ΑΝΑΚΕΦΑΛΑΙΩΣΗ

26 Οκτωβρίου 2011

Αντικειμενοστρεφής Προγραμματισμός

- Ένα νέο προγραμματιστικό μοντέλο (paradigm) το οποίο στηρίζεται στις κλάσεις και τα αντικείμενα.
- Κλάση: Μια αφηρημένη οντότητα με χαρακτηριστικά (μεταβλητές) και συμπεριφορά (μεθόδους).
- Αντικείμενο: Ένα στιγμιότυπο (instance) της κλάσης το οποίο έχει μια κατάσταση (οι τιμές των μεταβλητών) και εκτελεί ενέργειες (καλεί τις μεθόδους)

Ορισμός κλάσης

```
class MyClass
{
private:
    int privateData;
    int privateMethod();
public:
    int publicMethod();
};
```

Μην ξεχνάμε το ερωτηματικό!

Δεσμευμένες λέξεις με κόκκινο.

Δηλώσεις private δεδομένων και μεθόδων δεν είναι προσβάσιμα εκτός της κλάσης.

Δηλώσεις public δεδομένων και μεθόδων Προσβάσιμα εκτός της κλάσης.

Default: όλα είναι private

Καλός αντικειμενοστραφής σχεδιασμός: private data, public methods

```
int MyClass::publicMethod()
{
    privateData = 1;
    /* more code here */
}
```

Ορίζει το namespace της κλάσης

Ορισμός μεθόδων.

Μέσα στο namespace της κλάσης οι μεταβλητές μέλη είναι προσβασιμες

Παράδειγμα κλάσης

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

class Human {
private:
   int height;
   int age;

public:
  void Ages();
  void IsBorn();
  void Grows(int inc);
};

void Human::Ages(){
 age += 1;
}

void Human::IsBorn(){
  height = 40;
  age = 0;
}

void Human::Grows(int inc){
  height += inc;
}
```

Αντικείμενα

```
int main()
{
    MyClass myClassInstance;
    ...
    myClassInstance.publicMethod();
    ...
}
```

Ορισμός του αντικειμένου

Κλήση της public μεθόδου.

```
int main()
{
   Human peter;
   peter.IsBorn();
   peter.Ages();
   peter.Grows(10);
}
```

Πλεονεκτήματα

- Το αντικειμενοστρεφές μοντέλο αντιστοιχεί καλυτέρα με τον τρόπο που βλέπουμε τον κόσμο.
- Ο κώδικας που παράγεται είναι πιο εύκολο να διαβαστεί και να κατανοηθεί.
- Encapsulation και data hiding
  - Τα δεδομένα και οι συναρτήσεις που τα αλλάζουν ομαδοποιούνται μέσα στην κλάση.
  - Πιο εύκολη συντήρηση του κώδικα.
- Κληρονομικότητα
  - Ιεραρχία κλάσεων που μοιράζονται χαρακτηριστικά.
- Πιο εύκολη επαναχρησιμοποίηση του κώδικα.

Άσκηση

- Θέλουμε να φτιάξουμε ένα «παιχνίδι» το οποίο κάνει το εξής:
  - Έχουμε δυο αυτοκίνητα που ξεκινάνε από το σημείο 0 της ευθείας και τυχαία πάνω στις τιμές των ακεραίων.
    - Σε κάθε κίνηση διαλέγουν τυχαία να πάνε αριστερά, δεξιά, ή να μείνουν στο ίδιο σημείο.
  - Το παιχνίδι σταματάει όταν τα δυο αυτοκίνητα συγκρουστούν.

```
#include <iostream>

int main()
{
  int collision = 0;
  int counter = 0;

  // Αρχικοποίηση αυτοκινήτων

  while(!collision){
    // Τα αυτοκίνητα μετακινούνται
    // check for collision
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves"
      // τύπωσε την θέση της σύγκρουσης
}
```

Τι κλάσεις και αντικείμενα πρέπει να ορίσουμε?
Ποια θα είναι τα πεδία και ποιες οι μέθοδοι?

Class Car

```
class Car
{

private:
   int _pos;

public:
   void InitializePosition();
   void Move();
   int GetPosition();
};
```

Methods

```
void Car::InitializePosition()
{
   _pos = 0;
}

void Car::Move()
{
   _pos += floor((double(rand())/RAND_MAX)*3)-1;
}

int Car::GetPosition()
{
   return _pos;
}
```

```
#include <cmath>
#include <cstdlib>
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;

  // Αρχικοποίηση αυτοκινήτων

  while(!collision){
    // Τα αυτοκίνητα μετακινούνται
    // check for collision
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves"
       // τύπωσε την θέση της σύγκρουσης
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  // Αρχικοποίηση αυτοκινήτων

  while(!collision){
    // Τα αυτοκίνητα μετακινούνται
    // check for collision
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves"
       // τύπωσε την θέση της σύγκρουσης
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision){
    // Τα αυτοκίνητα μετακινούνται
    // check for collision
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves"
       // τύπωσε την θέση της σύγκρουσης
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision){
    carX.Move();
    carY.Move();
    collision = (carX.GetPosition() == carY.GetPosition());
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves"
       // τύπωσε την θέση της σύγκρουσης
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision){
    carX.Move();
    carY.Move();
    collision = (carX.GetPosition() == carY.GetPosition());
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves "
       << "at position " << carX.GetPosition() << endl;
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision){
    carX.Move();
    carY.Move();
    collision = (carX.GetPosition() == carY.GetPosition());
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves "
       << "at position " << carX.GetPosition() << endl;
}
```

Τι γίνεται αν θέλουμε να αλλάξουμε τον ορισμό της σύγκρουσης?

Class Car

```
class Car
{

private:
   int _pos;

public:
   void InitializePosition();
   void Move();
   int GetPosition();
   int Collide(Car)
};
```

```
int Car::Collide(Car other)
{
    return (_pos == other.GetPosition());
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int counter = 0;
  Car carX;
  Car carY;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision){
    carX.Move();
    carY.Move();
    collision = carX.Collide(carY);
    counter ++;
  }

  cout << "Cars collided after "  << counter <<" moves "
       << "at position " << carX.GetPosition() << endl;
}
```

Άσκηση

- Εκτός από τα αυτοκίνητα έχουμε και μία νάρκη, η οποία εμφανίζεται σε τυχαία σημεία στο διάστημα [-5,5]
- Το παιχνίδι τελειώνει είτε όταν συγκρουστούν τα οχήματα, ή όταν κάποιο από αυτά ενεργοποιήσει τη νάρκη.

Τι νέες κλάσεις χρειαζόμαστε να ορίσουμε?

Class Mine

```
class Mine
{
private:
  int _pos;

public:
  void Activate();
  int GetPosition();
};

void Mine::Activate()
{
  _pos = floor((double(rand())/RAND_MAX)*11)-5;
  _pos = floor((double(rand())/RAND_MAX)*11)-5;
}

int Mine::GetPosition()
{
  return _pos;
}
```

Class Car

```
class Car
{

private:
   int _pos;

public:
   void InitializePosition();
   void Move();
   int GetPosition();
   int Collide(Car)
   int Detonate(Mine);
};
```

```
int Car::Detonate(Mine someMine)
{
    return (_pos == someMine.GetPosition());
}
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  int collision = 0;
  int explosion = 0;
  int counter = 0;
  Car carX;
  Car carY;
  Mine randomMine;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision && !explosion){
    carX.Move();
    carY.Move();
    randomMine.Activate();
    collision = carX.Collide(carY);
    explosion = carX.detonate(randomMine) || carY.detonate(randomMine);
    counter ++;
  }
  if (collision){
     cout << "Cars collided after "  << counter <<" moves "
          << "at position " << carX.GetPosition() << endl;
  }
  if (explosion){
     cout << "Mine exploded after " << counter << " moves, "
          << "at position " << randomMine.GetPosition() << endl;
  }
}
```

C ΚΑΙ C++ ΟΜΟΙΟΤΗΤΕΣ ΚΑΙ ΔΙΑΦΟΡΕΣ

Χαρακτηριστικά της C++

ΕΙΣΟΔΟΣ/ΕΞΟΔΟΣ

Είσοδος / Έξοδος δεδομένων

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

main(){
     cout << “Hello world!\n”;
}
```

Ισοδύναμα:

```
cout << “Hello” << “world!\n”;
```

Είσοδος / Έξοδος δεδομένων

- Στη C η εγγραφή στην οθόνη και η ανάγνωση από το πληκτρολόγιο γίνεται με τα λεγόμενα ρεύματα (streams).
  - **`stdin`**, **`stdout`**
- Ότι θέλουμε να εκτυπώσουμε στην οθόνη με **`printf`** περνά από το **`stdout`** και εν συνεχεία μέσω του λειτουργικού εμφανίζεται στην οθόνη
- Ότι θέλουμε να διαβάσουμε περνά από το πληκτρολόγιο μέσω του λειτουργικού στο **`stdin`** και εν συνεχεία το διαβάζουμε με **`scanf`**.

```
printf("%d", i); <=> fprintf(stdout, "%d", i);
scanf("%d", &i); <=> fscanf(stdin, "%d", &i);
```

Είσοδος / Έξοδος δεδομένων

- Στη C++ ισχύει ότι και στην C, αλλά πέραν των γνωστών συναρτήσεων εισόδου εξόδου υπάρχουν πιο ευέλικτοι τρόποι.
- Τα ρεύματα που αντιστοιχούν στα **stdout, stdin** είναι τα **cout** και **cin**.
  - τα stdin, stdout είναι struct FILE *
  - τα **cin, cout** είναι αντικείμενα τύπου **istream** και **ostream** αντίστοιχα και προσφέρουν και συναρτήσεις που μπορώ να καλέσω σε αυτά.
- αντί για συναρτήσεις όπως οι printf, fprintf, scanf, fscanf έχουμε τους τελεστές <<, >> που εισάγουν δεδομένα στο **cout** και εξάγουν δεδομένα από το **cin**

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";

}
```

Είσοδος / Έξοδος δεδομένων

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

int main(){
  int i;
  float f;

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

Είσοδος / Έξοδος δεδομένων

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

int main(){
  int i;
  float f;

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

Χειριστές (manipulators): εντολές προς το ρεύμα εξόδου

Ο χειριστής **endl** έχει το ίδιο αποτέλεσμα με τον χαρακτήρα **\n**
Προκαλεί όμως και εκκένωση (flush) του output buffer.

Ο χειριστής setw

- Χρησιμοποιείται για να καθορίσει το πλάτος της εξόδου.
  - Η στοίχιση γίνεται στα δεξιά

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

int main(){
   cout << ":" << setw(10) << "alpha" << endl
        << ":" << setw(10) << "beta" << endl
        << ":" << setw(10) << "gamma" << endl;
}
```

```
:     alpha
:      beta
:     gamma
```

ΒΑΣΙΚΟΙ ΤΥΠΟΙ, ΜΕΤΑΒΛΗΤΕΣ

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 boolX;                  //1 byte
boolX = true; boolX = false;
```

Οι μεταβλητές τύπου bool είναι βολικές για να ξεχωρίζουμε τις λογικές μεταβλητές. Κατά τα αλλά συμπεριφέρονται ακριβώς όπως πριν.

Class Car

```
class Car
{

private:
   int _pos;

public:
   void InitializePosition();
   void Move();
   int GetPosition();
   bool Collide(Car)
   bool Detonate(Mine);
};
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  bool collision = 0;
  bool explosion = 0;
  int counter = 0;
  Car carX;
  Car carY;
  Mine randomMine;

  carX.InitializePosition();
  carY.InitializePosition();

  while(!collision && !explosion){
    carX.Move();
    carY.Move();
    randomMine.Activate();
    collision = carX.Collide(carY);
    explosion = carX.detonate(randomMine) || carY.detonate(randomMine);
    counter ++;
  }
  if (collision){
     cout << "Cars collided after "  << counter <<" moves "
          << "at position " << carX.GetPosition() << endl;
  }
  if (explosion){
     cout << "Mine exploded after " << counter << " moves, "
          << "at position " << randomMine.GetPosition() << endl;
  }
}
```

It would work even if we left these two variables to be `int`

Πίνακες

Integers

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

C Strings

```
#include <cstring>
char firstName[6];
main(){
    cin >> firstName; // get the string from input
    strcpy (firstName, "eddie"); // copy a value
    cout << firstName; // output string
}
```

Δηλώσεις Μεταβλητών

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

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

Βολικό γιατί μπορείς να δηλώσεις μια μεταβλητή όταν την χρειάζεσαι

```
#include <iostream>
#include <cmath>
#include <cstdlib>

/* Car class definition, method definitions */

int main()
{
  Car carX;
  carX.InitializePosition();
  Car carY;
  carY.InitializePosition();
  Mine randomMine;

  bool collision = 0;
  bool explosion = 0;
  int counter = 0;
  while(!collision && !explosion){
    carX.Move();
    carY.Move();
    randomMine.Activate();
    collision = carX.Collide(carY);
    explosion = carX.detonate(randomMine) || carY.detonate(randomMine);
    counter ++;
  }
  if (collision){
     cout << "Cars collided after "  << counter <<" moves "
          << "at position " << carX.GetPosition() << endl;
  }
  if (explosion){
     cout << "Mine exploded after " << counter << " moves, "
          << "at position " << randomMine.GetPosition() << endl;
  }
}
```

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 μεταβλητή `i` ζει μονό μέσα στο for loop
Ο πιο συχνός τρόπος δημιουργίας iterators

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}; // creates a table of size 4
int myArray[10] = {1,2,3,4};
// creates a table of size 10 and fills the rest with 0s
```

```
//size: 6, length: 6 -remember the last '\0'
char firstName[] = "eddie";
//size: 50, length: 6
char name[50] = "eddie";
strcpy(name,"eddie vedder"); // fits in there
```

Πολυδιάστατοι Πίνακες

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

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

const

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

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

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

Διαφορές const και #define

- 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`**.

ΒΑΣΙΚΕΣ ΕΝΤΟΛΕΣ

Εντολή if … else …

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

Λογικοί τελεστές για σύνθετες συνθήκες:

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

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

NOT: `!(cond1)`

Προσοχή: `= vs. ==`

Είναι καλό να χρησιμοποιείτε πάντα { } ακόμη και για μία μόνο εντολή

Εντολή Switch

```
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 **break** μας βγάζει έξω από το block στο οποίο είμαστε.

Το **default** καλύπτει την περίπτωση που χάνουμε κάποια case

Εντολές βρόγχων (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;
}
```

break και continue

- Χρήσιμα για τον έλεγχο ροής σε loops.

```
// 100 iterations or a collision/explosion
for(int i = 0; i < 100; i++){
    carX.Move();
    carY.Move();
    randomMine.Activate();
    collision = carX.Collide(carY);
    explosion = carX.detonate(randomMine)
                || carY.detonate(randomMine);
    counter ++;
    if (collision || explosion){
       break;
    }
  }
// cout << i; will give an error
```

break και continue

Αγνόησε συγκρούσεις ή εκρήξεις που γίνονται στο σημείο 0

```
while(!collision && !explosion){
    carX.Move();
    carY.Move();
    if (carX.GetPosition() == 0 || carX.GetPosition() == 0){
       continue;
    }
    randomMine.Activate();
    collision = carX.Collide(carY);
    explosion = carX.detonate(randomMine)
                || carY.detonate(randomMine);
    counter ++;
  }
```

break και continue

Τύπωσε μόνο τους αριθμούς που δεν διαιρούνται με το 3

```
for (int i = 0; i < 10; i ++){
    if (i%3 == 0){
      continue;
    }
    cout << i;
}
```

ΣΥΝΑΡΤΗΣΕΙΣ NAMESPACES

Συναρτήσεις

```
#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
         area = width * height / 2.0;
         return area;
}
```

Δηλωση Συνάρτησης

Ορίσματα Συνάρτησης

Επιστρεφόμενη τιμή Συνάρτησης

Χώροι ονομάτων (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 έχουμε κάποιες χρήσιμες συναρτήσεις που πήραμε από κάποιον άλλον.

```
…….

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

…….
int f(){return 1973}
```

Παράδειγμα (συνέχεια)

```
#include <iostream>
#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 << "English(sqr ml)" << squareArea(2.0);
   cout << f();
}
```

Compiler Error!!!

error: redefinition of `float squareArea(float)'

Παράδειγμα (συνέχεια)

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

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

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

main (){
   cout << "English(sqr ml)" << squareAreaMiles(2.0);
   cout << f();
}
```

Όχι καλή λύση, μπερδεμένος κώδικας, πιθανά λάθη

Παράδειγμα

- Στο αρχείο util.h έχουμε κάποιες χρήσιμες συναρτήσεις που πήραμε από κάποιον άλλον.

```
namespace PublicUtils
{
    ……….

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

    ……….
    int f(){return 1973}
}
```

Παράδειγμα (συνέχεια)

```
#include <iostream>
#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 << "English (sqr ml)" << Local::squareArea(2.0);
   cout << PublicUtils::f();
}
```

Παράδειγμα (συνέχεια)

```
#include <iostream>
#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);
   }
}
using namespace Local;
main (){
    cout << "English (sqr ml)" << squareArea(2.0);
    cout << PublicUtils::f();
}
```

Παράδειγμα (συνέχεια)

```
#include <iostream>
#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);
   }
}
using namespace Local;
main (){
   cout << "English (sqr ml) " << squareArea(2.0);
   cout << "Normal (sqr klm) "
        << PublicUtils::sqaureArea(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…

ΔΕΙΚΤΕΣ (POINTERS)
ΑΝΑΦΟΡΕΣ (REFERENCES)

Δείκτες (Pointers)

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

```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```

Δείκτες (Pointers)

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

```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```

Μνήμη

0x1000	
0x1001	
0x1002	
0x1003	
0x1004	
0x1005	
0x1006	
0x1007	

Δείκτες (Pointers)

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

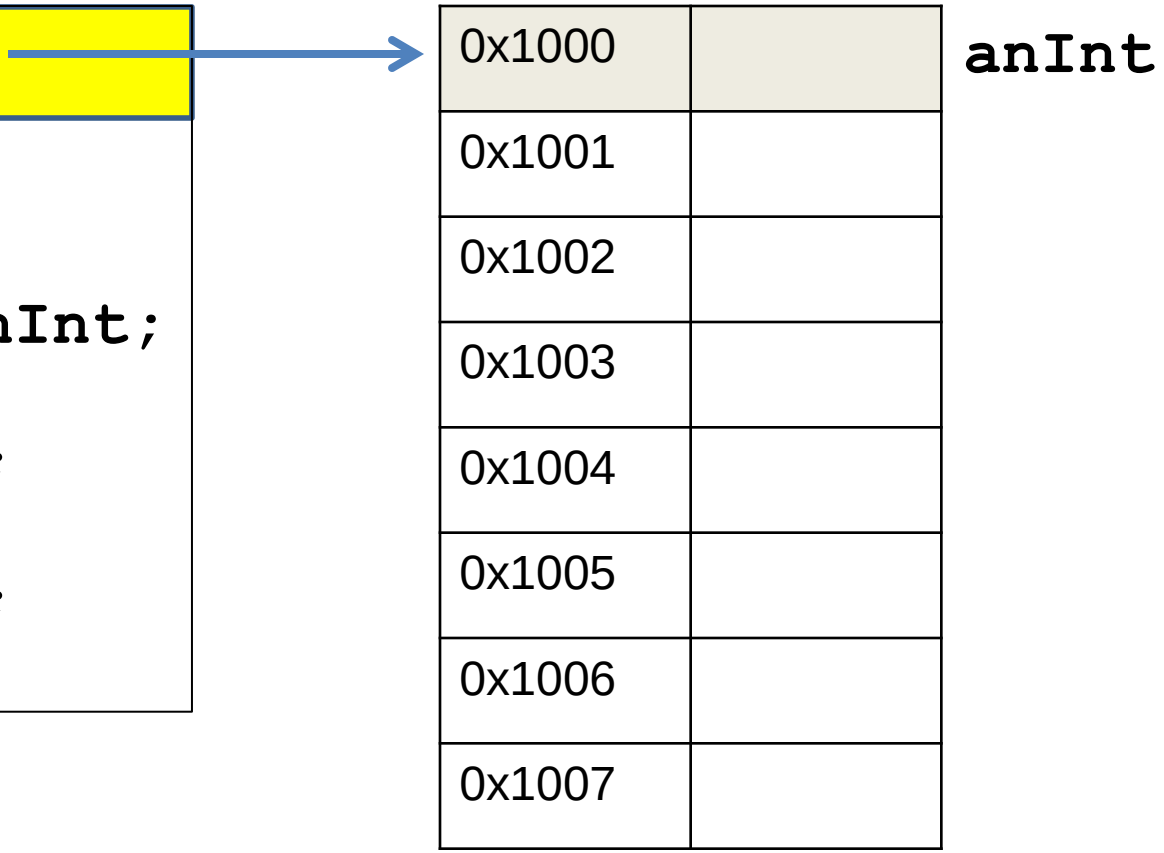
```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```



Δείκτες (Pointers)

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

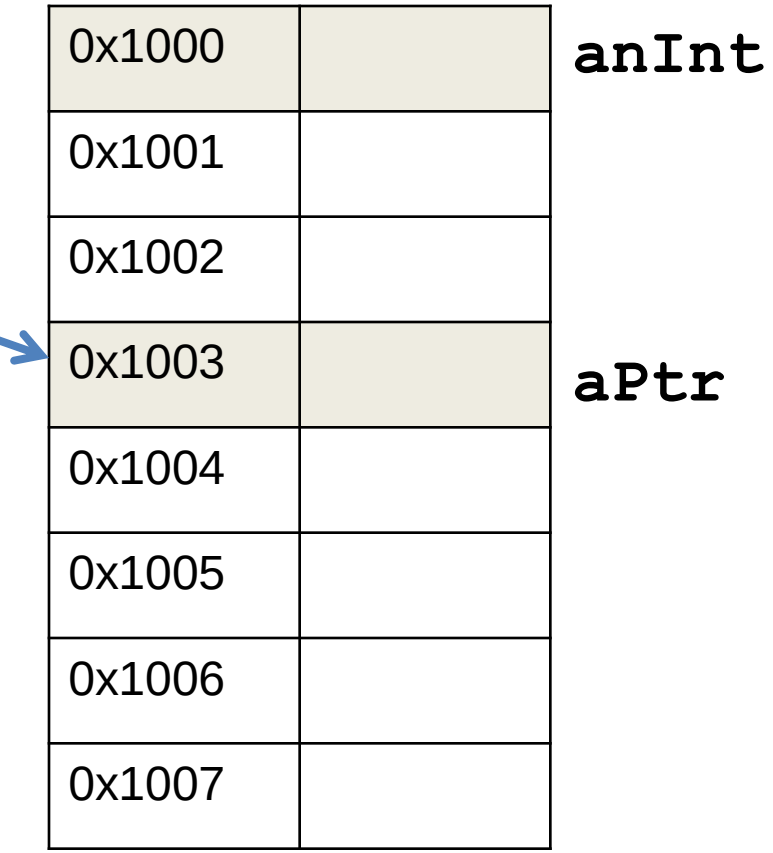
```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```



Δείκτες (Pointers)

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

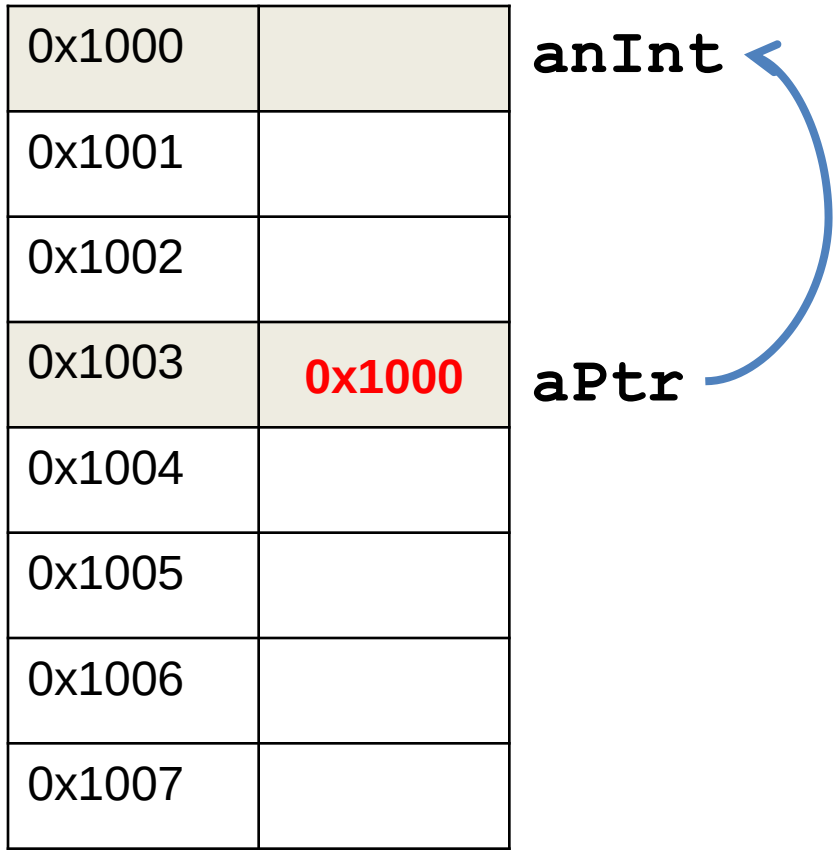
```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```



Δείκτες (Pointers)

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

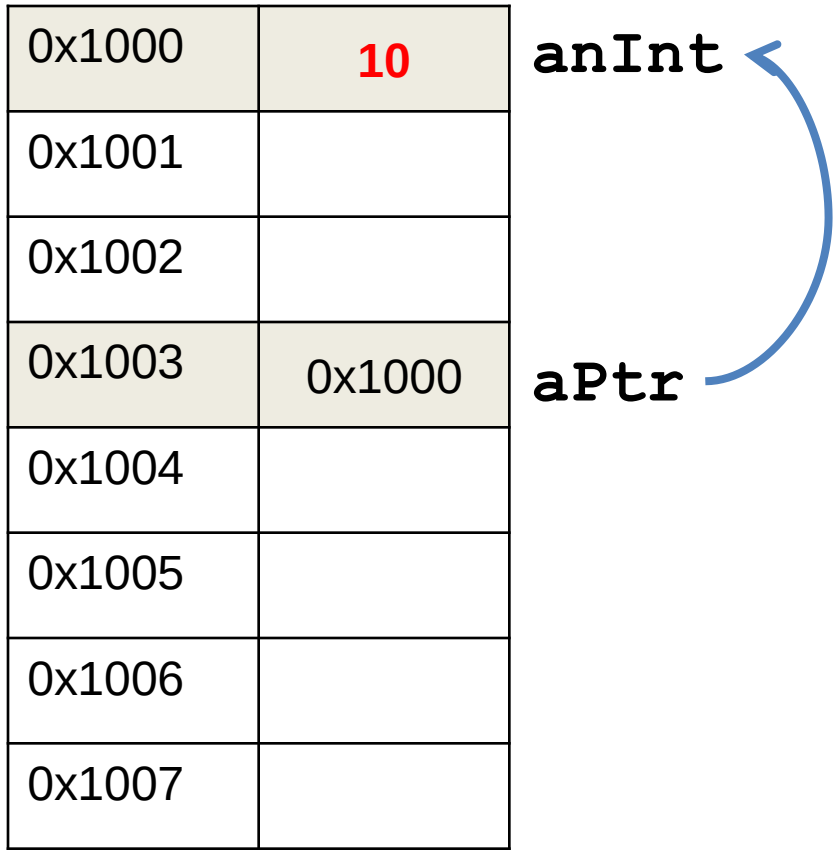
```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```



Δείκτες (Pointers)

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

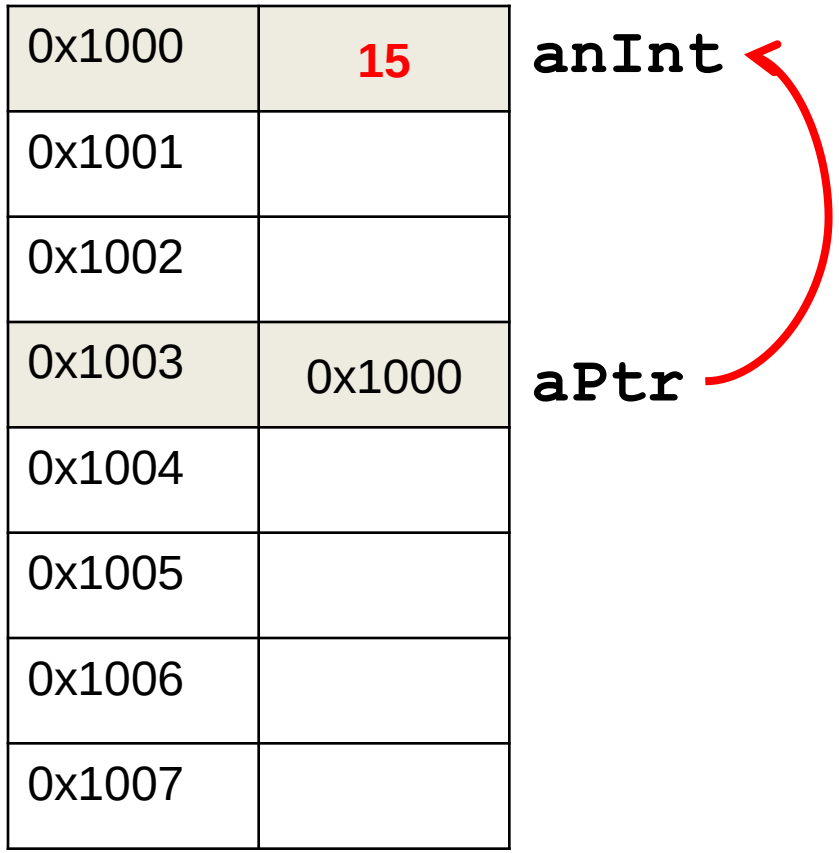
```
int anInt;

int *aPtr;

aPtr = & anInt;

anInt = 10;

*aPtr = 15;
```



Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;
```

0x1000	
0x1001	
0x1002	
0x1003	
0x1004	
0x1005	
0x1006	
0x1007	

Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;
```

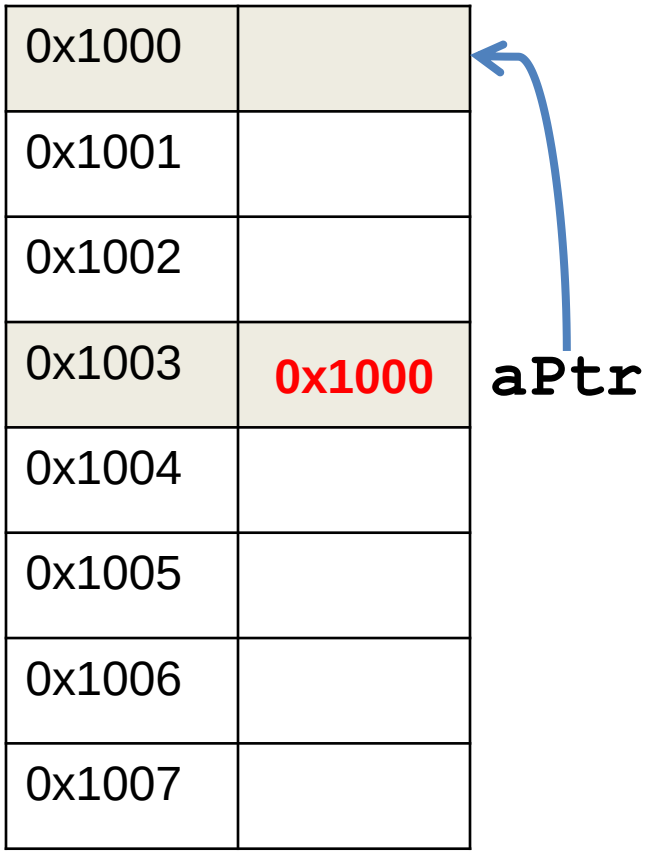
0x1000		
0x1001		
0x1002		
0x1003	0x0000	aPtr
0x1004		
0x1005		
0x1006		
0x1007		

Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;
```

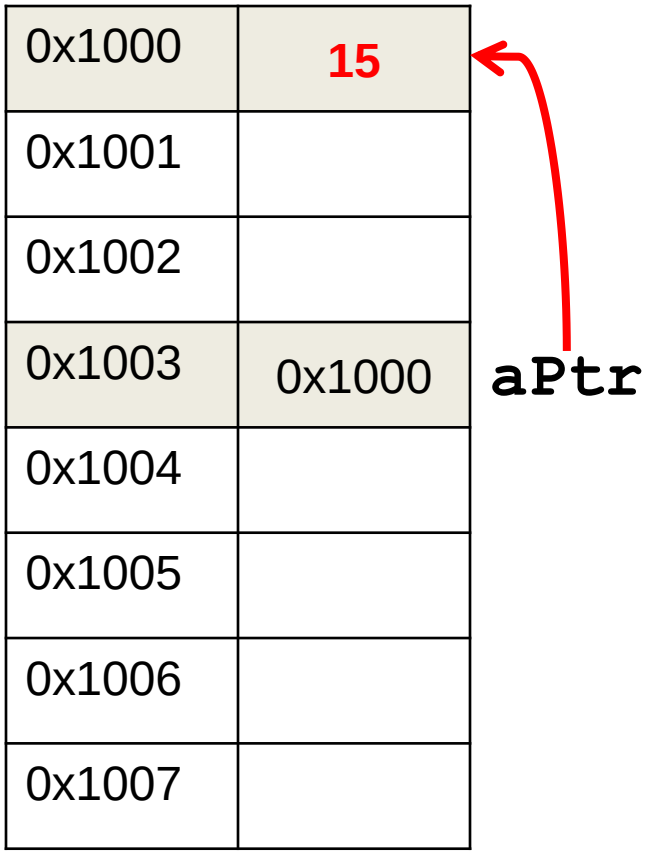


Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;
```



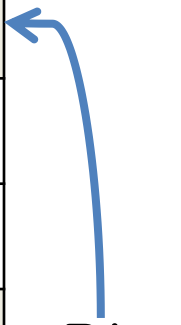
Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;
```

0x1000	15
0x1001	
0x1002	
0x1003	0x1000
0x1004	
0x1005	
0x1006	
0x1007	



Στη C++

```
int *aPtr = new int;

*aPtr = 15;
```

Δείκτες – Αποδέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;

free(aPtr);
```

0x1000	15
0x1001	
0x1002	
0x1003	0x0000
0x1004	
0x1005	
0x1006	
0x1007	

aPtr

Δείκτες – Αποδέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;

free(aPtr);
```

Στη C++

```
int *aPtr = new int;

*aPtr = 15;

delete aPtr;
```

0x1000	15
0x1001	
0x1002	
0x1003	0x0000
0x1004	
0x1005	
0x1006	
0x1007	

aPtr

Δείκτες – Αποδέσμευση Μνήμης

```
int *aPtr = null;

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

*aPtr = 15;

int *bPtr = aPtr;

free(aPtr);
```

0x1000	15	
0x1001		
0x1002		
0x1003	0x0000	aPtr
0x1004		
0x1005	0x1000	bPtr
0x1006		
0x1007		

Προσοχή! Η θέση της μνήμης στην οποία έδινε η μεταβλητή aPtr μπορεί να συνεχίσει να έχει την τιμή 15, και αν κάποιος άλλος δείκτης (bPtr) έδειχνε σε αυτή τη θέση να φαίνεται ότι μπορείτε να τη διαβάσετε. Η θέση μνήμης δεν είναι διαθέσιμη όμως και θα προκαλέσει προβλήματα, το πιο πιθανό segmentation fault.

Δείκτες – Δέσμευση Μνήμης

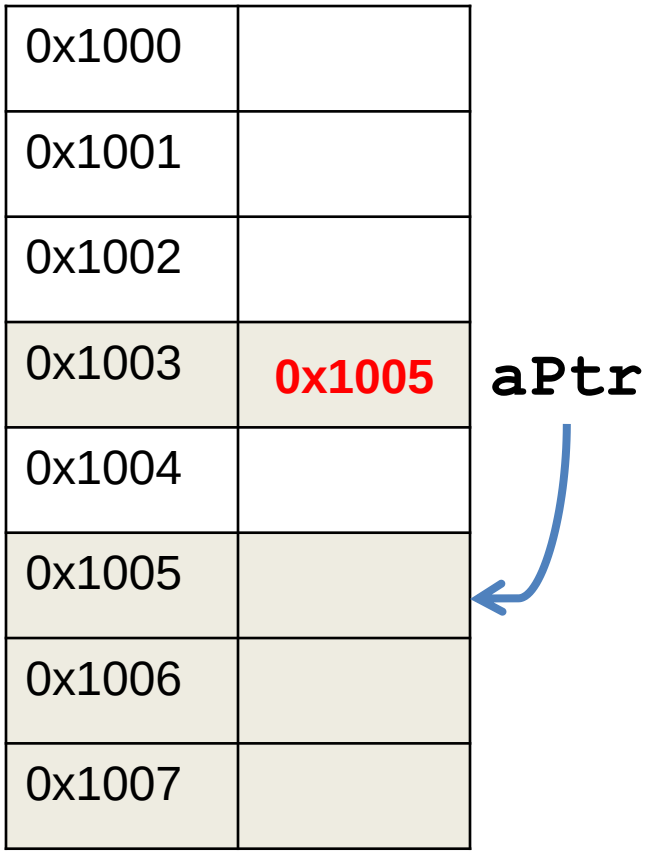
```
int *aPtr = null;

aPtr = (int *)malloc(3*sizeof(int));

*aPtr = 15;

*(aPtr + 2) =17;

free(aPtr)
```



Δείκτες – Δέσμευση Μνήμης

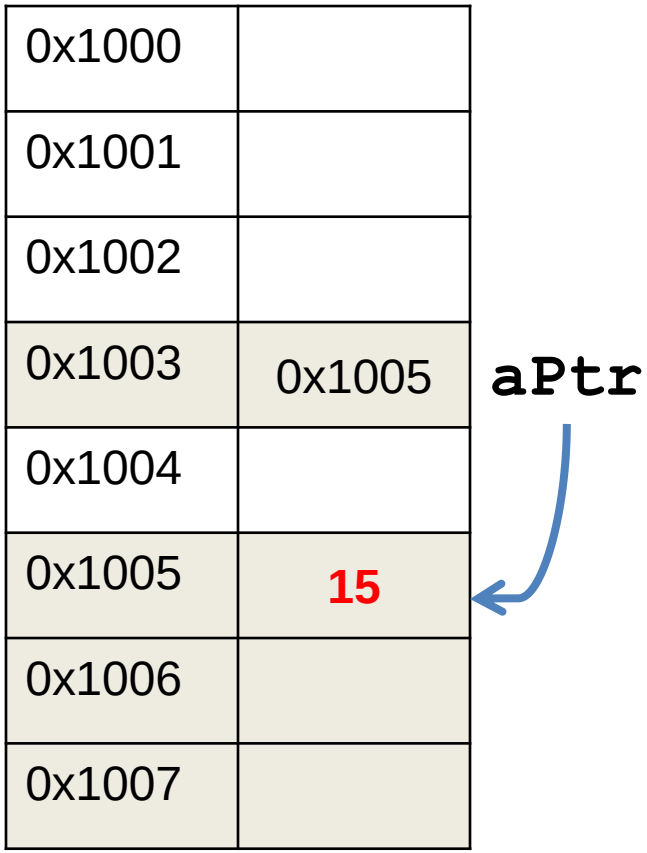
```
int *aPtr = null;

aPtr = (int *)malloc(3*sizeof(int));

*aPtr = 15; ⇔ aPtr[0] = 15;

*(aPtr + 2) =17;

free(aPtr)
```



Δείκτες – Δέσμευση Μνήμης

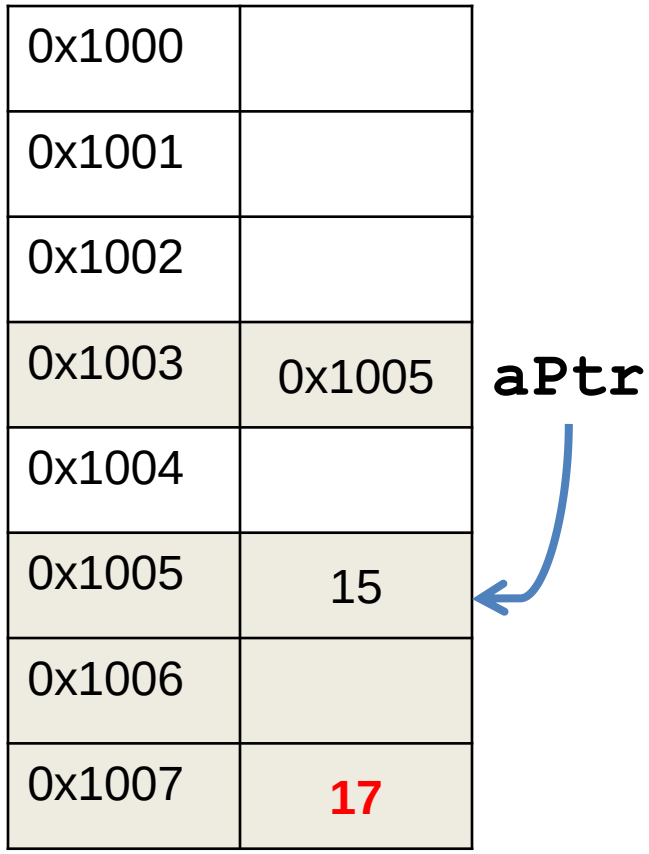
```
int *aPtr = null;

aPtr = (int *)malloc(3*sizeof(int));

*aPtr = 15; ⇔ aPtr[0] = 15;

*(aPtr + 2) =17; ⇔ aPtr[2] = 17;

free(aPtr)
```



Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

aPtr = (int *)malloc(3*sizeof(int));

*aPtr = 15; ⇔ aPtr[0] = 15;

*(aPtr + 2) =17; ⇔ aPtr[2] = 17;

free(aPtr)
```

0x1000		
0x1001		
0x1002		
0x1003	0x0000	aPtr
0x1004		
0x1005	15	
0x1006		
0x1007	17	

Δείκτες – Δέσμευση Μνήμης

```
int *aPtr = null;

aPtr = (int *)malloc(3*sizeof(int));

*aPtr = 15; ⇔ aPtr[0] = 15;

*(aPtr + 2) =17; ⇔ aPtr[2] = 17;

free(aPtr)
```

0x1000	
0x1001	
0x1002	
0x1003	0x0000
0x1004	
0x1005	15
0x1006	
0x1007	17

aPtr

Στη C++

```
int *aPtr = new int[3];

*aPtr = 15;
*(aPtr + 2) =17;

delete [] aPtr;
```

Αναφορές (References)

- Μια αναφορά σε μια μεταβλητή είναι σαν ένα συνώνυμο για τη μεταβλητή.
- Προσοχή: ΔΕΝ πρόκειται για pointers (αν και υπάρχει μια σχέση μεταξύ των δύο)

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

Η εντολή:
`int &wrongRef;`
θα δώσει compile error!

- Έτσι δηλώνουμε αναφορές
- κατά τη δήλωση πρέπει ΠΑΝΤΑ 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:

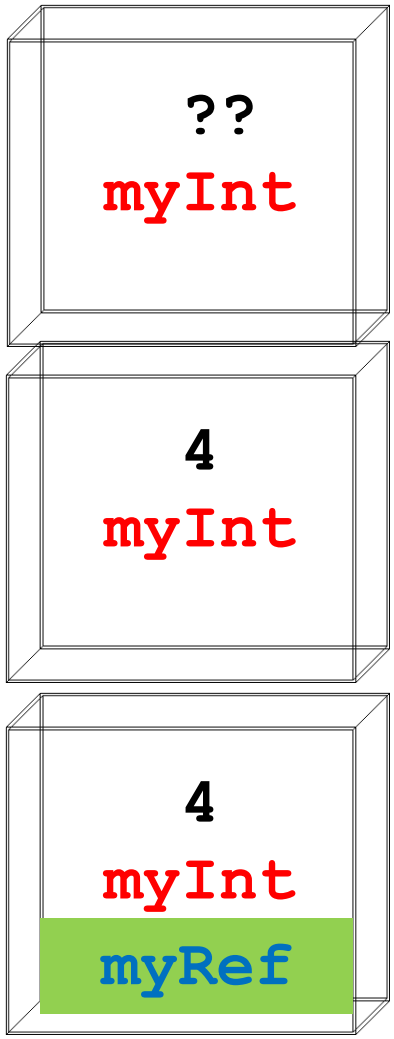
```
myInt++;
myRef++;
```

Αναφορές

```
int myInt;
```

```
myInt = 4;
```

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



Τι κρύβουν οι αναφορές

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

```
int * const myRef_p = &myInt;
```

Το = δεν είναι ανάθεση, δημιουργεί την αναφορά.
Δεν μπορούμε να ξαναναθέσουμε την αναφορά

```
myInt = 8;
```

```
myInt = 8;
```

```
myRef = 7;
```

```
*myRef_p = 7;
```

Οι αναφορές χρησιμοποιούν τους δείκτες για να αναφερθούν σε μια μεταβλητή αλλά δεν είναι σαν δείκτες γιατί αναφέρονται μόνιμα σε μια θέση μνήμης.

Τι κρύβουν οι αναφορές

```
#include <iostream>


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

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

refA = b;
cout << a << endl;


}
```

```
#include <iostream>


main() {
int a = 1;
int b = 3;
int * const refA_p = &a;

cout << *refA_p << endl;
cout << a << endl;

*refA_p = b;
cout << a << endl;


}
```

Σωστό ή Λάθος?

```
main(){
 int x = 1;      // Σωστό.
 int *p = new int;    // Σωστό. Ορίζει ένα νέο δείκτη σε int, και δεσμεύει μνήμη γι αυτόν.
 p = x;          // Λάθος! Το x είναι int και όχι pointer σε int
 *p = x;         // Σωστό. Η διεύθυνση στην οποία δείχνει ο p παίρνει την τιμή του x, 1.
                 // Ο p εξακολουθεί να δείχνει στην θέση μνήμης που δεσμεύσαμε αρχικά
 p = &x;         // Σωστό. Ο pointer p  τώρα δείχνει στην διεύθυνση του x
 int *p2;        // Σωστό. Ορισμός pointer σε int. Δεν χρειάζεται αρχικοποίηση. Δεν υπάρχει
                 // πρόβλημα που ο ορισμός είναι στη μέση του κώδικα και όχι στην αρχή.
 int *q = x;     // Λάθος! Σε αυτό το σημείο ορίζουμε τον δείκτη q και πρέπει είτε να
                 // δεσμεύσουμε μνήμη, είτε να του αναθέσουμε μια διεύθυνση.
 int &a;         // Λάθος! Δήλωση αναφοράς χωρίς αρχικοποίηση και σύνδεση με μεταβλητή.
 int &b = p;     // Λάθος! Η αναφορά είναι τύπου int  και την συνδέουμε με δείκτη σε int.
 int &c = x;     // Σωστό. Η αναφορά c συνδέθηκε με την μεταβλητή x.
 int* &d = p;    // Σωστό. Η αναφορά τύπου pointer σε int  συνδέεται με ένα pointer σε int.
 cout << *c;     // Λάθος! Το ίδιο σαν να ζητάμε *x. Το x δεν είναι pointer.
 cout << c;      // Σωστό. Θα τυπωθεί η τιμή του x.
 cout << *d;     // Σωστό. Θα τυπωθεί το περιεχόμενο της διεύθυνσης που δείχνει ο pointer p.
}
```

Που χρησιμεύουν?

Οι δηλώσεις αναφορών που παίζουν το ρόλο συνώνυμων μιας μεταβλητής σε ένα πρόγραμμα δεν έχουν πολύ μεγάλη χρησιμότητα όπως 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 δια αναφοράς.

Στην περίπτωση που χρησιμοποιήσουμε αναφορές σαν ορίσματα σε συναρτήσεις το πέρασμα δια αναφοράς γίνεται ακόμα πιο απλό.