HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 4

ΣΤΟΙΧΕΙΑ ΟΝΤΟΚΕΝΤΡΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αριθμός διαλέξεων 7, Διάλεξη 2η

CSD

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - *Εισαγωγή*
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

CSD

Εισαγωγή (1/2)

- *Νέες λέξεις-κλειδιά σε σύγκριση με τη C*

Κοινής χρήσης	Προχωρημένης χρήσης	Προηγμένης χρήσης
`class`	`operator`	`template`
`private`		`explicit`
`public`	`static_cast`	`mutable`
`protected`	`dynamic_cast`	`typeid`
`friend`	`reinterpret_cast`	
`inline`	`const_cast`	`namespace`
`virtual`		`using`
`const`	`try`	
`bool`	`catch`	
`true`	`throw`	
`false`		
`this`		
`new`		
`delete`		

Εισαγωγή (2/2)

- *Βασικές νέες προγραμματιστικές δομές οι οποίες εμπλέκονται ευρέως στη σχεδίαση και υλοποίηση κώδικα*

Επιπλέον προγραμματιστικά στοιχεία σε σύγκριση με τη C (κυρίως)
1. Δήλωση (declaration) και ορισμός (definition) κλάσεων
2. Συναρτήσεις και δεδομένα ως μέλη κλάσεων
3. Ποιοτικοί τελεστές πρόσβασης (access qualifiers)
4. Σχεδίαση κλάσεων και κληρονομικότητα
5. Υπερφόρτωση συναρτήσεων και τελεστών
6. Κατασκευή στιγμιότυπων κλάσεων και χρήση μελών
7. Δυναμική αντιστοίχηση κλήσεως συνάρτησης
8. Templates

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - *Class, constructor, destructor*
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Class, constructor, destructor (1/7)

- Ο constructor είναι η συνάρτηση που καλείται κατά την κατασκευή ενός στιγμιότυπου
- Ο destructor είναι αντίστοιχα η συνάρτηση που καλείται κατά την καταστροφή του
- Και οι δύο έχουν το ίδιο όνομα, που είναι ίδιο και με το όνομα της κλάσης. Ειδικότερα, ο destructor έχει το πρόθεμα ~
- Ο destructor δεν έχει καμία παράμετρο (ούτε void), και δεν επιδέχεται υπερφόρτωση
- Ο constructor μπορεί να έχει οσεσδήποτε παραμέτρους, ενώ μπορεί να οριστούν διάφοροι υπερφορτωμένοι constructors

Class, constructor, destructor (2/7)

```
struct Point { float x, y; };
class Shape {
     private:
     float x, y;
     public:
1:   Shape (float _x, float _y) { x = _x, y = _y; }
2α:  Shape (void) : x(0), y(0) {}
2β:  Shape (void) : Shape(0,0) {}
3α:   Shape (Point* point) : x(point->x), y(point->y){}
3β:   Shape (Point* point) : Shape(point->x, point->y){}
4:   ~Shape(){}
};
Shape shape1(10, 20);     // 1:
Shape shape2;             // 2:
Point point;
point.x = 40.56, point.y = 50.2;
Shape shape3(&point);     // 3:
```

…παράδειγμα

Class, constructor, destructor (3/7)

- Τόσο ο constructor όσο και ο destructor καλούνται αυτόματα κατά τη δημιουργία και καταστροφή
  - Μπορείτε ωστόσο να τις καλέσετε σαν κανονικές συναρτήσεις, με αποτέλεσμα να εκτελεστεί απλώς ο κώδικας που περιέχουν
    - *προσοχή, αυτό εφαρμόζεται σε αρκετά προχωρημένες τεχνικές*
- Ο destructor καλείται όταν το αντικείμενο καταστρέφεται, αμέσως πριν ελευθερωθεί τελικά η μνήμη που κατέχει το στιγμιότυπο
  - Συνήθως στον destructor περιλαμβάνουμε κώδικα που απελευθερώνει ότι επιπλέον δυναμική μνήμη ή πόρους έχει λάβει το στιγμιότυπο
    - και όχι τη μνήμη που καταλαμβάνει κάθε στιγμιότυπο μίας κλάσης **X**, η οποία είναι εξ ορισμού μεγέθους πάντα ίσου με ***sizeof(X)***, και υπολογίζεται από τον compiler κατά τη διάρκεια του compilation

Class, constructor, destructor (4/7)

```
#include <string.h>
#include <malloc.h>
class String {
   private:
   char* str;
   public:
   String (void) { str = strdup(“”); }
   String (char* s) { str = strdup(s); }
   String (String* s) { str = strdup(s->str); }
   ~String() { free(str); str = nullptr; }
};
String s1;                  // Το ίδιο είναι αν γράψουμε s1().
String s2(“hello, world”);
String s3(&s2);
s3.String::~String();       // Το ίδιο είναι και το s3.~String();,
s3.String::String(&s1);     // ενώ το s3.String(&s1); είναι error !
```

…παράδειγμα

Class, constructor, destructor (5/7)

- Πότε γίνεται η κλήση του destructor και constructor κατά τη δήλωση στιγμιότυπων

```
class Hello {
   public:
   Hello (const char* msg) { printf(“Hello(‘%s’)\n”, msg); }
   Hello (void) { printf(“Hello(void)\n”); }
  ~Hello() { printf(“~Hello()\n”); }
};

int main (int argc, char* argv[]) {
  Hello hello1, hello2(“world”);
  { Hello hello3(“a better world”); { Hello hello4(“with peace”); } }
}
```

Hello(void)	hello1	~ Hello()	hello4
Hello(‘world’)	hello2	~ Hello()	hello3
Hello(‘a better world’)	hello3	~ Hello()	hello2
Hello(‘with peace’)	hello4	~ Hello()	hello1

Class, constructor, destructor (6/7)

- Κανόνας κλήσης των destructors για τα δεδομένα – μέλη μίας κλάσης
  - Οι destructors των τοπικών δεδομένων καλούνται πάντα ***μετά*** τον destructor της κλάσης - το ίδιο ισχύει και για την περίπτωση κλήσης απευθείας ως απλή συνάρτηση

```
class Inner {
   public:
   ~Inner() { printf("~Inner()\n"); }
};
class Outer {
   private:
   Inner inner1, inner2;
   public:
   ~Outer(){ printf("~Outer()\n"); }
};
{ Outer outer;  outer.~Outer(); }
```

```
Εκτυπώνει πρώτα (λόγω της κλήσης ως συνάρτησης μέσω outer.~Outer(); )
~Outer()
~Inner() Προσοχή, καλείται και αυτό!!!
~Inner() Προσοχή, καλείται και αυτό!!!

...και έπειτα (λόγω της αυτόματης κλήσης destructor με την έξοδο από το block)
~Outer()
~Inner()
~Inner()
```

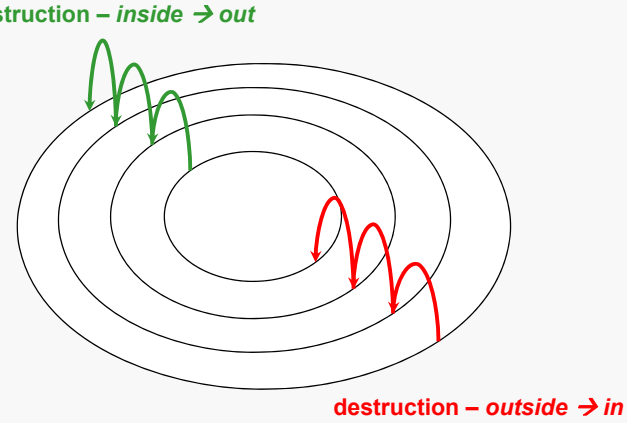
Class, constructor, destructor (7/7)

- Κανόνας κλήσης των constructors για τα δεδομένα – μέλη μίας κλάσης
  - Οι constructors των τοπικών δεδομένων καλούνται πάντα ***πριν*** τον constructor της κλάσης - το ίδιο ισχύει και για την περίπτωση απευθείας κλήσης ως απλή συνάρτηση

```
class Inner {
   public:
   Inner(void) { printf("Inner()\n"); }
};
class Outer {
   private:
   Inner inner1, inner2;
   public:
   Outer(void){ printf("Outer()\n"); }
};
{ Outer outer;  outer.Outer::Outer(); }
```

```
Εκτυπώνει πρώτα (λόγω της αυτόματης κλήσης constructor)
Inner()
Inner()
Outer()

...και έπειτα (λόγω της κλήσης ως συνάρτησης μέσω outer.Outer::Outer(); )
Inner() Προσοχή, καλείται και αυτό!!!
Inner() Προσοχή, καλείται και αυτό!!!
Outer()
```


Ένθετο



Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - *Class members and access qualification*
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Class members and access qualification (1/1)

- Υπάρχουν κυρίως δύο βασικοί ποιοτικοί τελεστές πρόσβασης, που ορίζουν ποια μέλη μίας κλάσης μπορούν να χρησιμοποιούνται έξω από την εμβέλεια ορισμού της κλάσης.

```
class Shape {
   private:
   float x, y;
   Shape (Shape* shape);

   public:
   float GetX (void) const { return x; }
   float GetY (void) const { return y; }
   void  Move (float dx, float dy);
   ...
};
```

```
void Shape::Move (float dx, float dy) {
   x += dx, y+= dy;
}

Shape sh;       // Ok: default constructor
sh.x = 10;      // Error: 'x' is private
sh.Move(10,0);  // Ok: 'Move' public.
sh.GetX()=30;   // Error: not an Lvalue.
Shape cpy(&sh); // Error: private constructor
```

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - *Member linkage qualification*
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Member linkage qualification (1/5)

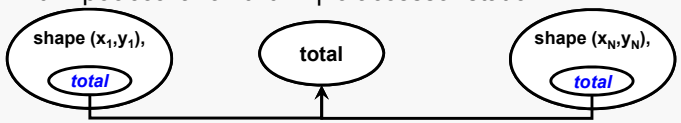
- Τα μέλη-δεδομένα μίας κλάσης μπορούν να οριστούν ως τοπικά (local / auto) και να επαναλαμβάνονται για κάθε στιγμιότυπο (default linkage), η ως καθολικά (static) και να είναι κοινόκτητα (shared) μεταξύ όλων των στιγμιότυπων.

```
// shape.h

class Shape {
   private:
   static unsigned int total;

   public:
   static unsigned int Total (void);
   Shape(...) { ... ++total; }
   ~Shape() { ... --total; }
};
```

```
// shape.c
#include "shape.h"

unsigned int Shape::total = 0;

unsigned int Shape::Total (void) {
   return total;
}

// main.c
Shape shape1, shape2, shape3;
printf("Total=%d\n", Shape::Total());
```


Member linkage qualification (2/5)

- Τα static δεδομένα είναι ουσιαστικά *κοινά δεδομένα* μεταξύ των στιγμιότυπων μίας κλάσης.
  - Επίσης «υπάρχουν» (δηλ. *έχουν ήδη το δικό τους χώρο μνήμης*) ανεξάρτητα από το εάν υπάρχουν στιγμιότυπα (και πόσα είναι αυτά)
  - Συνήθως τα static δεδομένα διαχειρίζονται από static συναρτήσεις
  - Αποφεύγετε πλέον τις καθολικές static ή non-static (extern συνήθως) μεταβλητές στη C++. Ενσωματώστε τις σε μία κλάση και προσθέστε το κατάλληλο accessor static API



Member linkage qualification (3/5)

- Public συναρτήσεις-μέλη που ορίζονται με static σύνδεση μπορούν να καλούνται με ειδικό συντακτικό χωρίς τη χρήση κάποιου στιγμιότυπου.
  - *Υπενθυμίζουμε ότι τέτοιου είδους συναρτήσεις είθισται να έχουν πρόσβαση σε static δεδομένα της κλάσης*

```
class Process {
   private:
   static ProcessTable* processTable;
   unsigned int id;
   public:
   static ProcessTable* Table(void)
          { return processTable; }
   unsigned int Id (void) const
          { return id; }
   ...
};
```

```
unsigned int id;
id  = Process::Id(); // Error: needs instance.

Process process;
id = process.Id();   // Ok, using instance.

ProcessTable* table;
table = Process::Table(); // Ok: static

table = process.Table();  // Ok, but avoid!
```

Member linkage qualification (4/5)

- Όσες συναρτήσεις υλοποιούνται στο σώμα δήλωσης μίας κλάσης, δηλ. **class { *εδώ* }** , αυτομάτως θεωρούνται ως **inline**, δηλ. ο κώδικας τους επαναλαμβάνεται από τον compiler σε κάθε σημείο κλήσης τους.
- Με τη χρήση *inline* συναρτήσεων ο παραγόμενος από τον compiler εκτελέσιμος κώδικας του προγράμματος τείνει να είναι μεγαλύτερος, αλλά γρηγορότερος.
  - Ωστόσο προκαλείται σημαντική αργοπορία στην φάση ανάπτυξης, αφού οι όποιες αλλαγές του κώδικα υλοποίησης μπορεί να γίνονται και μέσα σε header files, γεγονός που πάντα προκαλεί μεταγλώττιση περισσότερων αρχείων λόγω των εξαρτήσεων χρήσης του header file.
- → *Να χρησιμοποιείτε inline συναρτήσεις με οικονομία: μόνο εάν είναι υπερβολικά απλές, η πολύ απλές και κρίσιμες ως προς την ταχύτητα.*
- → *Ο χαρακτηρισμός inline μπορεί να δοθεί και σε συναρτήσεις που ορίζονται εκτός τους σώματος δήλωσης*

Member linkage qualification (5/5)

- Εάν υλοποιείτε συναρτήσεις - μέλη ***εκτός*** του σώματος ***της κλάσης***:
  - ***Εάν είναι μέσα στο header file τότε να τις ορίζετε πάντα ως inline***
    - Σε αντίθετη περίπτωση θα έχετε "**function redefinition link error**"
  - ***Εάν είναι μέσα στο source file*** τότε ο χαρακτηρισμός ***inline*** επιτρέπεται ***μόνο εάν δεν χρησιμοποιείται*** η συνάρτηση αυτή και ***σε άλλα αρχεία*** *(π.χ. private functions)*
    - Σε αντίθετη περίπτωση θα έχετε "**undefined function link error**"

`// shape.h`	`// shape.c`
Η παρακάτω υλοποίηση θα οδηγήσει σε link errors εάν το header γίνει included από πολλά αρχεία (multiple definitions of Shape::Move). `void Shape::Move(...) {...}`	Αποδεκτό, αλλά εάν και άλλα αρχεία καλούν την Shape::Move θα υπάρξει link error (γιατί ο compiler δεν θα φτιάξει αντίστοιχη συνάρτηση στον εκτελέσιμο κώδικα). `inline void Shape::Move(...} {...}`
Ο σωστός τρόπος να τοποθετούμε υλοποίηση συναρτήσεων εκτός κλάσης, αλλά εντός header file. `inline void Shape::Move(...} {…}`	Ο συνήθης τρόπος υλοποίησης και σύνδεσης, μέσα στο source της υλοποίησης της κλάσης. `void Shape::Move(...} {...}`

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - *Friend qualification*
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Friend qualification (1/2)

- Σε περίπτωση που σε κάποιες συναρτήσεις ***F*** ή κλάσεις ***C*** επιθυμείτε να επιτρέψετε την πρόσβαση στα private μέλη μίας κλάσης ***X***,
  - τότε θα πρέπει οι ***F*** και ***C*** να δηλώνονται στο σώμα της κλάσης ***X*** με τον χαρακτηρισμό **friend**
- Για τον ορισμό ενός friend, αρκεί η αναφορά μόνο του ονόματος της κλάσης, η της υπογραφής (signature / prototype) της συνάρτησης,
  - ενώ δεν είναι απαραίτητο να προηγείται και ο ορισμός τους
- → *Οι ορισμοί friend να είναι οι πρώτοι που εμφανίζονται στο σώμα της κλάσης, πριν οποιοδήποτε access qualifier (οι οποίοι δεν παίζουν ρόλο στο friend qualification)*

Friend qualification (2/2)

```
class X {
    friend class Y;
    friend class Z;
    friend void  F (X& x);

    private:
    void f (void);
    int  g (char*);
    ...

};

class Y {
   private:
   int h (X& x, char* s) {
      x.f();
      return x.g(s);
   }
   ...
}
```

```
void F (X& x, int a) {
  x.f();  // Error: void F(X&,int) not a friend.
}

void F (X& x) {
   x.f();  // Ok, void F(X&) is friend.
}

class X { // Extra. Operators can be friends.
   friend X operator+(X& a, X& b);
   ...
};
```

Η χρήση του friend αλόγιστα πρέπει να αποφεύγεται. Γενικά θα πρέπει να χρησιμοποιείται μόνο εάν δεν υπάρχει καλύτερος τρόπος οργάνωσης (π.χ. το να κάνουμε public απλώς τα μέλη που χρειάζονται είναι χειρότερη λύση).

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - *Instance references*
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Instance references (1/4)

- *Ένα reference (αναφορά) είναι ένα είδος απλού δείκτη, το οποίο ωστόσο δεν απαιτεί το συντακτικό των δεικτών*. Αναφέρεται σε σε μία μεταβλητή ή ένα στιγμιότυπο με τον ίδιο τρόπο που ο δείκτης δείχνει σε αυτό.
  - *Ουσιαστικά τόσο οι pointers όσο και τα references είναι διευθύνσεις μνήμης*
- Δεν μπορούμε να έχουμε πολλαπλά references όπως έχουμε πολλαπλούς δείκτες
  - δηλ. επιτρέπονται references σε όλους τύπους εκτός από references
- Οι μεταβλητές references που ορίζονται μέσα σε block πρέπει πάντα να αρχικοποιούνται στο σημείο της δήλωσης
- Δεν επιτρέπονται δείκτες σε references
  - Η λήψη της διεύθυνση με & μίας μεταβλητής reference επιστρέφει ένα δείκτη στη μεταβλητή η στιγμιότυπο που αναφέρεται το reference και όχι τη διεύθυνση μνήμης της μεταβλητής reference

Instance references (2/4)

παραδείγματα

```
int a;
int& intRef = a;  // Το intRef αναφέρεται στο a
intRef = 10;      // Το a γίνεται 10, δηλ. intRef συνώνυμο του a
int b = 20;
int* intPtr = &b; // Δείκτης στο b
intRef = *intPtr; // Το a γίνεται ίσο  με b, δηλ. 20
intPtr = &intRef; // Δηλ. το intPtr γίνεται ίσο με &a
*intPtr = 30;     // Δηλ. το a γίνεται 30

Shape  shape;
Shape& shapeRef = shape;  // Το shapeRef αναφέρεται στο shape
shapeRef.Move(10,10);     // Το ίδιο με shape.Move();

void Swap (int& a, int& b) {
   int temp = a;
   a = b;     // Είναι ουσιαστικά *&a = *&b
   b = temp;
}
Swap(a,b);    // Ανταλλαγή τιμών (δηλ. swap) – να αποφεύγεται
Swap(10, b);  // Error: μόνο Lvalues μπορούν να έχουν references
```

Instance references (3/4)

παραδείγματα

```
int& intRef;          // Error, πρέπει να αρχικοποιηθεί
int&& intRefRef;      // Error, reference σε reference δεν επιτρέπεται
typedef int& IntRef;  // Ok. Ορισμός ενός reference type
IntRef* intRefPtr;    // Error, pointers σε references δεν επιτρέπονται
void& voidRef;        // Error, δεν επιτρέπονται references σε void

class Window {
    public:
    Window& Move(int dx, int dy) { x+=dx, y+=dy; Display(); return *this; }
    Window& SetTitle (const char* title);
    Window& SetHasScrollBars (bool has);
    ...
    Window (void);
}

Window win;
win.Move(10,10)               // Επιστρέφει Window&, άρα μπορώ να καλέσω →
   .SetTitle("Hello")         // ...το οποίο επίσης είναι Window&, άρα →
   .SetHasScrollBars(true);   // ... και θα μπορούσα να είχα και άλλη κλήση.
```


Instance references (4/4)

- *Χρήση const references*
  - Ορίζει references σε αντικείμενα τα οποία δεν μπορούν να χρησιμοποιηθούν για να αλλάξουν τα αναφερόμενα αντικείμενα - read only ή αλλιώς immutable
  - Η πιο συνήθης χρήση τους είναι να αναφέρονται σε στιγμιότυπα που είναι παράμετροι σε συναρτήσεις
    - *Δεν επιτρέπεται η κλήση συναρτήσεων ενός στιγμιότυπου μέσω ενός const reference παρά μόνο εάν αυτές έχουν οριστεί ως const* - δηλ. δεν αλλάζουν τις τοπικές μεταβλητές του στιγμιότυπου

```
int f (const int& a, const int& b) {

  a = b/2;            // Error, const reference
  b = a/2;            // Error, const reference
  int* p1 = &a;       // Error, το &a είναι const int*, και όχι int*
  int& aa = a;        // Error, το a είναι const int&, και όχι int&
}
```


References - ένθετο

code generation με εξάλειψη χρήσης references

C++	C++/C

`void swap (int& a, int& b) {`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`}`
	`void swap (int* a, int* b) {`
	`  int temp = *a;`
	`  *a = *b;`
	`  *b = temp;`
	`}`

`int x, y;`
`swap(x, y);`
`int& z = x;`
`z = 20;`
`int* w = &z;`
`*w = 30;`
	`int x, y;`
	`swap(&x, &y);`
	`int* z = &x;`
	`*z = 20;`
	`int* w = z;`
	`*w = 30;`

μετατροπή μεταβλητής y σε reference ως &y, τύπος T& ως T*, χρήση x reference ως *x

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - *Function overloading*
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Function overloading (1/2)

- Η υπερφόρτωση δύναται να εφαρμοστεί τόσο τόσο σε καθολικές (global) συναρτήσεις, όσο και σε συναρτήσεις μέλη (member functions)
  - Επιτρέπεται η υπερφόρτωση και του constructor αλλά όχι του destructor
- Η υπερφόρτωση είναι νόμιμη εάν και μόνο εάν υπάρχουν διαφορετικές λίστες τυπικών ορισμάτων (formal arguments)
  - *Δεν ορίζεται υπερφόρτωση μίας συνάρτησης με διαφοροποίηση μόνο ως προς τον τύπο του επιστρεφόμενου αποτελέσματος*
  - *Υπάρχει μόνο μία εξαίρεση που αφορά την κληρονομικότητα*

Function overloading (2/2)

παραδείγματα

```
char* strconcat (char* str, char* s);
char* strconcat (char* str, int num);
char* strconcat (char* str, char c);
char* strconcat (char* str, double num);
char* strconcat (char* str, Employee* employee) {
   return strconcat(str, employee->tostring());
}
char* strconcat (char* str, bool val);

class Employee {...
   public: char* tostring (void); ...
};
```

```
strconcat(s, "Hello");
strconcat(s, 10);
strconcat(s, 't');
strconcat(s, 3.14);
strconcat(s, true);
Employee employee;
strconcat(s, &employee);
```


Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - *Operator overloading*
  - Dynamic memory allocation
  - Function pointers
  - Namespaces

Operator overloading (1/5)

- Όλοι, εκτός από τους παρακάτω τέσσερις τελεστές μπορούν να υπερφορτωθούν:
  - **. .* :: ? :**
- Για ορισμένους τελεστές η σημασιολογία της υπερφόρτωσης μπορεί να φαίνεται προφανής, αλλά ωστόσο υπάρχουν αρκετές περιπτώσεις προηγμένης χρήσης
- Αρκετά λάθη μπορούν να προκληθούν εάν δεν δοθεί η δέουσα προσοχή στην υπερφόρτωση δυαδικών τελεστών, όταν πρέπει αυτοί να επιστρέφουν προσωρινά αντικείμενα
- Η εκτέλεση κώδικα με υπερφόρτωση τελεστών συνήθως είναι βραδύτερη από ότι ο συνήθης κώδικας
  - έτσι θα πρέπει να αποφεύγεται σε περιπτώσεις όπου η μέγιστη ταχύτητα είναι κρίσιμος επιθυμητός στόχος

Operator overloading (2/5)

παραδείγματα

```
class Shape {
   public:          const T operator=(const T&)
   const Shape& operator=(const Shape& Rvalue) {  // a=b=c επιτρέπεται
      x = Rvalue.x, y=Rvalue.y;                   // To this είναι το Lvalue
      return *this;                               // Επιστρέφει το Lvalue
   }
}
```

Όταν ο operator = δεν έχει νόημα να επιστρέφει κάποια τιμή ή δεν είναι σωστό να συμβαίνει κάτι τέτοιο (π.χ. όταν η δημιουργία προσωρινών στιγμιότυπων δεν ορίζει καλή συμπεριφορά στο σύστημά μας), τότε πρέπει πάντοτε να έχει signature *void operator=(const T&)*.

Εάν έχετε την απορία για ποιο λόγο επιστρέφουμε προσωρινό στιγμιότυπο και όχι απευθείας το reference του l-value, τότε θεωρείστε την παρακάτω κλήση:

```
F(x=a, x=b, x=c)
```

Προφανώς για να είναι σωστή η συμπεριφορά θα πρέπει τα actual arguments να είναι *(a, b, c)*. Όμως εάν ο τελεστής = επιστρέφει reference, τότε τα actual arguments είναι *(x, x, x)* που μπορεί να είναι μία από τις τρείς περιπτώσεις (a, a, a) (b, b, b) (c, c, c) καμία από τις οποίες δεν είναι η σωστή.

Operator overloading (3/5)

παραδείγματα

```
Shape shape1(10,20), shape2, shape3;
shape3 = shape2 = shape1;  // Αυτόματη κλήση του υπερφορτωμένου τελεστή =
shape2.operator=(shape1);  // Ως συνάρτηση, ισχύει και αυτό το συντακτικό !

class Plus {               // Classes που υπερφορτώνουν το () λέγονται functors
   public:
   int operator()(int a, int b) { return a + b; } // Υπερφόρτωση τελεστή ()
   int operator()(int a, const char* s); // Μπορώ να έχω και εναλλακτικές εκδόσεις
}

Plus plus;                 // Δήλωση στιγμιότυπου
int a = plus(10,20);       // Αυτόματη κλήση υπερφορτωμένου τελεστή ()
a = plus.operator()(10,20); // ...ή με το εναλλακτικό μη συνηθισμένο συντακτικό
int b = Plus()(24, 50);    // ...ή και με χρήση προσωρινού στιγμιότυπου !
```


Operator overloading (4/5)

- Ιδιαίτερη προσοχή απαιτείται για την υπερφόρτωση των λογικών τελεστών όπως || && καθώς τότε ***ακυρώνεται*** το short-circuit boolen evaluation
  - *Δεν είναι παράνομο, απλά πρέπει να το γνωρίζουμε!*

```
class X {
          public:
          bool operator||(const X&) const;
          const X& f(const X&) const;
          …
};

X x1, x2;
bool a = x1.f(x2) || x2.f(x1);
ισοδύναμο με το:
a = x1.f(x2).operator||(x2.f(x1));
```

Δηλ. πάντα τα δύο ορίσματα γίνονται evaluated!

Operator overloading (5/5)

- Η λύση στη C++ είναι ο ορισμός ενός member function που υλοποίει την αυτόματη μετατροπή ενός στιγμιότυπου σε boolean τιμή (ειδικό feature της C++ για type casting)

```
class String {
          char* s;
          public:
          operator bool() { return s && *s; }
          operator T() {...} Γενική περίπτωση
          ...
};
```

Στον παρακάτω κώδικα εάν το s1.operator bool() επιστρέφει true το αντίστοιχο για το s2 δεν εκτελείται άρα έχουμε short-circuit boolean evaluation. Να χρησιμοποιείτε αυτή την τακτική για τους λογικούς τελεστές.

```
String s1, s2;
if (s1 || s2) {…}
```

CSD

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - *Dynamic memory allocation*
  - Function pointers
  - Namespaces

CSD

Dynamic memory allocation (1/3)

- Η δυναμική παραχώρηση μνήμης στη C++ χρειάζεται μόνο τον τύπο / κλάση του αντικειμένου, και προαιρετικά τον αριθμό των δυναμικά δημιουργούμενων αντικειμένων
  - ο μηχανισμός αυτόματα λαμβάνει την απαραίτητη ποσότητα μνήμης, και καλεί τον κατάλληλο constructor, και επιστρέφει τιμή συμβατή με δείκτη τύπου / κλάσης
- Παρομοίως, η απελευθέρωση μνήμης χρειάζεται έναν δείκτη τύπου / κλάσης
  - ο μηχανισμός καλεί αυτόματα τον (τους) destructor(-s) και έπειτα απελευθερώνει την μνήμη
- **`new new[] delete delete[]`**

CSD

Dynamic memory allocation (2/3)

παραδείγματα

```
int* aPtr = new int;          // Παραχώρηση μνήμης 1 αντικείμενου τύπου int
*aPtr = 10;                   // Η γνωστή χρήση μέσω δεικτών
delete aPtr;                  // Απελευθέρωση της μνήμης
aPtr = new int[10];           // Παραχώρηση μνήμης 10 αντικειμένων τύπου int
delete[] aPtr;                // Εάν λαμβάνουμε με new[], επιστρέφουμε με delete[]

Shape* shapePtr = new Shape;  // Ο default constructor καλείται αυτόματα
shapePtr->Move(10,20);        // Κλήση συνάρτησης-μέλους μέσω δείκτη στιγμιότυπου
(*shapePtr).Move(10,20);      // Διαφορετικό συντακτικό για την προηγούμενη κλήση
Shape shape(40,50);           // Δήλωση τοπικού (στη στοίβα) στιγμιότυπου
*shapePtr = shape;            // Εκχώρηση μέσω του υπερφορτωμένου τελεστή =
delete shapePtr;              // Απελευθέρωση μνήμης (αυτόματη κλήση destructor)

shapePtr = new Shape[2];      // Παραχώρηση 2 αντικειμένων Shape, οι default
                              // constructors καλούνται αυτόματα
shapePtr[0].Move(20,50);      // Φυσιολογική πρόσβαση στο 1ο (θέση 0) αντικείμενο
shapePtr[1] = shapePtr[0];    // Πρόσβαση στο 2ο (θέση 1), και εκχώρηση από το 1ο
delete[] shapePtr;            // Καλούνται αυτόματα και οι destructors
shapePtr = &shape;            // Λήψη διεύθυνσης τοπικού στιγμιότυπου σε δείκτη
delete shapePtr;              // Και ένα σίγουρο system crash
```


CSD

Dynamic memory allocation (3/3)

- *Απλοί κανόνες ασφαλείας*
  - Να φροντίζετε να απελευθερώνετε τη μνήμη που λαμβάνετε δυναμικά ***αμέσως*** όταν δεν την χρειάζεστε
  - Προσοχή στο γεγονός ότι είναι εύκολο, και πολύ επικίνδυνο, να συνεχίσετε να χρησιμοποιείτε μνήμη που έχετε ήδη απελευθερώσει
    - Οι δείκτες που γίνονται deleted, αλλά μπορεί να χρησιμοποιούνται ακόμη στο πρόγραμμά σας, καλό είναι να γίνονται null (έτσι θα σιγουρέψετε ένα άμεσο crash σε περίπτωση λάθους πρόσβασης)
  - Είναι πολύ σύνηθες με την χρήση δεικτών να συγχέετε τη δυναμική μνήμη με τη μνήμη στιγμιότυπων που «εδρεύουν» στη στοίβα
    - η πρόσβαση σε στιγμιότυπα στοίβας που έχουν ήδη γίνει popped, καταστρέφει τη στοίβα και δημιουργεί «δύσκολα» crashes (ακόμη και οι debuggers μπορεί να χάσουν το «δρόμο» με κατεστραμμένη στοίβα)

CSD

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - *Function pointers*
  - Namespaces

CSD

Function pointers (1/3)

- Υπάρχουν δύο κατηγορίες δεικτών σε συναρτήσεις, με διαφορετικό συντακτικό και σημασιολογία η κάθε μία:
  - Δείκτες σε συναρτήσεις που έχουν καθολική εμβέλεια (global functions, static member functions)
    - οι οποίες μπορούν να κληθούν μόνο με το αναγνωριστικό τους όνομα
  - Δείκτες σε μη καθολικές (non-static)συναρτήσεις – μέλη
    - οι οποίες καλούνται με το αναγνωριστικό τους όνομα, αλλά πάντα με τη χρήση και ενός στιγμιότυπου

CSD

Function pointers (2/3)

παράδειγμα – static functions

```
class Editor {
private:
    typedef void (*ErrorHandlerFunc) (unsigned int errCode);
    static void  ExitOnEror (unsigned int errCode);
    static void  LogOnError(unsigned int errCode);
    static void  UserDialogueOnError(unsigned int errCode);
    ErrorHandlerFunc errorHandler;
    void Error (unsigned int errCode) { (*errorHandler)(errCode); }
public:
    enum ErrorPolicy { Exit, Log, UserDialogue };
    Editor (ErrorPolicy policy) {
       switch (policy) {
           case Exit         : errorhandler = ExitOnEror; break;
           case Log          : errorhandler = LogOnError; break;
           case UserDialogue : errorhandler = UserDialogueOnError; break;
       }
    }
};
Editor editor1(Editor::Log);
Editor editor2(Editor::Exit);
Editor editor3(Editor::UserIntervention);
```


CSD

Function pointers (3/3)

παράδειγμα – member functions

```
class X {
    public:
    void f (int);
    void g (int);
    int a;
};

void (X::*pmf)(int) = &X::f;   // pmf δείκτης συνάρτησης- μέλους, εκχωρείται τη X::f
(*pmf)(10);                    // Error, χρειάζεται στιγμιότυπο, λάθος τρόπος κλήσης
X obj;
(obj.*pmf)(10);                // Ok, οι παρενθέσεις στο 1ο τμήμα απαραίτητες !
X* objPtr = &obj;
(objPtr->*pmf)(20);            // Ok, κλήση μέσω δείκτη στιγμιότυπου
pmf = &X::g;                   // Μπορεί να δείχνει και σε άλλη συμβατή συνάρτηση

int X::*pmi = &X::a;           // Παρόμοιο συντακτικό για δείκτες σε μέλη-δεδομένα
*pmi = 10;                     // Error, και εδώ χρειάζεται στιγμιότυπο !
obj.*pmi = 10;                 // Ok, παρόμοιο συντακτικό με τις συναρτήσεις
double X::*pmd;                // Ok, ασχέτως ανυπαρξίας μεταβλητής τέτοιου τύπου !
```

CSD

Περιεχόμενα

- Οι βασικές δομές – *περιήγηση από το πρίσμα της C++*
  - Εισαγωγή
  - Class, constructor, destructor
  - Class members and access qualification
  - Member linkage qualification
  - Friend qualification
  - Instance references
  - Function overloading
  - Operator overloading
  - Dynamic memory allocation
  - Function pointers
  - *Namespaces*

CSD

Namespaces (1/2)

- Ένα namespace είναι ένας ονομαζόμενος χώρος, στον οποίο επιτρέπονται όλων των ειδών οι δηλώσεις και οι ορισμοί
  - Το όνομα του namespace αυτομάτως γίνεται πρόθεμα στα αναγνωριστικά ονόματα των δηλώσεων / ορισμών που δηλώνονται ή ορίζονται στο χώρο του
  - Δηλ. δηλώσεις / ορισμοί με ίδια αναγνωριστικά ονόματα, αλλά σε διαφορετικά namespaces, είναι απολύτως νόμιμες
  - Τα namespaces έχουν υπόσταση μόνο σε compile-time, και χρησιμεύουν στην αποφυγή συγκρούσεων ονοματολογίας μεταξύ δηλώσεων / ορισμών διαφόρων τμημάτων / υπο-συστημάτων / μονάδων
- Τα namespaces μπορούν να οριστούν μέσα σε άλλα namespaces, ενώ μπορούν να σπάσουν σε διαφορετικά κομμάτια, σε διαφορετικά αρχεία

CSD

Namespaces (2/2)

παραδείγματα

```
// constants.h
namespace Constants {
    namespace Math { const double  pi = 3.1416; }
    namespace Ids { const char*   ProgramId = "DynaBase"; }
}

// util.h
namespace Util {
    extern float Max (float* numbers, unsigned int N);
}

// util.cpp
namspace Util {
    float Max (float* numbers, unsigned N) { ...υλοποίηση...  }
}
εναλλακτικά float Util::max (...) {...}
// application.cpp
float perimeter = 2 * Constants::Math::pi * circle.GetRadius();
strconcat(paramsStr, Constants::Names::ProgrameName);
float numbers[] = {0,1,2,3,4,5,6};
float maxNumber = Util::Max(numbers, sizeof(numbers) / sizeof(float));
```