ΣΤΟΙΧΕΙΑ ΤΗΣ ΓΛΩΣΣΑΣ C++

Constructors, Destructors, Pointers

IO Streams, File Streams

CONSTRUCTORS DESTRUCTORS

Η κλάση myString

```
class myString
{
private:
   char s[100];
public:
   char *GetString();
   void SetString(char const *);
};

char * myString::GetString()
{
   return s;
}

void myString::SetString(char const *sNew)
{
   strcpy(s,sNew);
}
```

Constructors

- Μια μέθοδος που ο μόνος ρόλος της ειναι να κατασκευάζει (construct) και να αρχικοποιεί το αντικείμενο.
- Συντακτικό:
  - `<classname>()`
  - Μπορεί να έχει ορίσματα.
  - ΔΕΝ εχει τυπο επιστροφής – ΟΥΤΕ void.
- Υλοποίηση:
  - `<classname>::<classname>()`

- Η συνάρτηση καλείται αυτόματα με την δημιουργία του αντικειμένου.
  - Είτε στη δήλωση του αντικειμένου.
  - Είτε δημιουργία με new όπου δεσμεύουμε μνήμη για ένα αντικείμενο.

myString

```
class myString
{
private:
   char *s;
public:
   myString();
   char *GetString();
   void SetString(char const *);
};
```

```
myString::myString()
{
   s = new char[100];
};
```

Ο constructor κάνει τη δέσμευση της μνήμης για το s, και εξασφαλίζει ότι δεν θα δημιουργηθεί αντικείμενο που δεν έχει την απαραίτητη μνήμη.

Υπερφόρτωση constructor.

```
class myString
{
private:
   char *s;
public:
   myString();
   myString(int C);
   char *GetString();
   void SetString(char const *);
};

myString::myString()
{
   s = new char[100];
};

myString::myString(int C)
{
   s = new char[C];
};
```

Υπερφόρτωση constructor.

```
class myString
{
private:
   char *s;
public:
   myString();
   myString(const char *);
   char *GetString();
   void SetString(char const *);
};

myString::myString()
{
   s = new char[100];
};
```

```
myString::myString(const char * x)
{
   s = new char[strlen(x)+1];
   strcpy(s,x);
};
```

Destructor

- Μια μέθοδος που ο μόνος ρόλος της είναι να αποδομεί (destruct) το αντικείμενο.
  - Ο destructor κάνει ένα clean-up.
- Συντακτικό:
  - `~<classname>()`
  - ΔΕΝ μπορεί να έχει ορίσματα.
  - ΔΕΝ εχει τυπο επιστροφής – ΟΥΤΕ void.
- Υλοποίηση:
  - `<classname>::~<classname>()`

- Η συνάρτηση καλείται αυτόματα με την αποδόμηση του αντικειμένου.
  - Είτε γιατί παύει να υπάρχει (βγαίνουμε από το scope που είναι ορισμένο).
  - Είτε αποδόμηση με delete.

myString

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
};
```

Η αποδέσμευση της μνήμης θα γίνει μέσα στον destructor.

```
myString::myString()
{
   s = new char[100];
};
```

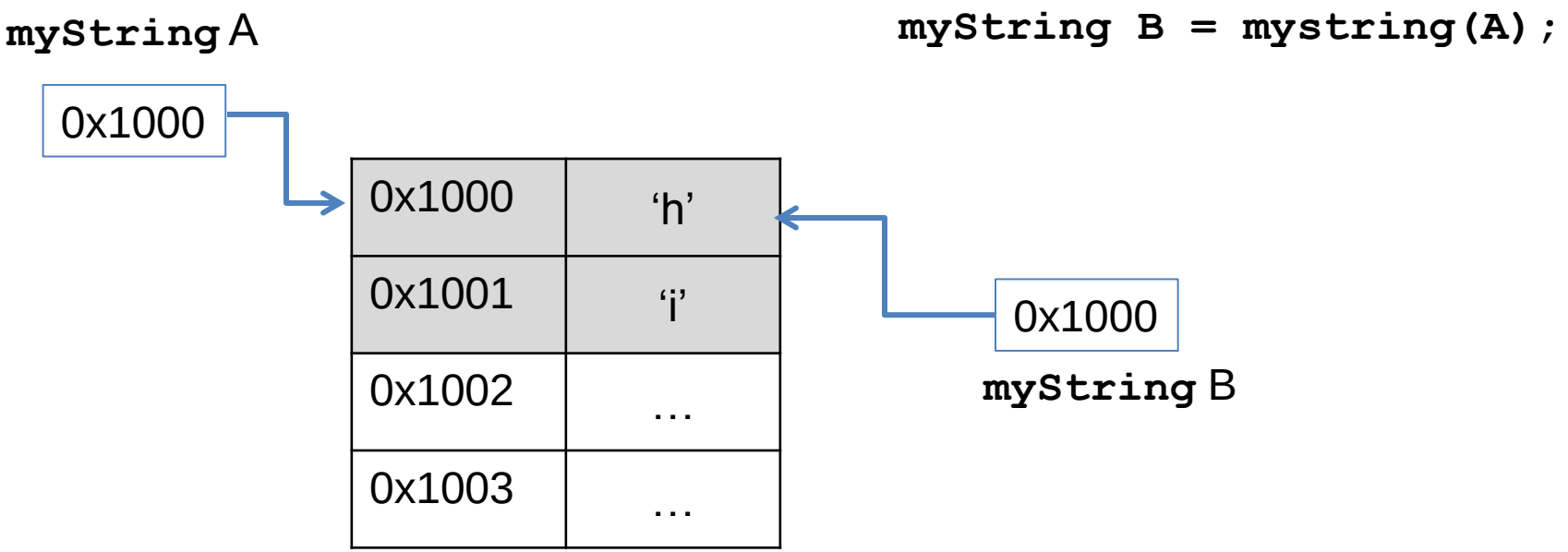
```
myString::~myString()
{
   delete [] s;
};
```

Μέλη μεταβλητές που είναι pointers

- Όπως και με όλους τους δείκτες, για τις μεταβλητές μέλη που είναι δείκτες υπάρχει κίνδυνος να βρεθούν να δείχνουν σε ένα χώρο μνήμης που έχει αποδεσμευτεί (dangling pointers).
- Οι constructors και destructors μπορούν να δημιουργήσουν επιπλέον προβλήματα γιατί μας αναγκάζουν να δεσμεύσουμε ή να αποδεσμεύσουμε μνήμη.

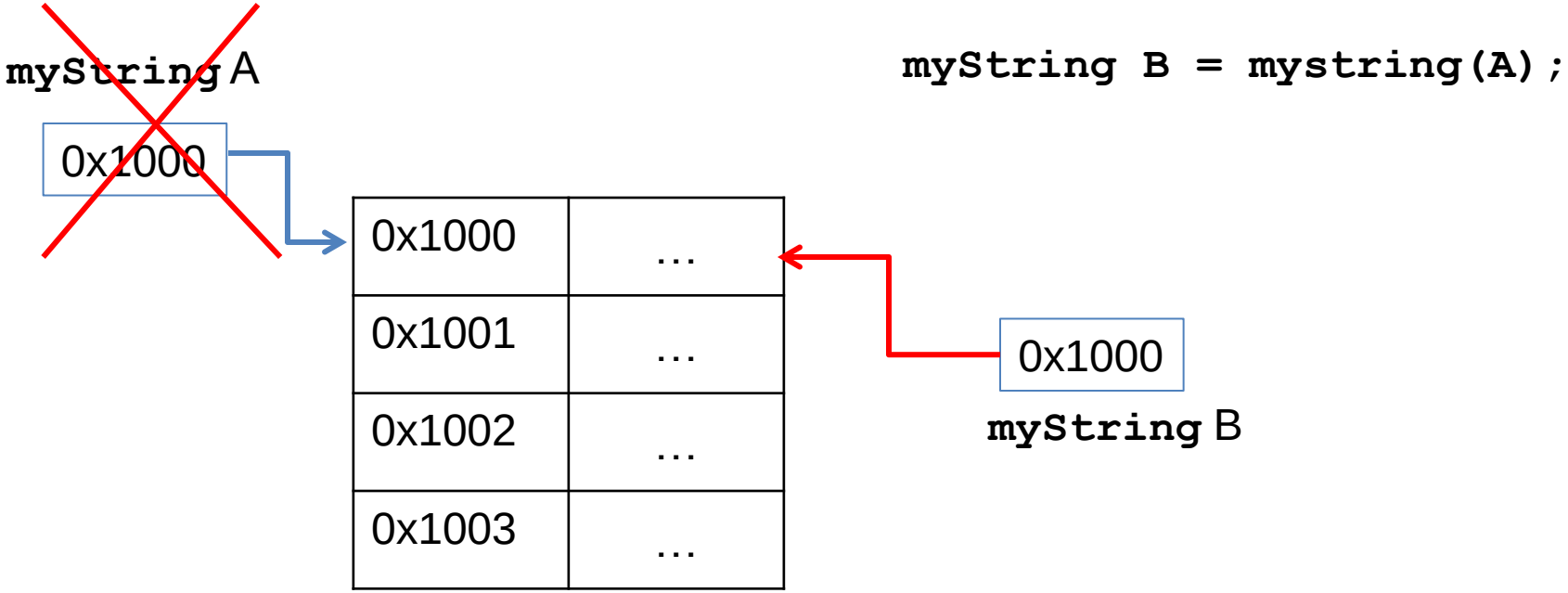
Σενάριο 1

- Ο Default Copy Constructor δημιουργεί ένα αντικείμενο Β αντιγράφοντας τα πεδία (συμπεριλαμβανομένων και των δεικτών) ενός αντικειμένου Α. Το αντικείμενο Α καταστρέφεται και ο destructor αποδεσμεύει τη μνήμη των δεικτών. Οι δείκτες του Β πλέον δείχνουν στο κενό.



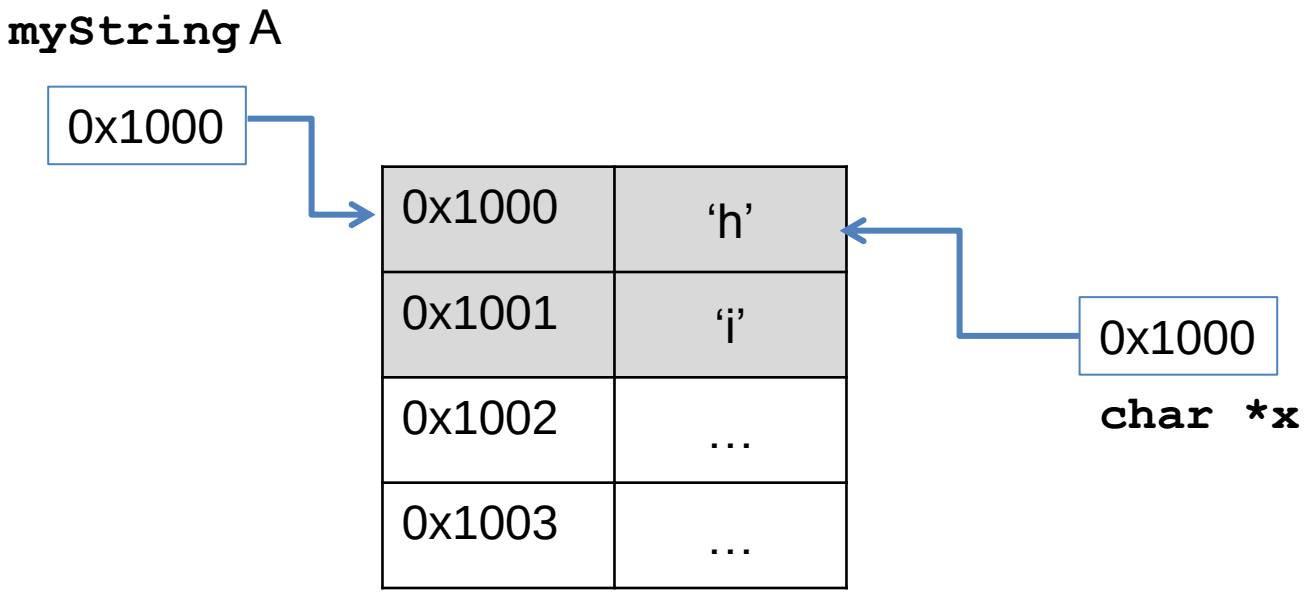
Σενάριο 1

- Ο Default Copy Constructor δημιουργεί ένα αντικείμενο B αντιγράφοντας τα πεδία (συμπεριλαμβανομένων και των δεικτών) ενός αντικειμένου A. Το αντικείμενο A καταστρέφεται και ο destructor αποδεσμεύει τη μνήμη των δεικτών. Οι δείκτες του B πλέον δείχνουν στο κενό.



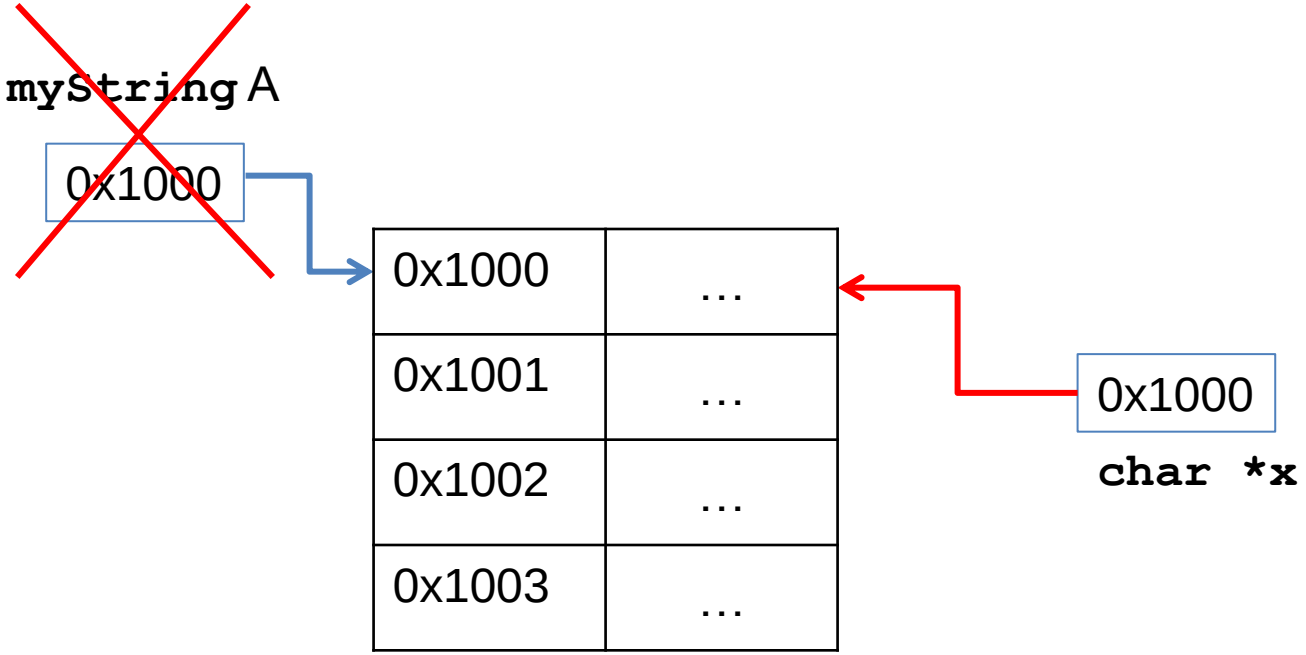
Σενάριο II

- Μια συνάρτηση της κλάσης (constructor ή άλλη) κάνει ένα δείκτη να δείχνει σε μια ήδη δεσμευμένη περιοχή μνήμης (shallow copy).



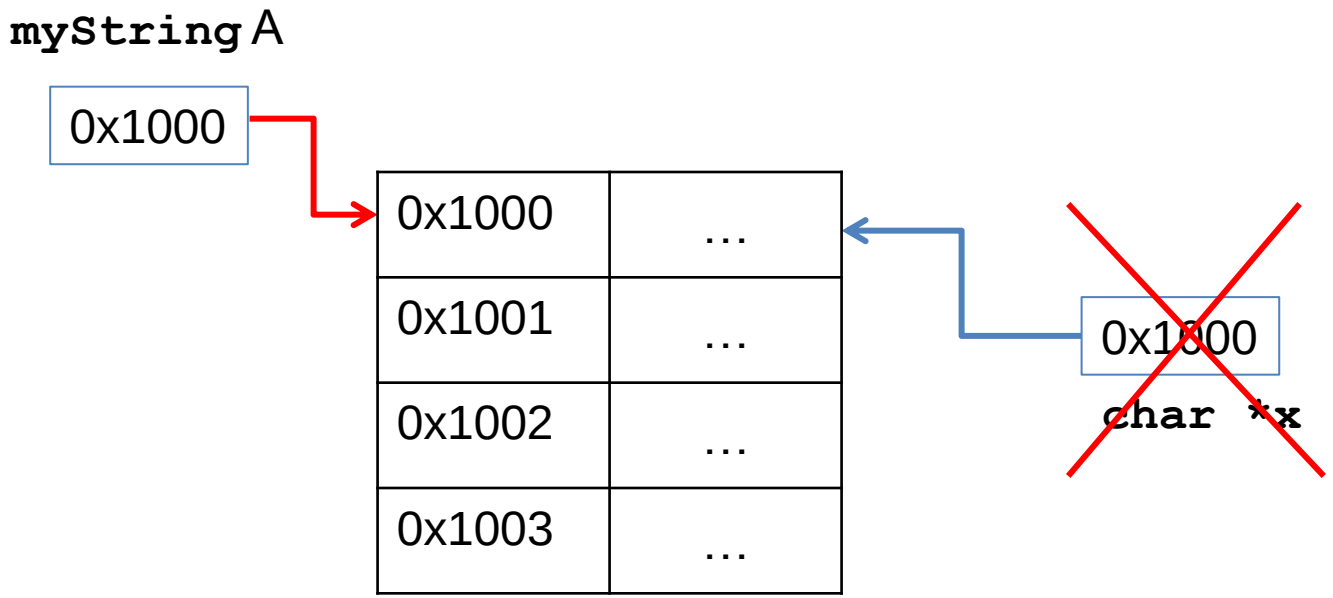
Σενάριο II

- Μια συνάρτηση της κλάσης (constructor ή άλλη) κάνει ένα δείκτη να δείχνει σε μια ήδη δεσμευμένη περιοχή μνήμης (shallow copy).
  - Το αντικείμενο καταστρέφεται, και η μνήμη αποδεσμεύεται με αποτέλεσμα οι υπόλοιποι δείκτες που έδειχναν σε αυτό το χώρο μνήμης τώρα να μη δείχνουν πουθενά.



Σεναριο II

- Μια συνάρτηση της κλάσης (constructor ή άλλη) κάνει ένα δείκτη να δείχνει σε μια ήδη δεσμευμένη περιοχή μνήμης (shallow copy).
  - Το αντικείμενο καταστρέφεται, και η μνήμη αποδεσμεύεται με αποτέλεσμα οι υπόλοιποι δείκτες που έδειχναν σε αυτό το χώρο μνήμης τώρα να μη δείχνουν πουθενά.
  - Η μνήμη αποδεσμεύεται κάπου μέσα στο πρόγραμμα. Ο δείκτης-μέλος πλέον δεν δείχνει πουθενά.



Παράδειγμα Ι: GetCopy()

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   myString GetCopy();
};
```

Υπενθύμιση: όταν μια συνάρτηση επιστρέφει ένα αντικείμενο τα πεδία του επιστρεφόμενου αντικειμένου αντιγράφονται στο αντικείμενο που κάνουμε ανάθεση.

Τι αποτέλεσμα θα έχει αυτός ο κώδικας?

```
myString myString::GetCopy()
{
   myString temp;
   temp.SetString(s);
   return temp;
}
```

```
int main()
{
   myString X;
   X.SetString("hello");
   myString Y;
   Y = X.GetCopy();
   cout << X.GetString()
        << " "
        << Y.GetString();
}
```

Παράδειγμα Ι: GetCopy()

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   myString GetCopy();
};
```

Ο κώδικας κρεμάει γιατί όταν βγαίνουμε από την συνάρτηση το αντικείμενο **temp** καταστρέφεται και ελευθερώνεται ο χώρος μνήμης που δείχνει το **s** στο αντικείμενο **Y**.

```
myString myString::GetCopy()
{
   myString temp;
   temp.SetString(s);
   return temp;
}
```

```
int main()
{
   myString X;
   X.SetString("hello");
   myString Y;
   Y = X.GetCopy();
   cout << X.GetString()
        << " "
        << Y.GetString();
}
```

Παράδειγμα ΙΙ: Copy()

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   void Copy(myString);
};
```

```
void myString::Copy(myString o)
{
  strcpy(s,o.GetString());
}
```

Τι αποτέλεσμα θα έχει αυτός ο κώδικας?

```
int main()
{
   myString X;
   X.SetString("hello");
   myString Y;
   Y.Copy(X);
   cout << X.GetString()
        << " "
        << Y.GetString();
}
```

Παράδειγμα II: Copy()

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   void Copy(myString);
};
```

Ο κώδικας κρεμάει γιατί:

- Όταν περνάμε το **X** ως όρισμα στην **Copy** δημιουργείται ένα τοπικό αντίγραφο με το default copy constructor που αντιγράφει όλα τα πεδία.
- Βγαίνοντας από την **Copy** το τοπικό αντίγραφο καταστρέφεται και ο δείκτης **s** του **X** κρεμάει.

```
void myString::Copy(myString o)
{
  strcpy(s,o.GetString());
}
```

```
int main()
{
  myString X;
  X.SetString("hello");
  myString Y;
  Y.Copy(X);
  cout << X.GetString()
       << " "
       << Y.GetString();
}
```

Λύση Ι: πέρασμα δια αναφοράς

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   void Copy(myString &);
};
```

Δεν δημιουργείται πλέον τοπικό αντίγραφο του αντικειμένου

```
void myString::Copy(myString &o)
{
  strcpy(s,o.GetString());
}
```

```
int main()
{
  myString X;
  X.SetString("hello");
  myString Y;
  Y.Copy(X);
  cout << X.GetString()
       << " "
       << Y.GetString();
}
```

Λύση ΙΙ: Επανορισμός του default copy constructor

```
class myString
{
private:
  char *s;
public:
  myString();
  myString(myString &)
  ~myString();
  char *GetString();
  void SetString(char const *);
  void Copy(myString);
};
```

```
myString::myString(myString &o)
{
  s = new char[100];
  strcpy(s,o.GetString());
}
```

```
void myString::Copy(myString o)
{
  strcpy(s,o.GetString());
}
```

```
int main()
{
  myString X;
  X.SetString("hello");
  myString Y;
  Y.Copy(X);
  cout << X.GetString()
       << " "
       << Y.GetString();
}
```

Παρένθεση: const

- Το αντικείμενο που περνάμε στην Copy το χρησιμοποιούμε μόνο για διάβασμα. Άρα θα έπρεπε να το περάσουμε ως σταθερή παραμετρο.

```
class myString
{
private:
   char *s;
public:
   myString();
   myString(myString &)
   ~myString();
   char *GetString();
   void SetString(char const *);
   void Copy(myString const &);
};
```

```
void myString::Copy(myString o)
{
   strcpy(s,o.GetString());
}
```

Πως ξέρουμε ότι η `GetString()` δεν θα αλλάξει το object `o`?

Παρένθεση: const

- Το αντικείμενο που περνάμε στην Copy το χρησιμοποιούμε μόνο για διάβασμα. Άρα θα έπρεπε να το περάσουμε ως σταθερή παραμετρο.

```
class myString
{
private:
   char *s;
public:
   myString();
   myString(myString &)
   ~myString();
   char *GetString() const;
   void SetString(char const *);
   void Copy(myString const &);
};
```

```
void myString::Copy(myString o)
{
   strcpy(s,o.GetString());
}
```

Πως ξέρουμε ότι η `GetString()` δεν θα αλλάξει το object `o`?

Την κάνουμε σταθερή συνάρτηση, δεν αλλάζει το αντικείμενο.

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

Διαβάστε [και γράψτε] ανάποδα ...

- Ο buf είναι const pointer σε χαρακτήρες
  - `char *const buf`
- Ο buf είναι pointer σε const χαρακτήρες
  - `char const *buf`

Δείκτες, Constructors και Destructors

- Οι constructors και destructors καλούνται και όταν δεσμεύουμε και αποδεσμεύουμε μνήμη για ένα pointer σε αντικείμενο.

```
int main()
{
  myString *X; // ορισμός ενός δείκτη τύπου myString
  X = new myString(); // κλήση του constructor, default size 100
  X->SetString("hello");
  myString *Y = new myString(*X);  // κλήση του overloaded
                                   // constructor,

  cout << X->GetString() << " " << Y->GetString();
  delete X;          // εκούσια αποδέσμευση μνήμης
  delete Y;
}
```

Δείκτες σε αντικείμενα

- Με τους δείκτες έχουμε πλήρη έλεγχο πότε δημιουργούνται και καταστρέφονται τα αντικείμενα.
  - Αυτό μας δίνει πολύ ευελιξία
  - Άλλα κάνει και τα λάθη πολύ πιο εύκολα.

Παράδειγμα Ι: GetCopy()

```
class myString
{
private:
   char *s;
public:
   myString();
   ~myString();
   char *GetString();
   void SetString(char const *);
   myString *GetCopy();
};
```

```
int main()
{
   myString X;
   X.SetString("hello");
   myString *Y = X.GetCopy();
   cout << X.GetString()
        << " "
        << Y->GetString();
   delete Y;
}
```

```
myString *myString::GetCopy()
{
   myString *temp = new myString();
   temp->SetString(s);
   return temp;
}
```

Η δεσμευμένη μνήμη παραμένει προσβάσιμη και αφού ο δείκτης **temp** καταστραφεί.
Δεν πρέπει να ξεχάσουμε να την αποδεσμεύσουμε όμως.

Πίνακες με δείκτες

- Θέλουμε να δημιουργήσουμε μια κατάσταση με τα ονόματα των φοιτητών μιας τάξης, την οποία να μπορούμε να τυπώνουμε. Διαφορετικές τάξεις έχουν διαφορετικά μεγέθη. Πως θα το σχεδιάσουμε?
  - Τα ονόματα θα τα κρατάμε σε αντικείμενα τύπου **`myString`** τα οποία ορίζονται με δυναμικό μέγεθος.
  - Η λίστα θα είναι σε ένα πίνακα με δείκτες σε αντικείμενα **`myString`**.
  - Ο πίνακας θα είναι μέρος μιας κλάσης που θα μας επιτρέπει την εκτύπωση.

Κλάση StudentClass

```
class StudentClass
{
private:
   myString **students;
   int classSize;
public:
   ClassStudents(int);
   ~ClassStudents();
   void EnterStudents();
   void PrintList();
};
```

Δείκτης σε δείκτη από **myString** για τον ορισμό του πίνακα από δείκτες σε **myString**

Κλάση StudentClass

```
class StudentClass
{
private:
   myString **students;
   int size;
public:
   ClassStudents(int);
   ~ClassStudents();
   void EnterStudents();
   void PrintList();
};
```

```
StudentClass::StudentClass(int s)
{
   students = new myString*[s];
   size = s;
}
```

Δημιουργία ενός πίνακα από **s** δείκτες σε αντικείμενα **myString**

Κλάση StudentClass

```
class StudentClass
{
private:
   myString **students;
   int size;
public:
   ClassStudents(int);
   ~ClassStudents();
   void EnterStudents();
   void PrintList();
};
```

```
void StudentClass::EnterStudents()
{
   for (int i = 0; i < size; i++){
      char s[100];
      cin >> s;
      students[i] = new myString(s);
   }
}
```

Δημιουργία των αντικείμενων **myString**. Εδώ καλείται ο constructor της κλάσης **myString**.

Κλάση StudentClass

```
class StudentClass
{
private:
   myString **students;
   int size;
public:
   ClassStudents(int);
   ~ClassStudents();
   void EnterStudents();
   void PrintList();
};
```

```
void StudentClass::PrintList()
{
   for (int i = 0; i < size; i++){
      cout << students[i]->GetString()
           << endl;
   }
}
```

students[i] είναι δείκτης σε αντικείμενο **myString**. Χρησιμοποιούμε τον τελεστη -> για να καλέσουμε τη μέθοδο **GetString()**.

Κλάση StudentClass

Ελευθερώνει τη μνήμη για καθένα από τα αντικείμενα που δημιουργήσαμε. Εδώ καλείται ο destructor της myString

Προσοχη! Πρέπει να είμαστε σίγουροι ότι δεν υπάρχουν άλλοι δείκτες στα αντικείμενα που δημιουργήσαμε αλλιώς το πρόγραμμα θα κρεμάσει

```
StudentClass::~StudentClass()
{
   for (int i = 0; i < size; i++){
      delete students[i];
   }
   delete [] students;
}
```

```
class StudentClass
{
private:
   myString **students;
   int size;
public:
   ClassStudents(int);
   ~ClassStudents();
   void EnterStudents();
   void PrintList();
};
```

Ελευθερώνει τη μνήμη για τον πίνακα με τους δείκτες σε αντικείμενα **myString**.

Για κάθε new χρειαζόμαστε ένα delete

ΡΕΥΜΑΤΑ ΕΙΣΟΔΟΥ/ΕΞΟΔΟΥ

Ρεύματα

- Στην C++ (και στη C) η είσοδος και έξοδος γίνεται μέσω των ρευμάτων εισόδου και εξόδου
- Τι είναι ένα ρεύμα? Μια αφαίρεση που αναπαριστά μια πηγή (για την είσοδο), ή ένα προορισμό (για την έξοδο) χαρακτήρων
  - Αυτό μπορεί να είναι ένα αρχείο, το πληκτρολόγιο, η οθόνη.
  - Όταν δημιουργούμε το ρεύμα το συνδέουμε με την ανάλογη πηγή, ή προορισμό.

Ρεύματα

- Το πρόγραμμα μας αλληλεπιδρά με τα ρεύματα μέσω των εντολών εισόδου/εξόδου.
  - Στην C, printf, fprintf για έξοδο, scanf, fscanf για είσοδο.
  - Στην C++ τα ρεύματα είναι αντικείμενα των κλάσεων εισόδου εξόδου. Τα cout και cin είναι αντικείμενα (των κλάσεων ostream και istream) για το standard output και standard input. Έχουμε πρόσβαση σε αυτά τα αντικείμενα μέσω της βιβλιοθήκης iostream.
  - Το πρόγραμμα μας χρησιμοποιεί μεθόδους και τελεστές της κλάσης για να διαβάσει ή να γράψει στην είσοδο/έξοδο.
- Μια εντολή εισόδου/εξόδου έχει αποτέλεσμα το λειτουργικό να πάρει ή να στείλει χαρακτήρες από/προς την αντίστοιχη πηγή/προορισμό.
  - Ένας buffer χρησιμοποιείται για προσωρινή αποθήκευση και επικοινωνία μεταξύ του λειτουργικού και του προγράμματος.

Είσοδος / Έξοδος δεδομένων

- Στη C++ 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;
  float f;

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

Το **endl** είναι ένας χειριστής (manipulator).

Χειριστές

- Οι χειριστές είναι εντολές μορφοποίησης που εισάγονται μέσα στο ρεύμα.
  - Ο χειριστής **`endl`** έχει ως αποτέλεσμα να εισαχθεί ένας χαρακτήρας αλλαγής γραμμής, και να αδειάσει ο buffer.
  - Υπάρχουν διάφοροι χειριστές που αλλάζουν την μορφή της εξόδου.
  - Π.χ. **`cout << setprecision(2) << (float)x;`**

  Καθορίζει την ακρίβεια (αριθμό δεκαδικών ψηφίων) στην έξοδο.
    - Οι χειριστές εφαρμόζονται πάντα σε αυτό που ακολουθεί.

Προκαθορισμένα αντικείμενα εισόδου/εξόδου

- **`cin`**: Αντικείμενο για είσοδο από το πληκτρολόγιο
- **`cout`**: Αντικείμενο για έξοδο στην οθόνη
- **`cerr`**: Αντικείμενο για την έξοδο σφαλμάτων.
  - Δεν αποθηκεύεται προσωρινά και δεν ανακατευθύνεται.
- **`clog`**: Αντικείμενο για την καταγραφή μηνυμάτων για την εξέλιξη του προγράμματος.

IOSTREAMs - reading lines

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

int main(){
   char buf[100];

   cin.get(buf, 10);
   cout << buf << endl;
   cin.get(buf, 10, '$');
   cout << buf << endl;
}
```

Διαβάζει 10-1 χαρακτήρες, ή μέχρι να συναντήσει τον χαρακτήρα '\n' και βάζει το αποτέλεσμα στον buff.
Ο χαρακτήρας '\n' ΔΕΝ διαβάζεται, παραμένει στο ρεύμα και θα είναι ο επόμενος χαρακτήρας να διαβαστεί.

Παρόμοιο με το παραπάνω αλλά σταματάει όταν δει τον χαρακτήρα '$'

IOSTREAMs - reading lines

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

int main(){
  char buf[100];

  cin.getline(buf, 10);
  cout << buf << endl;
  cin.getline(buf, 10, '$');
  cout << buf << endl;
}
```

Παρόμοια εντολή με την **get** αλλά η **getline** αφαιρεί τον delimiter χαρακτήρα '\n' (ή '$') από το ρεύμα, ΧΩΡΙΣ να τον τοποθετεί μέσα στη μεταβλητή buff.

Bit σφαλμάτων

- Οι κλάσεις istream και ostream είναι “παιδιά” της κλασης ios. Η κλάση ios (και συνεπώς και οι κλάσεις istream και ostream) εχει τις εξής σημαίες κατάστασης σφαλμάτων.
  - goodbit: δεν υπάρχουν σφάλματα
  - eofbit: τέλος αρχείου/εισόδου
  - failbit: αποτυχία λειτουργίας
  - badbit: μη εγγυρη λειτουργία (δεν υπαρχει το αντικείμενο)
  - heardfail: σφάλμα χωρίς δυνατότητα ανάκαμψης.

Συναρτήσεις σφαλμάτων

- Τις παραπάνω μεταβλητές μπορούμε να τις προσπελάσουμε μέσω των παρακάτω συναρτήσεων.
  - int eof(): επιστρέφει αληθές αν φτάσαμε στο τέλος της εισόδου (eofbit == true).
  - int fail(): επιστρέφει αληθές αν κάποια από τις σημαίες failbit, badbit, hardfail έχει ενεργοποιηθεί.
  - int bad(): επιστρέφει αληθές αν κάποια από τις σημαίες badbit, hardfail έχει ενεργοποιηθεί.
  - int good(): επιστρέφει αληθές αν καμία από τις σημαίες λάθους δεν έχει ενεργοποιηθεί.

IOSTREAMs - reading lines

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

int main(){
  char buf[100];

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

IOSTREAMs

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

int main(){
  int i; float f;

  while (cin.eof() != true){
    cin >> i >> f;
    cout << i << f << endl;
  }
}
```

IOSTREAMs - reading lines σε string

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

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

FILESTREAM

- 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,…)
- Ότι συνάρτηση είδαμε στα cin, cout ισχύει και σε μεταβλητές ifstream, ofstream.
- Στα filestreams η συσκευή εισόδου/εξόδου είναι ένα αρχείο αντί για το πληκτρολόγιο, ή την οθόνη.

FILESTREAM

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

int main(){
  ifstream in("in.txt");
  ofstream out("out.txt");

  if (!in.good() != true || out.good()!= true)
       exit(1);

  while (in.eof() != true){ // ή (!in.eof()) ή (!in)
    char buf[100];
    in.getline(buf, 100, '\n');
    out << buf << endl;
  }
}
```

Δημιουργεί ένα ρεύμα εισόδου που θα διαβάζει από το αρχείο “in.txt”

Δημιουργεί ένα ρεύμα εξόδου που θα γράφει στο αρχείο “out.txt”

Bit Καταστασης

- Τα bit καταστασης είναι πάλι της κλάσης ios και καθορίζουν τις διάφορες πτυχές του τρόπου ανοίγματος των αρχείων. Μερικά χρήσιμα bits:
  - ios::in: ανοιγμα για ανάγνωση
  - ios::out: άνοιγμα για εγγραφή
  - ios::app: ανοιγμα για εγγραφή στο τέλος του αρχείου (append).
  - ios::binary: ανοιγμα για γράψιμο ή διάβασμα binary αρχείου.

FILESTREAM

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

int main(){
  char buf[100];

  ifstream in("in.txt");
  ofstream out("out.txt", ios::app); //ios::out is default

  if(!in.good() || !out.good()) exit(1);

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

FILESTREAM

```
#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();
}
```

FILESTREAM

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

  fstream in; // Για είσοδο και έξοδο
  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;
  }
```

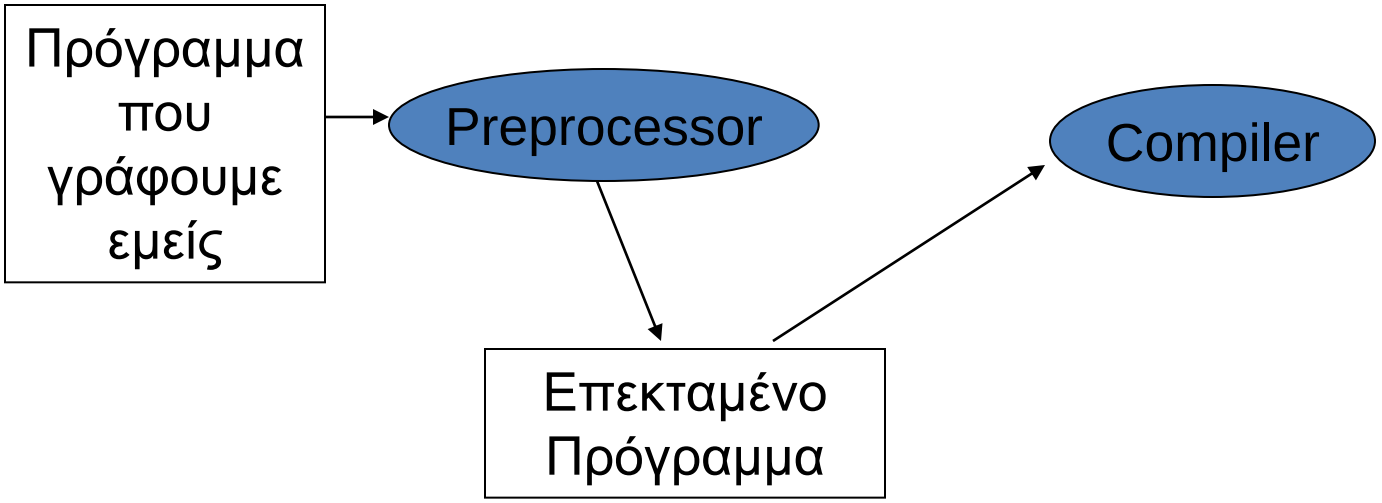
Παράδειγμα: Κλάση StudentClass

```
class StudentClass
{
private:
  myString **students;
  int size;
public:
  ClassStudents(int);
  ~ClassStudents();
  void EnterStudents(const char *);
  void PrintList();
};
```

```
void StudentClass::EnterStudents(const char *fname)
{
  ifstream fin(fname)
  for (int i = 0; i < size; i++){
    char s[100];
    fin >> s;
    students[i] = new myString(s);
  }
  fin.close();
}
```

PREPROCESSOR

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



Ο preprocessor χρησιμοποιεί τις εντολές με # για να δώσει οδηγίες στον επεξεργαστή

#include

- Ενσωματώνει στο πρόγραμμά μας αυτούσια αρχεία.

```
#include <iostream>
#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;
```

Εμβόλιμες Συναρτήσεις (Inline functions)

Είναι macros, οι οποίες χρησιμοποιούνται από τον compiler στην παραγωγή του εκτελέσιμου κώδικα. Χρήσιμες για πολύ μικρές συναρτήσεις.

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

Χρησιμοποιούνται κανονικά στο πρόγραμμα, π.χ.,

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

Οι συναρτήσεις που ορίζονται μέσα στον ορισμό της κλάσης είναι εξ ορισμού inline functions.