ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

2ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- Κλάσεις - Classes
- Ενθυλάκωση – Encapsulation
- Constructors / Destructors
- Default Arguments
- Function overloading
- Operator overloading
- Inline & static keyword
- Structs vs Classes
- Pointers & References
- Copy constructor
- Const keyword
- Οργάνωση σε αρχεία

Κλάσεις - Classes

- Στον αντικειμενοστραφή προγραμματισμό (Object Oriented Programming) τα προγράμματά μας μοντελοποιούνται με βάση τα δεδομένα τους
- Οι κλάσεις είναι οντότητες που έχουν κάποιο νόημα στο πρόγραμμά μας
- Οι κλάσεις αυτές μπορεί να έχουν κάποια χαρακτηριστικά (attributes – members) αλλά και κάποια λειτουργικότητα (functionality – methods)

Παράδειγμα: Class Vector

- Μοντελοποιώντας ένα διάνυσμα (vector).
  - Δεδομένα
    - Συντεταγμένες x, y, z
  - Λειτουργικότητα
    - add, dot product, cross product, equality, ...
- Ορίζουμε τα δεδομένα ως private members και τη λειτουργικότητα ως public member functions: ***encapsulation***

```
class Vector {
public:
  Vector Add(Vector v);
  Vector DotProduct(Vector v);
  Vector CrossProduct(Vector v);
  bool   IsEqual(Vector v);

  Vector(float x,float y,float z);

private:
  float x, y, z;
};
```

Ενθυλάκωση – Encapsulation (1/2)

- Στον οντοκεντρικό προγραμματισμό οι κλάσεις πρέπει να παρέχουν κάποια λειτουργικότητα αλλά ***πρέπει να κρύβουν*** την εσωτερική υλοποίηση
- Η λειτουργικότητα αυτή λέγεται συνήθως Application Programming Interface – API
- Στο προηγούμενο παράδειγμα, ο χρήστης δεν ξέρει ότι είχαμε 3 float μεταβλητές για την εσωτερική αναπαράσταση
  - Μπορεί αργότερα να χρησιμοποιήσουμε ένα πίνακα εσωτερικά, χωρίς να επηρεάσουμε τη λειτουργικότητα που παρέχουμε στο χρήστη.
- Αν οι μεταβλητές ήταν public, ο χρήστης θα μπορούσε να γράψει
  - Vector v;
    printf("(%f, %f, %f)\n", v.x, v.y, v.z);
  - Κάνοντας όμως την αλλαγή σε πίνακα ο κώδικας του χρήστη δε θα λειτουργούσε πλέον.

Ενθυλάκωση – Encapsulation (2/2)

- Για να έχουμε πρόσβαση στα private members παρέχουμε επιπλέον member functions που τα λέμε ***accessors***.
- Το keyword ***this*** είναι ένας pointer στο στιγμιότυπο της κλάσης που μας βοηθάει να προσπελάσουμε members που είναι υποσκιασμένες (shadowed) από τοπικές μεταβλητές.

```
class Vector {
public:
   Vector Add(Vector v);
   Vector DotProduct(Vector v);
   Vector CrossProduct(Vector v);
   bool   IsEqual(Vector v);

   float GetX(void) {return x;}
   void SetX(float x)
      { this->x = x; }
   void SetY(float y)
      { Vector::y = y; }
   // z...
private:
   float x, y, z;
};
```

Constructors / Destructor (1/2)

- **Constructor**
  - Καλείται όταν δημιουργείται το object
  - Έχει το ίδιο όνομα με την κλάση
  - Μια κλάση μπορεί να έχει πολλούς overloaded constructors
  - Χρησιμοποιείται ειδικό συντακτικό για την αρχικοποίηση των members
- **Destructor**
  - Καλείται όταν καταστρέφεται το object (όταν βγει εκτός scope είτε από delete)
  - Έχει το ίδιο όνομα με την κλάση με επιπλέον πρόθεμα ~
  - Μια κλάση μπορεί να έχει μόνο ένα destructor και μάλιστα χωρίς ορίσματα
- Αν δεν οριστεί καθόλου constructor ή destructor, ο compiler δημιουργεί τους default που δεν παίρνουν ορίσματα

```
class Vector {
public:
    Vector(void) : x(0), y(0), z(0) { puts("New vec!"); }
    Vector(float x, float y, float z) : x(x), y(y), z(z) {}
    ~Vector() { puts("A vec will die :'("); }
private:
    float x, y, z;
};
```

Constructors / Destructor (2/2)

- Διαφορά μεταξύ malloc/free και new/delete
  - malloc / free
    - Δέσμευση και αποδέσμευση της μνήμης για το στιγμιότυπο
  - new/delete
    - Δέσμευση και αποδέσμευση της μνήμης για το στιγμιότυπο
    - ***Κλήση του constructor και destructor***

```
int main(void) {
    Vector* vec1 = new Vector;  // default constructor is called
    Vector vec2(2,4,1);                    // parametrized constructor is called
    vec1->Add(vec2);              // vec1 + vec2
    if (vec1->IsEqual(vec2))               // vec1 == vec2
        printf("vec = (%f,%f,%f)\n", vec1->GetX(), vec1->GetY(), vec1->GetZ());
    delete vec1;                          // destructor of vec1 object called here
    return 0;
}                                         // destructor of vec2 object called here
```

Default Arguments (1/2)

- Στη C++ μπορούμε να δώσουμε default τιμές στα ορίσματα μια συνάρτησης.
- Αν κατά την κλήση δεν δώσουμε τιμή στο όρισμα αυτό, τότε χρησιμοποιείται η default τιμή
- Μπορούμε να χρησιμοποιήσουμε τις default τιμές σε οποιεσδήποτε συνάρτηση
- Υπάρχει όμως ο εξής **περιορισμός**:
  - Αν ένα formal argument έχει default τιμή, τότε και όλα τα επόμενα ορίσματα πρέπει να έχουν default τιμές.
    - void function(int x, int y=0, int z);//compile error
    - void function(int x, int y, int z = 0); //ok
    - void function(int x, int y=0, int z = 0); //ok

Default Arguments (2/2)

```
class Vector {
public:
   Vector(float x = 0, float y = 0, float z = 0)
       : x(x), y(y), z(z) {}
   ~Vector(void) {}
private:
   float x, y, z;
};
Vector vec1;         // (0,0,0)
Vector vec2(1,1);    // (1,1,0)
Vector vec3(2,1,4); // (2,1,4)
```

```
class Window { public: void SetVisible(bool val=true); };
Window w;
w.SetVisible(false);  //window is set to not visible
w.SetVisible();       //window is set to visible
```

Function Overloading (1/2)

- Είναι ένας μηχανισμός με τον οποίο μπορούμε να ορίσουμε διαφορετικές συναρτήσεις με το ίδιο όνομα
- Μπορούμε να κάνουμε overload οποιαδήποτε συνάρτηση
- Πρέπει όμως οι overloaded συναρτήσεις να έχουμε διαφορετική υπογραφή και μάλιστα να μη διαφέρουν μόνο στον επιστρεφόμενο τύπο
- Χρειάζεται ιδιαίτερη προσοχή όταν χρησιμοποιούμε function overloading και default ορίσματα

Function Overloading (2/2)

```
// Different signatures for all overloads, so everything ok
int    square(int x)    { return x*x; }
float  square(float x)  { return x*x; }
double square(double x) { return x*x; }

// Overloaded function differs only in return value
float  square(float x)  { return x*x; }
double square(float x)  { return x*x; }  //compile error here

// The following will not compile!
void foo (int x);
void foo (int x, int y=0);
foo(3, 5); // ok, calls second version
foo(3);    // ambiguous call: foo(3) or foo(3,0)?

// Neither  will this!
void foo (int x);
void foo (int x=0, int y=0);
foo();     // ok, calls second version
foo(3);    // ambiguous call: foo(3) or foo(3,0)?
```

Operator overloading (1/3)

- Στη C++ μπορούν να υπερφορτωθούν όλοι οι τελεστές, εκτός από τους παρακάτω τέσσερις:
  - **. .* :: ? :**
- Συντακτικό: <return type> operator <op> (<arguments>)
- Μπορούμε να έχουμε 1 ή 2 ορίσματα ανάλογα με τον τύπο του τελεστή (μοναδιαίος ή δυαδικός) και ανάλογα με το αν έχουμε global ή member συνάρτηση (οι member παίρνουν ένα όρισμα λιγότερο)
- Προσοχή στην υπερφόρτωση λογικών τελεστών γιατί «ακυρώνεται» το short-circuit boolean evaluation

Operator overloading (2/3)

```
class Point {
   int x, y;
public:
   Point(int x = 0, int y = 0) : x(x), y(y) {}
   const Point operator=(const Point& p) {  // a=b=c επιτρέπεται
      x = p.x, y=p.y;
      return *this;
   }
   const Point operator+(const Point& p) { return Point(x + p.x, y + p.y); }
};

Point p1(10,20), p2, p3;
p3 = p2 = p1;       // Αυτόματη κλήση του υπερφορτωμένου τελεστή =
p2.operator=(p1);  // 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 ()
}

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
```

Operator overloading (3/3)

- Ιδιαίτερη προσοχή θέλουν οι μοναδιαίοι τελεστές ++ και -- γιατί μπορούν να ερμηνευθούν ως ***prefix*** και ως ***postfix***
- Για να υπάρχει διάκριση μεταξύ τους, ο prefix operator δεν παίρνει παράμετρο, ενώ ο postfix παίρνει μια ***int*** παράμετρο
- Η παράμετρος αυτή δε χρησιμοποιείται **ποτέ** (για αυτό δεν της δίνουμε και όνομα) και παίρνει μια dummy τιμή από τον compiler
- Επίσης ο prefix operator επιστρέφει ***reference*** ενώ ο postfix αντίγραφο

```
class Integer {
  int i;
public:
  Integer(int i = 0) : i(i) {}
  Integer& operator++() {  //prefix operator
      ++i;
      return *this;
  }
  Integer operator++(int) {//postfix operator
      Integer tmp = *this;
      operator++(); //or simply ++i;
      return tmp;
  }
};

Integer i;          // i is now 0
printf("%d", i++);  // prints 0, i is now 1
printf("%d", ++i);     // i is now 2, prints
   2
```

Inline keyword

- Η κλήση συνάρτησης είναι “ακριβή” διαδικασία.
  - Βάζουμε στη στοίβα νέο activation record και κάνουμε jump
- Με το *inline* keyword δίνουμε εντολή στον compiler να αντιγράψει τον κώδικα της συνάρτησης στο σημείο κλήσης αντί να κάνει την κλήση
  - Ο compiler βέβαια ενδέχεται να αγνοήσει την εντολή αυτή!
- Υλοποίηση member function μέσα στο σώμα της κλάσης είναι inline
- Οι υλοποιήσεις αυτές πρέπει να είναι πολύ μικρές για να έχουμε μικρότερα binary αρχεία.

```
inline int square(int x) { return x*x; }
class Vector {
    float x, y, z;
public:
    Vector(float x, float y, float z) : x(x), y(y), z(z) {}
    float GetX(void) { return x; }  // by default treated inline
};
int x = square(2);  // function code copied here so this is equal to: int x = 2*2;
Vector v(0, 1, 2);  x = v.GetX(); // code from Vector::GetX also is copied here
```

Static Keyword

- **Static global variables and functions**
  - Μπορούν να χρησιμοποιηθούν μόνο μέσα στο αρχείο που δηλώνονται (όπως και στη C)
- **Static class member variables**
  - Οι μεταβλητές αυτές είναι κοινές για κάθε στιγμιότυπο της κλάσης και δεν ανήκουν σε κάποιο συγκεκριμένο στιγμιότυπο
  - Πρέπει πάντα να δηλωθούν και σε κάποιο cpp αρχείο
- **Class member functions**
  - Οι συναρτήσεις αυτές μπορούν να κληθούν χωρίς στιγμιότυπο
  - Δεν μπορούν να προσπελάσουν non-static class members και functions
- Προσπελαύνουμε static members και functions με τον operator **::**
  - <class_name>**::**<static_member>
  - <class_name>**::**<static_function>

```
//-- X.h file
Class X {
public:
    X(void) { ++count; }
    ~X()    { --count; }
    static int GetCount(void)
         { return count; }
private:
    static int count;
};
//-- X.cpp file
int X::count = 0;
int main(void) {
    X a, b, c;  // 3 constructor calls
    printf("%d\n", X::GetCount()); // 3
    X* x = new X;  // 1 more ctor call
    printf("%d\n", X::GetCount()); // 4
    delete x;      //dtor call
    printf("%d\n", X::GetCount()); // 3
}
```

Structs vs Classes

- Τα *structs* υπάρχουν στη C++ και συμπεριφέρονται όπως και οι κλάσεις με μόνη διαφορά ότι το default access modifier είναι το public.
- Ένα struct μπορεί να χρησιμοποιηθεί όπως και στη C
- Επιπλέον, μπορούμε να προσθέσουμε και λειτουργικότητα βάζοντας member functions και constructors/destructor.

```
class Vector1 {
   int x, y, z;  // private members
public:
   Vector1(int x, int y, int z) :
      x(x), y(y), z(z) {}
};

struct Vector2 {
   int x, y, z;  // public members
public:
   Vector2(int x, int y, int z);
private:
   int privateData;
};

Vector2 v;
v.x = 5.0;          //ok, public
v.privateData = 0; //error, private
```

Pointers in C++

- Οι pointers στη C++ συμπεριφέρονται ακριβώς όπως και στη C.
- Η μόνη διαφορά είναι ότι η C++ είναι πιο strongly typed από τη C, οπότε χρειάζεται explicit type casting για τις περισσότερες μετατροπές pointer.
- Επίσης συνηθίζεται να χρησιμοποιούμε το 0 αντί για το NULL, χωρίς αυτό να είναι απαραίτητο.

```
Vector vec(3,1,2);//instance
Vector* vecp = 0; //null pointer

//Assign vecp to point to vec
vecp = &vec;

//Assign vecp to point to a
//newly created instance
vecp = new Vector(1, 2, 3);

//Normal use of vecp with the
//member by pointer (->) and
//dereference (*) operators
Vector res = vecp->Add(vec);
res.DotProduct(*vecp);
```

References in C++ (1/2)

- Τα references μοιάζουν με τους pointers με κάποιες εξαιρέσεις:
  - Δεν έχουμε references arithmetic
  - Δεν χρειάζεται dereferencing
  - Αρχικοποίηση κατά τη δήλωση και δε μπορεί να αλλάξει τιμή αργότερα
  - Απαγορεύεται reference ή pointer σε reference
  - Δεν μπορούμε να έχουμε null references
- Call by reference
  - Κανονικά, στη C/C++ οι κλήσεις γίνονται “by value”
  - Συγκεκριμένα όταν περνάμε όρισμα ένα reference δεν το αντιγράφουμε αλλά περνάμε μια αναφορά του και έτσι μπορούμε να το αλλάξουμε

```
Vector vecA(3,5,1), vecB(4,1,2);
Vector& vecr;  //uninitilized, error
Vector& vecr = vecA;   //ok
vecr = vecB;     //vecA = vecB
vecr.Add(vecB);  //vecA + vecB

//-- Call by  reference
void swap(int& x, int& y) {
   int temp = x;
   x = y;
   y = temp;
}
int x = 3, y = 7;
swap(x, y);
printf(“x: %d, y: %d\n”, x, y);
//prints x: 7, y: 3
```

References in C++ (2/2)

- Μια συνάρτηση με const reference ως όρισμα μπορεί να δέχεται προσωρινά αντικείμενα ως ορίσματα
- Τα προσωρινά αντικείμενα δημιουργούνται από const τιμές που επιστρέφονται από συναρτήσεις ή δίνονται κατευθείαν από το χρήστη.
- **Προσοχή** σε references που βγαίνουν εκτός scope
  - Στο διπλανό παράδειγμα, με την κλήση της foo δεσμεύεται στη στοίβα μνήμη για το *Vector vec*
  - Κατά το return επιστρέφουμε το vec αλλά μετά την έξοδο απελευθερώνεται η μνήμη του
  - Τελικά καταλήγουμε με ένα reference στο που δείχνει σε *deallocated memory*

```
void f(int& i) {}
void g(const int& i) {}
f(1); //error const reference not allowed
g(1); // ok
```

```
int f(void) {return 0;}
void g(int& x) {}
void h(const int& x) {}
g(f()); // compile error!
//when f returns, a temporary object is
//created and then passed as an argument
//to g, which cannot accept it, as
//it accepts only ordinary references
h(f()); //ok,h accepts a const reference
```

```
Vector& foo(void) {
  Vector vec(2,4,5);
  return vec;//Compiler warning:returning
}            //address of local variable
Vector& v = foo();
printf("%d", v.x);  //probable crash
```

Copy Constructor

- Ο Copy Constructor είναι ένα ειδικό είδος constructor που παίρνει ως όρισμα ένα const reference σε στιγμιότυπο της κλάσης
- Στη C++ οι κλήσεις των συναρτήσεων γίνονται “by value”.
  - Όταν περνάμε ένα object ως όρισμα σε συνάρτηση, ο compiler το αντιγράφει (shallow copy) και το χρησιμοποιεί μέσα στο scope της
  - Κάποιες φορές το memory copy μπορεί να μην αρκεί γιατί μπορεί να έχουμε πολύπλοκα objects που χρειάζονται deep copy του στιγμιοτύπου
  - Με τον copy constructor χρησιμοποιείται για να κάνουμε override το memory copy

```
class Vector {
   int x, y, z;
public:
   Vector(int x, int y, int z) :
      x(x), y(y), z(z) {}
   Vector(const Vector& v) :
      x(v.x), y(v.y), z(v.z) {}

   Vector* Clone(void) const
     { return new Vector(*this); }
     //*this is a const reference
};

void foo(Vector vec) {}
void bar(const Vector& vec) {}
foo(Vector(3,2,5)); //by value
bar(Vector(3,2,5)); //by reference
```

Const Keyword

- **Const global/member variables**
  - Όπως και στη C, αρχικοποιούνται μόνο μια φορά και δε μπορούν να αλλάξουν
- **Const member functions**
  - Δε μπορεί να αλλάξει class member variables
    - Εκτός κι αν είναι ***mutable***
  - Μπορεί να καλέσει άλλες member functions μόνο αν είναι const
  - Οι static member functions δε μπορούν να δηλωθούν ως const
- Const references σε objects μπορούν να προσπελάσουν μόνο const member functions.
- **Συμβουλή:** Πάντα να ορίζετε τα member functions που δεν αλλάζουν member data ως const

```
class Vector {
   int x, y, z; mutable int totalGets;
public:
    Vector(int x, int y, int z);
    int GetX(void) const {
       ++totalGets; //ok as it is mutable
       return x;
    }
    void SetX(float _x) { x = _x; }
    // Set() can never be const
};
Vector v(4, 2, 5);
Vector& vec = v;
//now using v or vec is exactly the same
const Vector& cvec = vec;
vec.GetX(); //ok, non-const reference
vec.SetX(4);//ok, non-const reference
cvec.GetX();//ok, GetX is const
cvec.SetX(5);//error: cvec is const but
             //SetX is a non const member
```

Οργάνωση σε αρχεία (1/2)

- Τα αρχεία της C++ συνήθως οργανώνονται με τον εξής τρόπο:
  - Ορίζουμε τις κλάσεις μέσα σε header files (.h) με το όνομα της κλάσης
  - Στη συνέχεια γράφουμε τις υλοποιήσεις των class member functions σε source files (.cpp) με το ίδιο όνομα όπως το header file
    - Όπως είπαμε, οι υλοποιήσεις που τυχόν αφήσουμε στο header file θεωρούνται inline από τον compiler
  - Το source file πρέπει να κάνει include το header file
  - Κάθε άλλο αρχείο στο project μας που χρειάζεται να χρησιμοποιήσει την κλάση μας θα πρέπει να κάνει include το header file

Οργάνωση σε αρχεία (2/2)

Header file: ***Vector.h***

```
#ifndef VECTOR_H
#define VECTOR_H

class Vector {
public:
  Vector(int x, int y, int z);
  ~Vector();
  int GetX(void) const;
  static int Instances (void);
private:
  int x, y, z;
  static int instances;
};

#endif
```

Source file: ***Vector.cpp***

```
#include “Vector.h”

int Vector::instances = 0;

Vector::Vector(int x,int y,int z)

  :x(x),y(y),z(z){ ++instances; }

Vector::~Vector(){ --instances; }

int Vector::GetX(void) const
    { return x; }

int Vector::Instances(void)
    { return instances; }
```