ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ

https://courses.pclab.ece.ntua.gr/course/view.php?id=64

Διδάσκοντες:

Γιώργος Γκούμας
Άρης Παγουρτζής
Γιώργος Στάμου
Βασίλης Βεσκούκης
Θάνος Βουλόδημος
Γιώργος Αλεξανδρίδης
Γιώργος Σιόλας
Παρασκευή Τζούβελη

progtech@cslab.ece.ntua.gr

4/3/2022

Διαφάνειες παρουσιάσεων

- Η γλώσσα προγραμματισμού C++
- Αρχές αντικειμενοστρεφούς προγραμματισμού
- Δομές δεδομένων

ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ

https://courses.pclab.ece.ntua.gr/course/view.php?id=64

Διδάσκοντες:

Γιώργος Γκούμας
Άρης Παγουρτζής
Γιώργος Στάμου
Βασίλης Βεσκούκης
Θάνος Βουλόδημος
Γιώργος Αλεξανδρίδης
Γιώργος Σιόλας
Παρασκευή Τζούβελη

progtech@cslab.ece.ntua.gr

Credits διαφανειών, υλικού κ.π.ά.

Νίκος Παπασπύρου, Καθ.ΕΜΠ (nickie@softlab.ntua.gr)

Σύντομη επανάληψη της C++ (i)

- Γεια σου κόσμε!

```
#include <iostream>

using namespace std;

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

Σύντομη επανάληψη της C++ (ii)

- Δώσε μου δύο αριθμούς...

```
#include <iostream>

using namespace std;

int main() {
  int a, b;
  cout << "Give me two numbers: ";
  cin >> a >> b;
  cout << "You gave me " << a
       << " and " << b << endl;
}
```

Σύντομη επανάληψη της C++ (iii)

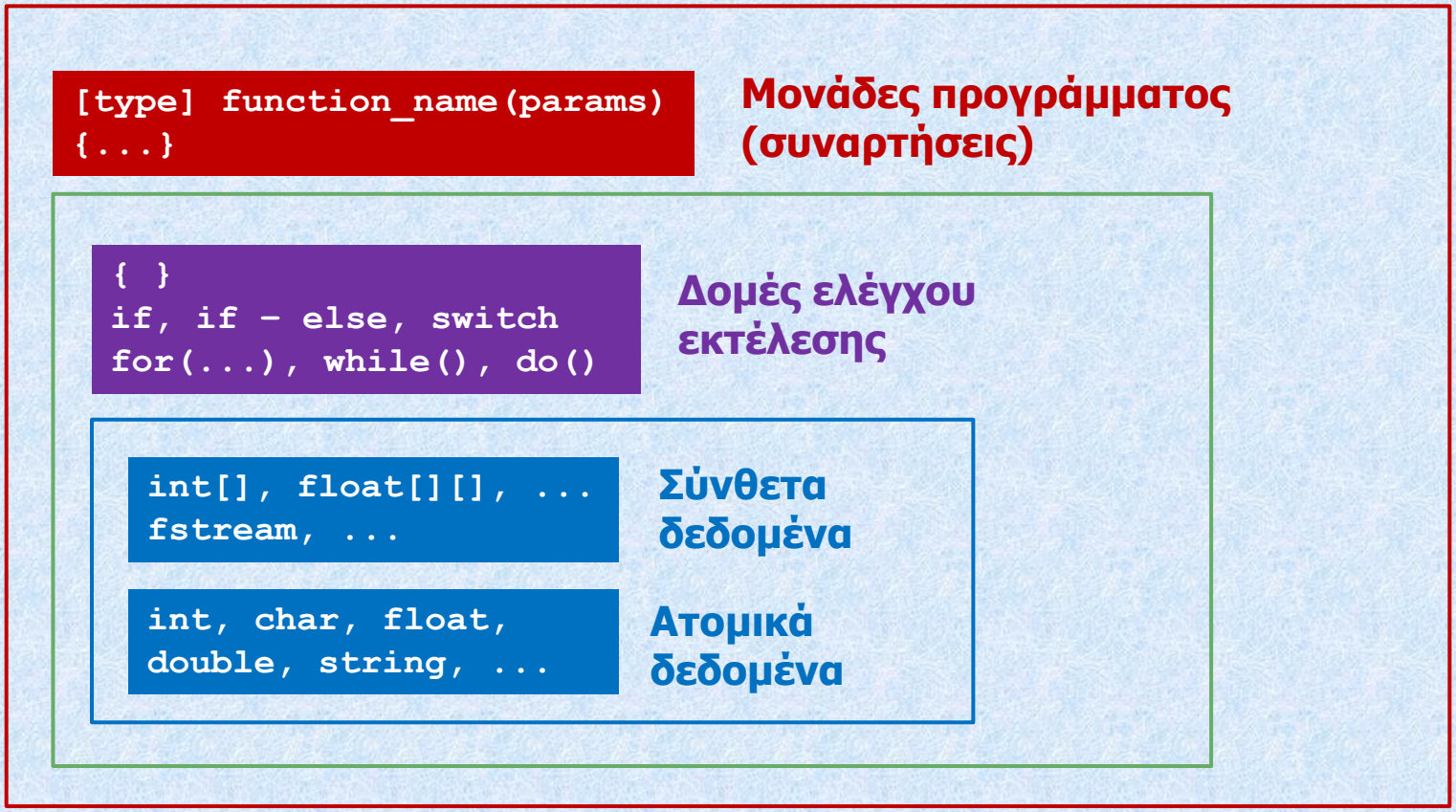
- ... και θα σου βρω το ΜΚΔ τους

```
int gcd(int x, int y) {
  while (x > 0 && y > 0)
    if (x > y) x %= y; else y %= x;
  return x + y;
}

int main() {
  cout << "Give me two numbers: ";
  int a, b;
  cin >> a >> b;
  cout << "GCD(" << a << ", " << b
       << ") = " << gcd(a, b) << endl;
}
```

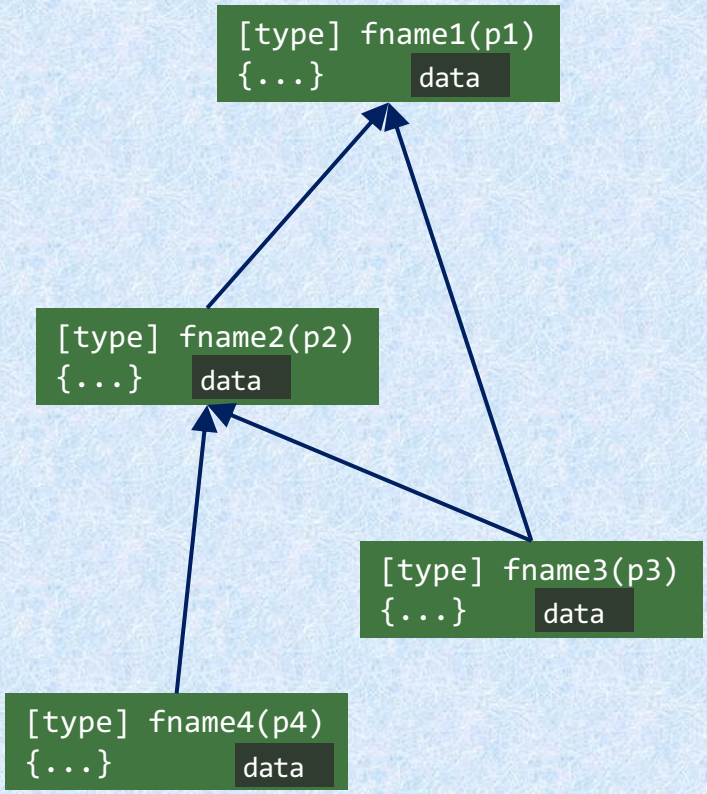
Προγραμματισμός - επανάληψη

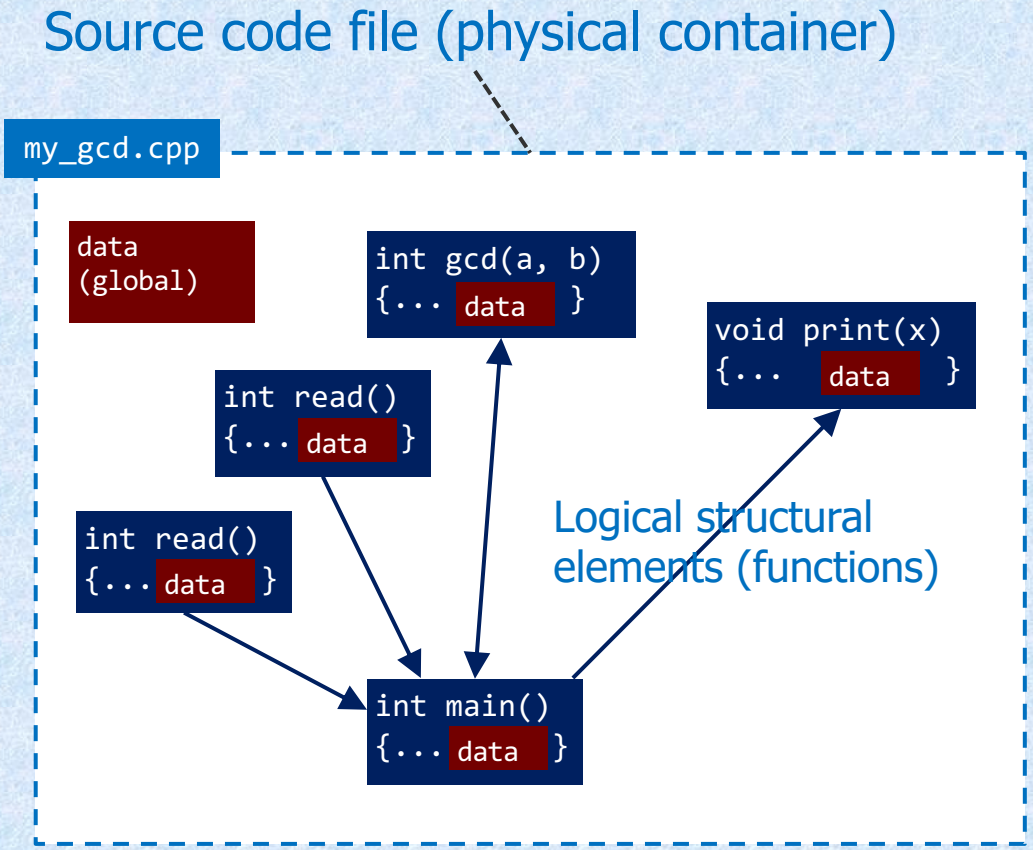
- **Algorithms + Data Structures = Programs**
  - **Δεδομένα**: τύποι, οργάνωση (δομές δεδομένων), λογική και φυσική υπόσταση
  - **Αλγόριθμοι**: τρόποι επίλυσης υπολογιστικών προβλημάτων που υλοποιούνται με εκφράσεις μιας γλώσσας προγραμματισμού: σειριακή, υπό συνθήκη, ή επαναληπτική εκτέλεση
- **Δομή προγράμματος**: συναρτήσεις, παράμετροι
- **Λοιπά**: μόνιμη αποθήκευση, δικαιώματα πρόσβασης, επικοινωνίες, ασφάλεια, κ.ά.

Προγραμματισμός - επανάληψη

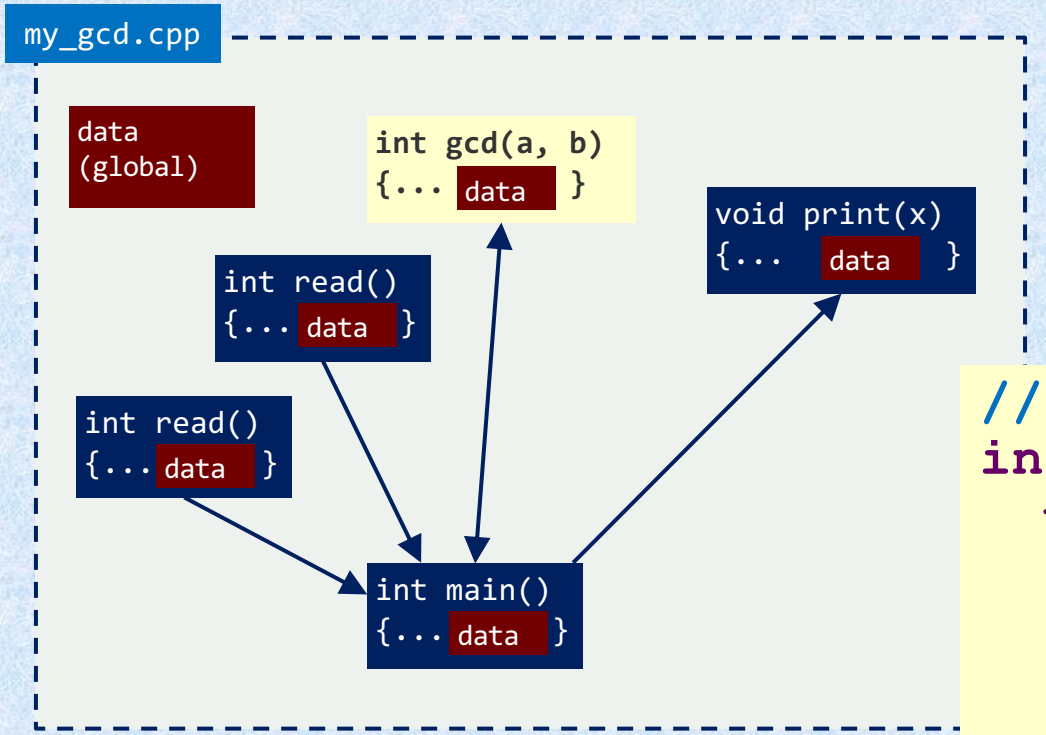


Προγραμματισμός - επανάληψη





Προγραμματισμός - επανάληψη




```
// function to calc gcd
int gcd(int x, int y) {
  while (x > 0 && y > 0)
    if (x > y)
      x %= y;
    else
      y %= x;
  return x + y;
}
```

Σημεία προσοχής

- Ο τρόπος γραφής ενός προγράμματος πρέπει να επιτρέπει να γίνει εύκολα κατανοητή η δομή και η συμπεριφορά του
  - Στοίχιση
  - Σχόλια
  - Ονόματα μεταβλητών
- Η παραγωγή "σωστών αποτελεσμάτων" δεν είναι από μόνη της κριτήριο ποιότητας ενός προγράμματος
- Ο,τι δεν μαθαίνω όταν εκπαιδεύομαι, δύσκολα "θα το κάνω στο μέλλον"

Ερωτήματα!

- Πώς αποφασίζω…
  - Πώς θα οργανώσω τα δεδομένα μου;
  - Πόσες/ποιες συναρτήσεις να κατασκευάσω
  - Ποιες παραμέτρους κάθε συνάρτηση

> *Να γράψετε ένα πρόγραμμα που να διαβάζει ένα φυσικό αριθμό $N > 0$ και να εκτυπώνει το μέγιστο φυσικό αριθμό του οποίου το παραγοντικό δεν υπερβαίνει το N.*

- «Διαισθητικά» καλές πρακτικές
  - Διαίρει και βασίλευε
  - Μην γράφεις τα πάντα από την αρχή: επανα-χρησιμοποίηση

Design principles

- Αρχή της μοναδικής ευθύνης (Single Responsibility Principle – SRP)
  - Code should be focused, narrow, and should do one thing and only one thing well.
  - In object-oriented design, a class should have only and only one responsibility so that it has to change less frequently.
  - Code should be cohesive: it should have one, and only one, reason to change.

Αρχή της μοναδικής ευθύνης

- …if there is a piece of code that does several things, then that code will need to change often; this is expensive because when a programmer comes to change it, they have to make sure that those changes do not break other things
- …when a piece of code is broken into several smaller pieces, each one of which does one and only one thing well, then the cost of changes is much lower

Design principles

- Open/closed
  - A class should be open for extension and closed for modification.
  - Classes should be written in a way that it is ready for adopting/adding new features but "not interested" in any modification.

Τι κάνει ένα πρόγραμμα «καλό»;

- Ορθότητα
- Ταχύτητα
- Καλή χρήση πόρων (μνήμη, CPU)
- Αναγνωσιμότητα
- Συντηρησιμότητα
- Επαναχρησιμοποίηση
- Γρήγορη κατασκευή

- Πολλά από τα παραπάνω συγκρούονται μεταξύ τους (ΠΟΙΑ;)

Ξεχωριστή μεταγλώττιση

Ξεχωριστή μεταγλώττιση

- Τα προγράμματα συνήθως αποτελούνται από > 1 αρχεία πηγαίου κώδικα και από έτοιμο πηγαίο κώδικα τρίτων ή/και βιβλιοθήκες της γλώσσας
- Κατά τη συγγραφή ενός προγράμματος πρέπει να συμπεριλαμβάνεται στη μεταγλώττιση και σύνδεση μόνο ό,τι απαιτείται (δηλ. όχι τα πάντα)
  - #include cmath
  - #include fstream
- Η κατάτμηση σε πολλά αρχεία πηγαίου κώδικα επιτρέπει την επαναχρησιμοποίηση (αναγκαία, αλλά όχι ικανή συνθήκη!)

Ξεχωριστή μεταγλώττιση (i)

gcd.cpp

```
int gcd(int x, int y) {
  while (x > 0 && y > 0)
    if (x > y) x %= y; else y %= x;
  return x + y;
}
```

gcd_demo.cpp

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

int main() {
  cout << "Give me two numbers: ";
  int a, b;
  cin >> a >> b;
  cout << "GCD(" << a << ", " << b
       << ") = " << gcd(a, b) << endl;
}
```

Ξεχωριστή μεταγλώττιση (ii)

- Αρχείο επικεφαλίδας (header file)

gcd.hpp

```
int gcd(int x, int y);
```

- Αρχείο υλοποίησης

gcd.cpp

```
#include "gcd.hpp"

int gcd(int x, int y) {
  while (x > 0 && y > 0)
    if (x > y) x %= y; else y %= x;
  return x + y;
}
```

Ξεχωριστή μεταγλώττιση (iii)

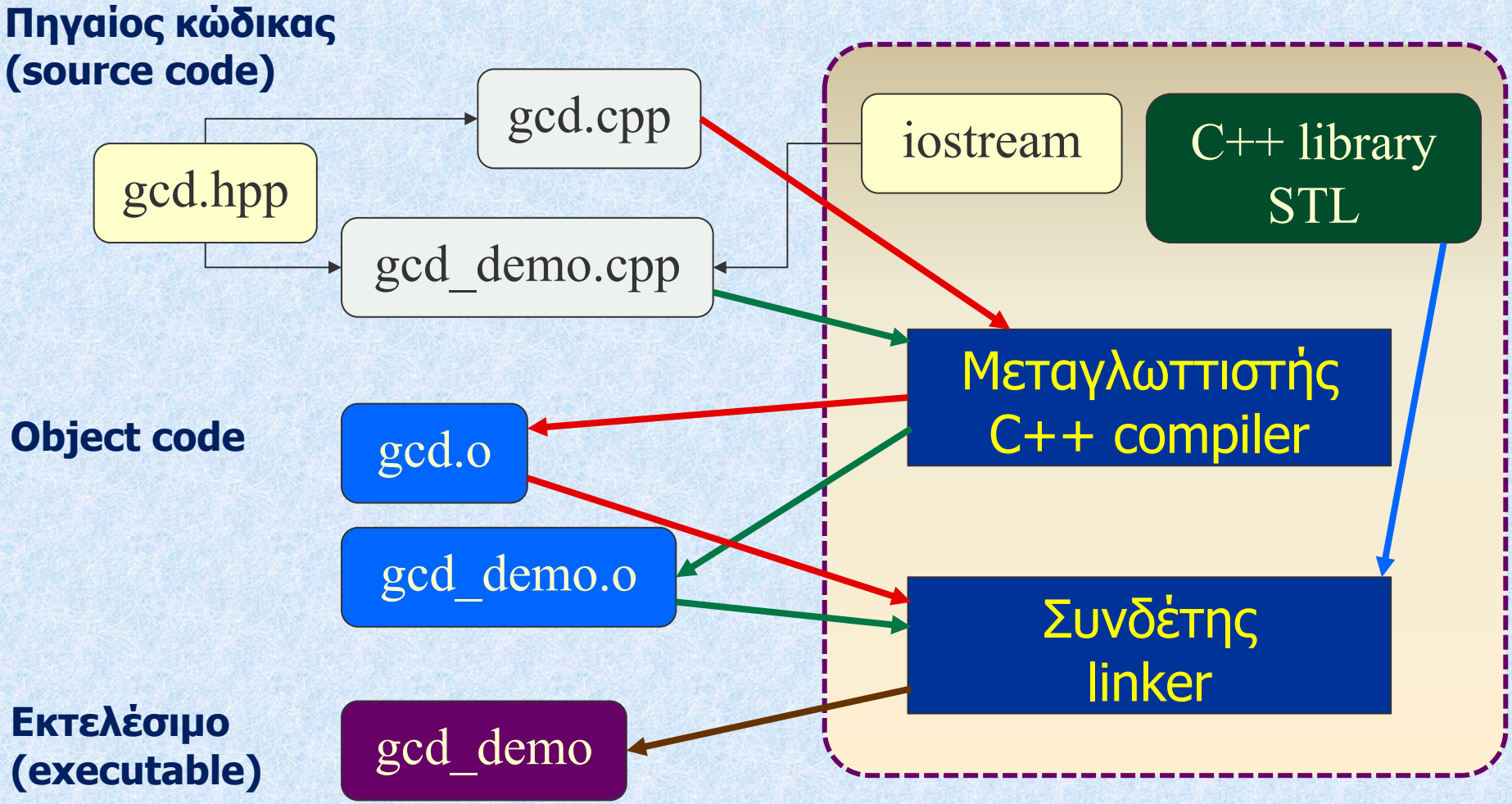
gcd.hpp

```
int gcd(int x, int y);
```

gcd_demo.cpp

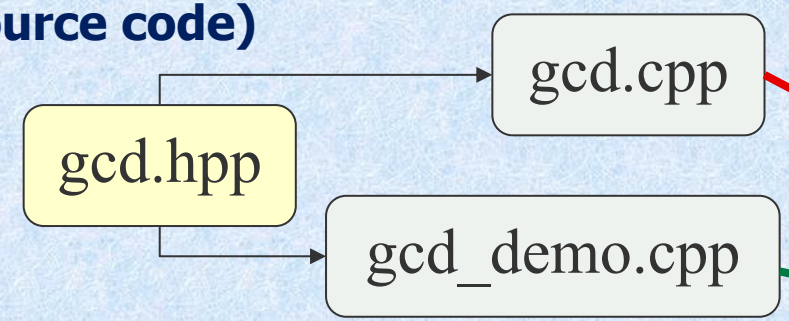
```
#include <iostream>
#include "gcd.hpp"

using namespace std;

int main() {
  cout << "Give me two numbers: ";
  int a, b;
  cin >> a >> b;
  cout << "GCD(" << a << ", " << b
       << ") = " << gcd(a, b) << endl;
}
```

Διαδικασία μεταγλώττισης (i)



Διαδικασία μεταγλώττισης (ii)

Πηγαίος κώδικας (source code)



Object code

Εκτελέσιμο (executable)


```
$ ls
gcd.hpp
gcd.cpp
gcd_demo.cpp

$ c++ -c gcd.cpp

$ c++ -c gcd_demo.cpp

$ ld -o gcd_demo
    gcd.o gcd_demo.o
    ...

$ ./gcd_demo
```

```
$ c++ -o gcd_demo
    gcd.cpp gcd_demo.cpp
```

Πίνακες, revisited

Πίνακες (i)

```
int main() {
  int a[10];
  for (int i = 0; i < 10; ++i)
    a[i] = 100*(i+1);
  for (int i = 0; i < 10; ++i)
    cout << a[i] << endl;
}
```

```
const int N = 42;
int main() {                    // με σταθερά
   int a[N]; ...
```

```
#define N 42
int main() {                    // με μακροεντολή
   int a[N]; ...
```

Πίνακες (ii)

- Μεγάλοι πίνακες:
  - είτε global (γιατί;;;)
  - είτε με δυναμική παραχώρηση μνήμης:

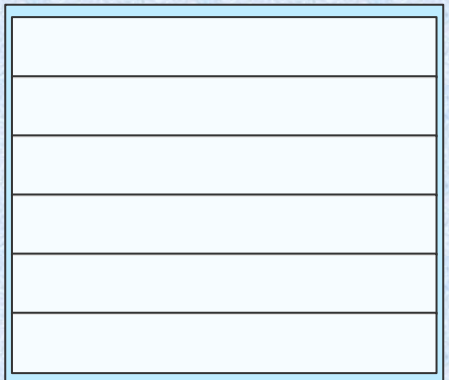
```
#define N 10000000
int main() {
  int *a = new int[N];
  for (int i = 0; i < N; ++i)
    a[i] = 10*(i+1);
  for (int i = 89; i < 100; ++i)
    cout << a[i] << endl;
  delete [] a;
}
```

Δέσμευση μνήμης

- Static
  - global ή static μεταβλητές
- Stack
  - μεταβλητές που δηλώνονται μέσα σε συναρτήσεις
- Heap
  - μεταβλητές με δυναμική δέσμευση
  - αποδέσμευση από τον προγραμματιστή ή το λειτουργικό, με το πέρας του προγράμματος

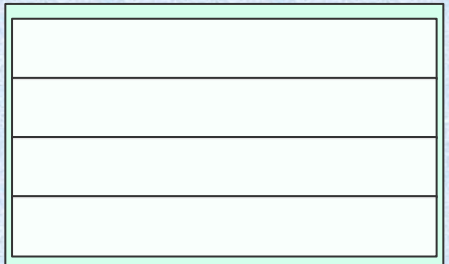
Δέσμευση μνήμης

static



- global και static μεταβλητές
- "ζουν" όσο τρέχει το πρόγραμμα
- αυτόματη αποδέσμευση μνήμης

stack



- τοπικές μεταβλητές
- συνεχείς (continuous) θέσεις
- αυτόματη δέσμευση και αποδέσμευση
- περιορισμός μεγέθους

heap

- τυχαίες θέσεις
- δυναμική δέσμευση
- απαιτείται αποδέσμευση
- μεγάλο μέγεθος

Δυναμική δέσμευση μνήμης

- Αιτούμαστε τη μνήμη και αν υπάρχει παίρνουμε ένα δείκτη σε αυτή (πού;;;), διαφορετικά NULL

- Δυναμική δέσμευση σε C: **`malloc`**

```
int *p;
...

p = (int *) malloc(sizeof(int));
if (p == NULL) {
  printf("Out of memory!\n");
  exit(1);
}
...
free(p)
```

type casting

Ζητούμενο μέγεθος μνήμης

Δυναμική δέσμευση μνήμης

- Δυναμική δέσμευση σε C++
  - Έλεγχος με ανίχνευση εξαίρεσης (exception) ή με ανίχνευση null, δηλώνοντας nothrow

```
int *p = new(nothrow) int;
if (!p) {
   cout << "Out of memory!\n";
}
```

  - Συνήθως (;) δεν ελέγχουμε

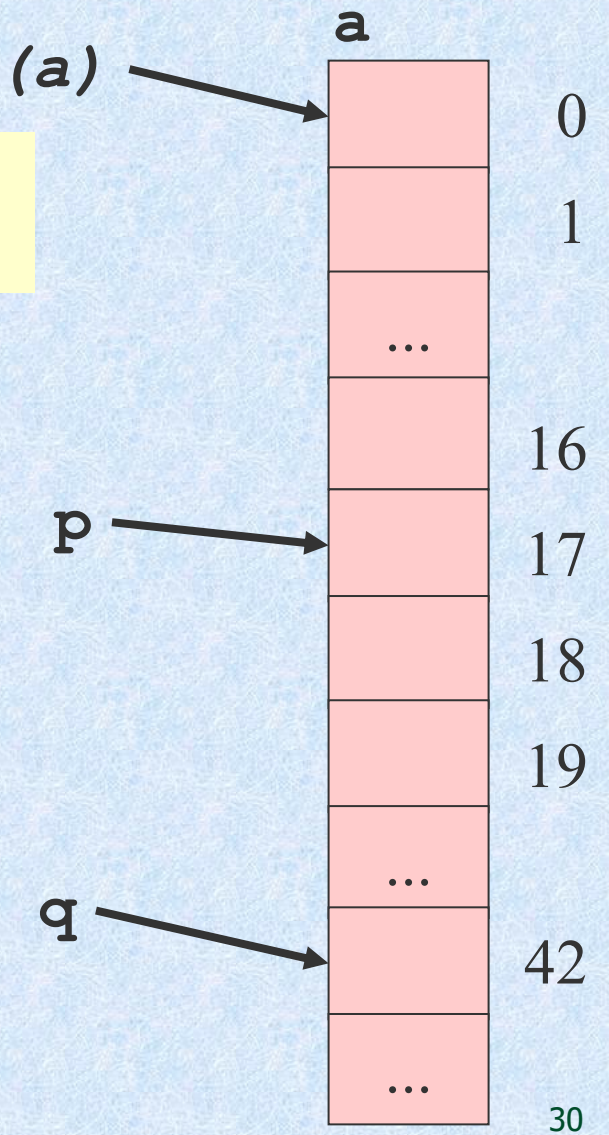
```
int *a = new int[N];
  for (int i = 0; i < N; ++i)...
```

Πίνακες και δείκτες

```
int a[N];
...
int *p;
p = &a[17];

cout << *p << endl;
cout << *(p+2) << endl;
cout << p[2] << endl;
cout << *(p-1) << endl;
cout << p[-1] << endl;

int *q = &a[42];
cout << q - p << endl;

p++;
q -= 4;
```

```
// ισοδύναμα:
int *p = &a[17];
```

25



Πέρασμα παραμέτρων, αναφορές, δείκτες

Πέρασμα κατ' αναφορά στη C++ (i)

```
void swap(int &x, int &y) {
  int t = x;
  x = y;
  y = t;
}

int main() {
  int a = 42, b = 17;
  swap(a, b);
  cout << a << endl;
  cout << b << endl;
}
```

Πέρασμα κατ' αναφορά (ii)

- Το ίδιο παράδειγμα με δείκτες

```
void swap(int *x, int *y) {
  int t = *x;
  *x = y;
  *y = t;
}

int main() {
  int a = 42, b = 17;
  swap(&a, &b);
  cout << a << endl;
  cout << b << endl;
}
```

Αναφορές (&) (i)

- Γενικότερος τύπος στη C++,
  όχι μόνο για πέρασμα με αναφορά
- Σαν τους δείκτες αλλά με τις εξής διαφορές:
  - Δε χρειάζονται αστεράκια κατά τη χρήση τους.
  - Όταν δηλώνονται πρέπει οπωσδήποτε να αρχικοποιούνται ώστε να «αναφέρονται» (να δείχνουν) σε κάποια άλλη μεταβλητή.
  - Μετά τη δήλωση και αρχικοποίησή τους, δεν υπάρχει τρόπος να τροποποιηθούν ώστε να αναφέρονται σε άλλη μεταβλητή

Αναφορές (ii)

```
int a = 42, b = 17;
int &c = a;   // Οι a και c είναι η ίδια μεταβλητή!
cout << a << " " << b << " " << c << endl;
//      42          17          42
a = 1;
//      1           17          1
c = 2;
//      2           17          2
c = b;
//      17          17          17
c = 3;
//      3           17          3
```

Δείκτες και αναφορές σε σταθερές (i)

- Μεταβλητές και σταθερές, επανάληψη

```
int a = 42;

cout << a << endl;
cin >> a;

const int b = 42;

cout << b << endl;
b = 17;                    // απαγορεύεται !!!
```

Δείκτες και αναφορές σε σταθερές (ii)

```
int a = 42;
const int b = 42;

int *p = &a;

cout << *p << endl;
*p = 17;

const int *q = &b;     // δείκτης σε const int

cout << *q << endl;
*q = 17;               // απαγορεύεται !!!

q = &a;

cout << *q << endl;
*q = 17;               // πάλι απαγορεύεται !!!

p = &b;                // απαγορεύεται !!! (γιατί;)
```

Δείκτες και αναφορές σε σταθερές (iii)

```
int a = 42;
const int b = 42;

int &c = a;

cout << c << endl;
c = 17;

const int &d = b;      // αναφορά σε const int

cout << d << endl;
d = 17;                // απαγορεύεται !!!

const int &e = a;

cout << e << endl;
e = 17;                // πάλι απαγορεύεται !!!

int &f = b;            // απαγορεύεται !!! (γιατί;)
```

Δείκτες και αναφορές σε σταθερές (iv)

- Σταθεροί δείκτες

```
int a = 42, z = 0;
const int b = 42;

int *p = &a;              // δείκτης σε int
const int *q = &b;        // δείκτης σε const int
int * const r = &a;       // σταθερός δείκτης σε int
r = &z;                   // απαγορεύεται !!!
*r = 17;                  // αυτό επιτρέπεται
const int * const q = &b; // σταθερός δείκτης
                          // σε const int
```

Δείκτες και αναφορές σε σταθερές (v)

```
struct person {
   char firstname[20], lastname[30];
   int phone;
   ...
};

person nickie;

void call(person p) { ... }
...
call(nickie);
```

Η κλήση προκαλεί την αντιγραφή της πραγματικής παραμέτρου **nickie** στην τυπική παράμετρο **p** (call by value!)

Δείκτες και αναφορές σε σταθερές (vi)

```
struct person {
   char firstname[20], lastname[30];
   int phone;
   ...
};

person nickie;

void call(person &p) { ... }
...
call(nickie);
```

Όχι αντιγραφή (call by reference!) αλλά η συνάρτηση **`call`** μπορεί να αλλάξει την παράμετρο **`nickie`**, αν το θέλει

Δείκτες και αναφορές σε σταθερές (vii)

```
struct person {
   char firstname[20], lastname[30];
   int phone;
   ...
};

person nickie;

void call(const person &p) { ... }
...
call(nickie);
```

Όχι αντιγραφή (call by reference!) και απαγορεύεται στη συνάρτηση **`call`** να αλλάξει την παράμετρό της

Και κάποιο συμπέρασμα

- Η συνάρτηση μεταβάλλει τις τιμές των παραμέτρων της
  - Πέρασμα ως αναφορά ή με δείκτες
- Η συνάρτηση ΔΕΝ μεταβάλλει τις τιμές των παραμέτρων της
  - Πέρασμα κατά τιμή: αντιγραφή της τιμής => κόστος σε χρόνο και μνήμη
  - Πέρασμα ως const αναφορά: δεν αντιγράφεται η τιμή => οικονομία σε χρόνο και μνήμη

Namespaces

Λίγα λόγια για namespaces (i)

- Η ζωή χωρίς το "**using namespace std;**"

```
#include <iostream>

int f(int x) {
  return x+1;
}

int main() {
  std::cout << f(41) << std::endl;
}
```

Λίγα λόγια για namespaces (ii)

- Πώς να ορίσετε το δικό σας namespace

```
#include <iostream>

namespace ns {

int cout(int x) {   // Μετονομάζω την 'f' σε 'cout'!
  return x+1;       // Κακή ιδέα!
}

}  // namespace ns

int main() {
  std::cout << ns::cout(41) << std::endl;
}
```

Λίγα λόγια για namespaces

- **QUIZ!**
  Για ποιο λόγο είναι χρήσιμα τα namespaces
  - (α) για έλεγχο της εμβέλειας των μεταβλητών
  - (β) για χρήση του ίδιου ονόματος πολλές φορές
  - (γ) για επαναχρησιμοποίηση
  - (δ) είναι χρήσιμα τα namespaces?!