Προγραμματισμός Ηλεκτρονικών Υπολογιστών

Παράδοση 11/1/2019, Νίκος Παπασπύρου.

Λίγα για namespaces

1. Η ζωή χωρίς το `using namespace std;`

```
#include <iostream>

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

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

2. Πώς να ορίσετε το δικό σας namespace.

```
#include <iostream>

namespace nickie {

int cout(int x) {  // I'm dumb enough to name f as 'cout' !
  return x+1;
}

}  // namespace nickie

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

Απλά αντικείμενα

0. Τι είναι οι μιγαδικοί αριθμοί; Διαβάστε το αντίστοιχο λήμμα στη Wikipedia.

1. Μιγαδικοί αριθμοί με `struct`.

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

struct complex {
  double re, im;
};

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

```
complex add_complex(complex c1, complex c2) {
  complex result;
  result.re = c1.re + c2.re;
  result.im = c1.im + c2.im;
  return result;
}

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

int main() {
  complex c1 = make_complex(3, 4);
  complex c2 = make_complex(1, 2);
  complex c = add_complex(c1, c2);
  print_complex(cout, c);
  print_complex(cerr, c);
}
```

2. Μιγάδικοί αριθμοί με `class`.

   Τα αντικείμενα συνδυάζουν τα δεδομένα (`re` και `im`) με τις πράξεις που εφαρμόζονται πάνω σε αυτά (`make`, `add` και `print`). Το δημόσιο μέρος ενός αντικείμενου (`public`) περιέχει συνήθως τις πράξεις ενώ το ιδιωτικό (`private`) περιέχει τα δεδομένα. Ο λόγος είναι η **αφαίρεση δεδομένων** (data abstraction) και η **απόκρυψη πληροφοριών** (information hiding). Δηλαδή, τα αντικείμενα μοιάζουν με τους αφηρημένους τύπους δεδομένων, στους οποίους το ιδιωτικό μέρος (η υλοποίηση) δεν είναι ορατό στους χρήστες τους. Η υλοποίηση (το ιδιωτικό μέρος) μπορεί να μεταβληθεί χωρίς να χρειαστεί να μεταβληθεί ο κώδικας που χρησιμοποιεί την κλάση (δηλαδή το δημόσιο μέρος της).

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

class complex {
public:
  static complex make(double r, double i);
  static complex add(complex c1, complex c2);
  static void print(ostream &out, complex c);
private:
  double re, im;
};

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

complex complex::add(complex c1, complex c2) {
  complex result;
  result.re = c1.re + c2.re;
  result.im = c1.im + c2.im;
  return result;
}
```

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

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

Προσέξτε τη χρήση της λέξης `static` μπροστά από τις μεθόδους `make`, `add` και `print`. Στην πραγματικότητα, αυτές οι τρεις δεν είναι παρά απλές συναρτήσεις που ονομάζονται πλέον `complex::make`, `complex::add` και `complex::print`. Βρίσκονται δηλωμένες μέσα στην κλάση έτσι ώστε να μπορούν να προσπελάζουν τα ιδιωτικά δεδομένα των αντικειμένων.

3. Προσθήκη μεθόδων και κατασκευαστή.

   Η `add` και `print` μπορούν να γραφούν ως μέθοδοι των αντικειμένων της κλάσης `complex`, καταργώντας τη μία παράμετρο. Π.χ., η κλήση

   ```
   complex::add(c1, c2)
   ```

   θα γράφεται τώρα

   ```
   c1.add(c2)
   ```

   Η μέθοδος `make` είναι στην πραγματικότητα ένας κατασκευαστής: κατασκευάζει ένα μιγαδικό αριθμό αν δοθεί το πραγματικό και το φανταστικό μέρος του.

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

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

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

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

   complex complex::add(complex c) {
     complex result;
   ```

```
  result.re = re + c.re;
  result.im = im + c.im;
  return result;
}

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

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

Προσοχή στα εξής σημεία:

- Οι **κατασκευαστές** (constructors) έχουν πάντοτε το ίδιο όνομα με την κλάση. Δεν έχουν τύπο επιστροφής.
- Παραπάνω ορίζονται δύο κατασκευαστές. Αυτός με τις δύο παραμέτρους αντιστοιχεί στην πράξη `make` που είδαμε προηγουμένως.
- Ο κατασκευαστής χωρίς παραμέτρους (default constructor) είναι απαραίτητος για να μπορεί να οριστεί η τοπική μεταβλητή `result` στη μέθοδο `add`.

4. Υπερφόρτωση – overloading.

   Στο παραπάνω παράδειγμα θα μπορούσαν να οριστούν και τρεις κατασκευαστές με επικεφαλίδες:

```
complex();
complex(double r);
complex(double r, double i);
```

   και αντίστοιχες υλοποίησεις:

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

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

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

   Ο μεταγλωττιστής είναι σε θέση να γνωρίζει ποιον θα καλέσει βλέποντας τα ορίσματα που έχουν δοθεί.

   Αυτό ονομάζεται **υπερφόρτωση** (overloading) και στη C++ μπορεί να γίνει με συναρτήσεις, τελεστές, μεθόδους και κατασκευαστές.

5. Default arguments.

   Βέβαια, στο προηγούμενο παράδειγμα, οι τρεις υπερφορτωμένες εκδόσεις του κατασκευαστή της κλάσης `complex` θα μπορούσαν να γραφούν σαν ένας μόνο κατασκευαστής, του οποίου οι τιμές των παραμέτρων εννοούνται αν παραλείπονται:

```
complex(double r = 0, double i = 0);
```

6. Με υπερφόρτωση τελεστών.

   Θα θέλαμε αντί add και print να χρησιμοποιούμε τους τελεστές + και << της C++.

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

class complex {
public:
  complex(double r=0, double i=0);
  friend complex operator+(complex c1, complex c2);
  friend ostream& operator<<(ostream &out, complex c);
private:
  double re, im;
};

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

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

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

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

   Προσοχή στα εξής σημεία:

   - Οι μέθοδοι add και print έχουν μετατραπεί σε υπερφορτωμένες εκδόσεις των τελεστών + και << (ο τελευταίος είναι αυτός που στέλνει ένα αντικείμενο στο cout).
   - Οι δύο τελεστές δεν είναι πλέον μέθοδοι, είναι **φίλες** (friend) συναρτήσεις της κλάσης. Αν δεν ήταν φίλες, δεν θα μπορούσαν να έχουν πρόσβαση στα ιδιωτικά πεδία re και im.
   - Ο τελεστής << παίρνει ως πρώτη παράμετρο μία αναφορά σε ostream, που ορίζεται στο <iostream> και είναι ο τύπος του cout.
   - Ο τελεστής << επιστρέφει ostream και μάλιστα επιστρέφει το ίδιο αντικείμενο που παίρνει ως πρώτη παράμετρο. Αυτό γίνεται για να μπορεί να χρησιμοποιείται πολλές φορές, όπως στην τελευταία γραμμή της main, που είναι ισοδύναμη με:

     ```
     (cout << c) << endl;
     ```