ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΤΕΧΝΙΚΕΣ

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

Διδάσκοντες:

Νίκος Παπασπύρου	(nickie@softlab.ntua.gr)
Άρης Παγουρτζής	(pagour@cs.ntua.gr)
Γιώργος Στάμου	(gstam@cs.ntua.gr)
Γιώργος Γκούμας	(goumas@cslab.ece.ntua.gr)
Γιώργος Αλεξανδρίδης	(gealexandri@islab.ntua.gr)
Γιώργος Σιόλας	(gsiolas@image.ntua.gr)
Παρασκευή Τζούβελη	(tpar@image.ntua.gr)

Διαφάνειες παρουσιάσεων

26/2/21

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

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;
}
```

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)

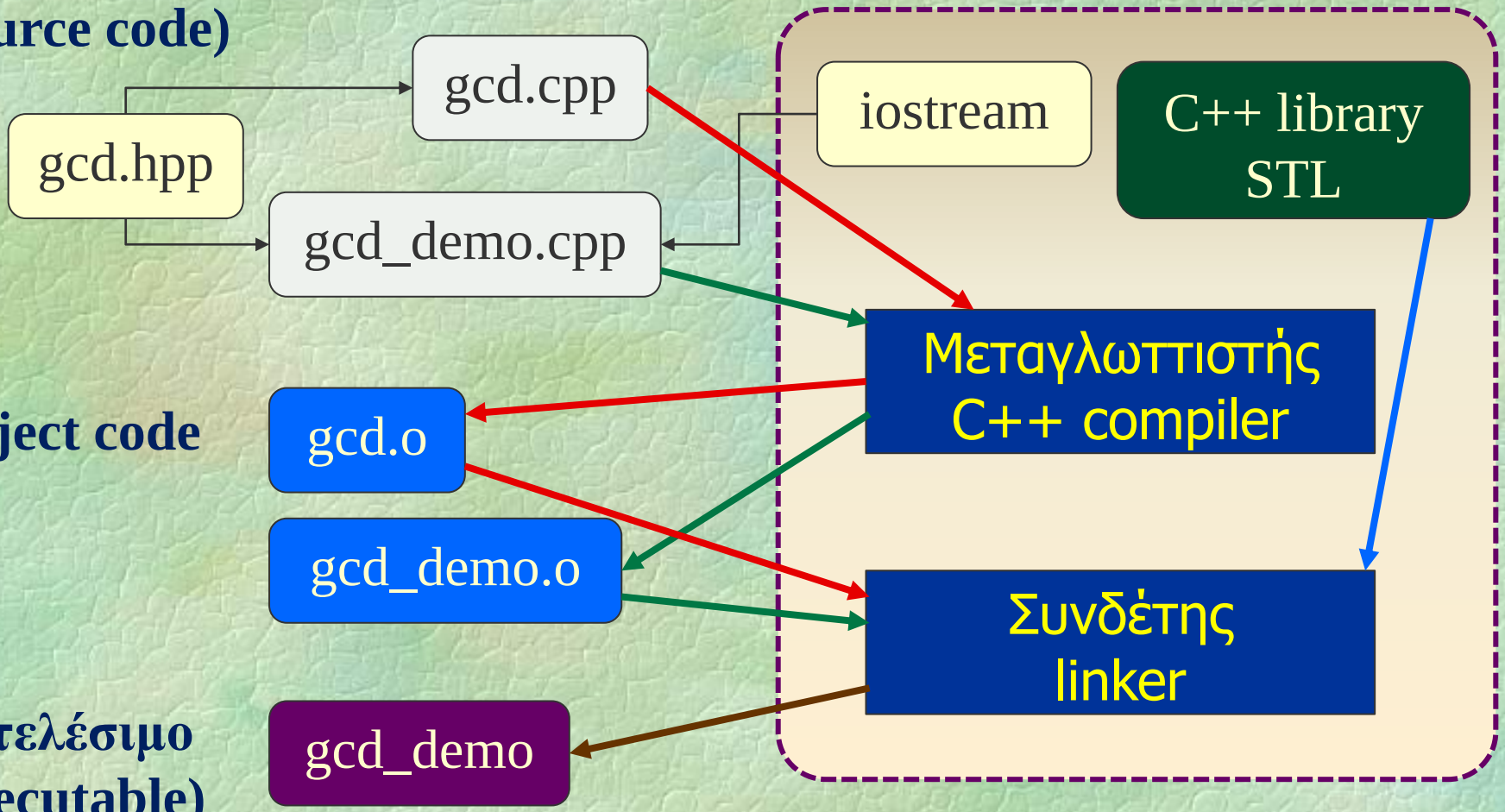
gcd.hpp

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

gcd_demo.cpp

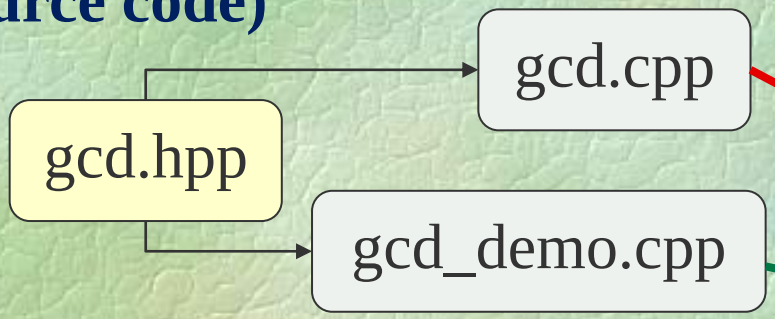
```
#include <iostream>
#include "gcd.hpp"

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)

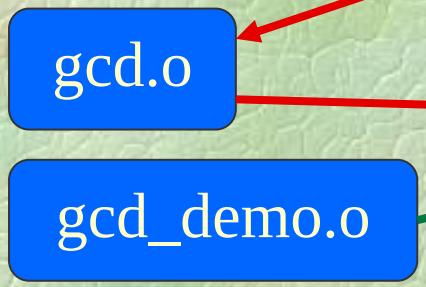


Διαδικασία μεταγλώττισης (ii)

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 (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]; ...
```

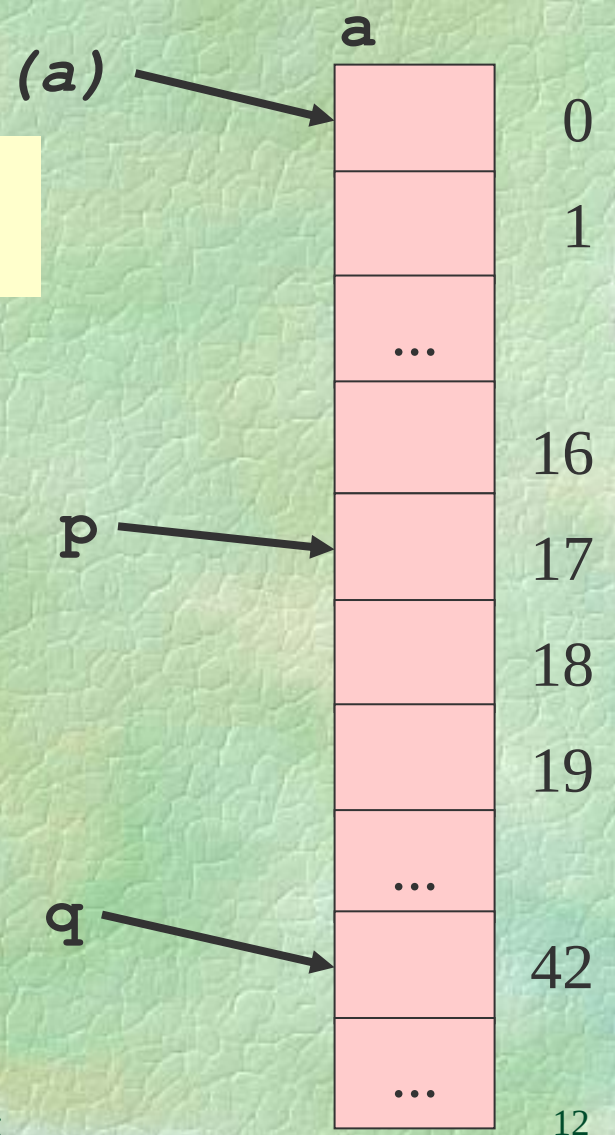
Πίνακες (ii)

- Μεγάλοι πίνακες:
  - είτε global
  - 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;
}
```

Πίνακες και δείκτες

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

// ισοδύναμα:
int *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;   → 25

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



Πέρασμα κατ' αναφορά (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 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;
a = 17;

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

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;
}
```

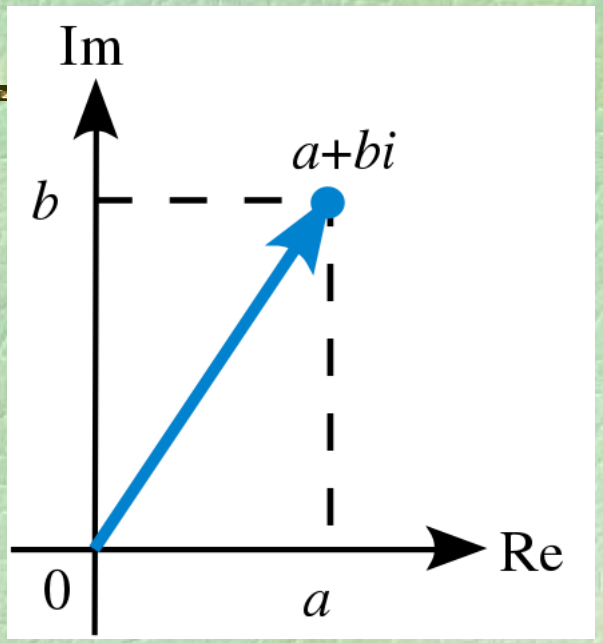
Τι είναι οι μιγαδικοί αριθμοί;

- Wikipedia: Στα μαθηματικά, οι μιγαδικοί αριθμοί είναι μία επέκταση του συνόλου των πραγματικών αριθμών με την προσθήκη του στοιχείου ***i***, που λέγεται φανταστική μονάδα, και έχει την ιδιότητα:

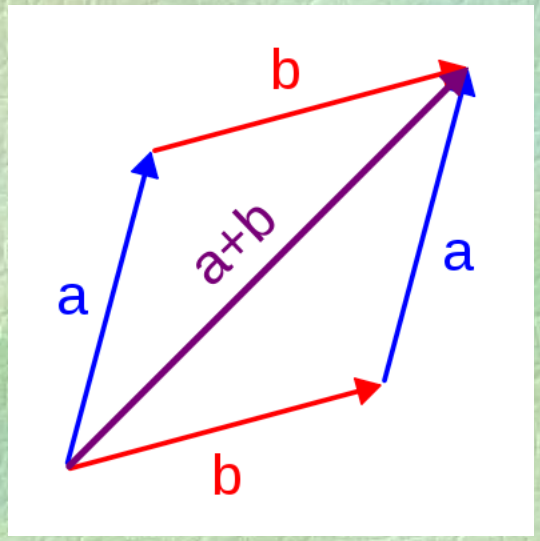
$$i^2 = -1$$

- Κάθε μιγαδικός αριθμός μπορεί να γραφτεί στη μορφή $a + ib$, όπου $a, b \in \mathbf{R}$

Μιγαδικοί αριθμοί



- a: πραγματικό μέρος
- b: φανταστικό μέρος
- Αριθμητικές πράξεις, π.χ. πρόσθεση



- Για περισσότερα, *ρωτήστε το μαθηματικό σας!*

Εισαγωγή στα αντικείμενα (i)

- Μιγαδικοί αριθμοί, με struct (όχι object-oriented)

```
struct complex { double re, im; };

complex c_make(double r, double i) {
  complex result;
  result.re = r; result.im = i;
  return result;
}

complex c_add(complex c1, complex c2) {
  return c_make(c1.re + c2.re,
                c1.im + c2.im);
}
```

Εισαγωγή στα αντικείμενα (ii)

- Μιγαδικοί αριθμοί με struct (συνέχεια)

```
void c_print(ostream &out, complex c) {
  out << c.re << "+" << c.im << "i" << endl;
}

int main() {
  complex c1 = c_make(3, 4);
  complex c2 = c_make(1, 2);
  complex c = c_add(c1, c2);
  c_print(cout, c);
  c_print(cerr, c);
}
```

Εισαγωγή στα αντικείμενα (iii)

- Κατ' αρχήν, **class ≈ struct**
- Διαφέρουν στην ορατότητα των πεδίων:
  - στο **struct** όλα είναι **public** by default
  - στο **class** όλα είναι **private** by default
- Τα αντικείμενα όμως σχεδιάζονται συνήθως με διαφορετική φιλοσοφία (object-oriented)
  - αφαίρεση δεδομένων / απόκρυψη πληροφοριών
  - ενθυλάκωση (encapsulation) = πεδία και μέθοδοι μέσα στα αντικείμενα

Εισαγωγή στα αντικείμενα (iv)

- Μιγαδικοί αριθμοί, με class (object-oriented)

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0);
   complex add(complex c);
   void print(ostream &out);
private:
   double re, im;
};
```

Δημόσιο και ιδιωτικό μέρος
Πεδία και μέθοδοι, κατασκευαστής

Εισαγωγή στα αντικείμενα (v)

```
complex::complex(double r, double i) {
  re = r; im = i;
}

complex complex::add(complex c) {
  return complex(re + c.re, im + c.im);
}

void complex::print(ostream &out) {
  out << re << "+" << im << "i" << endl;
}

int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1.add(c2);
  c.print(cout);
}
```

Εισαγωγή στα αντικείμενα (vi)

- Με υπερφόρτωση τελεστών: + και <<

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0);
   friend complex operator+(complex c1,
                            complex c2);
   friend ostream& operator<<(ostream &out,
                              complex c);
private:
   double re, im;
};
```

Εισαγωγή στα αντικείμενα (vii)

```
complex operator+(complex c1, complex c2) {
  return complex(c1.re + c2.re,
                 c1.im + c2.im);
}

ostream& operator<<(ostream &out,
                    complex c) {
  out << c.re << "+" << c.im << "i";
  return out;
}

int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1 + c2;
  cout << c << endl;
}
```

Ορισμός μεθόδων **εκτός** κλάσης

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0);
   double norm();
   ...
};
```

complex.hpp

```
complex::complex(double r, double i) {
   re = r; im = i;
}

double complex::norm() {
   return sqrt(re * re + im * im);
}
```

complex.cpp

Ίσως και σε ξεχωριστά αρχεία!

Ορισμός μεθόδων **εντός** κλάσης

```
class complex {
public:
   complex(double r = 0.0, double i = 0.0) {
      re = r; im = i;
   }
   double norm() {
      return sqrt(re * re + im * im);
   }
   ...
};
```

complex.hpp

Αναγκαστικά στο ίδιο αρχείο!

- Υλοποίηση: μέθοδοι `inline`

Καταστροφείς (destructors) (i)

- Καλούνται όταν καταστρέφονται τα αντικείμενα (αποδεσμεύεται η μνήμη τους)

```
class complex {
public:
   ...
   ~complex() {
      cout << "a complex number dies... :-("
           << endl;
   }
   ...
};
```

Καταστροφείς (destructors) (ii)

```
int main() {
  complex c1(3, 4), c2(1, 2);
  complex c = c1 + c2;
  cout << c << endl;
}
```

- Ο καταστροφέας θα κληθεί τρεις φορές για τα αντικείμενα `c1`, `c2` και `c`, όταν τελειώσει η `main` .
- Το ίδιο και εδώ (γιατί;)

```
int main() {
  complex c1(3, 4), c2(1, 2);
  cout << c1 + c2 << endl;
}
```

Μέθοδοι και φίλες συναρτήσεις (i)

- Οι τελεστές με δύο τελούμενα μπορούν να υλοποιηθούν με δύο τρόπους
- #1: ως μέθοδοι της κλάσης

```
class complex {
public:
   ...
   complex operator+(const complex &y) {
      return complex(re + y.re, im + y.im);
   }
   ...
};
```

Μέθοδοι και φίλες συναρτήσεις (ii)

- Οι τελεστές με δύο τελούμενα μπορούν να υλοποιηθούν με δύο τρόπους
- #2: ως φίλες συναρτήσεις

```
class complex {
public:
  ...
  friend complex operator+(const complex &x,
                           const complex &y) {
    return complex(x.re + y.re, x.im + y.im);
  }
  ...
};
```

Μέθοδοι και φίλες συναρτήσεις (iii)

- Και στις δύο περιπτώσεις η χρήση είναι ίδια: `c1 + c2`
- Στην περίπτωση #1 το αντικείμενο `c1` είναι αυτό για το οποίο καλείται η μέθοδος και το `c2` είναι η παράμετρος
- Στην περίπτωση #2 και τα δύο αντικείμενα είναι παράμετροι (πρόκειται για συνάρτηση, όχι για μέθοδο)

Μέθοδοι και φίλες συναρτήσεις (iv)

- Ο τελεστής εκτύπωσης όμως μπορεί να υλοποιηθεί μόνο ως φίλη συνάρτηση (γιατί;)

```
class complex {
public:
  ...
  friend ostream& operator<<(ostream &out,
                             const complex &x) {
    out << x.re << " + " << x.im << "i";
    return out;
  }
  ...
};
```

Υπερφόρτωση (overloading) (i)

```
int  f() { ... }                          // 1
int  f(int x) { ... }                     // 2
void f(bool x, int y) { ... }             // 3
int  f(const complex &c) { ... }          // 4
```

- Διαφέρουν στο πλήθος και τους τύπους των παραμέτρων

```
f();                          // 1
f(42);                        // 2
f(true, 17);                  // 3
complex i(0, 1);
f(i);                         // 4
f(42.0);                      // 4  γιατί;
```

Υπερφόρτωση (overloading) (ii)

```
int  f(int x) { ... }                    // 2
```

- Απαγορεύεται να προστεθεί η παρακάτω, που διαφέρει από την 2 μόνο στον τύπο επιστροφής

```
bool f(int y) { ... }
```

- Συμπέρασμα: πολύπλοκος μηχανισμός
  - περιπλέκεται λόγω των αυτόματων μετατροπών (coercions)
  - να χρησιμοποιείται με φειδώ

Περισσότεροι κατασκευαστές (i)

- Με χρήση υπερφόρτωσης
- Κοινός κατασκευαστής

```
complex(double r, double i) {
  re = r; im = i;
}
```

ή ισοδύναμα:

```
complex(double r, double i):
  re(r), im(i) {}
```

Η τελευταία μορφή διαχωρίζει την αρχικοποίηση των πεδίων του αντικειμένου από άλλες εντολές.

Περισσότεροι κατασκευαστές (ii)

- Default constructor

```
complex() { re = im = 0; }
```

ή ισοδύναμα:

```
complex(): re(0), im(0) {}
```

Κατασκευαστής χωρίς παραμέτρους, καλείται αυτόματα όταν έχουμε π.χ.

```
complex c;
```

Αν δεν οριστεί άλλος κατασκευαστής για μία κλάση, η C++ ορίζει αυτόματα έναν default

Περισσότεροι κατασκευαστές (iii)

- Copy constructor

```
complex(const complex &c) {
  re = c.re; im = c.im;
}
```

ή ισοδύναμα:

```
complex(const complex &c):
  re(c.re), im(c.im) {}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
complex c1(c);
complex c2 = c;
```

Περισσότεροι κατασκευαστές (iv)

- Copy constructor
  - Αν δεν οριστεί για μία κλάση, η C++ ορίζει αυτόματα έναν copy constructor που αντιγράφει ένα-ένα τα πεδία του αντικειμένου
  - Αυτό δεν είναι πάντα το επιθυμητό *γιατί;* (θα το συζητήσουμε σε λίγο στο παράδειγμα με **`weight`** και **`totalWeight`**)

Περισσότεροι κατασκευαστές (v)

- Κατασκευαστής που επιτρέπει αυτόματη μετατροπή τύπου

```
complex(double r) { re = r; im = 0; }
```

ή ισοδύναμα:

```
complex(double r): re(r), im(0) {}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
complex c1(5);
complex c2 = 5;

void f (const complex &c);
f(5);
```

Τελεστής ανάθεσης

```
const complex& operator=(const complex &y) {
  re = y.re;
  im = y.im;
  return *this;   // Αναφορά στο τρέχον αντικείμενο!
}
```

Καλείται αυτόματα όταν έχουμε π.χ.

```
c1 = c2;
```

Αν δεν οριστεί για μία κλάση, η C++ ορίζει αυτόματα έναν τελεστή ανάθεσης που αντιγράφει ένα-ένα τα πεδία του αντικειμένου

Μέθοδοι const (i)

```
class Person {
public:
   Person(double w): weight(w) {}
   double getWeight() {
      return weight;
   }
   void setWeight(double w) {
      weight = w;
   }
private:
   double weight;
};
```

Μέθοδοι const (ii)

```
int main() {
   Person me(75);
   cout << me.getWeight() << endl;
   me.setWeight(76);
   cout << me.getWeight() << endl;
   const Person constantlyFatPanda(630);
   // Αυτό είναι λάθος:
   constantlyFatPanda.setWeight(730);
   // Αλλά δυστυχώς και αυτό είναι λάθος:
   cout << constantlyFatPanda.getWeight()
        << endl;
}
```

Μέθοδοι const (iii)

```
class Person {
public:
   Person(double w): weight(w) {}
   double getWeight() const {
      return weight;
   }
   void setWeight(double w) {
      weight = w;
   }
private:
   double weight;
};
```

Αυτή η μέθοδος μπορεί να καλείται για σταθερά (**const**) αντικείμενα

Αυτή όχι!

Μέθοδοι const (iv)

```
int main() {
   Person me(75);
   cout << me.getWeight() << endl;
   me.setWeight(76);
   cout << me.getWeight() << endl;
   const Person constantlyFatPanda(630);
   // Αυτό εξακολουθεί να είναι λάθος:
   constantlyFatPanda.setWeight(730);
   // Αλλά τώρα αυτό είναι σωστό:
   cout << constantlyFatPanda.getWeight()
        << endl;
}
```

Στατικά πεδία και μέθοδοι (i)

```
class Person {
public:
   Person(double w);
  ~Person();
            double getWeight();
   static   double getTotalWeight();
private:
            double weight;
   static   double totalWeight;
};

double Person::totalWeight = 0.0;
```

Στατικά πεδία και μέθοδοι (ii)

```
Person::Person(double w): weight(w) {
   totalWeight += weight;
}
Person::~Person() {
   totalWeight -= weight;
}
double Person::getWeight() {
   return weight;
}
double Person::getTotalWeight() {
   return totalWeight;
}
```

Στατικά πεδία και μέθοδοι (iii)

```
int main() {
  Person x(65), y(100);
  cout << x.getWeight()
       << endl
       << x.getTotalWeight()
       << endl;
  Person *z = new Person(85);
  cout << Person::getTotalWeight()
       << endl;
  delete z;
  cout << y.getTotalWeight()
       << endl;
}
```

65

165

250

165