

12

Ενθυλάκωση

- *Η βασική ιδέα είναι ότι χωρίζουμε τα χαρακτηριστικά και τις μεθόδους σε public και private.*
  - *Private χαρακτηριστικά*: η επεξεργασία των τιμών τους για ένα αντικείμενο γίνεται μόνο μέσω των μεθόδων της κλάσης
  - *Public μέθοδοι*: μπορούν να χρησιμοποιηθούν οπουδήποτε στον κώδικα για την αλλαγή της κατάστασης ενός αντικειμένου της κλάσης
    - …τα σφάλματα δημιουργούνται σε σημεία του κώδικα στα οποία γίνεται επεξεργασία δεδομένων (π.χ. ανάθεση τιμών)
    - ….αν η επεξεργασία γίνεται μέσω συγκεκριμένων μεθόδων τα σφάλματα βρίσκονται εκεί και όχι διάσπαρτα σε όλο τον κώδικα !

13

Συνάθροιση & Σύνθεση Αντικειμένων

- Μπορούμε να κατασκευάσουμε αντικείμενα που αποτελούνται από άλλα επιμέρους αντικείμενα – επαναχρησιμοποιώντας κώδικα…..

```
class Door {
  int code; ….
  …….
};

class Room {
  Door x;
  …….
};
```

```
class Building {
   Room parts[10];
   Door main_door;
};
main () {
   Building xxx;
}
```

14

Κληρονομικότητα

- Φανταστείτε ένα πρόβλημα που έχει ως αντικείμενα πελάτες, υπαλλήλους και άλλα συναφή
- Τόσο οι **πελάτες** όσο και οι **υπάλληλοι** έχουν κοινά χαρακτηριστικά και συμπεριφορά (μεθόδους)
  - Θα πρέπει να φτιάξω 2 κλάσεις.
    - π.χ. Ύψος, βάρος κλπ.
  - Θαχω πολλές φορές τον ίδιο κώδικα.

15

Κληρονομικότητα

- Για να το αποφύγω και να έχω καλύτερη επαναχρησιμοποίηση δεδομένων και λειτουργιών.
  - *Ορίζω μια **βασική κλάση***
    - Αυτή θα ορίζει τα κοινά χαρακτηριστικά/μεθόδους
    - Η βασική κλάση επαναχρησιμοποιείται για τον ορισμό νέων κλάσεων μέσω του μηχανισμού της κληρονομικότητας (πχ πελάτης, υπάλληλος) στις οποίες ορίζονται **επιπλέον χαρακτηριστικά και μέθο**δοι

Διαδικαστικός vs. Αντικειμενοστρεφής Προγραμματισμός

Θεματολόγιο

- Σύγκριση διαδικαστικού και αντικειμενοστρεφούς τρόπου σκέψης με παραδείγματα
  - 1ο Παράδειγμα: διευκόλυνση ελέγχου και αποσφαλμάτωσης λογισμικού
  - 2ο Παράδειγμα: διευκόλυνση συντήρησης λογισμικού

1ο Παράδειγμα

1ο Παράδειγμα

- Έστω ένα απλό πρόγραμμα του χειρίζεται πληροφορία για ένα σύνολο ανθρώπων
  - Κάθε άνθρωπος χαρακτηρίζεται από
    - όνομα
    - ηλικία
    - …..

1ο Παράδειγμα

- Spaghetti τρόπος σκέψης….
  - όλος ο κώδικας στην main()

1ο Παράδειγμα

```
# include <stdio.h>
# include <stdlib.h>

struct Person {
  char * name;
  int age;
};
```

1ο Παράδειγμα

```
int main(){

  struct Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```


1ο Παράδειγμα

- Έστω κατά την εκτέλεση του προγράμματος προκύπτει πρόβλημα λόγω της απουσίας ελέγχου για το αν έχει επιτύχει η δυναμική δέσμευση μνήμης……

1ο Παράδειγμα

```
int main(){

  struct Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2"); /* BOOM!!! */
  printf ("%s", y.name);

  return 0;
}
```

1ο Παράδειγμα

- Για να το επιλύσει πρέπει να αναζητηθούν όλα τα σημεία του κώδικα στα οποία γίνεται δέσμευση μνήμης για το πεδίο name των εκάστοτε μεταβλητών τύπου Person…….
- αυτός που κάνει την αναζήτηση δεν είναι απαραίτητα αυτός που κατασκεύασε το πρόγραμμα
  - ομάδα ελέγχου
  - ομάδα συντήρησης

1ο Παράδειγμα

```
int main(){

  struct Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2"); /* BOOM!!! */
  printf ("%s", y.name);

  return 0;
}
```


1ο Παράδειγμα

- Προσθήκη ελέγχου σε **κάθε σημείο**……

1ο Παράδειγμα

```
int main(){
  struct Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  if(x.name == NULL) {printf("…..."); return 1;}
  x.age = 0;
  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  if(y.name == NULL) {printf("…..."); return 1;}
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```


1ο Παράδειγμα

- Διαδικαστικός τρόπος σκέψης … **αν σχεδιάζαμε εξαρχής καλύτερα το πρόγραμμα**…
  - κατασκευή μιας συνάρτησης για την αρχικοποίηση μεταβλητών τύπου Person
  - ο κώδικας που δεσμεύει δυναμικά μνήμη θα είναι συγκεντρωμένος σε μια συνάρτηση και όχι διάσπαρτος σε όλο το πρόγραμμα….

1ο Παράδειγμα

```
# include <stdio.h>

struct Person {
  char * name;
  int age;
};

void init(struct Person *p){
  p->name = (char *) malloc (10* sizeof(char));
  p->age = 0;
}
```

1ο Παράδειγμα

```
int main(){

  struct Person x, y;

  init(&x);
  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);


  init(&y);
  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```


1ο Παράδειγμα

- Έστω κατά την εκτέλεση του προγράμματος προκύπτει πρόβλημα λόγω της απουσίας ελέγχου για το αν έχει επιτύχει η δυναμική δέσμευση μνήμης……

1ο Παράδειγμα

- Για να το επιλύσει πρέπει να αναζητηθεί η συνάρτηση στην οποία γίνεται δέσμευση μνήμης για το πεδίο name των εκάστοτε μεταβλητών τύπου Person…….
  - **οι συνθήκες αναζήτησης έχουν βελτιωθεί** σε σχέση με την προηγούμενη έκδοση του προγράμματος (spaghetti τρόπος σκέψης)
  - μόλις βρούμε την συνάρτηση έχουμε τελειώσει…..
  - 1 αλλαγή αντί για πολλές
- αυτός που κάνει την αναζήτηση δεν είναι απαραίτητα αυτός που κατασκεύασε το πρόγραμμα
  - Πιθανότατα πρέπει να ελεγχθούν πολλές συναρτήσεις για να βρούμε ποια είναι αυτή που πρέπει να διορθωθεί

1ο Παράδειγμα

```
# include <stdio.h>

struct Person {
  char * name;
  int age;
};

void init(struct Person *p){
  p->name = (char *) malloc (10* sizeof(char));
  if (p -> name == NULL) {printf("…..."); exit(1);}
  p->age = 0;
}
```


1ο Παράδειγμα

```
int main(){

  struct Person x, y;

  init(&x);
  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);


  init(&y);
  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  f();

  return 0;
}
```


1ο Παράδειγμα

- Αντικειμενοστρεφής τρόπος σκέψης … **αν σχεδιάζαμε εξαρχής καλύτερα το πρόγραμμα**…
  - Ομαδοποίηση του τύπου δεδομένων Person και του κώδικα που χρησιμοποιείται για την επεξεργασία αυτών των δεδομένων
    - π.χ. της συνάρτησης init()
  - **…. ορισμός της κλάσης Person ….**

1ο Παράδειγμα

```
# include <cstdlib>
# include <cstring>

// αυτό είναι σχόλιο !!

class Person {
 public:                          // τι είναι αυτό το public ??

  char * name;                    // πεδία κλάσης
  int age;

  void init();                    // μέθοδοι κλάσης
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();

}; // προσοχή στο semicolon – είναι υποχρεωτικό
```


1ο Παράδειγμα

```
void Person::init(){
  name = (char *) malloc (10* sizeof(char));
  age = 0;
}

void Person::set_name(char *n) {
  strcpy(name, n);
}
void Person::set_age(int a) {
  age = a;
}

char *Person::get_name() {
  return name;
}
int Person::get_age() {
  return age;
}
```


1ο Παράδειγμα

```
int main(){

  Person x, y; // δήλωση αντικειμένου

  x.init(); // κλήση μεθόδου στο αντικείμενο

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());


  y.init();

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  return 0;
}
```

1ο Παράδειγμα

- Έστω ότι κατά την εκτέλεση του προγράμματος προκύπτει πρόβλημα λόγω της απουσίας ελέγχου για το αν έχει επιτύχει η δυναμική δέσμευση μνήμης……

1ο Παράδειγμα

```
int main(){

  Person x, y; // δήλωση αντικειμένου

  x.init(); // κλήση μεθόδου στο αντικείμενο

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());


  y.init();

  /* .... */
  y.set_name("Name2"); /* BOOOM */
  printf ("%s", y.get_name());

  return 0;
}
```


1ο Παράδειγμα

- Για επιλυθεί πρέπει να αναζητηθεί η συνάρτηση στην οποία γίνεται δέσμευση μνήμης για το πεδίο name των εκάστοτε μεταβλητών τύπου Person…….
  - **οι συνθήκες αναζήτησης έχουν βελτιωθεί** ακόμα περισσότερο σε σχέση με την προηγούμενη έκδοση του προγράμματος (διαδικαστικός τρόπος σκέψης)
  - από τις δεκάδες / εκατοντάδες συναρτήσεις/μεθόδους του προγράμματος γνωρίζουμε ποιες χρησιμοποιούνται για την επεξεργασία μεταβλητών τύπου Person από τη δήλωση της κλάσης…..

1ο Παράδειγμα

```
# include <cstdlib>
# include <cstring>


class Person {
 public:

   char * name;          // πεδία κλάσης
   int age;

   void init();          // μέθοδοι κλάσης
   void set_name(char *n);
   void set_age(int a);
   char *get_name();
   int get_age();

};  // προσοχή στο semicolon – είναι υποχρεωτικό
```

μόνο αυτές οι μέθοδοι είναι ύποπτες …… !!!

28

1ο Παράδειγμα

```
void Person::init(){
  name = (char *) malloc (10* sizeof(char));
  if (name == NULL) {printf(…); exit(1);}
  age = 0;
}

void Person::set_name(char *n) {
  strcpy(name, n);
}
void Person::set_age(int a) {
  age = a;
}

char *Person::get_name() {
  return name;
}
int Person::get_age() {
  return age;
}
```

29

1ο Παράδειγμα

```
int main(){

  Person x, y; // δήλωση αντικειμένου

  x.init(); // κλήση μεθόδου στο αντικείμενο

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());


  y.init();

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  return 0;
}
```

30

1ο Παράδειγμα

- Η χρήση κλάσεων από μόνη της λύνει το πρόβλημα μας ????
  - (δηλαδή την εύρεση των σημείων του κώδικα στο οποίο πρέπει να προστεθούν έλεγχοι)
- Γενικά όχι, αν πρόκειται για ένα λογισμικό μεγάλης κλίμακας, ο κώδικας που κατασκευάζεται από έναν προγραμματιστή μπορεί να χρησιμοποιηθεί από έναν άλλο προγραμματιστή
  - τι μπορεί να προκύψει σε αυτή την περίπτωση ???

31

1ο Παράδειγμα

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
 public:
  char * name;
  int age;

  void init();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

32

1ο Παράδειγμα

- Κάθε πεδίο ή μέθοδο που δηλώνουμε ως public σε μια κλάση μπορεί να χρησιμοποιηθεί σε όλο τον υπόλοιπο κώδικα ….
- Άρα, ένας απρόσεκτος προγραμματιστής που θα χρησιμοποιήσει αντικείμενα της κλάσης μας, μπορεί να αρχικοποιεί απευθείας τα πεδία αυτών των αντικειμένων, ξεχνώντας παράλληλα να κάνει τους απαραίτητους ελέγχους…..
  - άρα, να εισάγει -εν αγνοία του- λογικά σφάλματα ….

33

1ο Παράδειγμα

```
int main(){

  Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```


1ο Παράδειγμα

- Η λύση στο πρόβλημα αυτό ….
  - δηλαδή, της ανεξέλεγκτης πρόσβασης σε πεδία και μεθόδους μιας κλάσης από άλλους προγραμματιστές
- είναι η αρχή της **ενθυλάκωσης** (encapsulation):
  - σε κάθε κλάση, ο προγραμματιστής που την υλοποίησε, διαχωρίζει:
    - ποια πεδία και μέθοδοι (**public**) μπορούν να χρησιμοποιηθούν από όλο τον υπόλοιπο κώδικα
    - ποια πεδία και μέθοδοι (**private**) μπορούν να χρησιμοποιηθούν μόνο μέσω κλήσης των public μεθόδων της κλάσης…
  - μια συνήθης πρακτική την οποία πρέπει να ακολουθείτε και εσείς είναι ότι ***όλα τα πεδία τα δηλώνουμε* private**

1ο Παράδειγμα

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
private:
  char * name;
  int age;

public:
  void init();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

1ο Παράδειγμα

```
int main(){

  Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```

προσπάθεια πρόσβασης σε private πεδία =>

COMPILER ERRORS!!!

Παρενθετικά…

- τα προγράμματα C++ τα αποθηκεύουμε σε αρχεία .cpp
- τα κάνουμε compile χρησιμοποιώντας g++ αντί για gcc

2ο Παράδειγμα

2ο Παράδειγμα

- Έστω μια εφαρμογή διαχείρισης αριθμών φορολογικού μητρώου….

Ανάλυση απαιτήσεων…

- δύο βασικές αρμοδιότητες
  - καταχώρηση νέων αριθμών φορολογικού μητρώου
    - ΑΦΜ = ακέραιος αριθμός με 5 ψηφία
  - καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, …, 9

Σχεδίαση – Διαδικαστικός τρόπος σκέψης

Σχεδίαση

- καταχώρηση νέων αριθμών
  - `issueNew(…)` : ανάγνωση και αποθήκευση σε ένα πίνακα Α ενός νέου ΑΦΜ
  - `checkValidity(…)` : επαλήθευση αριθμών φορολογικού μητρώου που υπάρχουν στον πίνακα Α
    - έλεγχος αν είναι 5ψήφιοι.
- καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, …, 9
  - `countByLastDigit(…)` : καταγραφή του πλήθους των ΑΦΜ που καταλήγουν σε 0, 1, 2, …, 9 σε δεύτερο πίνακα Β
  - `printNumbers(…)` : εμφάνιση στην οθόνη του πίνακα Β

Υλοποίηση

```
#include <stdio.h>

void issueNew (int A[], int *size);

void checkValidity(int A[], int size);

void countByLastDigit(int A[], int size, int B[]);

void printNumbers(int B[]);
```


Υλοποίηση

```
int main(){
  int A[100];
  int currentSize = 0;
  int B[10];

  issueNew(A, &currentSize);
  issueNew(A, &currentSize);
  issueNew(A, &currentSize);

  checkValidity(A, currentSize);

  countByLastDigit(A, currentSize, B);
  printNumbers(B);

  return 0;
}
```

Υλοποίηση

```
void issueNew (int A[], int *size){
  int input;

  printf("Enter new number: ");
  scanf("%d", &input);
  A[*size] = input;
  (*size)++;
}
```


Υλοποίηση

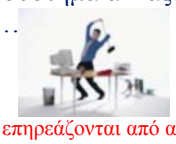
```
void checkValidity(int A[], int size){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 5) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\n");
}
```


Υλοποίηση

```
void countByLastDigit(int A[], int size, int B[]){
  int i;
  int lastDigit;

  for(i = 0; i < 10; i++)
    B[i] = 0;

  for(i = 0; i < size; i++){
    lastDigit = A[i] % 10;
    B[lastDigit]++;
  }
}

void printNumbers(int B[]){
  int i;
  printf("Count by last digit:\n");

  for (i = 0; i < 10; i++){
    printf("%d: %d\n", i, B[i]);
  }
}
```

Συντήρηση

- …το φορολογικό σύστημα αλλάζει και τα ΑΦΜ γίνονται 6ψήφια …



  - ποιες συναρτήσεις επηρεάζονται από αυτή την αλλαγή ?

Συντήρηση

```
void issueNew (int A[], int *size){
  int input;

  printf("Enter new number: ");
  scanf("%ld", &input);
  A[*size] = input;
  (*size)++;
}
```


Συντήρηση

Previous version of the code

```
void checkValidity(int A[], int size){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 5) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\n");
}
```

Previous version of the code

```
void checkValidity(int A[], int size){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 5) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\n");
}
```


Συντήρηση

New version of the code

```
void checkValidity(int A[], int size){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 6) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\nx");
}
```



```
void countByLastDigit(int A[], int size, int B[]){
  int i;
  int lastDigit;

  for(i = 0; i < 10; i++)
    B[i] = 0;

  for(i = 0; i < size; i++){
    lastDigit = A[i] % 10;
    B[lastDigit]++;
  }
}

void printNumbers(int B[]){
  int i;
  printf("Count by last digit:\n");

  for (i = 0; i < 10; i++){
    printf("%d: %d\n", i, B[i]);
  }
}
```

Σχεδίαση
Αντικειμενοστρεφής τρόπος σκέψης

Σχεδίαση

- Στον αντικειμενοστρεφή προγραμματισμό για κάθε αρμοδιότητα μιας εφαρμογής ορίζουμε μια νέα κλάση
  - ομαδοποιούμε δεδομένα και συναρτήσεις οι οποίες διαχειρίζονται αυτά τα δεδομένα
  - μια κλάση είναι σαν ένα struct στη C το οποίο αποτελείται από
    - ένα σύνολο πεδίων / χαρακτηριστικών
    - ένα σύνολο συναρτήσεων / μεθόδων

Υλοποίηση

```
#include <cstdio>

class ManageAFM{
private:
  int A[100];
  int size;
public:
  void init();
  void issueNew();
  void checkValidity();
  int getLastDigitsAndSize(int LastDigits[]);
  /* επιστρέφει το size και τα τελευταία ψηφία των A[i] */
};
```

Υλοποίηση

```
void ManageAFM::init(){
  size = 0;
}

void ManageAFM::issueNew (){
  int input;

  printf("Enter new number: ");
  scanf("%d", &input);
  A[size] = input;
  size++;
}
```


Υλοποίηση

```
void ManageAFM::checkValidity(){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 5) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\nx");
}
```


Υλοποίηση

```
int ManageAFM::getLastDigitsAndSize(int LastDigits[]){
  int i;
  for(i = 0; i < size; i++){
    LastDigits[i] = A[i] % 10;
  }
  return size;
}
```

Υλοποίηση

```
class AFMStatistics{
private:
  int B[10];
public:
  void init();
  void countByLastDigit(int LastDigits[], int size);
  void printNumbers();
};


void AFMStatistics::init(){
  int i;
  for(i = 0; i < 10; i++)
    B[i] = 0;
}
```


Υλοποίηση

```
void AFMStatistics::countByLastDigit(int LastDigits[], int size){
  int i;
  int lastDigit;

  for(i = 0; i < size; i++){
    lastDigit = LastDigits[i];
    B[lastDigit]++;
  }
}

void AFMStatistics::printNumbers(){
  int i;
  printf("Count by last digit:\n");

  for (i = 0; i < 10; i++){
    printf("%d: %d\n", i, B[i]);
  }
}
```



```
int main(){
  ManageAFM manager;
  AFMStatistics stats;

  int LastDigits[100];
  int currentSize;

  manager.init();
  manager.issueNew();
  manager.issueNew();
  manager.issueNew();

  manager.checkValidity();

  currentSize = manager.getLastDigitsAndSize(LastDigits);

  stats.init();
  stats.countByLastDigit(LastDigits, currentSize);
  stats.printNumbers();

  return 0;
}
```

Συντήρηση - τι κερδίσαμε ???

- ποιες μέθοδοι εξαρτώνται από τον πίνακα με τα ΑΦΜ (πίνακας Α)

Συντήρηση - τι κερδίσαμε ???

```
void ManageAFM::init(){
  size = 0;
}

void ManageAFM::issueNew (){
  int input;

  printf("Enter new number: ");
  scanf("%ld", &input);
  A[size] = input;
  size++;
}
```


Συντήρηση - τι κερδίσαμε ???

```
void ManageAFM::checkValidity(){
  int i;
  int length;
  int temp;

  for (i=0; i < size; i++){
    length = 1;
    temp = A[i];
    while(temp > 9){
      temp = temp / 10;
      length++;
    }
    if (length != 5) {
      printf("Error with number in position %d\n", i);
      return;
    }
  }
  printf("ALL OK!\n");
}
```

Συντήρηση - τι κερδίσαμε ???

```
int ManageAFM::getLastDigitsAndSize (int LastDigits[]){
  int i;
  for(i = 0; i < size; i++){
    LastDigits[i] = A[i] % 10;
  }
  return size;
}
```


Συντήρηση - τι κερδίσαμε ???

```
#include <cstdio>

class ManageAFM{
private:
  int A[100];
  int size;
public:
  void init();
  void issueNew();
  void checkValidity();
  int getLastDigitsAndSize (int LastDigits[]);
};
```


Συντήρηση - τι κερδίσαμε ???

- ποιες μέθοδοι εξαρτώνται από τον πίνακα με τα στατιστικά (πίνακας B) ??

Συντήρηση - τι κερδίσαμε ???

```
void AFMStatistics::init(){
  int i;
  for(i = 0; i < 10; i++)
    B[i] = 0;
}

void AFMStatistics::countByLastDigit (int LastDigits[], int size){
  int i;
  int lastDigit;

  for(i = 0; i < size; i++){
    lastDigit = LastDigits[i];
    B[lastDigit]++;
  }
}

void AFMStatistics::printNumbers(){
  int i;
  printf("Count by last digit:\n");

  for (i = 0; i < 10; i++){
    printf("%d: %d\n", i, B[i]);
  }
}
```


Συντήρηση - τι κερδίσαμε ???

```
class AFMStatistics{
private:
  int B[10];
public:
  void init();
  void countByLastDigit (int LastDigits[], int size);
  void printNumbers ();
};
```


Συντήρηση - τι κερδίσαμε ???

- οι μέθοδοι που εξαρτώνται από τα εκάστοτε δεδομένα μιας εφαρμογής είναι αυτές που τα χειρίζονται
- ποιες τα χειρίζονται ??
  - μόνο αυτές που ορίζονται στην ίδια κλάση με τα δεδομένα
  - στην κλάση της οποίας η αρμοδιότητα είναι η διαχείριση των δεδομένων

Συντήρηση - τι κερδίσαμε ???

- άρα για να ανακαλύψουμε ποιον κώδικα πρέπει να ελέγξουμε μετά από ένα σενάριο συντήρησης που αφορά τα δεδομένα μιας κλάσης αρκεί να εξετάσουμε ποιες μέθοδοι ορίζονται σε αυτή την κλάση

C++ fundamental building blocks

(χωρίς να αναφερθούμε, όμως, σε έννοιες του αντικειμενοστρεφούς προγραμματισμού)

(γεια σας και χαρά σας)

HELLO WORLD

Κύριο πρόγραμμα

```
//A Simple C++ program
#include <iostream>
using namespace std;

int main(){
  cout << "Hello world!" << endl;
  return 0;
}
```

Χρειαζόμαστε
(α) βιβλιοθήκες (π.χ., iostream for i/o)
(β) std namespace (to be explained later)

Χρειαζόμαστε τη main() ως την κεντρική συνάρτηση του προγράμματός μας

(εισαγωγικά)

ΕΙΣΟΔΟΣ / ΕΞΟΔΟΣ

Είσοδος / Έξοδος δεδομένων

- Στη C++ η βασική είσοδος και έξοδος γίνεται με τα ρεύματα εισόδου / εξόδου **cout** και **cin**, τα οποία είναι αντικείμενα τύπου istream και ostream αντίστοιχα
- Ο τελεστής << παίρνει μια (σύνθετη) έκφραση από το πρόγραμμά μας και την εισάγει στο cout, το οποίο με τη σειρά του την στέλνει στην οθόνη
- Ο τελεστής >> εξάγει δεδομένα από το cin (το οποίο τα παίρνει από το πληκτρολόγιο) και τα στέλνει στις μεταβλητές υποδοχής
- Η είσοδος / έξοδος της C υποστηρίζεται για λόγους συμβατότητας, επίσης

Απλή είσοδος / έξοδος: cout / cin

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

main(){
    cout << "Hello world!" << endl;
}
```

Ισοδύναμα:

```
cout << "Hello" << "world!\n";
cout << "Hello world!\n";
```

Είσοδος / Έξοδος δεδομένων

```
#include <iostream>
using namespace std;
int main(){

  int i;
  cin >> i;
  float f;
  cin >> f;
  char c;
  cin >> c;
  char buf[100];
  cin >> buf;

  cin >> i >> f >> buf;
}
```


Είσοδος / Έξοδος δεδομένων

```
#include <iostream>
using namespace std;
int main(){
  int i;
  cin >> i;
  cout << "i = ";
  cout << i;
  cout << "\n";

  float f;
  cin >> f;
  cout << "f = ";
  cout << f;
  cout << "\n";

  cin >> i >> f;
  cout << "i = " << i << "\n" << "f = " << f << "\n" ;
  cout << "i = " << i << endl << "f = " << f << endl;
}
```


(βασικοί τύποι, αρχικοποιήσεις, strings, const)

ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

Βασικοί τύποι

```
int xInt;                   //4 bytes (σπάνια 2)
short int xShortInt;        //2 bytes
long int xLong;             //4 bytes
float xFloat;               //4 bytes
double xDouble;             //8 bytes
char xChar;                 //1 byte

bool xBool;                 //1 byte
xBool = true; xBool = false;
```

10

Πίνακες

Integers

```
int dataArray[3];
```

C Strings

```
#include <cstring>
char firstName[6];
main(){
     strcpy (firstName, "eddie");
}
```

11

Δηλώσεις Μεταβλητών

- Στη C++ δηλώσεις μεταβλητών μπορούν να γίνουν παντού και όχι μόνο στην αρχή κάθε συνάρτησης (including main())

```
int main(){
  int x, y, z;
  if(…) {…}
  // ……..
  float f;
}
```

12

Δηλώσεις τοπικών μεταβλητών παντού

- Επιτρέπονται δηλώσεις μεταβλητών παντού μέσα στο πρόγραμμα.

```
void f ()
{
    int x = 0;
    cout << x;
    int y = x + 1;
    cout << y;
    for (int i=1; i < y; i++){
       cout << i;
    }
}
```

Η i ΔΕΝ ζει μετά το block του for

13

Αρχικοποίηση Μεταβλητών

```
int counter(0);          //equiv. to: int counter=0;
```

```
int myArray[4] = {1,2,3,4};   //equiv. to:
myArray[0] = 1;               //we start from 0
myArray[1] = 2; ...
//could also say
int myArray[] = {1,2,3,4};
```

```
//size: 6, length: 6 -remember the last '\0'
char firstName[] = "eddie";
//size: 50, length: 6
char firstName[50] = "eddie";
```

14

Πολυδιάστατοι Πίνακες

```
int myMatrix[2][4];

int myMatrix[2][4] = {
    {1,2,3,4},
    {5,6,7,8}
};
```

15

Const

Pay particular attention: frequently used for (input) parameter passing, as well as global variables!!

const

```
main(){
  int const SIZE = 20;

  //can be used to initialize an array
  char buffer[SIZE];

  //will NOT work – const variables do NOT change
  SIZE = 8;
}
```


Διαφορές const και #define

- Το `#define` αντικαθιστά την έκφραση με την ισοδύναμή της, μέσω του preprocessor (ΔΕΝ είναι μέρος της C++)
- Το `#define` είναι εξ' ορισμού global ορισμός, ενώ το `const` έχει δικό του scope.
- Η συντακτική ορθότητα μιας δήλωσης `const` ελέγχεται αμέσως, ενώ για το `#define` μόνο αφού γίνει η αντικατάσταση. Το όποιο λάθος εμφανίζεται στη γραμμή του προγράμματος και όχι στη δήλωση του `#define`.
- **Το `#define` είναι συχνή πηγή σύγχυσης και bugs, ιδίως όταν ο κώδικας σπάσει σε πολλά αρχεία => θα αποφεύγουμε τα `#define` και θα χρησιμοποιούμε const μεταβλητές!**

const

- Πρακτικός κανόνας: όταν δηλώνω μια μεταβλητή σαν const ΔΕΝ μπορώ να αλλάξω οτιδήποτε βρίσκεται **αριστερά** του keyword const.

- `char const *buf`
  Μπορώ να αλλάξω τον pointer, αλλά όχι τα δεδομένα του array
- `char *const buf`
  Δεν μπορώ να αλλάξω τον pointer, αλλά μπορώ να αλλάξω τα δεδομένα του array

19

Διαβάστε [και γράψτε] από δεξιά...

- Ο buf είναι const pointer (άρα δεν αλλάζει) σε χαρακτήρες

  ← `char *const buf`

- Ο buf είναι pointer σε const χαρακτήρες (οι οποίοι, συνεπώς δεν αλλάζουν)

  ← `char const *buf`

20

C++ strings

21

Strings

- Στη C++ υπάρχει έτοιμος τύπος δεδομένων που ονομάζεται string για τη διαχείριση συμβολοσειρών.
- Σε μια μεταβλητή τύπου string μπορούμε να αποθηκεύσουμε ακολουθίες χαρακτήρων χωρίς να ανησυχούμε για το αν χωράνε στην μεταβλητή αυτή.
- Η διαχείριση των bytes που δεσμεύονται από τη μεταβλητή γίνεται δυναμικά ανάλογα με το μέγεθος της ακολουθίας που αποθηκεύουμε σε αυτή.
- Επίσης δεν χρειάζεται να ανησυχούμε για το αν η ακολουθία τερματίζεται με '\0'.
- Χρειάζεται να συμπεριλάβουμε τη βιβλιοθήκη string στην include list του προγράμματος

Strings – Δήλωση και αρχικοποίηση

```
#include <iostream>
#include <string>
using namespace std;
int main(){
 string imBlank;
 string heyMom("where is my coat?");
 string heyPap = "whats up?";
 string heyGranPa(heyPap);
 string heyGranMa = heyMom;
}
```


Strings – Δήλωση και αρχικοποίηση

```
#include <iostream>
#include <string>
using namespace std;
int main(){
 string s1("I saw elvis in a UFO.");
 string s2(s1, 0, 8);

 string s3 = s1 + "Am I crazy?"; //append!!
 string s4 = s1.substr(2, s1.length()-10);
                  //take a substring: starting pos, how many chars
}
```

Strings – Επεξεργασία

```
#include <iostream>
#include <string>
using namespace std;
int main(){
 string s1("I saw elvis in a UFO.");
 cout << s1.size() << "\n";
 cout << s1.length() << "\n";   // Very important, very frequently used!
 cout << s1.capacity() << "\n";
 string s2 = " thought I ";
 s1.insert(1, s2);              // insert in position 1, i.e. right after 'I'
 cout << s1.capacity() << "\n";
 string s3 = "I've been working too hard";
 s1.append(s3);                 //append, again, equiv. to +
 s1.append(", or am I crazy?);
 s1.resize(10);                 // increase/reduce the capacity
 cout << s1 << "\n";
}
```


Strings – Επεξεργασία

```
#include <iostream>
#include <string>
using namespace std;
int main(){
 string s("Hello mother");
 string s1("fa");
 s.replace(6, 2, s1);
        // start at 7th character (counting starts with 0), replace 2 chars with the whole of s1
        // Syntax: starting  pos, how many chars to replace, replacement
 cout << s << "\n";
}
```


Strings – Επεξεργασία

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  string s("Hello mother. How are you mother? Im fine mother.");
  string s1("fa");
  string s3("mo");
  int start = 0;
  unsigned int found = 1;

  found = s.find(s3, start);
  while (found != string::npos){
    s.replace(found, s3.length(), s1);
    start = found + s1.length();
    cout << s << "\n";
    found = s.find(s3, start);
  }
  cout << s << "\n";
}
```

Strings – Επεξεργασία

```
s.rfind(s1, s.length()-1);
```
// finds s1 in s backwards starting from the last character

```
s.find_first_of("@$.", pos);
```
// find position of first occurrence of a char in "…" starting from pos

```
s.find_last_of("@$.", pos);
```
// find position of last occurrence of a char in "…"

Strings – Επεξεργασία με τελεστές

```
s= s1 + s2 + s3;
s += s5 + s6;
s[10] = 'c';
s.at(10) = 'c';
s1 == s2
s1 != s2
s1 >= s2
s1 <= s2
s1 > s2
s1 < s2
```


Strings and backwards compatibility with C

- Για λόγους συμβατότητας προς τα πίσω, υποστηρίζονται και οι συμβολοσειρές της C
- Στη C τα strings είναι ακολουθίες χαρακτήρων που τερματίζονται με το χαρακτήρα '\0'.
- Αποθηκεύονται σε πίνακες χαρακτήρων.
- Η επεξεργασία τους διευκολύνεται από έτοιμες συναρτήσεις όπως strcpy, strcmp,…

- How to convert a string s to a (C-style) char array:

```
const char *p = s.c_str();
```

(συνθήκες, βρόγχοι και τα σχετικά)

ΒΑΣΙΚΕς ΕΝΤΟΛΕς

If … Else … Statement

```
if (condition) {
   statement;
   ...
}
else {
 statement;
   ...
}
```

Λογικοί τελεστές για σύνθετες συνθήκες:

OR: `((cond1)||(cond2))`

AND: `((cond1)&&(cond2))`

NOT: `!(cond1)`

Προσοχή: `= vs. ==`

Switch Statement

```
switch (condition) {
  case constant1:
      statement;
      ...
      break;
  case constant2:
      statement;
      ...
      break;
  ...
  default:
      statement;
      ...
      break;
}
```

Προσοχή: αν παραληφθεί το **break**, εκτελείται η επόμενη εντολή!

Δεν το παραλείπουμε ποτέ!!!

Το `default` καλύπτει την περίπτωση που χάνουμε κάποια case

Loop Statements

```
while (condition){
   statement;
   ...
}

for (declaration-initialization; condition; iteration){
   statement;
   ...
}

//Example of counting...
for (int i=0; i<5; i++){        //runs 5 times
   cout << i*5+3;
}
```

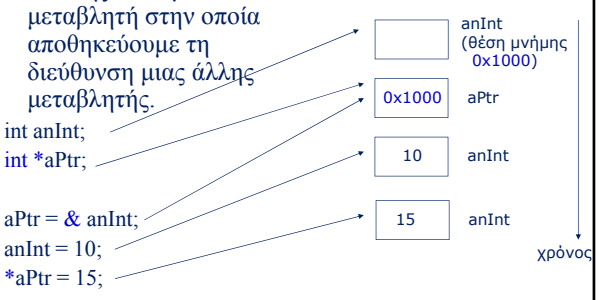

Συναρτήσεις (εισαγωγικά)

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

float triangleArea (float width, float height);
main(){
       float area = triangleArea(1.0, 2.0);
       cout << "The area of the triangle is "  <<
  area;
}
float triangleArea(float width, float height){
        float area; // local στην triangleArea
        area = width * height / 2.0;
        return (area);
}
```


ΔΕΙΚΤΕΣ ΚΑΙ ΑΝΑΦΟΡΕΣ

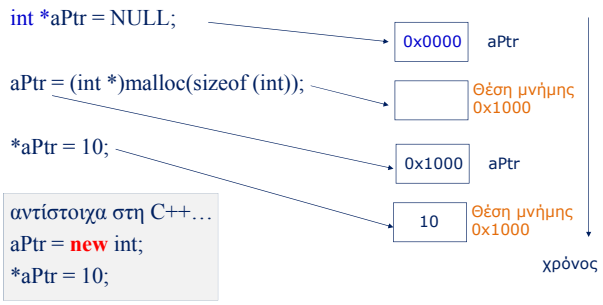
Δείκτες

- Δείκτης είναι μια μεταβλητή στην οποία αποθηκεύουμε τη διεύθυνση μιας άλλης μεταβλητής.



37

Δείκτες – Δέσμευση Μνήμης

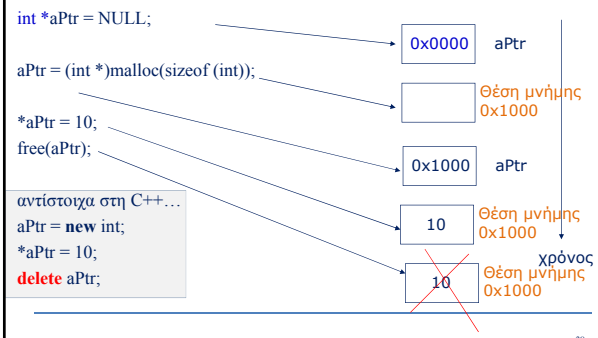


αντίστοιχα στη C++…

```
aPtr = new int;
*aPtr = 10;
```

38

Δείκτες – Αποδέσμευση Μνήμης



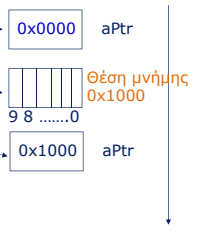
αντίστοιχα στη C++…

```
aPtr = new int;
*aPtr = 10;
delete aPtr;
```

39

Δείκτες – Πίνακες - Δέσμευση Μνήμης

```
int *aPtr = NULL;

aPtr = (int *)malloc(10*sizeof (int));

*aPtr = 10; ⇔ aPtr[0] = 10;
*(aPtr + 2) = 6; ⇔ aPtr[2] = 6;
free(aPtr);
```



αντίστοιχα στη C++…

```
aPtr = new int [10];
*aPtr = 10;
delete [] aPtr;
```


Αναφορές (References)

- Μια ***αναφορά*** σε μια μεταβλητή είναι σαν ένα συνώνυμο για τη μεταβλητή.
- **Προσοχή**: ΔΕΝ πρόκειται για pointers (αν και υπάρχουν κάποιες ομοιότητες, βέβαια – βλ. παρακάτω...)

```
...
int myInt;
int &myRef = myInt;
...
```

Έτσι δηλώνουμε αναφορές:

κατά τη δήλωση πρέπει να καθορίζουμε τη μεταβλητή της οποίας είναι συνώνυμα

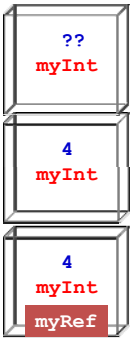
Αναφορές (References)

- Όταν δηλώνουμε μια μεταβλητή, αυτό σημαίνει ότι της παραχωρούμε κάποιο χώρο στη μνήμη και ένα όνομα στο χώρο αυτό
- Όταν δηλώνουμε μια αναφορά σε μια μεταβλητή, είναι σαν να δίνουμε παραπάνω από ένα ονόματα στο συγκεκριμένο χώρο στη μνήμη.
- Οι παρακάτω εντολές είναι ισοδύναμες:

```
myInt++;
myRef++;
```

Αναφορές

```
int myInt;
```




```
myInt = 4;
```



```
int &myRef =
myInt;
```


43

Τι κρύβουν ….

```
int myInt = 5;
int &myRef = myInt;
```

→ `int * const myRef_p = &myInt;`

```
myInt = 8;
```

→ `myInt = 8;`

```
myRef = 7;
```

→ `*myRef_p = 7;`

44

Τι κρύβουν ….

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

int main() {
        int a(1), b(3);
        int& refA = a;

        cout << refA << endl;
        cout << a << endl;

        refA = b;
        cout << a << endl;
        return 0;
}
```

`int * const refA_p = &a;`

→ `cout << *refA_p << endl;`

→ `*refA_p = b;`

45

Πού χρησιμεύουν ….

- Οι δηλώσεις αναφορών που παίζουν το ρόλο συνώνυμων μιας μεταβλητής σε ένα πρόγραμμα δεν έχουν πολύ μεγάλη χρησιμότητα . Το ίδιο ισχύει για τις δηλώσεις δεικτών που τοποθετούνται σε κάποια υπάρχουσα μεταβλητή

```
int x = 10;
int *x_p = &x;
```

- Εκεί που βοηθούν δραστικά τόσο οι δείκτες όσο και οι αναφορές είναι σε συναρτήσεις στο πέρασμα παραμέτρων δι' αναφοράς.
- **Στην περίπτωση που χρησιμοποιήσουμε αναφορές σαν ορίσματα σε συναρτήσεις το πέρασμα δι' αναφοράς γίνεται ακόμα πιο απλό.**

46

ΠΈΡΑΣΜΑ ΠΑΡΑΜΈΤΡΩΝ ΣΕ ΣΥΝΑΡΤΉΣΕΙς

47

Πέρασμα δια τιμής

```
#include <iostream>
Using namespace std;
void increase(int myParameter)
{
    myParameter++;
    cout << myParameter << "\n";
}
int main(){
    int aValue = 3;

    increase(aValue);
    cout << aValue << "\n";
}
```

Δημιουργείται τοπική κόπια του aValue

48

Πέρασμα δι'αναφοράς με δείκτη

```
#include <iostream>
using namespace std;
void increase(int *myPointer)
{
    (*myPointer)++;
    cout << *myPointer << "\n";
}
int main(){
    int aValue = 3;

    increase(&aValue);
    cout << aValue << "\n";
}
```

49

Πέρασμα δι'αναφοράς με αναφορά

```
#include <iostream>
using namespace std;
void increase(int &myRef)
{
    myRef++;
    cout << myRef << "\n";
}
int main(){
    int aValue = 3;

    increase(aValue);
    cout << aValue << "\n";
}
```

50

Πέρασμα δι' αναφοράς με αναφορά – τι κρύβει…

```
#include <iostream>
using namespace std;
void increase(int &myRef)
{
    myRef++;
    cout << myRef << "\n";
}
int main(){
    int aValue = 3;

    increase(aValue);
    cout << aValue << "\n";
}
```

Increase(int *const myRef_p)

(*myRef_p)++;

cout << *myRef_p << "\n";

localIncrease(&aValue)

51

Παράδειγμα – μεταβλητές που δεν αλλάζουν

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

void showConst(int &counter, const int &newCounter){
       counter = counter + newCounter;
    /* will not pass!!!  newCounter++; */
}
int main(){
   int aCounter = 0;
   int anotherCounter = 3;
   showConst(aCounter, anotherCounter);
   cout << aCounter << "\n";
}
```


Arrays as parameters

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

void printArray(int arg[], int length){
  int i;
  for (i=0;i<length;i++){
      cout << arg[i] << "\n";
  }
}
main(){
  int firstArray[] = {5,10,15};
  int secondArray[] = {3,6,9,12};
  printArray(firstArray,3);
  printArray(secondArray,4);
}
```


Διαφορές pointers και references

- Μια αναφορά δεν στέκει ποτέ μόνη της σε ένα πρόγραμμα. Πρέπει να δηλωθεί οπωσδήποτε σαν αναφορά σε κάποια μεταβλητή (απόρροια του γεγονότος ότι οι const μεταβλητές θέλουν αρχικοποίηση).
- Ένας pointer μπορεί κάλλιστα να μη δείχνει πουθενά.
- Αν μια αναφορά δείχνει σε μια μεταβλητή, ΔΕΝ μπορώ να την βάλω να δείξει σε άλλη μεταβλητή (απόρροια του γεγονότος ότι οι const μεταβλητές δεν αλλάζουν).
- Έναν pointer μπορώ να τον μετακινώ κατά βούληση.

Ομοιότητες pointers και references

- Μπορούν να περνούν αμφότερα σαν παράμετροι σε μια συνάρτηση.
- Ομοίως, μπορούν να αποτελούν την τιμή επιστροφής μιας συνάρτησης.

Ισοδύναμο αποτέλεσμα

```
void byPointer(int *value){
   *value +=5;
}

main(){
   int i = 3;
   byPointer(&i);
}
```

```
void byRef(int &value){
   value +=5;
}

main(){
   int i = 3;
   byPointer(i);
}
```


Συνοπτικά για το πέρασμα παραμέτρων

`function (int var)` Πέρασμα τιμής	Η μεταβλητή περνιέται αυτούσια ως παράμετρος στη function, μπορεί να αλλάξει τοπικά στη function, αλλά οι αλλαγές δεν περνούν εξωτερικά στο πρόγραμμα.
`function (const int var)` Πέρασμα τιμής	Όπως πριν, αλλά χωρίς να μπορεί να αλλάξει η τιμή της μεταβλητής εσωτερικά της function
`function (int &var)` Αναφορά	Περνιέται μια αναφορά ως παράμετρος. Ότι αλλαγές γίνουν στην αναφορά, ανακλώνται και εξωτερικά στο πρόγραμμα
`function (const int &var)` **Σταθερή Αναφορά**	Όπως πριν, αλλά χωρίς να μπορεί να αλλάξει η τιμή της αναφοράς εσωτερικά της function
`function (int array [])`	Η C++ αυτομάτως μετατρέπει τα arrays σε αναφορές
`function (int *var)`	Περνά ένας pointer ως παράμετρος της function

Συνοπτικά για το πέρασμα παραμέτρων

Excellent guideline at Google Code Style:
- **use const references for input variables that are not to be changed**
- **use pointers for output variables that will be changed, eventually**

Significantly enhances **maintainability** and usage: the reader knows what is input + what is output
If the maintainer accidentally tries to mess with the (const) input, compiler catches it

```
void updateLicense(const int &newNumber, int *licenceNo)
```


Default τιμές στο πέρασμα παραμέτρων

```
void f (int x = 1);
void g (int a, int b = 0){
   cout << "a: " << a << " b: " << b << "\n";
}

int main ()
{
    f(5);
    f();
    g(1, 2);
    g(5);
}
void f (int x){ … //could be (int x =1 )
}
```


Default τιμές στο πέρασμα παραμέτρων

```
void h(int a, int b = 0, int c, int d = 0){
   // ……..
}

int main (){
   h(10, 5, 20);
}
```

Avoid default values if you can: source of much confusion!!

- στην περίπτωση αυτή σίγουρα a = 10
  - το 5 μπορεί να αντιστοιχεί στο b οπότε c = 20 και d = 0
  - Η' το b = 0, c = 5 και d = 20
- προς αποφυγή του παραπάνω προβλήματος το οποίο δεν μπορεί να επιλύσει ο compiler **αν η i-οστή παράμετρος στη δήλωση μιας συνάρτησης με N παραμέτρους έχει default τιμή, όλες όσες την ακολουθούν πρέπει επίσης να έχουν δηλωμένες default τιμές**.

με βάση το παραπάνω αν κατά την κλήση η συνάρτηση έχει ι <= κ < N πραγματικές παραμέτρους, οι N-κ τελευταίες λαμβάνουν τις default τιμές

Κονσόλα -- Αρχεία

C++ I/O

C++ IO Streams

IOSTREAMs - reading lines

```
#include <iostream>
using namespace std;
int main(){
   char buf[100]; char whatIsLeft[200];

/* reads max of 9 chars or up to '\n' and leaves the '\n' behind, i.e. the next char we read is \n */
   cout << "Give a string, at most 9 chars, ending with endl \n";
   cin.get(buf, 10, '\n');
   cout << buf << endl;

   cout << "Here, normally, you should give another string \n";
   cin.get(whatIsLeft, 10, '\n');
   cout << whatIsLeft << endl;

}
//Try declaring buf as string (including #include <string>) to see what happens
```

IOSTREAMs - reading lines

```
#include <iostream>
using namespace std;
int main(){
   char buf[100]; char whatIsLeft[200];

/* reads max of 9 chars or up to '\n' including the '\n' which is not included in buf */
   cout << "Give a string of at most 9 characters, ending with
   endl \n";
   cin.getline(buf, 10, '\n');
   cout << buf << endl;

   cout << "Here, normally, you should give another string \n";
   cin.getline(whatIsLeft, 10, '\n');
   cout << whatIsLeft << endl;

}

// Try it with input that exceeds the limit of 10 chars and with input less than 10 chars
```


IOSTREAMs - reading lines

```
#include <iostream>
using namespace std;
int main(){
   char buf[100];

   while (cin.eof() != true){ // or cin.good() == true
       cout << "Give another string: ";
       cin.getline(buf, 100, '\n');
       cout << buf << endl;
   }
}

//Eof is given via Ctrl+D
```


IOSTREAMs

```
#include <iostream>
using namespace std;
int main(){
   float b; int a;

   while (cin.good() == true){ // or cin.eof() != true
       cout << "Give a float and an int: ";
       cin >> b >> a;
       cout << b <<"\t" << a << endl;
   }
}
//Try it separating the 2 numbers with space, enter, or tab
```

IOSTREAMs - reading lines σε string

```
#include <iostream>
#include <string>
using namespace std;
int main(){
  string s;

  while (getline(cin, s)){
    cout << s << endl;
  }
}
```

Very useful!

Can handle strings of (i) arbitrary size and (ii) several whitespaces

Contrast this to `cin >> s` that can withstand only one (1) word in the input stream

//Try it with strings having several whitespaces and big size

C++ File Streams

File streams

- Για ανάγνωση από αρχείο δημιουργούμε μεταβλητές (ρεύματα) τύπου ifstream
- Για γράψιμο σε αρχείο δημιουργούμε μεταβλητές (ρεύματα) τύπου ofstream
- Ανάγνωση γίνεται με >> για βασικούς τύπους (int, float, char, char[], string…) ΚΑΙ get, getline για char[]
- εγγραφή γίνεται με << για βασικούς τύπους (int, float, char, char[], string,…)
- Γενικά ότι συνάρτηση είδαμε στα cin, cout ισχύει και σε μεταβλητές ifstream, ofstream.

Copy one file to another, line by line

```
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;
int main(){
  char buf[100];

  ifstream in("in.txt");
  ofstream out("out.txt");
  if(in.good()!= true) exit(EXIT_FAILURE);

  while (in.eof() != true){
    in.getline(buf, 100, '\n');
    out << buf << endl;
  }
  in.close();
  out.close();
}
```

Same old story:

Μια μεταβλητή για κάθε αρχείο

ATTN: αν κάτι πάει στραβά, πιάστε το νωρίς!!

Διαβάζουμε γραμμή – γραμμή μέχρι τέλους του αρχείου && … we do sth with it … (here: copy it to out.txt)

housekeeping

70

Append a file to another file, line by line

```
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;
int main(){
  char buf[100];
  char c;

  ifstream in("in.txt");
  ofstream out("out.txt", ios::app);   // ios::out is the default
  if(in.good()!= true) exit(1);
  while (in.eof() != true){
    in.getline(buf, 100, '\n');
    out << buf << endl;
  }
  in.close();
  out.close();
}
```

71

Append a file to another file, line by line

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
int main(){
  char buf[100];

  ifstream in;
  ofstream out;
  in.open("in.txt");
  out.open("out.txt", ios::app); // ios::out default
  if(in.good()!= true) exit(1);
  while (in.eof() != true){
    in.getline(buf, 100, '\n');
    out << buf << endl;
  }
  in.close();
  out.close();
}
```

Η διαφορά με το προηγούμενο παράδειγμα είναι στη χρήση της open

72

Extra methods

```
  in.clear();                  // sets eof flag back to 0
  in.seekg(0,ios::beg);        // offset from the beginning
  while (in.eof() != true){
    in.getline(buf, 100, '\n');
    cout << buf << endl;
  }

  out.seekp(0,ios::beg);       // offset from the beginning
  out << "BBBBBBBBBBB";
  in.close();
  out.close();
}

// seekp seekg χρήση και με ios::end και αρνητικό offset, ή ios::curr
```


Try it for yourselves (part 1)

```
#include <iostream>
#include <cstdlib>
#include <fstream>
using namespace std;
int main(){
  char buf[100];
  char c; int i; float f;

  fstream in;
  fstream out; // ios::out is default
  in.open("in.txt", ios::in);
  out.open("out.txt", ios::out|ios::in); // ios::app to append
  if(out.good()!= true) exit(1);
  while (in.eof() != true){
    in.getline(buf, 100, '\n');
    out << buf << endl;
  }
```

…continued: part 2

```
  in.clear();
  in.seekg(0,ios::beg);
  while (in.eof() != true){
    in.getline(buf, 100, '\n');    //print file on screen
    cout << buf << endl;
  }

  out.seekg(0,ios::beg);
  while (out.eof() != true){
    out.getline(buf, 100, '\n');
    cout << "XXX" << buf << endl; //print on screen file's info with XXX in front…
  }
  out.clear();
  out.seekp(0,ios::beg);                //replace file's starting bytes with BBBB…
  out << "BBBBBBBBBBB";
  in.close();
  out.close();
}
```

(structs, υπερφόρτωση συναρτήσεων, namespaces, …)

POT POURRI

Disclaimer

- Δεν γίνεται μονομιάς να μάθετε ολόκληρη τη C++
  - Υπάρχουν θέματα που θα τα ανακαλύψετε σιγά σιγά, έξω από το πλαίσιο του μαθήματος
  - Υπάρχουν και στοιχεία που θα πρέπει να αντιμετωπίσετε με πολύ προσοχή – ενδεχομένως και να αποφασίσετε να μην τα χρησιμοποιήσετε καθόλου…
- Στη συνέχεια δίνουμε μια πρώτη εικόνα από στοιχεία της C++ στα οποία δε θα επεκταθούμε ιδιαίτερα…
- Θυμηθείτε ότι ο σκοπός του μαθήματος είναι να διδάξουμε βασικές αρχές αντικειμενοστρεφούς προγραμματισμού και όχι C++

Structs & Κλάσεις

Structs

- Ο πιο απλός τρόπος δήλωσης ενός struct είναι ως εξής:

```
struct structName {
  fieldType fieldName;
  fieldType fieldName;
  ...
};
```


Τα structs είναι τύποι δεδομένων

- Δήλωση struct

```
struct complex {  //μιγαδικός
    double re;
     double im;
};
```

- Δήλωση μεταβλητής τύπου complex

```
complex z;
```

  - Σε **αντίθεση με τη C** δεν χρειάζεται να γράψουμε

```
struct complex z;
```


Συναρτήσεις μέσα σε structs

- Σε αντίθεση με τη C μπορούμε να ορίσουμε μεθόδους για τα structs

```
struct complex {      //μιγαδικός
  double re;
  double im;
  void show();
};

void complex::show(){
  cout << "(" << re << "," << im << \ ")\n";
}
```

Συναρτήσεις μέσα σε structs

- Εντελώς αντίστοιχα μπορώ να ορίσω την κλάση

```
class complex {       //μιγαδικός
public:
  double re;
  double im;
  void show();
};
void complex::show(){
  cout << "(" << re << "," << im << \ ")\n";
}
```


Δηλώσεις και χρήση αντικειμένων

```
int main ()
{
  complex a;

  a.re = 1.0;
  a.im = 2.0;
  a.show();

  return 0;
}
```

Εναλλακτικά (α-λα-C):

```
#include <iostream.h>
...
void show (complex &p){
      ...
}
main(){
      complex a;
      ...
      show(a);
}
```


Δείκτες σε αντικείμενα

```
int main ()
{
  complex *pa, *pb;
  complex a;

  pa = &a;
  pb = new complex;
  a.re = 1.0; pa ->re = 1.0; (*pa).re = 1.0;
  a.im = 2.0; pa->im = 2.0;  (*pa).im = 2.0;
  a.show();   pa->show();    (*pa).show();

  pb->re = 3.0; pb->im = 4.5;
  pb->show();

  delete pb;
  return 0;
}
```

Υπερφόρτωση (Overloading) Συναρτήσεων

Υπερφόρτωση συναρτήσεων

- Στη C++ επιτρέπεται να δηλώσουμε συναρτήσεις με το ίδιο όνομα αλλά:
  - με διαφορετικό αριθμό παραμέτρων
  - με διαφορετικούς τύπους παραμέτρων
  - **ΠΡΟΣΟΧΗ!** Όχι συναρτήσεις που διαφέρουν μόνο στον τύπο του αποτελέσματος που επιστρέφουν.

Υπερφόρτωση συναρτήσεων

```
#include <iostream>

using namespace std;

int max (int a, int b){
  if (a > b) return a;
  else return b;
}

int max (int a[], int size){
  int max = a[0];

  for (int i = 0; i < size; i++)
    if (a[i] > max) max = a[i];

  return max;
}
```

Υπερφόρτωση συναρτήσεων

```
int main(){
  int A[] = {-2, 4, 5, 6};
  int a = 3, b = 5;

  int ret;

  ret = max(A, 4);
  cout << ret << endl;

  ret = max(a, b);
  cout << ret << endl;
}
```


Υπερφόρτωση συναρτήσεων

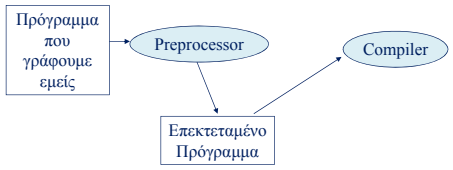
```
#include <iostream>

using namespace std;

int max (int a, int b){
  if (a > b) return a;
  else return b;
}

float max (int a, int b){
   if (a > b) return a;
   else return (float) b;
}
```


Υπερφόρτωση συναρτήσεων

```
int main(){
  int a = 3, b = 5;

  int ret;

  ret = max(a, b);
  cout << ret << endl;

  max(a, b); // Πρόβλημα ποια θα καλεστεί ???
             // Compile Error
}
```

C++ Preprocessor

Preprocessor της C++ (#include, #define, …)



#include

- Ενσωματώνει στο πρόγραμμά μας αυτούσια αρχεία.
- `#include <iostream.h>`
- `#include "mydefinitions.h"`
- `#include "../../mydefinitions.h"`
  DOS/Windows: "`..\..\mydefinitions.h`"

#define

- `#define SIZE 20`
- Σημαίνει ότι οπουδήποτε βλέπει ο preprocessor `SIZE`, το αντικαθιστά με `20`.

```
//illegal definitions
#define X = 5
#define X 5;
```

- Σαφώς καλύτερα να χρησιμοποιεί κανείς `const`

```
const int SIZE = 20;
```


Inline Functions

Inline functions

Είναι συναρτήσεις οι οποίες χρησιμοποιούνται από τον compiler στην παραγωγή του εκτελέσιμου κώδικα: για επιτάχυνση της διαδικασίας, ενσωματώνονται στον κώδικα της καλούσας συνάρτησης, αντί να κληθούν ως χωριστές συναρτήσεις.

Χρήσιμες ΜΟΝΟ για πολύ μικρές συναρτήσεις.

```
inline int square (int value){
      return (value * value);
}
```

Χρησιμοποιούνται κανονικά στο πρόγραμμα, π.χ.,

```
main(){
      …
      mySquareArea = square(squareEdge);
      …
}
```

Namespaces

Namespaces

- Πολλές φορές στη C φτιάχνουμε συναρτήσεις των οποίων τα ονόματα έρχονται σε σύγκρουση με έτοιμες συναρτήσεις της γλώσσας
  - σαν αποτέλεσμα έχουμε λάθη μετάφρασης που δεν εξηγούνται εύκολα…
  - και κόστος επιδιόρθωσης του κώδικα
- Ένα πρόγραμμα μπορεί να συνθέτει κώδικα από δύο ή περισσότερα άτομα που μπορεί να χρησιμοποιούν συνώνυμες συναρτήσεις ή κλάσεις.
  - σαν αποτέλεσμα πάλι μπορεί να έχουμε λάθη μετάφρασης του συνολικού κώδικα
- στη C++ υπάρχει ο μηχανισμός namespace που μπορεί να χρησιμοποιηθεί για την αποφυγή συγκρούσεων

```
namespace X {
……..
};
```


Παράδειγμα

- **Στο αρχείο: `util.h` υπάρχουν χρήσιμες συναρτήσεις που πήραμε από κάποιον άλλο προγραμματιστή**

```
……..
float squareArea(float length){
        return (length*length);
}
……..
int f(){….}
```

Παράδειγμα (συνέχεια)

```
#include <iostream.h>
#include "util.h"

//1 square mile is 2.59 square km
//the following function returns square miles
//whereas length is in km

float squareArea(float length){
        return (length*length / 2.59);
}

main (){
  cout << "Normal " << squareArea(2.0);
   f();
}
```

100

Παράδειγμα

- Στο αρχείο: `util.h`

```
namespace UtilJohn {

    float squareArea(float length){
        return length*length;
     }
     int f(){….}

}
```

101

Παράδειγμα (συνέχεια)

```
#include <iostream.h>
#include "util.h"

namespace Local{
                //1 square mile is 2.59 square km
                //the following function returns square miles
                //whereas length is in km
    float squareArea(float length){
        return (length*length / 2.59);
    }
}
main (){
   cout << "Normal " << Local::squareArea(2.0);
   UtilJohn::f();
}
```

102

Παράδειγμα (συνέχεια)

```
#include <iostream.h>
#include "util.h"
namespace Local{      //returns square km for length in km
                      //1 square mile is 2.59 square km
                      //the following function returns square miles
                      //whereas length is in km
    float squareArea(float length){
        return (length*length / 2.59);
    }
}
using namespace Local;
main (){
   cout << "Normal " << squareArea(2.0);
   UtilJohn::f();
}
```


Παράδειγμα (συνέχεια)

```
#include <iostream.h>
#include "util.h"
namespace Local{      //returns square km for length in km
                      //1 square mile is 2.59 square km
                      //the following function returns square miles
                      //whereas length is in km
    float squareArea(float length){
        return (length*length / 2.59);
    }
}
using namespace Local;
main (){
   cout <<"English "<< UtilJohn::squareArea(2.0)
                << "\n";
   cout << "Normal " << squareArea(2.0);
}
```


Το `std` namespace

- Μπορείτε να δηλώνετε το **`std`** namespace:

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

- για να μη χρειάζεται να το γράφετε συνέχεια

πχ `std::cout <<` ….

Αν και κάποιοι μεταφραστές το κάνουν αυτόματα…

Exceptions

Εξαιρέσεις

- Αρκετές φορές στην πράξη υλοποιούμε συναρτήσεις που δέχονται σαν όρισμα κάποια δεδομένα και επιστρέφουν ένα αποτέλεσμα κάποιου τύπου (π.χ. int).
- Το αποτέλεσμα σε πολλές περιπτώσεις μπορεί να είναι οποιαδήποτε τιμή αυτού του τύπου (int).
- Παρόλα αυτά αν κάτι πάει στραβά στην εκτέλεση της συνάρτησης θέλουμε να επιστρέφει κάτι σε αυτόν που την κάλεσε ώστε να καταλάβει ότι κάτι πήγε στραβά (π.χ. απέτυχε το άνοιγμα ενός αρχείου).
- Για αυτό το κάτι, προφανώς δεν μπορούμε να δεσμεύσουμε μια τιμή αν αυτή μπορεί να επιστρέφεται και κάτω από κανονικές συνθήκες (π.χ. -1).
- Αντί αυτού μπορούμε να χρησιμοποιήσουν το μηχανισμό των εξαιρέσεων…..

Εξαιρέσεις

- Γενικά μπορούμε να πούμε ότι μια εξαίρεση είναι μια δομή (struct) που επιστρέφει μια συνάρτηση όταν κάτι πάει λάθος.
- Η επιστροφή γίνεται με throw αντί για return.
- Η συλλογή αυτής της δομής - εάν και εφόσον επιστραφεί – γίνεται με τη χρήση της εντολής try/catch.

Εξαιρέσεις

```
#include <iostream>
#include <string>
using namespace std;
struct MyError {
  int code;
  string cause;
};
// -1 is a valid calculation of the function
float test_function(string s) {
  MyError e;
  ifstream in;

  in.open(s);
  if (in.good() != true) {
        e.code = 555;
        e.cause = "open file failed";
        throw e;
  }
  else {
   float sum;
    // ..... read float numbers from file and return the sum
   return sum;
  }
}
```


Εξαιρέσεις

```
.....

main(){
 int y; float k;
 try {

  k = test_function ("lala.txt");     // test_function is invoked
                                      // k = ... if ok
 } catch (MyError error) {            // error = ... if a problem occurs
   cout << error.code << error.cause << endl;
 }
```


Εξαιρέσεις - new

```
.....

main(){
 int * x;

 try {
  x = new int [1000];
 } catch (bad_alloc bad){
   cout << "Bad allocation integer array";
 }
}
```

Υπερφόρτωση Τελεστών

Υπερφόρτωση τελεστών

- Στη C οι τελεστές (+, -, *, ==, >, <, ….) ορίζονται και μπορεί να χρησιμοποιηθούν μόνο μεταξύ μεταβλητών ή σταθερών κάποιου βασικού τύπου (int, float, double, char,……).
- Στη C++ μπορούμε να επανα-ορίσουμε τη λειτουργία των τελεστών για περιπτώσεις μεταβλητών των οποίων ο τύπος ορίστηκε από τον προγραμματιστή.
  - μεταβλητές τύπου struct
  - αντικείμενα κάποιας κλάσης
    - περισσότερα θα πούμε στην περίπτωση αντικειμένων…..

Υπερφόρτωση τελεστών

```
struct complex {
   double re, im;
};

complex operator + (complex x, complex y)
{
   complex result;
   result.re = x.re + y.re;
   result.im = x.im + y.im;
   return result;
}

int operator == (complex x, complex y)
{
   return (x.re == y.re)
        && (x.im == y.im);
}
```

Υπερφόρτωση τελεστών

```
int main ()
{
   complex a, b, c;

   a.re = 1.0; a.im = 2.0;
   b.re = 4.0; b.im = 1.0;

   if (a == b)
      cout << "they are equal\n";
   else
      cout << "they are different\n";
   c = a + b;

   return 0;
}
```

Κλάσεις, Αντικείμενα, Ενθυλάκωση, Δημιουργία & Καταστροφή Αντικειμένων

Θεματολόγιο

- Classes, objects and encapsulation
- Constructors
- Destructors
- Static πεδία και μέθοδοι

Classes, objects and encapsulation

Κλάσεις και αντικείμενα

- Όπως είδαμε, το βασικό στοιχείο σχεδίασης και δόμησης του αντικειμενοστρεφούς κώδικα είναι η κλάση (class)
- Μία κλάση είναι ένα καλούπι από το οποίο παράγονται αντικείμενα (objects), τα οποία:
  - Αναπαριστούν ομοειδείς οντότητες του πραγματικού κόσμου
  - Έχουν ίδια δομή

Κλάσεις και αντικείμενα

- Είδαμε ότι κάθε κλάση είναι μια ενότητα λογισμικού, η οποία σχεδιάζεται με σκοπό
  - Να διευκολύνει την συνεργασία των προγραμματιστών
  - Να διευκολύνει την επαναχρησιμοποίηση έτοιμου κώδικα από άλλους
  - Να συμπτύξει στο ίδιο σημείο ενός project (στο ίδιο αρχείο κατά βάση) κώδικα και δεδομένα
- Για το σκοπό αυτό:
  - Αντί για να διαχωρίζουμε δεδομένα και κώδικα, ο σχεδιαστής μιας κλάσης εξάγει προς τους υπόλοιπους προγραμματιστές μία δημόσια διαπροσωπεία (public interface), η οποία λέει στους υπόλοιπους προγραμματιστές, τι λειτουργίες μπορεί να προσφέρουν τα αντικείμενα της κλάσης αυτής
  - Έτσι, εκτός από τα δεδομένα που κουβαλά ένα αντικείμενο (τα οποία αξιοποιούν τα πεδία της κλάσης) εκτελεί και συναρτήσεις (οι οποίες εκτελούν τις μεθόδους της κλάσης)

Προσοχή: αντικειμενοστρεφής σχεδίαση

- Η κλάση είναι το καλούπι, αλλά τα αντικείμενα είναι αυτά που εκτελούν τις λειτουργίες
- Όταν σχεδιάζουμε τον κώδικα, σκεπτόμαστε πώς κάποια αντικείμενα που θα δημιουργηθούν κατά την εκτέλεση του προγράμματος θα επιτελούν λειτουργίες!
  - Σχεδιάζουμε, λοιπόν, έχοντας υπόψη λειτουργίες που θα επιτελούνται και όχι δομές δεδομένων

Ενθυλάκωση

- Η αρχή της **ενθυλάκωσης** (encapsulation):
  - σε κάθε κλάση, ο προγραμματιστής που την υλοποίησε, διαχωρίζει:
    - ποια πεδία και μέθοδοι (**public**) μπορούν να χρησιμοποιηθούν από όλο τον υπόλοιπο κώδικα
    - ποια πεδία και μέθοδοι (**private**) μπορούν να χρησιμοποιηθούν μόνο μέσω κλήσης των public μεθόδων της κλάσης…
  - μια συνήθης πρακτική την οποία πρέπει να ακολουθείτε και εσείς είναι ότι **όλα τα πεδία τα δηλώνουμε private**

Παράδειγμα ενθυλάκωσης

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
private:
  char * name;
  int age;

public:
  void init();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

Η φίλη μας εδώ φτιάχνει
• ένα αρχείο person.h το οποίο μοιράζεται με το developer που θα φτιάξει τη main
• ένα αρχείο person.cpp στο οποίο υλοποιούνται οι μέθοδοι

Παράδειγμα ενθυλάκωσης

```
int main(){

  Person x, y;

  x.name = (char *) malloc (10* sizeof(char));
  x.age = 0;

  /* .... */
  strcpy (x.name, "Name1");
  printf ("%s", x.name);

  y.name = (char *) malloc (10* sizeof(char));
  y.age = 0;

  /* .... */
  strcpy (y.name, "Name2");
  printf ("%s", y.name);

  return 0;
}
```

προσπάθεια πρόσβασης σε private πεδία =>

COMPILER ERRORS!!!

Παράδειγμα ενθυλάκωσης

```
int main(){

   Person x, y;

   x.set_name("Lio");
   printf ("%s", x.get_name());

  return 0;
}
```

x,y αντικείμενα που θα επιτελέσουν λειτουργίες

Οι δουλειές που κάνουμε είναι (α) να δώσουμε όνομα στο x και (β) να τυπώσουμε το όνομά του

Το αν η αναπαράσταση εσωτερικά γίνεται με char * ή string δεν (πολυ)απασχολεί τον developer που φτιάχνει τη main
-- κυρίως, όμως, αδιαφορεί για το πώς υλοποιούνται οι get_name, set_name, …

Και το person.cpp

```
void Person::init(){
  name = (char *) malloc (10* sizeof(char));
  if (name == NULL) {printf(…); exit(1);}
  age = 0;
}

void Person::set_name(char *n) {
  strcpy(name, n);
}
void Person::set_age(int a) {
  age = a;
}

char *Person::get_name() {
  return name;
}
int Person::get_age() {
  return age;
}
```


Constructors

Constructors

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
 private:
   char * name;
   int age;
public:
   void init();
   void set_name(char *n);
   void set_age(int a);
   char *get_name();
   int get_age();
};
```


Constructors

```
void Person::init(){
   name = (char *)malloc(10*sizeof(char));
   age = 0;
}

void Person::set_name(char *n) {
   strcpy(name, n);
}

void Person::set_age(int a) {
   age = a;
}

char *Person::get_name() {
   return name;
}

int Person::get_age() {
   return age;
}
```


Constructors

```
int main(){
   Person x, y;

   x.init();

   /* .... */
   x.set_name("Name1");
   printf ("%s", x.get_name());

   y.init();

   /* .... */
   y.set_name("Name2");
   printf ("%s", y.get_name());

   f();

   return 0;
}
```

Constructors

- Σε ένα από τα αρχικά μας παραδείγματα (κλάση Person) είδαμε πώς ο αντικειμενοστρεφής τρόπος σκέψης και σχεδίασης βοηθά στο να προσθέτουμε εύκολα ελέγχους αποσφαλμάτωσης στο πρόγραμμά μας.
- Το πρόγραμμα χρησιμοποιεί μεθόδους για την επεξεργασία των δεδομένων της κλάσης.
- Άρα όποιοι έλεγχοι πρέπει να προστεθούν και όποια λάθη πρέπει να ανιχνευτούν – βρίσκονται σε αυτές τις μεθόδους.
  - **ο υπόλοιπος κώδικας δεν χρειάζεται ψάξιμο….**
    - **π.χ., προσθήκη ελέγχου για το αν έχει γίνει σωστή δέσμευση μνήμης…**

Constructors

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
 private:
   char * name;
   int age;
public:
   void init();
   void set_name(char *n);
   void set_age(int a);
   char *get_name();
   int get_age();
};
```


Constructors

```
void Person::init(){
   name = (char *)malloc(10*sizeof(char));
   if(name == NULL) exit(1);
   age = 0;
}

void Person::set_name(char *n) {
   strcpy(name, n);
}

void Person::set_age(int a) {
   age = a;
}

char *Person::get_name() {
   return name;
}

int Person::get_age() {
   return age;
}
```

Constructors

```
int main(){
  Person x, y;

  x.init();

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());

  y.init();

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  f();

  return 0;
}
```


Constructors

- Επίσης είδαμε ότι το encapsulation βοηθάει στο να επιβάλουμε σε κάθε προγραμματιστή που χρησιμοποιεί την κλάση μας
  - να χρησιμοποιεί μεθόδους της κλάσης για την **σωστή διαχείριση των δεδομένων** που χαρακτηρίζουν τα αντικείμενα της κλάσης.
    - π.χ., σωστή αρχικοποίηση μόνο μέσω της χρήσης της μεθόδου init()….μιας που το πεδίο name είναι private

Constructors

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
 private:
  char * name;
  int age;
public:
  void init();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

Constructors

```
void Person::init(){
  name = (char *)malloc(10*sizeof(char));
  if(name == NULL) exit(1);
  age = 0;
}

void Person::set_name(char *n) {
  strcpy(name, n);
}

void Person::set_age(int a) {
  age = a;
}

char *Person::get_name() {
  return name;
}

int Person::get_age() {
  return age;
}
```


Constructors

```
int main(){
  Person x, y;

  // x.name = (char *)malloc(10*sizeof(char)); NOT ALLOWED
  x.init();

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());

  // x.name = (char *)malloc(10*sizeof(char)); NOT ALLOWED
  y.init();

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  f();

  return 0;
}
```


Constructors

- …και πάλι όμως δεν είμαστε απόλυτα σίγουροι ότι δεν θα προκύψουν σφάλματα στο πρόγραμμα…
  - μια συνήθης περίπτωση είναι σφάλματα που οφείλονται στην μη αρχικοποίηση
  - … **τι γίνεται αν ο προγραμματιστής παραλείψει την κλήση στην init()** για κάποιο αντικείμενο?

Constructors

```
int main(){
  Person x, y;

  x.init();

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());


  // What if we forget to write y.init()???

  /* .... */
  y.set_name("Name2"); // BOOOOM !!!!
  printf ("%s", y.get_name());

  f();

  return 0;
}
```


Constructors

.....πώς μπορούμε να το αποφύγουμε αυτό ??

- Δήλωση **constructors** στις κλάσεις μας
  - μέθοδοι που **καλούνται αυτόματα** όταν δημιουργείται ένα αντικείμενο.
  - Συντακτικό:
    `<classname>()`
    - μπορεί να έχει παραμέτρους
    - ΔΕΝ έχει τύπο επιστροφής – ΟΥΤΕ καν void
  - Υλοποίηση:
    `<classname>::<classname>(){//.....}`
  - **πότε καλούνται ???**
    - δήλωση ενός αντικειμένου.
    - δημιουργία αντικειμένου με new.

Constructors

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
private:
  char * name;
  int age;

 public:
  Person();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

Constructors

```
Person::Person(){
  name = (char *)malloc(10*sizeof(char));
  if(name == NULL) exit(1);
  age = 0;
}

void Person::set_name(char *n) {
  strcpy(name, n);
}
void Person::set_age(int a) {
  age = a;
}

char *Person::get_name() {
  return name;
}
int Person::get_age() {
  return age;
}
```


Constructors

```
int main(){

  Person x, y; // καλείται αυτόματα ο constructor

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  Person *p = new Person(); // καλείται αυτόματα ο constructor

  p->set_name("Name4");
  printf ("%s", p->get_name());

  delete p;

  return 0;
}
```


Destructors

Destructors

```
int main(){

  Person x, y; // καλείται αυτόματα ο constructor

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  Person *p = new Person; // καλείται αυτόματα ο constructor

  p->set_name("Name4");
  printf ("%s", p->get_name());

  delete p;

  return 0;
}
```

Όταν τελειώσει το πρόγραμμα πώς αποδεσμεύουμε τη μνήμη που δεσμεύτηκε για τα πεδία του κάθε αντικειμένου ??

μέθοδος αντίστοιχη της init() ???

Destructors

- Μέθοδοι που καλούνται όταν ένα αντικείμενο καταστρέφεται, δηλαδή παύει να υπάρχει, ήτοι,
  - **όταν τελειώνει η εκτέλεση του μπλοκ εντολών στο οποίο έχει δηλωθεί,**
  - **όταν διαγράφεται η μνήμη στην οποία δείχνει ένας pointer στο αντικείμενο με delete**.
- Απαραίτητοι για λειτουργίες εκκαθάρισης, π.χ.
  - αποδέσμευση μνήμης
  - κλείσιμο αρχείων
- Συντακτικό:

```
~<classname>() // δεν παίρνει παραμέτρους
```

- Υλοποίηση:

```
<classname>::~<classname>()
```


Destructors

```
# include <cstdio>
# include <cstdlib>
# include <cstring>

class Person {
private:
  char * name;
  int age;

 public:
  Person();
  ~Person();
  void set_name(char *n);
  void set_age(int a);
  char *get_name();
  int get_age();
};
```

Destructors

```
Person::Person(){
  name = (char *)malloc(10*sizeof(char));
  if(name == NULL) exit(1);
  age = 0;
}

Person::~Person(){
  free(name);
}

void Person::set_name(char *n) {
  strcpy(name, n);
}
void Person::set_age(int a) {
  age = a;
}
……..
```


Destructors

```
int main(){

  Person x, y;

  /* .... */
  x.set_name("Name1");
  printf ("%s", x.get_name());

  /* .... */
  y.set_name("Name2");
  printf ("%s", y.get_name());

  Person *p = new Person;

  p->set_name("Name4");
  printf ("%s", p->get_name());

  delete p; // καλείται αυτόματα ο destructor

  return 0; // καλείται αυτόματα ο destructor για τα x, y
}
```


Υπερφόρτωση constructors

Υπερφόρτωση constructors

- Γενικά σε μια κλάση μπορούμε να έχουμε περισσότερους constructors με διαφορετικές παραμέτρους.
- Κάθε ένας από αυτούς υλοποιεί μια διαφορετική διαδικασία αρχικοποίησης αντικειμένων που μπορεί να βολεύει κατά περίπτωση…

Ορισμός μιγαδικών

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

class Complex
   {
   private:
      double re, im;
   public:
      void set (double r, double i);
      Complex add (Complex c);
      int equal (Complex c);
   };
```


Ορισμός μεθόδων

```
void Complex::set (double r, double i)
{
   re = r;
   im = i;
}

Complex Complex::add (Complex c)
{
   Complex result;
   result.re = re + c.re;
    result.im = im + c.im;
   return result;
}

int Complex::equal (Complex c)
{
    if ((re == c.re) && (im == c.im))
            return 1;
    else
            return 0;
}
```

Ορισμός και χρήση αντικειμένων

```
int main(){
  Complex a, b, c, d;

  a.set(3, 4);
  b.set(1, 0);

  c = a.add(b);
  d = b.add(a);
  if (c.equal(d) == 1)
   cout << "correct.\n";
  else
   cout << "mistake!\n";

  return 0;
}
```


Υπερφόρτωση constructors

- Μιγαδικοί αριθμοί

```
class Complex
{  // ...
 public:
   Complex ();
   Complex (double r, double i);
   Complex (const Complex & c);
};
```


Υπερφόρτωση constructors

- Μιγαδικοί αριθμοί

```
int main(){

   Complex c1;
   Complex c2(1.0, 4.0);
   Complex c3(c1);
.....
  return 0;
}
```

Υπερφόρτωση constructors

- Default constructor

```
Complex::Complex ()
{
   re = im = 0.0;
}
```

Χρήση: `Complex c1;`

`Complex *pc = new Complex`

- Copy constructor

```
Complex::Complex (const Complex & c)
{
   re = c.re;
   im = c.im;
}
```

Χρήση: `Complex c1(c2);`

`Complex *pc = new Complex(c2)`

43

Υπερφόρτωση constructors

- Parameterized constructors

```
Complex::Complex (double r, double i)
{
   re = r;
   im = i;
}
```

Χρήση:

`Complex c(5.0,6.0);`

`Complex *pc = new Complex (5.0, 6.0);`

44

Υπερφόρτωση constructors

- Αν ΔΕΝ υπάρχει κανένας constructor υλοποιημένος για την κλάση μας, φτιάχνεται ένας default και ένας copy αυτόματα.
  - Ο default θέτει όλα τα πεδία 0 ή NULL αν είναι δείκτες
  - Ο copy αντιγράφει τις τιμές των πεδίων του ορίσματος στα πεδία του νέου αντικειμένου
- Αν υπάρχει έστω και ένας constructor υλοποιημένος (πχ ένας παραμετροποιημένος) ο default δεν παράγεται αυτόματα.

45

Προσοχή στα members που είναι pointers!!!

```
class Name{
public:
  char *s;
  Name();
  Name(char *y)                  //parameterized constructor
  Name (Name &x);                //copy constructor
  ~Name ();                      //destructor
};
Name::Name(){
  cout <<  "a new name is born " << endl;
  s = NULL; // no memory allocation so far
}
```


Προσοχή στα members που είναι pointers!!!

```
Name::Name (char *y){
  s = y; // NOT A DEEP COPY
}
Name::Name (Name &x){
  s = x.s; // NOT A DEEP COPY
}
Name::~Name (){
   cout << s << " is deleted\n";
   delete[] s;
}
```


Προσοχή στα members που είναι pointers!!!

```
main(){
 char *x = new char[10];
 strcpy(x, "abcdefg");
 Name aName(x);
 Name bName(aName);
 delete [] x;
 cout << aName.s << endl; // BOOM!!!!!
}
```

Καλύτερη εκδοχή ...

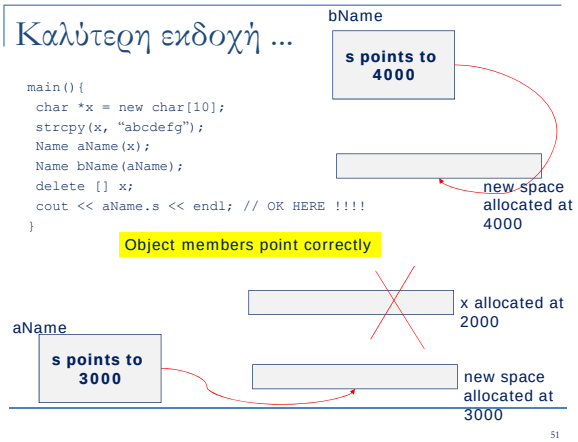
```
class Name{
public:
     char *s;
     Name();           //default constructor
     Name (char *n);   //parameterized con.
     Name (Name &x);
    ~Name ();          //destructor
};
Name::Name(){
  cout <<  "a new name is born " << endl;
  s = NULL; // no memory allocation so far
}
```


Καλύτερη εκδοχή ...

```
Name::Name (char *n){
   cout << n << " is copied\n";
   s = new char [strlen(n)+1];
   strcpy(s,n);  // DEEP COPY HERE
}
Name::Name (Name &x){
  s = new char [strlen(x.s)+1];
  strcpy(s, x.s); // DEEP COPY
}

Name::~Name (){
   cout << s << " is deleted\n";
   delete[] s;
}
```


Καλύτερη εκδοχή ...

```
main(){
 char *x = new char[10];
 strcpy(x, "abcdefg");
 Name aName(x);
 Name bName(aName);
 delete [] x;
 cout << aName.s << endl; // OK HERE !!!!
}
```



Στατικά Πεδία και Μέθοδοι

Στατικά πεδία

- Υπάρχει περίπτωση σε ένα πρόγραμμα να υπάρχουν δεδομένα που χαρακτηρίζουν μια ολόκληρη κλάση αντικειμένων και όχι μόνο κάποιο συγκεκριμένο αντικείμενο της κλάσης
  - π.χ., στατιστικά για το σύνολο αντικειμένων της κλάσης που χρησιμοποιείται σε ένα πρόγραμμα…
  - συμφέρει τα δεδομένα αυτά να είναι μέλη της κλάσης διότι έτσι διευκολύνεται η συντήρηση……

Στατικά πεδία

- Στατικά πεδία είναι πεδία των οποίων η τιμή χαρακτηρίζει μια ολόκληρη κλάση αντικειμένων και όχι μόνο κάποιο συγκεκριμένο αντικείμενο της κλάσης.
  - Γενικά, αν για κάποιο πεδίο της κλάσης θέλω
    1. υπάρχει ίδια τιμή **για όλα τα αντικείμενα της κλάσης**.
    2. και αν μεταβληθεί αυτή η τιμή για ένα αντικείμενο να μεταβάλλεται αυτόματα και για όλα τα υπόλοιπα….
  - Αυτό το επιτυγχάνω δηλώνοντας το πεδίο ως **`static`**.

Στατικά πεδία και μέθοδοι

- Τα πεδία που χαρακτηρίζουν μια ολόκληρη κλάση μπορούμε να τα διαχειριστούμε καλώντας μεθόδους σε ένα οποιοδήποτε αντικείμενο της κλάσης.

Στατικά πεδία και μέθοδοι

- Έστω ένα πρόγραμμα διαχείρισης πληροφοριών για ανθρώπους
- πέρα από τα προσωπικά στοιχεία καθενός μας ενδιαφέρει και η διαχείριση στατιστικών που αφορούν στο σύνολο των ανθρώπων που διαχειρίζεται το πρόγραμμα όπως για παράδειγμα ο μέσος όρος ηλικίας τους….

Στατικά πεδία και μέθοδοι

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

class Person {
private:
  double age;
  static double totalAge;
  static int number;
public:
  Person (double a);
  ~Person ();

  void printAge();
  void printAvgAge ();
};
```

Στατικά πεδία και μέθοδοι

```
double Person::totalAge = 0.0;
int Person::number = 0;

Person::Person (double a)
{
  age = a;
  totalAge += a;
  number ++;
}

Person::~Person ()
{
  totalAge -= age;
  number --;
}
```


Στατικά πεδία και μέθοδοι

```
void Person::printAge ()
{
  cout << age << " old.\n";
}

void Person::printAvgAge ()
{
  cout << totalAge/number << " old.\n";
}
```


Στατικά πεδία και μέθοδοι

```
int main ()
{
  // …..
  Person x(65), y(100);

  x.printAge();
  x.printAvgAge();

  Person * z = new Person(85);

  delete z;
  y.printAvgAge();

  return 0;
}
```

Στατικά πεδία και μέθοδοι

- Τα στατικά πεδία που χαρακτηρίζουν μια ολόκληρη κλάση μπορούμε να τα διαχειριστούμε καλώντας μεθόδους σε ένα οποιοδήποτε αντικείμενο της κλάσης.
- Τι γίνεται όμως αν σε ένα πρόγραμμα θέλουμε να επεξεργαστούμε ένα στατικό πεδίο ενώ δεν έχουμε δημιουργήσει κανένα αντικείμενο της κλάσης ???
  - Οι **στατικές μέθοδοι** είναι μέθοδοι μιας κλάσης που μπορούν να κληθούν αυτόνομα –
    - δεν χρειάζεται να κληθούν πάνω σε ένα αντικείμενο της κλάσης και χρησιμοποιούνται για την πρόσβαση σε static πεδία της κλάσης.
      - ουσιαστικά είναι κάτι σαν **global functions**, και χρησιμοποιούνται για την επεξεργασία στατικών πεδίων μιας κλάσης ανεξαρτήτως των αντικειμένων της κλάσης.

Στατικά πεδία και μέθοδοι

- Έστω ένα πρόγραμμα διαχείρισης πληροφοριών για ανθρώπους
  - πέρα από τα προσωπικά στοιχεία καθενός μας ενδιαφέρει και η διαχείριση στατιστικών που αφορούν στο σύνολο των ανθρώπων που διαχειρίζεται το πρόγραμμα όπως για παράδειγμα ο μέσος όρος ηλικίας τους….
  - το πρόγραμμα ζητάει από το χρήστη επαναληπτικά να επιλέξει μεταξύ 2 επιλογών
    - αν ο χρήστης επιλέξει την πρώτη δημιουργεί ένα νέο αντικείμενο Person
    - αν επιλέξει τη δεύτερη επιλογή τυπώνεται στην οθόνη ο μέσος όρος ηλικίας των αντικειμένων Person

στην περίπτωση που ο χρήστης επιλέξει στην 1η επανάληψη τη 2η επιλογή πρέπει το πρόγραμμα με κάποιο τρόπο να τυπώσει το ΜΟ ηλικίας χωρίς να έχει δημιουργηθεί κάποιο αντικείμενο Person

… αυτό μπορούμε να το πετύχουμε αν οι μέθοδοι που υπολογίζουν το ΜΟ είναι στατικές….

Στατικά πεδία και μέθοδοι

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

class Person {
private:
  double age;
  static double totalAge;
  static int number;
public:
  Person (double a);
  ~Person ();

  void printAge();
  static int getNumber();
  static void printAvgAge ();
};

double Person::totalAge = 0.0;
int Person::number = 0;
```

Στατικά πεδία και μέθοδοι

```
Person::Person (double a)
{
  age = a;
  totalAge += a;
  number ++;
}

Person::~Person ()
{
  totalAge -= age;
  number --;
}

void Person::printAge ()
{
  cout << age << " old.\n";
}
```


Στατικά πεδία και μέθοδοι

```
int Person::getNumber(){
  return number;
}

void Person::printAvgAge ()
{
  if (number !=0)
    cout << totalAge/number << " old.\n";
  else
    cout << "0 old\n";
}
```


Στατικά πεδία και μέθοδοι

```
int main (){
  Person *people[100];
  int choice = -1;
  double age;

  while (choice != 0) {
    cin >> choice;
    if (choice == 1) {
      cin >> age;
      people[Person::getNumber()] = new Person(age);
      people[Person::getNumber()]->printAvgAge();
    } else
      if (choice == 2) {
        // δεν ξέρουμε πάντα αν υπάρχουν αντικείμενα μιας κλάσης
        // για να επεξεργαστούμε μέσω αυτών στατικά πεδία μιας κλάσης
        // ΑΡΑ οι στατικές μέθοδοι είναι απαραίτητες
        Person::printAvgAge();
      }
  }
  return 0;
}
```

Κλήση

~~Αντί για~~
~~**object.**methodname(parameters);~~
έχουμε
Class::methodname(parameters);

Δείκτης this

- Δείχνει στο αντικείμενο επάνω στο οποίο καλείται μια μέθοδος
- Παράδειγμα

```
void Complex::set (double r, double i)
{
   this->re = r;
   this->im = i;
}
```

Συνάθροιση Αντικειμένων (Aggregation)

Θεματολόγιο

- Συνάθροιση Αντικειμένων
  - Παράδειγμα
  - Βελτιστοποιημένο Παράδειγμα

Παράδειγμα

- Έστω ένα απλό πρόγραμμα που διαχειρίζεται τα προσωπικά στοιχεία ενός συνόλου προσώπων.
- Κάθε πρόσωπο χαρακτηρίζεται από
  - όνομα
  - επίθετο
  - διεύθυνση κατοικίας
    - οδός
    - αριθμός
    - πόλη
    - …

Παράδειγμα

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

class Person{
private:
  char fname[40];
  char lname[40];

  char street[40];
  int number;
  char city[40];

public:
  Person(char fn[], char ln[], char str[], int no, char ct[]);

  char *getAddress();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], char str[], int no, char ct[]){
  strcpy(fname, fn);
  strcpy(lname, ln);

  strcpy(street, str);
  strcpy(city, ct);
  number = no;
}

char * Person::getAddress(){
  char *address = new char[strlen(street)+strlen(city)+1];
  sprintf(address, "%s %d %s", street, number, city);
  return address;
}

char * Person::getPersonalDetails(){
  char *details = new char[strlen(fname)+strlen(lname)+1];
  sprintf(details, "%s %s", fname, lname);
  return details;
}
```

Παράδειγμα

```
int main(){
  char fn[40], ln[40], str[40], ct[40];
  int no;

  cin >> fn >> ln >> str >> no >> ct;
  Person *pit = new Person (fn, ln, str, no, ct);
  cin >> fn >> ln;
  Person *ang = new Person (fn, ln, str, no, ct);

  char *details = pit -> getPersonalDetails();
  char *address = pit -> getAddress();
  cout << "First person - Details : "
       << details << " - Address : " << address << endl;
  delete details;
  delete address;

  details = ang -> getPersonalDetails();
  address = ang -> getAddress();
  cout << "First person - Details : " << details
       << " - Address : " << address << endl;
  delete pit;
  delete ang;
  delete details;
  delete address;
}
```

Παράδειγμα

- Γενικά στο πρόγραμμά μας μπορεί να προκύψει η περίπτωση περισσότερα από ένα Person αντικείμενα να μένουν στην ίδια διεύθυνση.
  - Κατά συνέπεια, έτσι όπως έχει σχεδιαστεί το πρόγραμμα, δαπανά μνήμη για την αποθήκευση της ίδιας πληροφορίας.
  - Για την επίλυση του προβλήματος πρέπει να γίνει μια καλύτερη σχεδίαση του προγράμματός μας στην οποία θα βασιστούμε στην τεχνική της **συνάθροισης αντικειμένων (aggregation)**.
    - Έστω ότι για την επεξεργασία διευθύνσεων ορίζουμε μια επιπλέον κλάση Address.

Συνάθροιση Αντικειμένων (Aggregation)

- Πολλές φορές σε ένα πρόβλημα θέλουμε να κωδικοποιήσουμε το γεγονός ότι
  - ένα αντικείμενο της κλάσης X (πχ ένας άνθρωπος) χαρακτηρίζεται από ένα αντικείμενο της κλάσης Y (πχ μια διεύθυνση), **και**
  - ένα αντικείμενο της κλάσης Y (πχ η ίδια διεύθυνση) μπορεί να χαρακτηρίζει πολλαπλά αντικείμενα της κλάσης X (πχ περισσότερους από έναν άνθρωπο που μένουν στο ίδιο σπίτι)
    - (και πιθανόν πολλαπλά αντικείμενα (πχ κτίρια) μιας άλλης κλάσης Z…)
- Προσοχή: **η ύπαρξη των αντικειμένων της Y δεν εξαρτάται άμεσα από την ύπαρξη των αντικειμένων της X** (πχ το πρόγραμμα διαχειρίζεται μια λίστα από διευθύνσεις ακόμα και αν δεν υπάρχουν άνθρωποι που μένουν σε αυτές)
  - μεταξύ των δύο κλάσεων υπάρχει όπως λέμε μια **σχέση συνάθροισης**.
    - δηλ., κάθε αντικείμενο της X αποτελείται από ένα «κοινόχρηστο» αντικειμένου της Y
    - τα αντικείμενα της X «μοιράζονται» αντικείμενα της Y

Συνάθροιση Αντικειμένων (Aggregation)

- Στην περίπτωση αυτή κατασκευάζουμε τις κλάσεις X, Y και η κλάση X έχει δείκτη σε αντικείμενα της κλάσης Y.
  - Μια που η ύπαρξη των αντικειμένων της κλάσης Y (δημιουργία, καταστροφή, κλπ), δεν εξαρτάται από την ύπαρξη (δημιουργία, καταστροφή κλπ) των αντικειμένων των κλάσεων X, Z, …
    - τα αντικείμενα της Y στα οποία δείχνει ο δείκτης της κλάσης X δεν κατασκευάζονται ούτε καταστρέφονται από την κλάση X (πχ στον constructor - destructor)
    - συνήθως τα αντικείμενα της Y τα δίνουμε σαν παραμέτρους αρχικοποίησης στον constructor της X

Παράδειγμα

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

class Address{
private:
  char street[40];
  int number;
  char city[40];

public:
  Address(char str[], int no, char ct[]);
  char *getAddress();
};

Address::Address(char str[], int no, char ct[]){
  strcpy(street, str);
  strcpy(city, ct);
  number = no;
}
```

Παράδειγμα

```
Address::Address(char str[], int no, char ct[]){
  strcpy(street, str);
  strcpy(city, ct);
  number = no;
}

char * Address::getAddress(){
  char *address = new
   char[strlen(street)+strlen(city)+1];

  sprintf(address, "%s %d %S", street, number, city);

  return address;
}
```

Παράδειγμα

```
class Person{
private:
  char fname[40];
  char lname[40];

  Address *paddress;

public:
  Person(char fn[], char ln[], Address* addr);

  char *getAddress();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], Address *addr){
  strcpy(fname, fn);
  strcpy(lname, ln);
  paddress = addr; // δεν κατασκευάζονται εδώ τα Address αντικείμενα
}

char * Person::getAddress(){
  char *address = paddress->getAddress();

  return address;
}

char * Person::getPersonalDetails(){
  char *details = new char[strlen(fname)+strlen(lname)+1];

  sprintf(details, "%s %s", fname, lname);

  return details;
}
```

Παράδειγμα

```
int main(){
  char fn[40], ln[40], str[40], ct[40]; int no;

  cin >> str >> no >> ct;
  Address *addr = new Address (str, no, ct);

  cin >> fn >> ln;
  Person *pit = new Person (fn, ln, addr);
  cin >> fn >> ln;
  Person *ang = new Person (fn, ln, addr);

  char *details = pit -> getPersonalDetails();
  char *address = pit -> getAddress();
  cout << "First person - Details : "
       << details << " - Address : " << address << endl;

  details = ang -> getPersonalDetails();
  address = ang -> getAddress();
  cout << "First person - Details : " <<
       details << " - Address : " << address << endl;

  delete pit; delete ang; // το addr δεν πρέπει
                          // να καταστραφεί στο destructor γιατί είναι κοινό
  delete details; delete address;
  delete addr;
}
```

Σύνθεση Αντικειμένων (Composition)

Σύνθεση Αντικειμένων (Composition)

- Πολλές φορές σε ένα πρόβλημα θέλουμε να κωδικοποιήσουμε το γεγονός ότι **μια έννοια Χ** (π.χ., κτίριο) **αποτελείται από άλλες έννοιες** Y, Z (π.χ., πόρτες, παράθυρα), **οι οποίες γενικά δεν μπορεί να χαρακτηρίζουν από κοινού και άλλες έννοιες του προβλήματος μας**
  - πχ μια πόρτα είναι επιμέρους συνθετικό ενός μόνο κτιρίου και δεν μπορεί να χαρακτηρίζει 2 διαφορετικά κτίρια…
- Τότε μεταξύ των κλάσεων αυτών υπάρχει μια σχέση σύνθεσης
  - δηλ. το κάθε αντικείμενο της περιέχουσας κλάσης (π.χ., κτίριο) αποτελείται από ένα ή περισσότερα αντικείμενα των περιεχόμενων κλάσεων (π.χ., πόρτες, παράθυρα) τα οποία δεν τα μοιράζεται με άλλα αντικείμενα της περιέχουσας κλάσης (π.χ., άλλα κτίρια)….

Σύνθεση Αντικειμένων (Composition)

- Στην περίπτωση αυτή κατασκευάζουμε τις κλάσεις X, Y, Z και η κλάση X έχει ως πεδία
  - είτε αντικείμενα των κλάσεων Y, Z
  - είτε δείκτες σε αντικείμενα των κλάσεων Y, Z
- Τα αντικείμενα της X αποτελούν **σύνθεση** αντικειμένων των κλάσεων Y, Z …

Σύνθεση Αντικειμένων (Composition)

- **Η δημιουργία και καταστροφή των αντικειμένων στα οποία δείχνουν οι δείκτες είναι αρμοδιότητα της κλάσης Χ**
  - αυτό είναι λογικό μια που τα επιμέρους αντικείμενα των Υ, Ζ που χαρακτηρίζουν ένα αντικείμενο της Χ ΔΕΝ χαρακτηρίζουν άλλα αντικείμενα της Χ ή αντικείμενα άλλων κλάσεων του προγράμματος μας

Παράδειγμα

- Έστω ένα απλό πρόγραμμα που διαχειρίζεται πληροφορίες για τις οικογένειες μιας πόλης
- Μια **οικογένεια** χαρακτηρίζεται από
  - επίθετο
  - πλήθος τέκνων
  - και αποτελείται από επιμέρους **μέλη**
    - πατέρας
    - μητέρα
    - τέκνα
- Κάθε **μέλος** ανήκει σε μια μόνο οικογένεια και χαρακτηρίζεται από
  - όνομα
  - ηλικία
  - φύλλο

Παράδειγμα

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

class Member{
private:
  char fname[40];
  int age;
  int gender;
public:
  Member(char fn[], int age, int gender);
  int getAge();
};

Member::Member(char fn[], int ag, int gend){
  strcpy(fname, fn);
  age = ag;
  gender = gend;
}
int Member::getAge() {
  return age;
}
```

Παράδειγμα

```
class Family{
private:
  char lname[40];
  int nextChild;

  Member *father;
  Member *mother;
  Member *children[20];

  Member *createMember(char n[], int age, int gd);

public:
  Family(char ln[]);
  void setFather(char n[], int age, int gd);
  void setMother(char n[], int age, int gd);
  void addChild(char n[], int age, int gd);
  float avgAge();
  ~Family();
};
```


Παράδειγμα

```
Family::Family(char ln[]){
  strcpy(lname, ln);
  nextChild = 0;
  father = mother = NULL;
}

void Family::setFather(char n[], int age, int gd){
// τα επιμέρους συνθετικά της Family
// δημιουργούνται από την ίδια την κλάση
  father = createMember(n, age, gd);
}

void Family::setMother(char n[], int age, int gd){
// τα επιμέρους συνθετικά της Family
// δημιουργούνται από την ίδια την κλάση
  mother = createMember(n, age, gd);
}
```


Παράδειγμα

```
void Family::addChild(char n[], int age, int gd){
   // τα επιμέρους συνθετικά της Family
   // δημιουργούνται από την ίδια την κλάση
  children[nextChild++] = createMember(n, age, gd);
}

Member *Family::createMember(char n[], int age, int gd){
  if((age >= 0) && (gd >= 1) && (gd <= 2))
    return new Member(n, age, gd);
  else {
    cerr << "Error in Member creation !" << endl;
    return NULL;
  }
}
```

Παράδειγμα

```
float Family::avgAge() {
  float avg;
  int total_members = 0;

  for (int i = 0; i < nextChild; i++)
    avg += children[i]->getAge();
  total_members = nextChild;
  if(father != NULL) {
   avg += father->getAge();
    total_members++;
  }
  if(mother != NULL) {
   avg += mother->getAge();
    total_members++;
  }
  return avg/total_members;
}
```


Παράδειγμα

```
Family::~Family(){
// τα επιμέρους συνθετικά της Family καταστρέφονται από την ίδια την κλάση
// Όταν καταστρέφεται το σύνθετο αντικείμενο (οικογένεια), καταστρέφονται
// και τα συστατικά του
  cout << "Cleaning up father" << endl;
  if (father != NULL) delete father;
  cout << "Cleaning up mother" << endl;
  if (mother != NULL) delete mother;
  for(int i = 0; i < nextChild; i++){
    cout << "Cleaning up child : " << i+1 << endl;
    delete children[i];
  }
}
```


Παράδειγμα

```
int main(){
  Family family("FamilyName");

  family.setFather("father", 30, 1);
  family.setMother("mother", 25, 0);
  family.addChild("child1", 5, 1);
  family.addChild("child2", 7, 1);

  cout << "Average Family Age : " <<
             family.avgAge() << endl;
}
```

Κληρονομικότητα
(is – a Inheritance)

Θεματολόγιο

- Κληρονομικότητα
  - Παράδειγμα
  - Βελτιστοποιημένο Παράδειγμα – Κληρονομικότητα
  - Encapsulation : public – private - protected πεδία
  - Encapsulation : public – private - protected κληρονομικότητα
  - Ανάθεση αντικειμένων και Κληρονομικότητα
  - Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και Κληρονομικότητα
  - Πέρασμα παραμέτρων και Κληρονομικότητα
  - Πέρασμα παραμέτρων 2 και Κληρονομικότητα
  - Άλλα θέματα

Παράδειγμα

- Έστω ένα απλό πρόγραμμα που διαχειρίζεται πληροφορίες για τους πελάτες και τους υπαλλήλους μιας εταιρίας….

Παράδειγμα

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

using namespace std;


class Employee {
private:
  char fname[40];
  char lname[40];
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
class Customer {
private:
  char fname[40];
  char lname[40];
  int credit;
  char creditType[10];

public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  char *getPersonalDetails();
  int getCredit();
};
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal){
  strcpy(fname, fn);
  strcpy(lname, ln);
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
  strcpy(creditType, ct);
  credit = 0;
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

char *Customer::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = jonh.getPersonalDetails();
  sal = jonh.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  pet.chargeCredit(250);
}
```

Παράδειγμα

- Η υπάρχουσα σχεδίαση και υλοποίηση με τις 2 κλάσεις είναι προβληματική μια που οι δύο έννοιες (υπάλληλος, πελάτης) έχουν κάποια κοινά χαρακτηριστικά και κώδικα ο οποίος επαναλαμβάνεται.
- Γενικά αυτό δεν διευκολύνει τη συντήρηση …

Παράδειγμα

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

using namespace std;


class Employee {
private:
  char fname[40];
  char lname[40];
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
class Customer {
private:
  char fname[40];
  char lname[40];
  int credit;
  char creditType[10];

public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  char *getPersonalDetails();
  int getCredit();
};
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal){
  strcpy(fname, fn);
  strcpy(lname, ln);
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
  strcpy(creditType, ct);
  credit = 0;
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

char *Customer::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = jonh.getPersonalDetails();
  sal = jonh.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  pet.chargeCredit(250);
}
```

Κληρονομικότητα

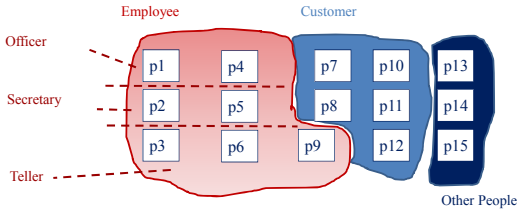
- Το πρόβλημα:
  - Σε πολλές εφαρμογές έχουμε διάφορες κλάσεις αντικειμένων οι οποίες έχουν κοινά χαρακτηριστικά/πεδία και μεθόδους
    - πχ. τόσο οι εργαζόμενοι όσο και οι πελάτες μιας εταιρίας χαρακτηρίζονται από ένα όνομα, επίθετο, κλπ….
  - ένα σφάλμα μπορεί να εισαχθεί σε πολλά διαφορετικά σημεία
  - μια αλλαγή θα πρέπει να γίνει σε πολλά διαφορετικά σημεία
  - μια διόρθωση, ένας έλεγχος θα πρέπει να γίνει σε πολλά διαφορετικά σημεία

Κληρονομικότητα

- Γιατί υπάρχει το παραπάνω χαρακτηριστικό?
- Υποψιαζόμαστε ότι οι υπάλληλοι και οι πελάτες μοιράζονται χαρακτηριστικά γιατί όλοι ανήκουν σε ένα ευρύτερο υπερσύνολο, αυτό των ανθρώπων

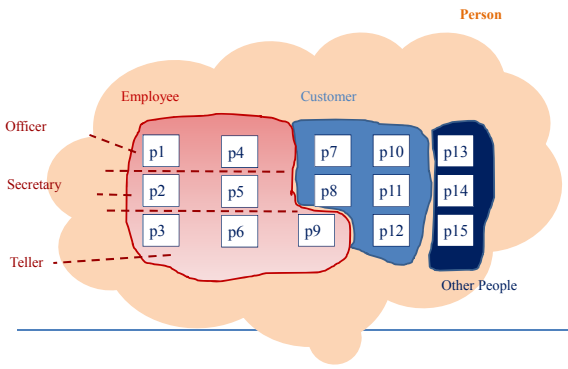
- Η σχέση του συνόλου των ανθρώπων με τα σύνολα των πελατών και των υπαλλήλων είναι σχέση υπερσυνόλου
- Σημαντική παρατήρηση: οι πελάτες/υπάλληλοι ΕΙΝΑΙ ΚΑΙ μέλη του συνόλου των ανθρώπων

Εξειδίκευση



17

Εξειδίκευση



18

Κληρονομικότητα

- Για καλύτερη οργάνωση και συντήρηση του κώδικα (για να μην επαναλαμβάνεται ο ίδιος κώδικας πολλές φορές)
  - φτιάχνουμε μια **γενική/βασική κλάση** που να περιλαμβάνει τα κοινά χαρακτηριστικά/πεδία και μεθόδους δύο ή περισσοτέρων κλάσεων και
  - εν συνεχεία **επεκτείνουμε** τη βασική φτιάχνοντας κλάσεις που **κληρονομούν** από αυτή.

Κληρονομικότητα

- Τι σημαίνει επέκταση...
  - Στις παραγόμενες κλάσεις **δηλώνουμε επιπλέον χαρακτηριστικά και λειτουργίες**.
  - Τα αντικείμενα έχουν **όλα τα χαρακτηριστικά και λειτουργίες που δηλώνονται στην βασική κλάση,**
  - **καθώς και τα επιπλέον χαρακτηριστικά και λειτουργίες που δηλώνονται στην παραγόμενη κλάση**

Κληρονομικότητα

- Τα αντικείμενα των παραγόμενων κλάσεων συμπεριφέρονται και ως αντικείμενα της βασικής κλάσης (π.χ., στο πέρασμα παραμέτρων)
- Η επέκταση ονομάζεται και σχέση **IS_A** («είναι»).

Κληρονομικότητα

- Για τα αντικείμενα των παραγόμενων κλάσεων
  - Κατά τη δημιουργία τους **πρώτα εκτελείται ο κώδικας του constructor της βασικής κλάσης** και εν συνεχεία εκτελείται ο κώδικας **του constructor της παραγόμενης κλάσης.**
  - Οι **destructors καλούνται με την αντίστροφή σειρά**….

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  ~Person();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

Person::~Person(){
   cout << "~Person() called\n";
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  ~Employee();
  int getSalary();
};


Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

Employee::~Employee(){ cout << "~Employee() called"; }

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  ~Customer();
  void chargeCredit(int amount);
  int getCredit();
};

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

Customer::~Customer(){ cout << "~Customer called\n"; }

void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = jonh.getPersonalDetails();
  sal = jonh.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  pet.chargeCredit(250);


  // οι destructors εκτελούνται αντίστροφα ….
}
```

Encapsulation: public-private-protected πεδία και μέθοδοι

Κληρονομικότητα – Πεδία και Μέθοδοι

- πεδία και μέθοδοι **public**: ορατά από όλους
  - αλλαγές / διορθώσεις μπορεί να επηρεάσουν τον υπόλοιπο κώδικα
- πεδία και μέθοδοι **private**: ορατά από την υλοποίηση των μεθόδων της κλάσης
  - αλλαγές / διορθώσεις / έλεγχοι συγκεντρώνονται στον κώδικα της κλάσης και δεν επηρεάζουν τον υπόλοιπο κώδικα
- πεδία και μέθοδοι **protected**: ορατά από
  - την υλοποίηση των μεθόδων της κλάσης, και επιπλέον από
  - την υλοποίηση των μεθόδων των κλάσεων που τα κληρονομούν
    - αλλαγές / διορθώσεις / έλεγχοι συγκεντρώνονται στον κώδικα της κλάσης
    - επηρεάζουν τον κώδικα των παραγόμενων κλάσεων
    - δεν επηρεάζουν τον υπόλοιπο κώδικα

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
};
```

τα private πεδία της βασικής δεν μπορούν να χρησιμοποιηθούν απευθείας στις παραγόμενες
…ούτε βέβαια στον υπόλοιπο κώδικα…

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};


Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
   //  cout << "First Name : " << fname << " - Last Name : " << lname << endl;
   // Compile error !!
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
class Customer : public Person {
private:
  int credit;
  char creditType[10];

public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = jonh.getPersonalDetails();
  sal = jonh.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  pet.chargeCredit(250);

}
```

Παράδειγμα - protected

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

using namespace std;

class Person {
protected:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
};
```

τα protected πεδία της βασικής μπορούν να χρησιμοποιηθούν απευθείας στις παραγόμενες ΟΧΙ όμως στον υπόλοιπο κώδικα !

Παράδειγμα - protected

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα - protected

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};


Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
   cout << "First Name : " << fname << " - Last Name : " << lname << endl;
   // OK NO Compile error !!
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα - protected

```
class Customer : public Person {
private:
  int credit;
  char creditType[10];

public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα - protected

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  char *details = jonh.getPersonalDetails();
  sal = jonh.getSalary();

  cout << details << " - Salary : " << sal << endl;
  delete [] details;

  // cout << pet.fname; Compile error !!! fname protected
  pet.chargeCredit(250);

}
```

Encapsulation: public - private - protected κληρονομικότητα

public, private, protected κληρονομικότητα

- Κληρονομικότητα με public:
  - τα πεδία και οι μέθοδοι της βασικής κλάσης διατηρούν τους περιορισμούς εμβέλειας που έχουν
    - για όλους όσους χρησιμοποιούν αντικείμενα της παραγόμενης κλάσης,
    - καθώς και για κλάσεις που κληρονομούν από την παραγόμενη κλάση.

Κλάσεις και εμβέλεια (συνέχεια)

- Κληρονομικότητα με private:
  - τα public και protected πεδία και οι μέθοδοι της βασικής κλάσης X γίνονται private
    - για όλους όσους χρησιμοποιούν αντικείμενα της **παραγόμενης κλάσης Y**,
    - καθώς και για κλάσεις **Z, K, L, που κληρονομούν** από την **παραγόμενη κλάση**.
  - Για την ίδια την παραγόμενη κλάση Y οι περιορισμοί εμβέλειας παραμένουν ως έχουν δηλωθεί στην βασική κλάση X.
  - => χρήσιμο για διευκόλυνση της συντήρησης
    - αν αλλάξουμε μια public μέθοδο μιας βασικής κλάσης και
    - έχουμε **ένα κώδικα που χρησιμοποιεί αντικείμενα μιας παραγόμενης κλάσης** που προέκυψε με private κληρονομικότητα,
      - ξέρουμε ότι αυτός ο κώδικας δεν επηρεάζεται από την αλλαγή

Παράδειγμα

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

using namespace std;

class Person {
protected:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
};
```

Παράδειγμα

```
class Employee : private Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : private Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα

```
class VIPCustomer : public Customer {
private:
  int vipCode;
public:
  VIPCustomer(char fn[], char ln[], char ct[], int code);
  int getVIPCode();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
  cout << "First Name : " << fname <<
        " - Last Name : " << lname << endl;   // ok because protected
  cout << getPersonalDetails() << endl;       // ok because public
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) :
   Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
VIPCustomer::VIPCustomer(char fn[], char ln[],
                char ct[], int code) : Customer(fn, ln, ct){
  vipCode = code;
}

int VIPCustomer::getVIPCode(){
  // cout << getPersonalDetails();
  // cout << fname << lname;
  // Compile error - public method BUT private inheritance
  return vipCode;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  // cout << jonh.getPersonalDetails() << endl;
  // Compile error - public method BUT private inheritance
  cout << "Salary : " << jonh.getSalary() << endl;

  pet.chargeCredit(250);

  return 1;
}
```

Κλάσεις και εμβέλεια (συνέχεια)

- Κληρονομικότητα με protected:
  - τα public πεδία και οι μέθοδοι της βασικής κλάσης X γίνονται protected
    - για όλους όσους χρησιμοποιούν **αντικείμενα της παραγόμενης κλάσης**.
    - καθώς και **για κλάσεις Z, K, L, που κληρονομούν από την παραγόμενη κλάση**.
  - Για την ίδια την παραγόμενη κλάση Y οι περιορισμοί εμβέλειας παραμένουν ως έχουν δηλωθεί στην βασική κλάση.

Παράδειγμα

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

using namespace std;

class Person {
protected:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
};
```

Παράδειγμα

```
class Employee : protected Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : protected Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα

```
class VIPCustomer : public Customer {
private:
  int vipCode;
public:
  VIPCustomer(char fn[], char ln[], char ct[], int code);
  int getVIPCode();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
  cout << "First Name : " << fname <<
      " - Last Name : " << lname << endl; // ok because protected
  cout << getPersonalDetails() << endl; // ok because public method
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[],
                   char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
VIPCustomer::VIPCustomer(char fn[], char ln[], int code) :
   Customer(fn, ln){
  vipCode = code;
}

int VIPCustomer::getVIPCode(){
  cout << getPersonalDetails();
  // OK -  public method that becomes protected
  // because of protected inheritance
  return vipCode;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;
  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  // cout << jonh.getPersonalDetails() << endl;
  // Compile error - public method BUT protected inheritance
  cout << "Salary : " << jonh.getSalary() << endl;

  pet.chargeCredit(250);

  return 1;
}
```

Ανάθεση Αντικειμένων και Κληρονομικότητα

Ανάθεση Αντικειμένων και Κληρονομικότητα

- Στην ανάθεση μεταξύ δύο αντικειμένων x = y πρέπει
  - τα δύο αντικείμενα να είναι της ίδιας κλάσης ή
  - η κλάση του αντικειμένου που ανατίθεται (y) να κληρονομεί (άμεσα ή έμμεσα) από την κλάση του αντικειμένου στο οποίο γίνεται η ανάθεση (x)

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

Customer::Customer(char fn[], char ln[],
                   char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  cout << jonh.getPersonalDetails() <<
        " - Salary : " << jonh.getSalary() << endl;

  pet.chargeCredit(250);
  cout << pet.getPersonalDetails() <<
        " - Credit : " << pet.getCredit() << endl;

  ……
```

Παράδειγμα

```
  Person johnPrivateLife = jonh; // upcasting
  Person petPrivateLife = pet;  // upcasting

  cout << johnPrivateLife.getPersonalDetails() << endl;
  cout << petPrivateLife.getPersonalDetails() << endl;
  // petPrivateLife.chargeCredit(130);
  // compile error ..

  cin >> fname >> lname;
  petPrivateLife.setPersonalDetails(fname, lname);

  cout << petPrivateLife.getPersonalDetails() << endl;
  cout << pet.getPersonalDetails() <<
        " - Credit : " << pet.getCredit() << endl;
}
```

Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και Κληρονομικότητα

Ανάθεση διευθύνσεων αντικειμένων σε δείκτες και Κληρονομικότητα

- Στην ανάθεση της διεύθυνσης ενός αντικειμένου σε ένα pointer p = & x πρέπει
  - ο p να είναι pointer της κλάσης στην οποία ανήκει το αντικείμενο ή
  - pointer κλάσης από την οποία κληρονομεί η κλάση του αντικειμένου.
- Το ποιες μέθοδοι μπορούν να κληθούν μεσω του pointer, **καθορίζεται από τον τύπο του pointer** και όχι από τον τύπο του αντικειμένου.

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cout << "Give fname, lname, sal" << endl;
  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cout << "Give fname, lname" << endl;
  cin >> fname >> lname;
  Customer pet(fname, lname, (char *) "VISA");

  cout << jonh.getPersonalDetails() <<
        " - Salary : " << jonh.getSalary() << endl;

  pet.chargeCredit(250);
  cout << pet.getPersonalDetails() <<
        " - Credit : " << pet.getCredit() << endl;

  ……
```

Παράδειγμα

```
……
  Person *johnPrivateLife = &jonh;
  Person *petPrivateLife = &pet;

  cout << johnPrivateLife->getPersonalDetails() << endl;
  cout << petPrivateLife->getPersonalDetails() << endl;
  // petPrivateLife->chargeCredit(130);
  // compile error ..

  cout << "Give fname, lname" << endl;
  cin >> fname >> lname;
  petPrivateLife->setPersonalDetails(fname, lname);

  cout << petPrivateLife->getPersonalDetails() << endl;
  cout << pet.getPersonalDetails() <<
        " - Credit : " << pet.getCredit() << endl;
}
```

Πέρασμα παραμέτρων και Κληρονομικότητα

Πέρασμα παραμέτρων και Κληρονομικότητα

- Είδαμε στα προηγούμενα παραδείγματα ότι
  - Σε ένα αντικείμενο βασικής κλάσης μπορεί να ανατεθεί ένα αντικείμενο παραγόμενης κλάσης
  - Σε ένα δείκτη σε αντικείμενα βασικής κλάσης μπορεί να ανατεθεί η διεύθυνση ενός αντικειμένου παραγόμενης κλάσης
- Αυτές οι δύο δυνατότητες αποκτούν εξαιρετική χρησιμότητα αν σκεφτούμε την περίπτωση που το αντικείμενο βασικής κλάσης ή ο δείκτης βασικής κλάσης είναι παράμετρος σε μια άλλη συνάρτηση ή μέθοδο του προγράμματος μας.
  - **Η συνάρτηση μας μπορεί να εκτελεστεί επιτυχημένα χωρίς καμία αλλαγή για αντικείμενα οποιασδήποτε κλάσης παράγεται από τη βασική κλάση**….
  - => η συνάρτηση μας είναι **επαναχρησιμοποιήσιμη**

Παράδειγμα

- Έστω ότι στο πρόγραμμά μας θέλουμε να προσθέσουμε την επιπλέον δυνατότητα να μετατρέπει τις πληροφορίες που διαχειρίζεται για τους πελάτες και τους υπαλλήλους σε html σελίδες και να τις εκτυπώνει ….

Παράδειγμα – Πέρασμα δια τιμής

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα – Πέρασμα δια τιμής

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα – Πέρασμα δια τιμής

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα – Πέρασμα δια τιμής

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα – Πέρασμα δια τιμής

```
void HTMLFormattedPrint( Person p ){
  cout << "<HTML>" << p.getPersonalDetails()
       << "</HTML>" << endl;
  p.setPersonalDetails("", "");
}
```

για να δούμε τη διαφορά περάσματος δια τιμής, περάσματος με δείκτη ή αναφορά

Παράδειγμα – Πέρασμα δια τιμής

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  .....
  HTMLFormattedPrint(pet);
  HTMLFormattedPrint(jonh);

  cout << pet.getPersonalDetails() <<
       " - Credit : " << pet.getCredit() << endl;
  return 0;
}
```

Το αντικείμενο περνάει δια τιμής – οι αλλαγές δεν είναι εμφανείς στη main ()

Παράδειγμα – Πέρασμα δείκτη

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα –Πέρασμα δείκτη

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
};
```

Παράδειγμα – Πέρασμα δείκτη

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα – Πέρασμα δείκτη

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}
```

Παράδειγμα – Πέρασμα δείκτη

```
void HTMLFormattedPrint( Person *p ){
  cout << "<HTML>" << p->getPersonalDetails()
       << "</HTML>" << endl;
  p->setPersonalDetails("", "");
}
```

για να δούμε τη διαφορά περάσματος δια τιμής, περάσματος με δείκτη ή αναφορά

Παράδειγμα – Πέρασμα δείκτη

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  ..........
  HTMLFormattedPrint(&pet);
  HTMLFormattedPrint(&jonh);

  cout << pet.getPersonalDetails() <<
       " - Credit : " << pet.getCredit() << endl;
  return 0;
}
```

Το αντικείμενο περνάει δι'αναφοράς– οι αλλαγές είναι εμφανείς στη main ()

Παράδειγμα

- λόγω της κληρονομικότητας, μια μόνο επιπλέον συνάρτηση μας αρκεί για να υλοποιήσουμε την επιπλέον δυνατότητα τόσο για τους πελάτες όσο και για τους υπάλληλους του καταστήματος….

Παράδειγμα

- παρόλα αυτά το πρόγραμμά μας μπορεί να μετατρέψει σε html μόνο τις πληροφορίες που περιλαμβάνονται σαν χαρακτηριστικά στη βασική κλάση Person …. ΓΙΑΤΙ ??
  - τι γίνεται αν θέλουμε να μετατρέψουμε σε html και τα επιμέρους χαρακτηριστικά που περιλαμβάνουν οι κλάσεις Employee και Customer ?

Κληρονομικότητα και Πολυμορφισμός

Παράδειγμα

```
using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}

void Customer::chargeCredit(int amount){
  credit -= amount;
}

int Customer::getCredit(){
  return credit;
}

char *Customer::getCreditType(){
  return creditType;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Employee *p){
  cout << "<HTML> " << p->getPersonalDetails() << " " <<
              p->getSalary() << " </HTML>" << endl;
}
```

Person

Employee

```
void HTMLFormattedPrint (Customer *p){
  cout << "<HTML> " << p->getPersonalDetails() << " " <<
       p->getCreditType() << " " << p->getCredit()  <<
       " </HTML>" << endl;
}
```

Person

Customer

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  pet.chargeCredit(250);

  HTMLFormattedPrint(&pet);
  HTMLFormattedPrint(&jonh);

  return 0;
}
```

Παράδειγμα

- **η σχεδίαση και η υλοποίηση μας πλέον δεν είναι καλή**
  - ο κώδικας που μετατρέπει τις πληροφορίες που χειρίζεται το πρόγραμμα σε html δεν είναι πλέον επαναχρησιμοποιήσιμος ….
  - χρειαζόμαστε μια συνάρτηση `HTMLFormattedPrint()` για κάθε κλάση του προγράμματός μας…
  - αν στο μέλλον προσθέσουμε και άλλες κλάσεις που παράγονται από την κλάση Person θα χρειαστούμε αντίστοιχες συναρτήσεις για την μετατροπή των στοιχείων σε html ….
  - … συνεπώς η συντήρηση του προγράμματος γίνεται προβληματική ….

Παράδειγμα

- Πώς αλλιώς θα μπορούσαμε να διαχειριστούμε το πρόβλημα μας ? ….
  - **στόχος μας είναι ο κώδικας που μετατρέπει τις πληροφορίες που χειρίζεται το πρόγραμμα σε html να είναι επαναχρησιμοποιήσιμος** ….

Πολυμορφισμός

- να χρησιμοποιήσουμε ένα είδος πολυμορφισμού ??
  - **τι είναι πολυμορφισμός ?**
    - **method overloading (υπερφόρτωση μεθόδων)**
      - **σε μια κλάση ορίζονται 2 ή περισσότερες μέθοδοι με το ίδιο όνομα και διαφορετικές παραμέτρους – διαφορετικό πρωτότυπο**
      - (το είδαμε στην περίπτωση πολλαπλών constructors)
    - **method overriding (επαναορισμός μεθόδων)**
      - **η μέθοδος μιας βασική κλάσης ορίζεται ξανά στην παραγόμενη κλάση με νέα υλοποίηση και το ίδιο πρωτότυπο**
      - **με αυτή τη τεχνική μπορούμε να πετύχουμε τον στόχο !!**
    - **static polymorfism…**

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getPersonalDetails(){
   // σε αυτό το παράδειγμα καλείται και η μέθοδος που επαναορίσαμε
   // για να μην εχουμε αναδρομή καθορίζουμε ότι καλούμε τη μέθοδο
   // που κληρονομούμε από την Person
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + 10];
  sprintf(ret, "%s %d", name, basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]): Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getPersonalDetails(){
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + strlen(creditType) + 10];
  sprintf(ret, "%s %s %d", name, creditType, credit);

  return ret;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p){
  cout << "<HTML>" << p->getPersonalDetails() << "</HTML>" <<
   endl;
}

int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");
  pet.chargeCredit(250);

  HTMLFormattedPrint(&pet);
  HTMLFormattedPrint(&jonh);
  return 0;
}
```

η συνάρτηση έχει παράμετρο Person *p
μέσω του δείκτη καλείται η **getPersonalDetails**
στο αντικείμενο στο οποίο δείχνει ο p

→θα θέλαμε να κληθεί η επαναορισμένη μέθοδος
→δηλαδή η getPersonalDetails της Customer ή της Employee ανάλογα με το αντικείμενο στο οποίο δείχνει ο p

Παράδειγμα

- **ΠΡΟΒΛΗΜΑ!!!**
  - **όταν εκτελούμε το πρόγραμμά μας ο κώδικας της συνάρτησης `HTMLFormattedPrint` καλεί μέσω του δείκτη `p` την `getPersonalDetails` της κλάσης Person και όχι την επαναορισμένη `getPersonalDetails` της κλάσης του αντικειμένου (`Customer` ή `Employee`) στο οποίο δείχνει ο δείκτης `p`..**

Παράδειγμα

- με άλλα λόγια στη C++ αν έχω
  - μια βασική κλάση Α και παραγόμενες κλάσεις Β, Γ και οι παραγόμενες κλάσεις επαναορίζουν μια μέθοδο f() της Α
  - ένα κώδικα (μια συνάρτηση, μια μέθοδος, κλπ.) στον οποίο
    - χρησιμοποιείται ένας δείκτης p βασικής κλάσης Α που μπορεί να δείχνει σε αντικείμενα των κλάσεων Α, Β, Γ
    - μέσω του δείκτη καλείται η μέθοδος f() στο αντικείμενο στο οποίο δείχνει ο δείκτης p
    - το ποια υλοποίηση της μεθόδου f() θα καλεστεί **καθορίζεται από τον τύπο του δείκτη**
      - πρακτικά, αφού εδώ ο δείκτης είναι της κλάσης Α, καλείται η υλοποίηση της μέθοδου f() της βασικής κλάσης Α
      - η επιλογή γίνεται από τον compiler κατά τη διάρκεια της μετάφρασης του προγράμματος….

Παράδειγμα

- αν εμείς θέλουμε η επιλογή της μεθόδου f() που θα καλεστεί να **καθορίζεται από τον τύπο του αντικειμένου στο οποίο δείχνει ο δείκτης p** (δηλ καλείται η υλοποίηση της μέθοδου f() των παραγόμενων κλάσεων Β, Γ, ..) ΠΡΕΠΕΙ η μέθοδος f() να δηλωθεί στην κλάση Α ως **virtual** μέθοδος…
  - στην ουσία στην περίπτωση των virtual μεθόδων δίνεται η οδηγία στον compiler ότι η επιλογή της υλοποίησης της f() δεν θα γίνει κατά τη μετάφραση του προγράμματος, με βάση τον τύπο του δείκτη, αλλα κατά την εκτέλεση του προγράμματος με βάση τον τύπου του αντικειμένου στο οποίο δείχνει ο δείκτης κάθε φορά που εκτελείται ο κώδικας….
    - **late binding**

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  virtual char *getPersonalDetails();
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getPersonalDetails();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[], int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getPersonalDetails(){
   // σε αυτό το παράδειγμα καλείται και η μέθοδος που επαναορίσαμε
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + 10];
  sprintf(ret, "%s %d", name, basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]): Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getPersonalDetails(){
  char *name = Person::getPersonalDetails();
  char * ret = new char [strlen(name) + strlen(creditType) + 10];
  sprintf(ret, "%s %s %d", name, creditType, credit);

  return ret;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p){
  cout << "<HTML>" << p->getPersonalDetails() << "</HTML>" <<
   endl;
}

int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");
  pet.chargeCredit(250);

  HTMLFormattedPrint(&pet);
  HTMLFormattedPrint(&jonh);
  return 0;
}
```

Πολυμορφισμός και Αφηρημένες Κλάσεις

Παράδειγμα

- Έστω ότι στο πρόγραμμά μας θέλουμε να προσθέσουμε μια μέθοδο που μετατρέπει τις πληροφορίες που διαχειρίζεται για τους πελάτες και τους υπαλλήλους σε html σελίδες και να τις εκτυπώνει ….
  - με βάση την τιμή μιας παραμέτρου, θέλουμε η μέθοδος να μετατρέπει σε html
    - μόνο τα βασικά χαρακτηριστικά της κλάσης Person
    - μόνο τα οικονομικά επιμέρους χαρακτηριστικά που περιλαμβάνουν οι κλάσεις Employee και Customer

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  virtual char* getFinancialDetails();
  // virtual method, μέθοδος χωρίς υλοποίηση
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getFinancialDetails();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getFinancialDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}

char* getFinancialDetails() {
   return NULL;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[],
                   int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getFinancialDetails(){
  char * ret = new char [30];
  sprintf(ret, " Basic Salary %d ", basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getFinancialDetails(){
  char * ret = new char [strlen(creditType) + 40];
  sprintf(ret, " Credit Type %s Credit Amount %d ", creditType, credit);
  return ret;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p, int choice){
  if(choice == 1)
    cout << "<HTML>" << p->getPersonalDetails() <<
              "</HTML>" << endl;
  else
    if(choice == 2)
      cout << "<HTML>" << p->getFinancialDetails() <<
                  "</HTML>" << endl;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  pet.chargeCredit(250);

  HTMLFormattedPrint (&pet, 1);
  HTMLFormattedPrint (&jonh, 1);

  HTMLFormattedPrint (&pet, 2);
  HTMLFormattedPrint (&jonh, 2);

  return 0;
}
```

Παράδειγμα

- στην ουσία κατασκευάσαμε μια βασική κλάση που περιλαμβάνει μια τετριμμένη μέθοδο μόνο και μόνο για να μπορούμε να την κάνουμε override στις παραγόμενες κλάσεις και να την καλούμε στην μέθοδο που μετατρέπει τα χαρακτηριστικά των αντικειμένων σε html….
  - η τετριμμένη μέθοδος όμως επιστρέφει NULL και είναι αρκετά επικίνδυνη …
  - κανείς δεν μπορεί να εγγυηθεί ότι ένας προγραμματιστής που θα επεκτείνει την κλάση Person θα κάνει override την τετριμμένη μέθοδο
    - αν δεν γίνει override η μέθοδος η NULL τιμή που επιστρέφει μπορεί να δημιουργήσει προβλήματα στον υπόλοιπο κώδικα ….

Αφηρημένες βασικές κλάσεις

- για να εξασφαλίσουμε ότι όσοι χρησιμοποιούν την κλάση Person θα κάνουν override την τετριμμένη μέθοδο → χρήση αφηρημένης βασικής κλάσης ….
  - μια κλάση ονομάζεται αφηρημένη αν περιέχει τουλάχιστον μια αφηρημένη μέθοδο που δεν περιλαμβάνει υλοποίηση
    - δηλώνεται ως virtual methodName() = **0**;
  - η αφηρημένη κλάση λειτουργεί σαν καλούπι για την κατασκευή παραγόμενων που προσφέρουν εναλλακτικές υλοποιήσεις στις αφηρημένες μεθόδους….
  - η δημιουργία αντικειμένων αφηρημένης κλάσης δεν επιτρέπεται από τον compiler
  - η μη υλοποίηση αφηρημένων μεθόδων δεν επιτρέπεται από τον compiler

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];
public:
  Person(char fn[], char ln[]);
  char *getPersonalDetails();
  virtual char* getFinancialDetails() = 0;
  // pure virtual method, μέθοδος χωρίς υλοποίηση
  void setPersonalDetails(char fn[], char ln[]);
};
```

Παράδειγμα

```
class Employee : public Person {
private:
  int basicSalary;

public:
  Employee(char fn[], char ln[], int sal);
  int getSalary();
  char *getFinancialDetails();
};

class Customer : public Person {
private:
  int credit;
  char creditType[10];
public:
  Customer(char fn[], char ln[], char ct[]);
  void chargeCredit(int amount);
  int getCredit();
  char *getCreditType();
  char *getFinancialDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::setPersonalDetails(char fn[], char ln[]){
  strcpy(fname, fn);
  strcpy(lname, ln);
}

char *Person::getPersonalDetails(){
  char *ret = new char [strlen(fname) + strlen(lname) + 40];
  sprintf(ret, "First Name %s - Last Name %s", fname, lname);

  return ret;
}
```

Παράδειγμα

```
Employee::Employee(char fn[], char ln[],
                     int sal) : Person(fn, ln) {
  basicSalary = sal;
}

int Employee::getSalary(){
  return basicSalary;
}

char *Employee::getFinancialDetails(){
  char * ret = new char [30];
  sprintf(ret, " Basic Salary %d ", basicSalary);

  return ret;
}
```

Παράδειγμα

```
Customer::Customer(char fn[], char ln[], char ct[]) : Person(fn, ln) {
  credit = 0;
  strcpy(creditType, ct);
}
void Customer::chargeCredit(int amount){
  credit -= amount;
}
int Customer::getCredit(){
  return credit;
}
char *Customer::getCreditType(){
  return creditType;
}

char *Customer::getFinancialDetails(){
  char * ret = new char [strlen(creditType) + 40];
  sprintf(ret, " Credit Type %s Credit Amount %d ", creditType, credit);
  return ret;
}
```

Παράδειγμα

```
void HTMLFormattedPrint (Person *p, int choice){
  if(choice == 1)
    cout << "<HTML>" << p->getPersonalDetails() <<
              "</HTML>" << endl;
  else
    if(choice == 2)
      cout << "<HTML>" << p->getFinancialDetails() <<
                    "</HTML>" << endl;
}
```

Παράδειγμα

```
int main(){
  char fname[40];
  char lname[40];
  int sal;
  int credit;

  cin >> fname >> lname >> sal;
  Employee jonh(fname, lname, sal);
  cin >> fname >> lname;
  Customer pet(fname, lname, "VISA");

  pet.chargeCredit(250);

  HTMLFormattedPrint (&pet, 1);
  HTMLFormattedPrint (&jonh, 1);

  HTMLFormattedPrint (&pet, 2);
  HTMLFormattedPrint (&jonh, 2);

  return 0;
}
```

Άλλα θέματα

Πολλαπλή Κληρονομικότητα - Αμφισημία

```
class Base1{
  public:
     void f();
};
class Base2{
  public:
     void f();
};
```

```
class Derived: public
      Base1, public Base2{
      ...
}
```

```
main(){
      Derived d;
      d.Base1::f();
      d.Base2::f();
}
```

Πολλαπλή Κληρονομικότητα - Virtual Inheritance

```
class Base {
  int i;
  char *name;
public:
  Base(){
    cout << "alala2\n";
  }
  Base (int ii) {
    cout << "alala\n";
    i = ii;
    name = new char [1000];
  }
  ~Base () {}
  void print() {
    cout << i << endl;
  }
};
```

```
class Child1 : public virtual Base {
public:
  Child1 (int ii) : Base(ii) {}
  ~Child1 () {}
};


class Child2 : public virtual Base {

public:
  Child2 (int ii) : Base(ii) {}
  ~Child2 () {}
};
```

Πολλαπλή Κληρονομικότητα - Virtual Inheritance

```
class GrandChild : public  Child1,  public Child2 {
public:
  GrandChild(int ii) : Child1(ii), Child2(ii) {}
  ~GrandChild(){}
};

main() {
  GrandChild gch(7);
}


// alala2
// alala
```

Κληρονομικότητα & Πολυμορφισμός
(Inheritance & Polymorphism)

Θεματολόγιο

- Κληρονομικότητα & Πολυμορφισμός
  - Παράδειγμα
  - Βελτιστοποιημένο Παράδειγμα – is-a κληρονομικότητα και πολυμορφισμός
  - Αφηρημένες κλάσεις
  - Άλλα θέματα

Παράδειγμα

- Έστω ένα απλό πρόγραμμα που διαχειρίζεται πληροφορίες για ένα σύνολο προσώπων
- Κάθε πρόσωπο χαρακτηρίζεται από
  - όνομα
  - επίθετο
- Για κάθε πρόσωπο υπάρχει η δυνατότητα αποθήκευσης των στοιχείων του σε ένα αρχείο με 2 εναλλακτικούς τρόπους
  - σε HTML μορφή
  - σε κωδικοποιημένη μορφή
    - κάθε χαρακτήρας του ονοματεπώνυμού του αντικαθίσταται με τον επόμενό του στο αλφάβητο

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];

  FILE *fp;
public:
  Person(char fn[], char ln[], char file[]);
  void logPersonalDetails (int choice);
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], char file[]){
  fp = fopen(file, "a");
  if(fp == NULL) exit(1);
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::logPersonalDetails(int choice){
  char *msg = new char[strlen(fname)+strlen(lname)+20];
  sprintf(msg, "First Name %s - Last Name %s", fname, lname);
  char *buf = new char[strlen(msg)+30];

  if(choice == 1)
    sprintf(buf, "<HTML> %s </HTML>\n", msg);
  else
    if(choice == 2)
      for (int i = 0; i < strlen(msg); i++)
        buf[i] = msg[i]+1;
  fprintf(fp, "%s\n", buf);
}
```

Παράδειγμα

```
int main(){
  Person x("jim", "morison", "morison.html");

  Person y("george", "bush", "encrypted.txt");

  x.logPersonalDetails(1);

  y.logPersonalDetails(2);
}
```

Παράδειγμα

- Έστω ότι σε ένα πιθανό σενάριο συντήρησης θέλουμε να προσθέσουμε μια επιπλέον εναλλακτική αποθήκευσης των στοιχείων των αντικειμένων Person σε αρχείο….
- Μια πρόχειρη λύση είναι να αλλαχθεί ο κώδικας της κλάσης Person ???

Παράδειγμα

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

using namespace std;

class Person {
private:
  char fname[40];
  char lname[40];

  FILE *fp;
public:
  Person(char fn[], char ln[], char file[]);
  void logPersonalDetails (int choice);
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], char file[]){
  fp = fopen(file, "a");
  if(fp == NULL) exit(1);
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::logPersonalDetails(int choice){
  char *msg = new char[strlen(fname)+strlen(lname)+20];
  sprintf(msg, "First Name %s - Last Name %s", fname, lname);
  char *buf = new char[strlen(msg)+30];

  if(choice == 1)              προσθήκη επιπλέον εναλλακτικών !!
    sprintf(buf, "<HTML> %s </HTML>\n", msg);
  else
    if(choice == 2)
      for (int i = 0; i < strlen(msg); i++)
        buf[i] = msg[i]+1;
  fprintf(fp, "%s\n", buf);
}
```

Παράδειγμα

- Μια πρόχειρη λύση είναι να αλλαχθεί ο κώδικας της κλάσης Person ….???
- Τέτοιου είδους στρατηγικές γενικά πρέπει να αποφεύγονται διότι το κόστος αλλαγής υπάρχοντος κώδικα είναι μεγάλο
  - κάποιος πρέπει να κατανοήσει τι κάνει ο υπάρχον κώδικας
  - να κάνει τις αλλαγές
  - να κάνει εξαρχής έλεγχο αν δουλεύει σωστά ο υπόλοιπος κώδικας

Κληρονομικότητα & Πολυμορφισμός

- Γενικά είναι επιθυμητό είναι να έχουμε τη δυνατότητα να προσθέτουμε νέες δυνατότητες σε ένα πρόγραμμα χωρίς να χρειαστεί να αλλάξουμε δραστικά τον υπάρχοντα κώδικα …
- Πως γίνεται αυτό ??
  - συνδυάζουμε κληρονομικότητα & πολυμορφισμό
  - τι είναι πολυμορφισμός ?
    - method overloading
      - σε μια κλάση ορίζονται 2 ή περισσότερες μέθοδοι με το ίδιο όνομα και διαφορετικές παραμέτρους
      - (το είδαμε στην περίπτωση πολλαπλών constructors)
    - **method overriding**
      - **η μέθοδος μιας βασική κλάσης ορίζεται ξανά στην παραγόμενη κλάση με νέα υλοποίηση και το ίδιο πρωτότυπο**
      - **(αυτό το είδος θα χρησιμοποιήσουμε για το πρόβλημά μας)**
    - static polymorfism…

Κληρονομικότητα & Πολυμορφισμός

- Πώς γίνεται αυτό ??
  1. για κάθε αρμοδιότητα του προγράμματος για την οποία υπάρχει δυνατότητα διαφορετικών εναλλακτικών υλοποιήσεων (πχ διαφορετικές μορφές αποθήκευσης των δεδομένων σε ένα αρχείο) και κατά συνέπεια η δυνατότητα μελλοντικών επεκτάσεων στο πρόγραμμά μας με μια νέα επιπλέον υλοποίηση ορίζουμε μια βασική κλάση
  2. εν συνεχεία για κάθε διαφορετική εναλλακτική κατασκευάζουμε μια παραγόμενη κλάση η οποία ξανα-ορίζει (overrides) τις public μεθόδους της βασικής κλάσης
  3. υλοποιούμε / παραμετροποιούμε τον υπόλοιπο κώδικα χρησιμοποιώντας δείκτες (ή αναφορές) της βασικής κλάσης
     - οι δείκτες αυτοί μπορούν να δείχνουν σε αντικείμενα οποιασδήποτε κλάσης προκύπτει από τη βασική
     - επομένως τα μόνα σημεία τα οποία πρέπει να αλλαχθούν σε μια μελλοντική επέκταση είναι τα σημεία στο οποία αρχικοποιούνται οι δείκτες αυτοί
     - ο υπόλοιπος κώδικας στον οποίο καλούνται μέθοδοι σε αντικείμενα στα οποία δείχνουν οι δείκτες δεν χρειάζεται αλλαγές

Κληρονομικότητα & Πολυμορφισμός

- Πώς γίνεται αυτό ??
  1. για κάθε αρμοδιότητα του προγράμματος για την οποία υπάρχει δυνατότητα διαφορετικών εναλλακτικών υλοποιήσεων (πχ διαφορετικές μορφές αποθήκευσης των δεδομένων σε ένα αρχείο) και κατά συνέπεια η δυνατότητα μελλοντικών επεκτάσεων στο πρόγραμμά μας με μια νέα επιπλέον υλοποίηση ορίζουμε μια βασική κλάση

Παράδειγμα

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

using namespace std;

class Log {
private:
  FILE *fp;
public:
  Log(char fname[]);
  void logMessage(char msg[]);
};

Log::Log(char fname[]){
  fp = fopen(fname, "a");
  if(fp == NULL) exit(1);
}

void Log::logMessage(char msg[]){
  fprintf(fp, "%s\n", msg);
}
```

Κληρονομικότητα & Πολυμορφισμός

- Πώς γίνεται αυτό ??
  2. εν συνεχεία για κάθε διαφορετική εναλλακτική κατασκευάζουμε μια παραγόμενη κλάση η οποία ξανα-ορίζει (overrides) κάποιες από τις public μεθόδους της βασικής κλάσης
     - συνήθως η κάθε παραγόμενη δεν προσθέτει νέες public μεθόδους απλά ξανα-ορίζει κάποιες από τις public μεθόδους της βασικής κλάσης

Παράδειγμα

```
class HTMLLog : public Log {
private:
// Δεν περιλαμβάνει κανένα νέο πεδίο !!
public:
  HTMLLog(char fname[]);
  void logMessage(char msg[]);
// ξαναορίζει τη μέθοδο της βασικής κλάσης
};


HTMLLog::HTMLLog(char fname[]) : Log(fname){}

void HTMLLog::logMessage(char msg[]){
  char *buf = new char[strlen(msg)+30];
  sprintf(buf, "<HTML> %s </HTML>\n", msg);
  Log::logMessage(buf);
// σε πολλές περιπτώσεις μέρος της υλοποίησης
// της επαναορισμένης μεθόδου καλεί την αντίστοιχη μέθοδο της βασικής
}
```

Παράδειγμα

```
class EncryptLog : public Log {
private:

public:
  EncryptLog(char fname[]);
  void logMessage(char msg[]);
};


EncryptLog::EncryptLog(char fname[]) : Log(fname){}

void EncryptLog::logMessage(char msg[]){
  char *buf = new char[strlen(msg)+1];
  for (int i = 0; i < strlen(msg); i++)
    buf[i] = msg[i]+1;
  Log::logMessage(buf);
}
```

Κληρονομικότητα & Πολυμορφισμός

- Πώς γίνεται αυτό ??
  3. υλοποιούμε / παραμετροποιούμε τον υπόλοιπο κώδικα χρησιμοποιώντας δείκτες (ή αναφορές) της βασικής κλάσης
     - οι δείκτες αυτοί μπορούν να δείχνουν σε αντικείμενα οποιασδήποτε κλάσης προκύπτει από τη βασική
     - επομένως τα μόνα σημεία τα οποία πρέπει να αλλαχθούν σε μια μελλοντική επέκταση είναι τα σημεία στο οποία αρχικοποιούνται οι δείκτες αυτοί
     - ο υπόλοιπος κώδικας στον οποίο καλούνται μέθοδοι σε αντικείμενα στα οποία δείχνουν οι δείκτες δεν χρειάζεται αλλαγές

Παράδειγμα

```
// υπόλοιπος κώδικας βασισμένος σε δείκτες στη βασική

class Person {
private:
  char fname[40];
  char lname[40];

// υπόλοιπος κώδικας βασισμένος σε δείκτες στη βασική
  Log *log;

public:
// υπόλοιπος κώδικας παραμετροποιημένος
// με δείκτες στη βασική
  Person(char fn[], char ln[], Log *l);
  void logPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], Log *l){
  log = l;
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::logPersonalDetails(){
  char *msg = new char[strlen(fname)+strlen(lname)+20];
  sprintf(msg, "First Name %s - Last Name %s", fname, lname);

  log->logMessage(msg);
}
```

Παράδειγμα

```
int main(){
  Log *l1 = new HTMLLog("morison.html");
  Log *l2 = new EncryptLog("encrypted.txt");

  Person x("jim", "morison", l1);

  Person y("george", "bush", l2);

  Person z("vlad", "poutin", l2);

  x.logPersonalDetails();

  y.logPersonalDetails();

  z.logPersonalDetails();
}
```

Για να προσθέσω μια νέα εναλλακτική υλοποιώ απλώς μια νέα κλάση που κληρονομεί από τη Log και αλλάζω μόνο τη main()

δεν πειράζω καθόλου την κλάση Person !!!

Κληρονομικότητα & Πολυμορφισμός

- Δουλεύουν όμως όλα σωστά ???
  - στα δύο αρχεία (morison.html και encrypted.txt) παρατηρούμε ότι τα στοιχεία κάθε αντικειμένου **δεν είναι σε html μορφή ή κωδικοποιημένα ☹**
  - **ΓΙΑΤΙ ??**

Παράδειγμα

```
// υπόλοιπος κώδικας βασισμένος σε δείκτες στη βασική

class Person {
private:
  char fname[40];
  char lname[40];

// υπόλοιπος κώδικας βασισμένος σε δείκτες στη βασική
  Log *log;

public:
// υπόλοιπος κώδικας παραμετροποιημένος
// με δείκτες στη βασική
  Person(char fn[], char ln[], Log *l);
  void logPersonalDetails();
};
```

Παράδειγμα

```
Person::Person(char fn[], char ln[], Log *l){
  log = l;
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::logPersonalDetails(){
  char *msg = new char[strlen(fname)+strlen(lname)+20];
  sprintf(msg, "First Name %s - Last Name %s", fname, lname);

  log->logMessage(msg);

// Για το x αντικείμενο ο log δείχνει σε ένα αντικείμενο κλάσης HTMLLog
// ποια logMessage() μέθοδος θα εκτελεστεί ??
// της βασικής κλάσης η οποία γράφει το msg στο αρχείο χωρίς μετατροπές
// ή της κλάσης στην οποία δείχνει η δείκτης ??
}
```

Κληρονομικότητα & Πολυμορφισμός

- Σύμφωνα με αυτά που είπαμε στην προηγούμενη ενότητα
  - αν ένας δείκτης X *p δείχνει σε ένα αντικείμενο τότε:
    - το ποιες μέθοδοι μπορούν να εκτελεστούν χρησιμοποιώντας το δείκτη καθορίζεται από τον τύπο του δείκτη
    - μπορούν να εκτελεστούν οι μέθοδοι της κλάσης X
    - ισχύει και στην περίπτωση μεθόδων της βασικής X που έχουν ξαναοριστεί στην παραγόμενη Y
      - αν το p δείχνει σε αντικείμενο της παραγόμενης κλάσης Y και f() μια επαναορισμένη μέθοδος
      - p -> f(); έχει ως συνέπεια την εκτέλεση της μεθόδου της βασικής κλάσης X

Κληρονομικότητα & Πολυμορφισμός

- Οπότε ???
  - οπότε για να «ξεφύγουμε» από τον προηγούμενο κανόνα πρέπει να δηλώσουμε τις μεθόδους της βασικής κλάσης ως **virtual**
    - σε αυτή την περίπτωση αν ένας δείκτης X *p δείχνει σε ένα αντικείμενο τότε:
      - το ποιες μέθοδοι θα εκτελεστούν χρησιμοποιώντας το δείκτη καθορίζεται πλέον από τον τύπο του αντικειμένου στο οποίο δείχνει ο δείκτης
        - **αν το p δείχνει σε αντικείμενο της παραγόμενης κλάσης Y και f() μια επαναορισμένη μέθοδος**
        - p -> f(); έχει ως συνέπεια την εκτέλεση της μεθόδου της παραγόμενης κλάσης Y
        - **ΠΡΟΣΟΧΗ :** αν **η f() ΔΕΝ είναι επαναορισμένη μέθοδος**, αλλά **απλά μέθοδος της κλάσης Y** τότε **ΔΕΝ μπορεί να κληθεί** (compile error) χρησιμοποιώντας το δείκτη p …

Κληρονομικότητα & Πολυμορφισμός

- Γενικά δηλώνοντας μια μέθοδο f() virtual, «λέμε» στον compiler ότι η επιλογή σχετικά με το ποια μέθοδος θα καλεστεί μέσω του δείκτη p (η ίδια η f() ή κάποια ξαναορισμένη f()) θα γίνει
  - κατά τη διάρκεια εκτέλεσης του προγράμματος, με βάση τον τύπο του αντικειμένου στο οποίο δείχνει ο p,
    - μηχανισμός **late binding**
  - και όχι κατά τη μετάφραση του προγράμματος, με βάση τον τύπο του δείκτη p (είναι η μόνη σίγουρη πληροφορία που έχει ο compiler κατά τη μετάφραση του προγράμματος)

Παράδειγμα

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

using namespace std;

class Log {
private:
  FILE *fp;
public:
  Log(char fname[]);
  virtual void logMessage(char msg[]);
};

Log::Log(char fname[]){
  fp = fopen(fname, "a");
  if(fp == NULL) exit(1);
}

void Log::logMessage(char msg[]){
  fprintf(fp, "%s\n", msg);
}
```

Παράδειγμα

```
class HTMLLog : public Log {
private:
// Δεν περιλαμβάνει κανένα νέο πεδίο !!
public:
  HTMLLog(char fname[]);
  void logMessage(char msg[]);
// ξαναορίζει τη μέθοδο της βασικής κλάσης
};


HTMLLog::HTMLLog(char fname[]) : Log(fname){}

void HTMLLog::logMessage(char msg[]){
  char *buf = new char[strlen(msg)+30];
  sprintf(buf, "<HTML> %s </HTML>\n", msg);
  Log::logMessage(buf);
// σε πολλές περιπτώσεις μέρος της υλοποίησης
// της επαναορισμένης μεθόδου καλεί την αντίστοιχη μέθοδο της βασικής
}
```

Παράδειγμα

```
class EncryptLog : public Log {
private:

public:
  EncryptLog(char fname[]);
  void logMessage(char msg[]);
};


EncryptLog::EncryptLog(char fname[]) : Log(fname){}

void EncryptLog::logMessage(char msg[]){
  char *buf = new char[strlen(msg)+1];
  for (int i = 0; i < strlen(msg); i++)
    buf[i] = msg[i]+1;
  Log::logMessage(buf);
}
```

Αφηρημένες Κλάσεις (Abstract Classes)

Κληρονομικότητα & Πολυμορφισμός

- Γενικά είναι επιθυμητό είναι να έχουμε τη δυνατότητα να προσθέτουμε νέες δυνατότητες σε ένα πρόγραμμα χωρίς να χρειαστεί να αλλάξουμε δραστικά τον υπάρχοντα κώδικα …

Κληρονομικότητα & Πολυμορφισμός

- Πώς γίνεται αυτό ??
  1. για κάθε αρμοδιότητα του προγράμματος για την οποία υπάρχει δυνατότητα διαφορετικών εναλλακτικών υλοποιήσεων και κατά συνέπεια η δυνατότητα μελλοντικών επεκτάσεων στο πρόγραμμά μας με μια νέα επιπλέον υλοποίηση ορίζουμε μια βασική κλάση
  2. εν συνεχεία για κάθε διαφορετική εναλλακτική κατασκευάζουμε μια παραγόμενη κλάση η οποία ξανα-ορίζει (overrides) τις public μεθόδους της βασικής κλάσης
  3. υλοποιούμε / παραμετροποιούμε τον υπόλοιπο κώδικα χρησιμοποιώντας δείκτες (ή αναφορές) της βασικής κλάσης
     - οι δείκτες αυτοί μπορούν να δείχνουν σε αντικείμενα οποιασδήποτε κλάσης προκύπτει από τη βασική
     - επομένως τα μόνα σημεία τα οποία πρέπει να αλλαχθούν σε μια μελλοντική επέκταση είναι τα σημεία στο οποία αρχικοποιούνται οι δείκτες αυτοί
     - ο υπόλοιπος κώδικας στον οποίο καλούνται μέθοδοι σε αντικείμενα στα οποία δείχνουν οι δείκτες δεν χρειάζεται αλλαγές

Αφηρημένες Κλάσεις

- Στο προηγούμενο παράδειγμα οι διαφορετικές εναλλακτικές υλοποιήσεις έχουν κοινά χαρακτηριστικά και κώδικα τα οποία τοποθετούνται στην υλοποίηση της βασικής κλάσης
  - στο προηγούμενο παράδειγμα δεν ίσχυε αυτό …

```
class Log {
private:
  FILE *fp;
public:
  Log(char fname[]);
  virtual void logMessage(char msg[]);
};

Log::Log(char fname[]){
  fp = fopen(fname, "a");
  if(fp == NULL) exit(1);
}

void Log::logMessage(char msg[]){
  fprintf(fp, "%s\n", msg);
}
```

Αφηρημένες Κλάσεις

- Γενικά είναι δυνατόν οι διαφορετικές εναλλακτικές υλοποιήσεις να μην έχουν κοινά χαρακτηριστικά ή κώδικα
  - στην περίπτωση αυτή η βασική κλάση μπορεί να μην έχει χαρακτηριστικά
  - επίσης μπορεί να ορίζονται μέθοδοι χωρίς υλοποίηση αποκλειστικά και μόνο για
    - να μπορούν ξαναοριστούν στις παραγόμενες κλάσεις και
    - να κληθούν μέσω δεικτών βασικής κλάσης που δείχνουν σε αντικείμενα κάποιας παραγόμενης κλάσης…..
  - αυτές οι μέθοδοι λέγονται **pure virtual μέθοδοι** και θα λέγαμε λειτουργούν σαν «καλούπι» για την κατασκευή παραγόμενων κλάσεων
  - μια κλάση που περιλαμβάνει **pure virtual μεθόδους** λέγεται **αφηρημένη κλάση**
    - γενικά δεν μπορούν να δημιουργηθούν αντικείμενα μιας αφηρημένης κλάσης

Παράδειγμα

- Έστω ένα απλό πρόγραμμα που διαχειρίζεται πληροφορίες για ένα σύνολο ατόμων
- Κάθε άτομο χαρακτηρίζεται από
  - όνομα
  - επίθετο
- Για κάθε άτομο υπάρχει η δυνατότητα καταγραφής των στοιχείων του
  - υπάρχουν δύο εναλλακτικές δυνατότητες καταγραφής
    - αποθήκευσης των στοιχείων του σε ένα αρχείο
    - αποθήκευση σε μια βάση δεδομένων

Παράδειγμα

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

using namespace std;

// Abstract class
class Log {
private:
public:
   // pure virtual method
   virtual void logMessage(char msg[]) = 0;
};
```

Παράδειγμα

```
class FileLog : public Log {
private:
  FILE *fp;
public:
  FileLog();
  // υλοποίηση της pure virtual μεθόδου
  void logMessage(char msg[]);
};

FileLog::FileLog(){ // default constructor for Log()
  fp = fopen("fileLog.txt", "a");
  if(fp == NULL) exit(1);

}

void FileLog::logMessage(char msg[]){
  fprintf(fp, "%s\n", msg);
}
```

Παράδειγμα

```
class DatabaseLog : public Log {
private:

public:
  DatabaseLog();
  void logMessage(char msg[]);
};


DatabaseLog::DatabaseLog(){
// κώδικας σύνδεσης με τη βάση δεδομένων
// ……
}

void DatabaseLog::logMessage(char msg[]){
  printf("%s", msg);
// κώδικας καταγραφής στοιχείων στη βάση δεδομένων
// ……
}
```

Παράδειγμα

```
class Person {
private:
  char fname[40];
  char lname[40];

  Log *log;

public:
  Person(char fn[], char ln[], Log *l);
  void logPersonalDetails();
};

Person::Person(char fn[], char ln[], Log *l){
  log = l;
  strcpy(fname, fn);
  strcpy(lname, ln);
}

void Person::logPersonalDetails(){
  char *msg = new char[strlen(fname)+strlen(lname)+20];
  sprintf(msg, "First Name %s - Last Name %s", fname, lname);

  log->logMessage(msg);
}
```

Παράδειγμα

```
int main(){
  Log *l1 = new DatabaseLog();
  Log *l2 = new FileLog();
  //Log *l3 = new Log();
  // Δεν επιτρέπεται η δημιουργία αντικειμένων αφηρημένης κλάσης

  Person x("jim", "morison", l1);

  Person y("george", "bush", l2);

  Person z("vlad", "poutin", l2);

  x.logPersonalDetails();

  y.logPersonalDetails();

  z.logPersonalDetails();
}
```

Άλλα Θέματα

Virtual Destructors

Virtual Destructors

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

class Base {
private:
public:
  Base();
  virtual ~Base();
  virtual void doSomething();
};

Base::Base(){
  cout << "Base() called \n";
}

Base::~Base(){
  cout << "~Base() called \n";
}

void Base::doSomething(){
  cout << "doing some BASE work\n";
}
```

Virtual Destructors

```
class Derived : public Base {
private:
  char *data;
public:
  Derived();
  ~Derived();
  void doSomething();
};
Derived::Derived() : Base(){
  data = new char[100];
  cout << "Derived() called \n";
}
Derived::~Derived(){
  delete [] data;
  cout << "~Derived() called \n";
}
void Derived::doSomething(){
  cout << "doing some MORE work\n";
  Base::doSomething();
}
```

Virtual Destructors

```
int main(){
  int choice = -1;
  Base *p;
  cin >> choice;
  if (choice == 1)
    p = new Derived();
  else
    p = new Base();

  p->doSomething();

  delete p;
// αν ο destructor δεν είναι virtual
// το πρόγραμμα αφήνει σκουπίδια στη μνήμη
}
```

Dynamic Casting

Dynamic Casting

- Έστω έχουμε ένα δείκτη μιας κλάσης από την οποία έχουν προκύψει με κληρονομικότητα άλλες κλάσεις.
  - π.χ. μας τον έχουν περάσει όρισμα σε μια συνάρτηση
- Έστω θέλω να μάθω τον τύπο του αντικειμένου στο οποίο δείχνει ο δείκτης και ανάλογα με τον τύπο αυτό να καλέσω συναρτήσεις αυτού του τύπου.
- Τι κάνω ?
  - μηχανισμός dynamic casting
  - εξετάζω αν μπορώ να μετατρέψω το δείκτη της βασικής κλάσης σε ένα δείκτη κάποιας παραγόμενης κλάσης
  - αν γίνεται αυτό καλώ τη συνάρτηση της παραγόμενης κλάσης πάνω στο δείκτη που πήρα από τη μετατροπή

Dynamic Casting

```
class C {
private:
   int fc;
public:
   C(){...}
   virtual ~C(){} // Για να παίξει το dynamic casting πρέπει
                  // η βασική κλάση να είναι πολυμορφική
   void print_c(){ cout << fc;}
};

class D : public C {
private:
   int fd;
public:
   D(){...}
   ~D(){}
   void print_d(){ cout << fd;}
};
```

Dynamic Casting

```
class B : public C {
private:
   int fb;
public:
   B(){...}
   ~B(){}
 void print_b(){cout << fb;}
};

void print_test(C *c) {
   B *testB; D *testD;

   c->print_c();
   testB = dynamic_cast<B*> (c);
   testD = dynamic_cast<D*> (c);
   if (testB != NULL) testB->print_b();
   if (testD != NULL) testB->print_d();
}
```

Dynamic Casting

```
main(){

   B *b = new B;
   D *d = new D;

   print_test(b);
   print_test(d);
}
```

Standard Template Library (featuring template classes) && Makefiles

STANDARD TEMPLATE LIBRARY

Συλλογή (υπερ)πολύτιμων προγραμματιστικών εργαλείων

- Η STL αποτελεί μια συλλογή από έτοιμα προγραμματιστικά εργαλεία, τα οποία βοηθούν τον προγραμματιστή να επιτελέσει μεγάλο κομμάτι της δουλειάς του χρησιμοποιώντας **έτοιμες πρότυπες δομές δεδομένων και αλγορίθμους**
- **Η επαναχρησιμοποίηση έτοιμου κώδικα** (πολλώ δε μάλλον αν αυτός προέρχεται εγγενώς από την ίδια τη γλώσσα => έχει χρησιμοποιηθεί, ελεγχθεί, αποσφαλματωθεί, βελτιστοποιηθεί πλειστάκις) **είναι θεμελιώδης στον αντικειμενοστρεφή προγραμματισμό**
- Αντίστοιχες συλλογές υπάρχουν σε όλες τις ώριμες αντικειμενοστρεφείς γλώσσες προγραμματισμού

STL

- Υπάρχουν τριών ειδών οντότητες που μας ενδιαφέρουν
  - Αποδέκτες (Containers)
    - Οι διαφορετικοί τρόποι για να αποθηκεύουμε τα δεδομένα μας.
  - Επαναλήπτες (Iterators)
    - Μας παρέχουν δείκτες στα στοιχεία του container και μας επιτρέπουν να διατρέχουμε τα στοιχεία του αποδέκτη.
  - Αλγόριθμοι (Algorithms)
    - Υλοποιημένοι αλγόριθμοι που μας δίνουν βασικές λειτουργίες πάνω στους αποδέκτες

STL Sequential Containers

Container	Χαρακτηριστικά	+/-
Πίνακας C++	Σταθερό μέγεθος	+ Γρήγορη τυχαία πρόσβαση. - Αργή προσθήκη σε ενδιάμεση θέση - Σταθερό μέγεθος
vector	Δυναμικός πίνακας	+ Γρήγορη τυχαία πρόσβαση. - Αργή προσθήκη σε ενδιάμεση θέση - Προσθήκη μόνο στο τέλος
list	Διπλά συνδεδεμένη λίστα	+ Γρήγορη προσθήκη σε ενδιάμεση θέση - Αργή τυχαία πρόσβαση.
deque	Δυναμικός πίνακας με πρόσβαση και από τις δύο μεριές	+ Γρήγορη τυχαία πρόσβαση. + Προσθήκη σε αρχή και τέλος - Αργή προσθήκη σε ενδιάμεση θέση

vector

Μέθοδος
`int size()`
`void push_back(const TYPE &)`
`void pop_back()`
`TYPE & back()`
`TYPE & operator [](int)`
`bool empty()`
`iterator insert(iterator, const TYPE &)`
`iterator erase(iterator)`

list

Μέθοδος
`int size()`
`void push_back(const TYPE &)`
`void pop_back()`
`TYPE & back()`
`bool empty()`
`iterator insert(iterator, const TYPE &)`
`iterator erase(iterator)`
`void push_front(const TYPE &)`
`void pop_front()`
`TYPE & front()`

Επίσης, μεθόδους: reverse, merge, unique

deque

- Έχει τις μεθόδους και των δύο παραπάνω.
- Και προσθήκη στην αρχή και το τέλος, και τυχαία πρόσβαση.

iterators

`list<int>::iterator iter:` Δήλωση του iterator

`L.begin():` Συνάρτηση που επιστρέφει iterator στην αρχή του container

`L.end():` Συνάρτηση που επιστρέφει iterator στο τέλος του container

`iter++:` Κάνει τον iterator να δείχνει στο επόμενο στοιχείο της λίστας

`*iter:` Το περιεχόμενο της θέσης στην οποία δείχνει ο iterator, μπορεί να χρησιμοποιηθεί και για να πάρει και για να δώσει τιμή.

```
#include <iostream>
#include <list>

using namespace std;

int main(){
  list<int> L;
  int x;

  do{
    cin >> x;
    L.push_back(x);
  }while (x != -1);

  cout << "list elements: ";
  for (list<int>::iterator iter = L.begin(); iter != L.end(); iter ++){
    *iter += 2;
    cout << *iter << " ";
  }
  cout << endl;
}
```

iterators

```
#include <iostream>
#include <list>

using namespace std;

int main(){
  list<int> L;
  int x;

  do{
    cin >> x;
    L.push_back(x);
  }while (x != -1);

  cout << "list elements in reverse: ";
  list<int>::reverse_iterator riter;
  for (riter = L.rbegin(); riter != L.rend(); iter ++){
    cout << *iter << " ";
  }
  cout << endl;
}
```

Εκτύπωση των στοιχείων σε αντίστροφη σειρά χρησιμοποιώντας έναν **reverse_iterator**

iterators

```
#include <iostream>
#include <list>

using namespace std;

int main(){
  list<int> L;
  int x;

  do{
    cin >> x;
    L.push_back(x);
  }while (x != -1);

  list<int>::iterator  iter;
  for (iter = L.begin(); iter != L.end(); iter ++){
    if (*iter == 8){ break;}
  }
  if (iter != L.end()) { // υπάρχει στοιχείο με τιμή 8
    L.erase(iter);
  }
}
```

Βρες και σβήσε το πρώτο στοιχείο της λίστας με την τιμή 8, εφόσον υπάρχει.

Associative Containers

Container	Χαρακτηριστικά
set	Αποθηκεύει μόνο αντικείμενα-κλειδιά Δεν επιτρέπει πολλαπλές τιμές
map	Κρατάει ζεύγη κλειδιών και τιμών (key-value pairs). Συσχετίζει αντικείμενα-κλειδιά με αντικείμενα-τιμές. Το κάθε κλειδί μπορεί να συσχετίζεται με μόνο μία τιμή
multiset	Επιτρέπει πολλαπλές εμφανίσεις ενός κλειδιού
multimap	Επιτρέπει πολλαπλές τιμές για ένα κλειδί.

Παράδειγμα set

```
#include <iostream>
#include <set>

using namespace std;

int main(){
  set<string> S;

  string name;
  while (!cin.eof()){
    cin >> name;
    S.insert(name);
  }

  set<string>::iterator iter = S.find("bob");

  if (iter == S.end()){ cout << "bob is not in\n";}
  else{cout << "bob is in\n";}

  for (set<string>::iterator iter =  S.begin(); iter != S.end(); iter ++){
    cout << *iter << " ";
  }
  cout << endl;
}
```

13

Set με δικές μας κλάσεις

- Τι γίνεται αν θέλουμε να βάλουμε αντικείμενα από μία δική μας κλάση σε ένα Set?
  - Τι σημαίνει ότι το set περιέχει ήδη ένα αντικείμενο?
- Για να χρησιμοποιήσουμε το Set, η κλάση θα πρέπει να έχει ορισμένους τον τελεστή == και τον τελεστή < .

14

Παράδειγμα Set

```
class Person {
private:
  int id;
  string fname;
  string lname;
public:
  Person();
  Person(int);
  void SetDetails(int id, string fn, string ln);
  bool operator == (const Person &) const;
  bool operator < (const Person &) const;
  int getId() const;
  void PrintDetails() const;
};
```

15

```
Person::Person() {}

Person::Person(int i)
{ id = i; }

void Person::SetDetails(int i, string f, string l){
  id = i;  fname = f;  lname = l;
}

bool Person::operator == (const Person &p) const
{ return (id == p.getId()); }

bool Person::operator < (const Person &p) const
{ return (id < p.getId()); }

int Person::getId() const
{ return id; }

void Person::PrintDetails() const
{
  cout << "id: " << id
       <<" first name:" << fname << " last name:" << lname << endl;
}
```



```
int main(){
  set<Person> S;
  Person P[3];

  int id;
  string fname, lname;
  for(int i =0 ; i< 3; i ++){
    cin >>id >> fname >> lname;
    P[i].SetDetails(id, fname, lname);
    S.insert(P[i]);
  }

  cin >> id;
  Person sp(id);
  set<Person>::iterator iter = S.find(sp);
  if (iter == S.end()){
    cout << "id not found\n";
  }else{
    iter->PrintDetails();
  }
}
```

Η αναζήτηση γίνεται με ένα αντικείμενο της κλάσης

Παράδειγμα map

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

int main(){
  map<int,Person*> M;
  int id;

  while (!cin.eof()){
    string fname,lname;
    cin >> id >> fname >> lname;
    Person *p =new Person; p->setDetails(id,fname,lname);
    M[id] = p;
  }

  cin >> id;
  map<int,Person*>::iterator iter = M.find(id);
  if (iter == M.end()){
    cout << "id is not in\n";
  }else{
    M[id]->PrintDetails();
  }
}
```

Το κλειδί στην περίπτωση αυτή είναι int, αλλά μπορεί να είναι οποιοσδήποτε τύπος δεδομένων που πληροί τις προϋποθέσεις για να είναι κλειδί σε Set.

Παράδειγμα map iterator

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

int main(){
  map<int,Person*> M;
  int id;

  while (!cin.eof()){
    string fname,lname;
    cin >> id >> fname >> lname;
    Person *p =new Person; p->setDetails(id,fname,lname);
    M[id] = p;
  }

  map<int,Person*>::iterator iter = M.find(id);
  for (iter = M.begin(); iter != M.end(); i++){
    cout << (*iter).first << ":";
    (*iter).second->PrintPersonalDetails();
  }
}
```


Αλγόριθμοι

- Στην STL υπάρχουν ήδη υλοποιημένοι αλγόριθμοι που μπορούν να εφαρμοστούν σε containers.
- Ο κάθε αλγόριθμος χρειάζεται κάποιον τύπο iterator, ο οποίος καθορίζει και το αν μπορούμε να τον εφαρμόσουμε σε ένα συγκεκριμένο τύπο container.

Αλγόριθμοι

- Οι αλγόριθμοι που υποστηρίζονται από την STL εξυπηρετούν βασικές λειτουργίες πάνω σε συλλογές αντικειμένων. Χονδρικά μπορούμε να πούμε τις εξής οικογένειες αλγορίθμων
  - Ανεύρεση (find, find_first_of, …)
  - Μέτρηση (count, accumulate, max_element, min_element, …)
  - Μετατροπή (copy, replace, remove, …)
  - Ταξινόμηση και ανεύρεση (sort, binary_search, …)

find

- Βρίσκει την πρώτη εμφάνιση μίας τιμής μέσα σε ένα container.
- Δουλεύει για όλους τους containers
- Συντακτικό:
  - `iterator find(start, finish, value)`
- Παραδειγμα:

```
include<algorithm>
vector<int> v
vector<int>::iterator i =
    find(v.begin(), v.end(), 100)
```

- Επιστρέφει iterator στην θέση του στοιχείου αν είναι μέσα στον container, ή `v.end()` αν δεν είναι.

22

sort

- Ταξινομεί τα στοιχειά του αποδέκτη.
- Χρειάζεται τυχαία πρόσβαση, και άρα δουλεύει μόνο για vectors και deque
- Συντακτικό:
  - `sort(start, finish)`
  - Ταξινομεί σε αύξουσα σειρά
- Παράδειγμα:

```
include <algorithm>
vector<int> v;
sort(v.begin(), v.end());
```

23

Αντικείμενα συναρτήσεων

- Τι γίνεται όμως αν θέλουμε να ταξινομήσουμε τα στοιχεία με κάποια άλλη σειρά?
- Τότε πρέπει να περάσουμε ως όρισμα στην sort ένα αντικείμενο-συνάρτησης.
  - Βασικά ένα αντικείμενο μιας template class στην οποία υπερφορτώνουμε τον τελεστή ().
- Η C++ έχει ήδη υλοποιήσει κάποια από αυτά τα αντικείμενα.

```
vector<int> v;
sort(v.begin(), v.end(),
greater<int>());
```

24

```
class ComparePersons
{
public:
  bool operator ()(const Person &p1, const Person &p2) const
  {
    return p1 < p2;
  }
};


int main(){
  vector<Person> V;
  Person P[3];

  for(int i =0 ; i< 3; i ++){
    int id;
    string fname, lname;
    cin >>id >> fname >> lname;
    P[i].SetDetails(id, fname, lname);
    V.push_back(P[i]);
  }

  sort(V.begin(),V.end(),ComparePersons());
  sort(P,P+3,ComparePersons());

  for (int i =0; i < 3; i ++){
    P[i].PrintDetails();
  }
}
```



```
bool compare(const Person &p1, const Person &p2)
{
  return p1 < p2;
}


int main(){
  vector<Person> V;
  Person P[3];

  for(int i =0 ; i< 3; i ++){
    int id;
    string fname, lname;
    cin >>id >> fname >> lname;
    P[i].SetDetails(id, fname, lname);
    V.push_back(P[i]);
  }

  sort(V.begin(),V.end(),compare);
  sort(P,P+3,compare);

  for (int i =0; i < 3; i ++){
    P[i].PrintDetails();
  }
}
```


for_each

- Καλεί μια συνάρτηση, στην οποία περνάμε ως όρισμα, για κάθε στοιχείο του container.
- Μπορεί να χρησιμοποιηθεί για όλους τους containers.
- Συντακτικό:
  - **`for_each(start, finish, function)`**
  - Η συνάρτηση **`function`** παίρνει όρισμα τον τύπο δεδομένων που περιέχει ο container, και εφαρμόζεται σε όλα τα στοιχεια μεταξύ **`start`** και **`finish`**.
- Παράδειγμα:

```
void inc(int *i){ cout << ++(*i) << endl;}
vector<int *> v;
for_each(v.begin(), v.end(), inc);
```

Αυξάνει όλες τις τιμές κατά ένα και τις τυπώνει

Φίλες συναρτήσεις

- Χρησιμοποιώντας το keyword friend, μπορούμε να κάνουμε μια συνάρτηση να βλέπει τα private μέλη μιας κλάσης.
  - Σε ορισμένες περιπτώσεις αυτό είναι βολικό άλλα δεν θα πρέπει να χρησιμοποιείται πολύ συχνά.

Παράδειγμα Set

```
class Person {
private:
  int id;
  string fname;
  string lname;
public:
  Person();
  Person(int);
  void SetDetails(int id, string fn, string ln);
  friend bool operator == (const Person&,const Person &);
  friend bool operator < (const Person &,const Person &);
  int getId() const;
  void PrintDetails() const;
};
```



```
Person::Person() {}

Person::Person(int i)
{ id = i; }

void Person::SetDetails(int i, string f, string l){
  id = i;  fname = f;  lname = l;
}

bool operator == (const Person &p1, const Person &p2)
{ return (p1.id == p2.id); }

bool operator < (const Person &p1, const Person &p2)
{ return (p1.id < p2.id); }

int Person::getId() const
{ return id;}

void Person::PrintDetails() const
{
  cout << "id: " << id
       <<" first name:" << fname << " last name:" << lname << endl;
}
```

```
int main(){
  set<Person> S;
  Person P[3];

  int id;
  string fname, lname;
  for(int i =0 ; i< 3; i ++){
    cin >>id >> fname >> lname;
    P[i].SetDetails(id, fname, lname);
    S.insert(P[i]);
  }

  cin >> id;
  Person sp(id);
  set<Person>::iterator iter = S.find(sp);
  if (iter == S.end()){
    cout << "id not found\n";
  }else{
    iter->PrintDetails();
  }
}
```

Ο κώδικας παραμένει ο ίδιος

31

Το υπόβαθρο για την STL

TEMPLATE CLASSES

32

Class Templates

- Με τα Class Templates μπορούμε να ορίσουμε μία κλάση ή οποία να δουλεύει για πολλούς διαφορετικούς τύπους δεδομένων στα πεδία της.
  - Χρησιμοποιούνται κυρίως για κλάσεις «αποδέκτες» οι οποίες αποθηκεύουν δεδομένα

33

Συντακτικό

```
template<class DATA_TYPE>
class SomeClass<DATA_TYPE>
{
private:
        ...
        DATA_TYPE someData;
        ...
public:
        ...
        DATA_TYPE SomeMethod(DATA_TYPE *);
        ...
};

template<class DATA_TYPE>
DATA_TYPE SomeClass<DATA_TYPE>::SomeMethod(DATA_TYPE *x)
{
        ...
        if (someData.methodX() == x.methodX())
        ...
}
```

Απλά ένα όνομα που μετά θα αντικατασταθεί από το όνομα του τύπου δεδομένων

Ο τύπος δεδομένων που θα αντικαταστήσει το **DATA_TYPE** θα πρέπει να υλοποιεί την **methodX**

34

Class Template: Array

```
template <class DATA_TYPE>
class Array
{
private:
  DATA_TYPE **A;
  int size;
public:
  Array(int s);
  DATA_TYPE *& operator [](int);
  void Sort();
  void Print();
};
```

35

```
template <class DATA_TYPE>
Array<DATA_TYPE>::Array(int s)
{
  size = s;
  A = new DATA_TYPE*[size];
}

template <class DATA_TYPE>
DATA_TYPE *& Array<DATA_TYPE>::operator [] (int i)
{
  return A[i];
}

template <class DATA_TYPE>
void Array<DATA_TYPE>::Sort()
{
  for (int i = 0; i < 10; i ++){
    for (int j = i+1; j < 10; j ++){
      if (*A[i] < *A[j]){
        DATA_TYPE *temp = A[i];
        A[i] = A[j];
        A[j] = temp;
      }
    }
  }
}
```

```
template <class DATA_TYPE>
void Array<DATA_TYPE>::Print()
{
  for (int i = 0; i < 10; i ++){
    A[i]->Print();
  }
}
```

36

Χρήση στη main()

```
int main()
{
  srand(time(NULL));
  Array<IntElement> iA(10);
  Array<PointElement> pA(10);

  for (int i = 0; i < 10; i ++){
    iA[i] = new IntElement(rand()%10);
    pA[i] = new PointElement(rand()%10, rand()%10);
  }

  iA.Print();
  pA.Print();

  iA.Sort();
  pA.Sort();

  iA.Print();
  pA.Print();
}
```


Χρήση στη main()

```
struct point
{
  int x;
  int y;
};

int main()
{
  srand(time(NULL));
  Array<point> pA(10);
  Array<int> iA(10);

  for (int i = 0; i < 10; i ++){
    iA[i] = new int(rand()%10);
    pA[i] = new point; pA[i]->x = rand()%10; pA[i]->y = rand()%10;
  }

  iA.Sort(); //OK! Int has < operator
  pA.Sort(); // COMPILE ERROR: point does not have < operator
  iA.Print(); // COMPILE ERROR: int does not have Print method
  pA.Print(); // COMPILE ERROR: point does not have Print method
}
```

Το Compiler Error εμφανίζεται μόνο όταν χρησιμοποιούμε την μέθοδο που έχει πρόβλημα

ΚΩΔΙΚΑΣ ΣΕ ΠΟΛΛΑ ΑΡΧΕΙΑ

Διαχείριση μεγάλων προγραμμάτων

- Σε μεγάλα projects όπου έχουμε μεγάλη ποσότητα κώδικα και πολλαπλές κλάσεις δεν είναι δυνατόν να έχουμε όλο τον κώδικα σε ένα αρχείο.
- Στον αντικειμενοστρεφή προγραμματισμό ο διαμερισμός του κώδικα σε πολλά αρχεία γίνεται τελείως φυσικά
  - Η κάθε κλάση τοποθετείται σε ξεχωριστό αρχείο.

Επικοινωνία μεταξύ κλάσεων

- Τι γίνεται όμως όταν μία κλάση Α θέλει να δημιουργήσει αντικείμενα μιας άλλης κλάσης Β?
  - Για να μπορέσει ο compiler να δεσμεύσει μνήμη για το αντικείμενο, θα πρέπει να ξέρει τι είναι αυτό για το οποίο δεσμεύει μνήμη.
  - Άρα η κλάση Α θα πρέπει να ξέρει τον ορισμό της κλάσης Β.
  - Πρέπει να κάνουμε include τον ορισμό της κλάσης Β στην κλάση Α.

Αρχεία επικεφαλίδες (header files)

- Η θεσμοθετημένη πλέον πρακτική είναι για κάθε κλάση να δημιουργείται ένα αρχείο επικεφαλίδα (.h file) που περιέχει τον ορισμό της κλάσης.
  - Π.χ. για την κλάση Person, θα δημιουργήσουμε ένα αρχείο Person.h που θα περιλαμβάνει τον ορισμό της κλάσης Person.
- Ο κώδικας για την υλοποίηση της κλάσης, τοποθετείται σε ένα αρχείο κώδικα (.cpp file)
  - Οι μέθοδοι της Person υλοποιούνται στο αρχείο Person.cpp

Inclusion

- Όποιος χρειάζεται τον ορισμό της Person κάνει include τον ορισμό το αρχείο Person.h
  - `#include "Person.h"`
- Βολεύει οι include εντολές να είναι μέσα στα header files. Μετά το κάθε .cpp αρχείο κάνει include μόνο το δικο του header file
  - Δηλαδή το Person.cpp κάνει include μόνο το Person.h header file.

Πολλαπλοί ορισμοί

- Αν έχουμε πολλά header αρχεία που το ένα κάνει include το άλλο, τότε υπάρχει κίνδυνος σε ένα αρχείο να δηλώσουμε πολλαπλές φορές την ίδια κλάση.
  - Στην περίπτωση αυτή θα πάρουμε λάθος από τον compiler.
- Για να το αποφύγουμε αυτό δίνουμε εντολή στον preprocessor να μην ορίσει δύο φορές την ίδια κλάση

Εντολή προεπεξεργαστή

```
#if !defined(__CLASSNAME__)
#define __CLASSNAME__

Class ClassName
{
      ...
      ...
      ...
}

#endif
```

Με αυτό τον τρόπο εξασφαλίζουμε ότι η κλάση θα οριστεί μόνο μία φορά. Τότε θα οριστεί και η μεταβλητή `__CLASSNAME__` οπότε την επόμενη φορά δεν θα μπούμε μέσα στο if

Το αρχείο Person.h

```
using namespace std;

#if !defined(__PERSON__)
#define __PERSON__

class Person {
private:
  int id;
  string fname;
  string lname;
public:
  Person();
  Person(int);
  void SetDetails(int id, string fn, string ln);
  bool operator == (const Person &) const;
  bool operator < (const Person &) const;
  int getId() const;
  void PrintDetails() const;
};

#endif
```


Το αρχείο Person.cpp

```
#include <iostream>
#include "Person.h"

Person::Person() {}

Person::Person(int i) { id = i; }

void Person::SetDetails(int i, string f, string l)
{ id = i;  fname = f;  lname = l; }

bool Person::operator == (const Person &p2)
{ return (id == p2.id); }

bool Person::operator < (const Person &p2)
{ return (id < p2.id); }

int Person::getId() const
{ return id;}

void Person::PrintDetails() const
{
  cout << "id: " << id
       <<" first name:" << fname << " last name:" << lname << endl;
}
```

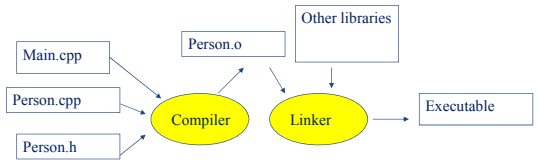

Το αρχείο main.cpp

```
#include <iostream>
#include <set>
#include "Person.h"

int main(){
  set<Person> S;
  Person P[3];

  int id;
  string fname, lname;
  for(int i =0 ; i< 3; i ++){
    cin >>id >> fname >> lname;
    P[i].SetDetails(id, fname, lname);
    S.insert(P[i]);
  }

  cin >> id;
  Person sp(id);
  set<Person>::iterator iter = S.find(sp);
  if (iter == S.end()){
    cout << "id not found\n";
  }else{ iter->PrintDetails(); }
}
```

makefiles

- Τα makefiles ρυθμίζουν τον τρόπο με τον οποίο συνδέουμε πολλά αρχεία μεταξύ τους
- Όταν μεταφράζουμε τον *πηγαίο* (source) κώδικα, ο μεταγλωττιστής παράγει ένα αρχείο με *τελικό* (object) κώδικα.
- Ο linker συνδέει τα αρχεία τελικού κώδικα με τον τελικό κώδικα από τις βιβλιοθήκες και παράγει το εκτελέσιμο.

makefiles



makefiles

- Το αρχείο makefile (ή σχεδόν ισοδύναμα και Makefile) περιέχει εντολές για το πώς θα συνδέσουμε και θα μεταγλωττίσουμε τα αρχεία μας.
- Μερικές *οδηγίες* (options ή directives):

```
# This is a comment
# Options for the compilation of C, C++ programs:
#  -O         Optimize object code
#  -c         Compile and assemble, but do not link
#  -o <file>  Place the output into <file>
```

Χρήση

- Τα περιεχόμενα του τρέχοντος directory είναι:
`Person.h, Person.cpp, main.cpp, makefile`
- Η μεταγλώττιση γίνεται ως εξής:
`make main`

File: makefile

```
# Person.o DEPENDS on Person.h and Person.cpp
# We create only the .o[bject] file and do not
# link it to an executable
# main DEPENDS on the Person.o file and main.cpp
# We compile it and we produce the executable
# main.exe

main: Person.o main.cpp
    g++ Person.o -o main.exe main.cpp
Person.o: Person.h Person.cpp
    g++ -O -c Person.cpp
```

Οι γραμμές αυτές θα πρέπει να ξεκινάνε οπωσδήποτε με tab !

Εναλλακτικά

- Ένας πιο απλός τρόπος για να κάνετε compile κατ' ευθείαν το πρόγραμμά σας:

```
g++ -o main.exe main.cpp Person.cpp
```

Σε κάποιους compilers: `-omain.exe` (χωρίς κενό)

Ένα μεγαλύτερο παράδειγμα.

- Θα δούμε πως μπορούμε να σπάσουμε το παράδειγμα με το Array σε πολλαπλά αρχεία.

Utils.h

Ένα αρχείο με διάφορες βιβλιοθήκες που χρειάζονται οι περισσότερες κλάσεις

```
#include <iostream>
#include <cstring>
#include <stdio.h>
#include <cmath>
#include <ctime>
#include <cstdlib>

using namespace std;
```


Element.h

```
#if !defined(__ELEMENT__)
#define __ELEMENT__

class Element
{
public:
  virtual bool operator < (Element &other) = 0;
  virtual void Print() = 0;
};

#endif
```

Για την κλάση Element δεν χρειάζεται να ορίσουμε .cpp αρχείο.

Array.h

```
#if !defined(__ARRAY__)
#define __ARRAY__

#include "Utils.h"
#include "Element.h"

class Array
{
private:
  Element **A;
  int size;
public:
  Array(int s);
  Element *& operator [](int);
  void Sort();
  void Print();
};

#endif
```


Array.cpp

```
#include "Array.h"

Array::Array(int s)
{
  size = s;
  A = new Element*[size];
}

Element *& Array::operator [] (int i)
{ return A[i]; }

void Array::Sort()
{
  for (int i = 0; i < 10; i ++)
    for (int j = i+1; j < 10; j ++)
      if (*A[i] < *A[j]){
        Element *temp = A[i]; A[i] = A[j]; A[j] = temp;
      }
}

void Array::Print()
{
  for (int i = 0; i < 10; i ++) { A[i]->Print();}
}
```


IntElement.h

```
#if !defined(__INT_ELEMENT__)
#define __INT_ELEMENT__

#include "Utils.h"
#include "Element.h"

class IntElement: public Element
{
private:
  int val;
public:
  IntElement(int);
  int GetVal();
  bool operator < (Element &);
  void Print();
};

#endif
```

IntElement.cpp

```
#include "IntElement.h"

IntElement::IntElement(int i)
{ val = i; }

int IntElement::GetVal()
{ return val; }


bool IntElement::operator <(Element &cOther)
{
  IntElement &other = dynamic_cast<IntElement &>(cOther);
  if (val < other.GetVal()){
    return true;
  }
  return false;
}

void IntElement::Print()
{ cout << val << endl; }
```

61

PointElement.h

```
#if !defined(__POINT_ELEMENT__)
#define __POINT_ELEMENT__

#include "Utils.h"
#include "Element.h"

struct point
{
  int x;
  int y;
};

class PointElement: public Element
{
private:
  point val;
public:
  PointElement(int,int);
  point GetVal();
  bool operator < (Element &);
  void Print();
};

#endif
```

62

PointElement.cpp

```
#include "PointElement.h"

PointElement::PointElement(int x,int y)
{ val.x = x; val.y = y;}

point PointElement::GetVal()
{ return val; }


bool PointElement::operator <(Element &cOther)
{
  PointElement &other = dynamic_cast<PointElement &>(cOther);
  if (val.x < other.GetVal().x){ return true; }
  if (val.x == other.GetVal().x){
    if (val.y < other.GetVal().y ){
      return true;
    }else { return false; }
  }else { return false; }
}

void PointElement::Print()
{
  cout << val.x << " " << val.y << endl;
}
```

63

main.cpp

```
#include "Utils.h"
#include "Array.h"
#include "IntElement.h"
#include "PointElement.h"

int main()
{
  srand(time(NULL));
  Array iA(10);
  Array pA(10);

  for (int i = 0; i < 10; i ++){
    iA[i] = new IntElement(rand()%10);
    pA[i] = new PointElement(rand()%10, rand()%10);
  }

  iA.Print(); pA.Print();
  iA.Sort(); pA.Sort();
  iA.Print(); pA.Print();
}
```


makefile

```
main: main.cpp Array.o IntElement.o PointElement.o Utils.h
	g++ -o main.exe Array.o IntElement.o PointElement.o main.cpp
Array.o: Array.h Array.cpp Element.h Utils.h
	g++ -c Array.cpp
IntElement.o: IntElement.h IntElement.cpp Element.h Utils.h
	g++ -c IntElement.cpp
PointElement.o: PointElement.h PointElement.cpp Element.h Utils.h
	g++ -c PointElement.cpp
```