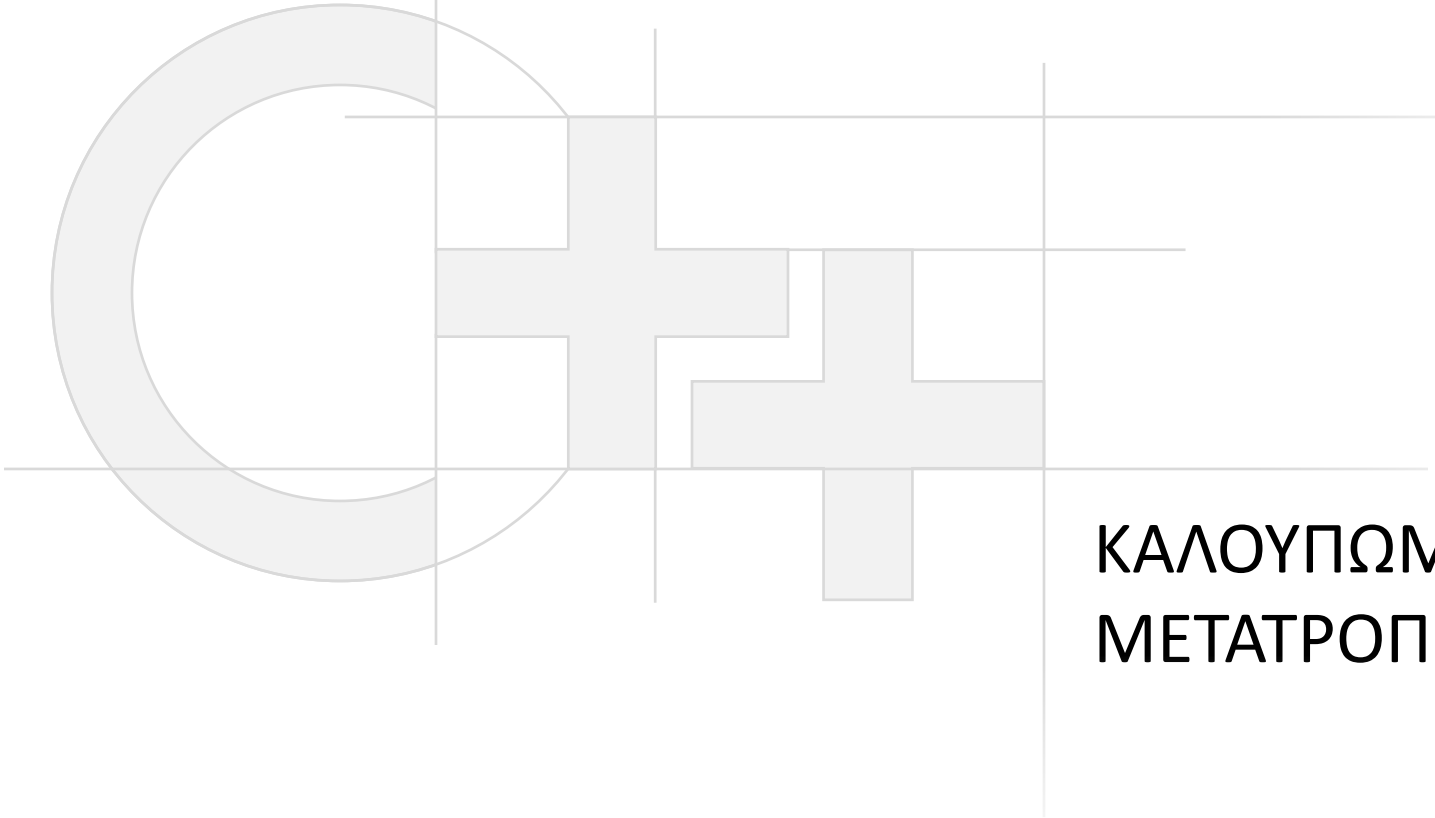
Προγραμματισμός Υπολογιστών με C++



ΚΑΛΟΥΠΩΜΑΤΑ & ΜΕΤΑΤΡΟΠΕΣ

Γεώργιος Παπαϊωάννου (2013-14)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Είδη καλουπωμάτων
  - Μετατροπές και έλεγχοι τύπου

- Τελευταία ενημέρωση: Οκτώβριος 2013

Μετατροπή και Προαγωγή Τύπων

- Όπως και στη Java, στη C++ γίνεται έμμεση μετατροπή (implicit cast) βασικών τύπων (π.χ. int →float)
- Γίνεται αυτόματα η άμεση μετατροπή κλάσεων σε οποιαδήποτε άλλη διαθέτει κατασκευαστή που παίρνει ως όρισμα την πρώτη
- Μπορούμε να αρχικοποιήσουμε στιγμιότυπα κλάσεων με άλλους τύπους (βλ. προϋποθέσεις παρακάτω)
  - Π.χ. string str = "abc" // Το "abc" είναι const char [ ]
- Μπορούμε να μετατρέψουμε απευθείας μια κλάση στη βασική αυτής ή άλλο πρόγονο (up-casting), χωρίς καλούπωμα
  - Π.χ. istream * stream = new ifstream("file.txt");

Μετατροπή σε Κλάση (Implicit Cast)

- Για να μπορούμε να πραγματοποιήσουμε μια μετατροπή:
  - Δημιουργούμε έναν κατασκευαστή που παίρνει ως όρισμα τον τύπο της τιμής ανάθεσης
- Παράδειγμα:

```
class A
{
    float value;
public:
       A(int val)
       {
           value = val;          // εδώ έχουμε προαγωγή τύπου
       }
};
…
A data = 13;                   // «ανάθεση τιμής» με κλήση του κατασκευαστή
```

Ρητή Μετατροπή σε Κλάση (Explicit Cast)

- Αν θέλουμε να απαγορεύσουμε την παραπάνω συμπεριφορά μιας κλάσης:
  - Δηλώνουμε ως explicit τον κατασκευαστή
- Παράδειγμα:

```
class A {
    float value;
public:
    explicit A(int val) {   // ρητή δήλωση κατασκευαστή
       value = val;
    }
};
…
A data = 13;                // ανάθεση με κλήση του explicit κατασκευαστή
                            // απαγορεύεται. Σωστό: A data = A(13);
```

Τελεστές Μετατροπής Τύπου 1

```
class rtResult
{
public:
   enum rt_result_code_t { RT_OK=0, RT_INVALID_PARAMS, RT_INVALID_OPERATION };

   rt_result_code_t code = RT_OK;
   std::string message;

   rtResult() {};
   rtResult(rt_result_code_t c, const std::string m = "") {
      message = m; code = c;
   }

   operator bool() const {
      return code == RT_OK;
   }
};
```

Τελεστές Μετατροπής Τύπου 2

- Μας επιτρέπουν να μετατρέψουμε εμείς ρητά μια κλάση σε κάποια άλλη ή κάποιο βασικό τύπο.
- Στο παράδειγμά μας, μπορώ να γράψω πλέον:

```
rtResult res = ParseCommandLineArgs();
if (!res) {  // εδώ μετατρέπουμε το αντικείμενο rtResult σε bool
   std::cout << res.message;
   return;
}
```

Τι Έγινε το Κλασικό Καλούπωμα (...) ;

- Υπάρχει και ισχύει
- Είναι κι αυτό explicit cast και μάλιστα:
  - Επιτρέπει μετατροπή σε συμβατούς βασικούς τύπους
  - Επιτρέπει μετατροπή σε κλάσεις ίδιου επιπέδου ή προγόνους (αλλά προφανώς δε χρειάζεται)
  - Δεν επιτρέπει άμεση μετατροπή σε απόγονο – down-casting (εκτός κι αν υπάρχει κατάλληλος κατασκευαστής, τον οποίο και καλεί)
  - Προσοχή: **Επιτρέπει μετατροπή από οποιονδήποτε δείκτη σε οποιονδήποτε άλλο χωρίς έλεγχο**! Αν δεν πρέπει να γίνει η μετατροπή, ο compiler δε θα βγάλει σφάλμα!

Καλούπωμα ( ) - Παραδείγματα

```
class Bottle {
private:
    float volume;
public:
    Bottle(float vol): volume(vol) {};
    const float getVolume() {return volume;}
};

class SmallBottle : public Bottle {
public:
    SmallBottle() : Bottle(1.0f) {}
};

int main() {
    Bottle bottle = Bottle(2.0f);
    SmallBottle bs = (SmallBottle)bottle; // Down-casting: Απαγορεύεται. Δεν
                                          // υπάρχει κατασκευαστής που να παίρνει
                                          // όρισμα Bottle
    SmallBottle * p_bs = (SmallBottle *) &bottle; // Επιτρέπεται! Μπορεί να
    return 0;                                     // προκαλέσει σοβαρά run-time
}                                                 // σφάλματα
```

Άλλα Καλουπώματα

- Ναι, στη C++ υπάρχουν άλλες 4 μορφές καλουπώματος, όχι μόνο 1!
- Διαφέρουν στο επίπεδο που σέβονται ή παρακάμπτουν την κληρονομικότητα
- Κάποια αφορούν ειδικές μετατροπές
- Διαφέρουν στην αλγοριθμική πολυπλοκότητά τους
- Έχουμε:
  - static_cast
  - const_cast
  - dynamic_cast
  - reinterpret_cast

static_cast

- Δήλωση: `static_cast<return_type> ( expression )`
- Επιτρέπει τη μετατροπή δεικτών σε κλάσεις και προς τα κάτω στην κληρονομική αλυσίδα (down-casting)
  - Αντιγράφει τα κοινά πεδία
  - Δεν εξασφαλίζει τι γίνεται με τα επιπρόσθετα πεδία
  - Δεν ελέγχει

static_cast – Παράδειγμα Κακής Χρήσης

```
class Bottle {
private:
    float volume;
public:
    Bottle(float vol): volume(vol) {};
    const float getVolume() {return volume;}
    const bool operator< (const Bottle & right) const {return volume<right.volume;}
};

class SmallBottle : public Bottle {
public:
    string label;
SmallBottle() : Bottle(1.0f), label("bell") {}
};
...
Bottle bottle = Bottle(2.0f);
SmallBottle bs = * static_cast<SmallBottle*>(&bottle); // Δε βγάζει compile error, όμως:
       // Κατά την αντιγραφή του αντικειμένου (default copy constructor)
       // πάει να αντιγράψει το label από μνήμη εκτός του αντικειμένου (δεν υπάρχει)
       // στην κλάση Bottle! →Access Violation
```

dynamic_cast

- Δήλωση: `dynamic_cast<return_type> ( expression )`
- **Κάνει ασφαλή μετατροπή, όπου γίνεται, μεταξύ δεικτών πολυμορφικών τύπων**, ακόμα και με πολλαπλή κληρονομικότητα
- Αν αποτύχει η μετατροπή, επιστρέφεται ο (nullptr)
- Σημείωση: το dynamic_cast είναι αργό σε σχέση με τα υπόλοιπα καλουπώματα

dynamic_cast και RTTI

- Χρησιμοποιεί το RTTI (Run-Time Type Information) των κλάσεων, γι αυτό, **πρέπει να υπάρχει πολυμορφική μορφή** στην `expression`:
  - Πρέπει να έχει δηλωθεί τουλάχιστον μια μέθοδος ως virtual

Προσοχή:

- Αν δεν έχει RTTI η κλάση στην οποία εφαρμόζουμε το dynamic cast, τότε αυτό είναι compilation error

dynamic_cast: Υλοποίηση μιας instanceof ( ) (1)

- Το dynamic_cast αντιστοιχεί με ένα συνδυασμό instanceof της Java συν την τελική μετατροπή στον τύπο ελέγχου
- Οπότε μπορούμε να φτιάξουμε μια αντίστοιχη συνάρτηση της instanceof:

```
#define instanceof(A,B) ( dynamic_cast<B*>(&A)==nullptr ?     \\
                          false : true )
```

dynamic_cast: Υλοποίηση μιας instanceof ( ) (2)

```
#define instanceof(A,B) ( dynamic_cast<B*>(&A)==nullptr ? false:true )

class Bottle {
private:
   float volume;
public:
   Bottle(float vol): volume(vol) {};
   virtual const float getVolume() {return volume;}
   const bool operator< (const Bottle & right) const {return
volume<right.volume;}
};
...

SmallBottle bottle = SmallBottle();
bool dc = instanceof(bottle,Bottle);   // true
```

const_cast

- Δήλωση: `const_cast<return_type> ( expression )`
- Το const_cast χρησιμοποιείται για να «επιβάλει» ή να «αναιρέσει» το const από μια μεταβλητή, συνήθως κατά την ανάθεσή ή τη χρήση της ως όρισμα
- Χρήσιμο όταν περνάμε ορίσματα που διαφέρουν μόνο ως προς το const και ξέρουμε ότι δεν θα επηρεάσει το casting τη λειτουργία των συναρτήσεων

reinterpret_cast

- Δήλωση: `reinterpret_cast<return_type> ( expression )`
- Το πιο «ισχυρό» cast operation
- Αγνοεί παντελώς τους τύπους που πρόκειται να μετατραπούν!
- Βλέπει το expression σαν ένα block μνήμης το οποίο και το «βαφτίζει» ως κάτι άλλο
- Ουσιαστικά μας επιτρέπει να δούμε οτιδήποτε ως οποιονδήποτε τύπο!
- Προφανώς είναι επικίνδυνη η χρήση του!
- Αλλά ταυτόχρονα χρήσιμη (βλ. συνέχεια)

reinterpret_cast: Πότε;

- Το reinterpret_cast ενδέχεται να μας χρειαστεί όταν πρέπει να περάσουμε δεδομένα γνωστού τύπου ως ορίσματα που είναι αυστηρά προσδιορισμένα αλλιώς

Παράδειγμα:

- Έστω ότι έχουμε μια συνάρτηση που χειρίζεται events και που παίρνει 2 ορίσματα:
  - Τον τύπο του event
  - Τα δεδομένα του event
- Επειδή το τι είναι τα δεδομένα εξαρτάται από το είδος του μηνύματος, αυτό το γνωρίζουμε μόνο κατά την εκτέλεση
- Όμως, αν μας έρθει ένας τύπος μηνύματος, τότε σίγουρα ξέρουμε τον τύπο των δεδομένων και μπορούμε να κάνουμε άμεσα τη μετατροπή:

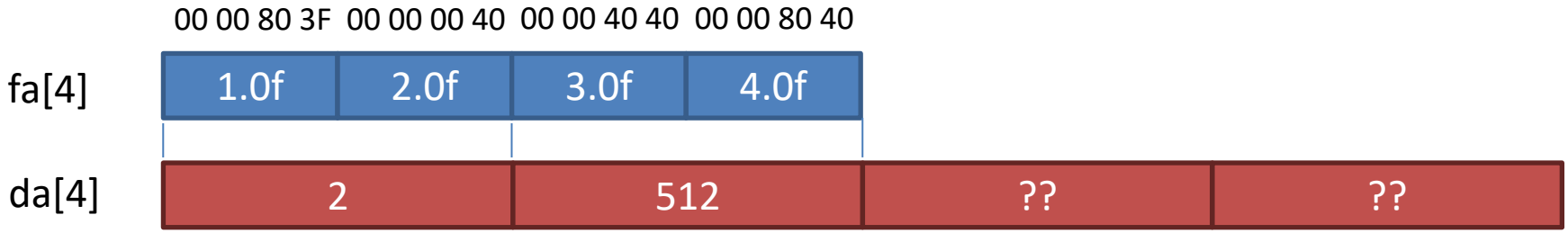
reinterpret_cast: Παράδειγμα

```
void processMessage( unsigned int msg_type, void * msg_data )
{
    switch (msg_type)
    {
    case EVENT_MOUSE_DOWN:
    case EVENT_MOUSE_UP:
        MouseData * data = reinterpret_cast<MouseData *> (msg_data);
        data->...
        break;
    case EVENT_WINDOW_CLOSE:
        Window * wnd = reinterpret_cast<Window *> (msg_data);
        wnd->...
        break;
    ...
    }
}
```

reinterpret_cast: (Αντι) Παράδειγμα

```
float fa[] = {1.0f, 2.0f, 3.0f, 4.0f};
double * da = reinterpret_cast<double*>(fa);
cout << da[0] << " " << da[1] << " " << da[2] << " " << da[3];
// τυπώνει: 2  512  -9.25596e+061  -9.25596e+061
```

Διότι:



reinterpret_cast: Άλλη χρήση

- Όταν υλοποιούμε πρωτόκολλα πάνω από χαμηλού επιπέδου κανάλια μεταφοράς (streams από bytes), π.χ. sockets:
  - Εμείς διαβάζουμε μια ακολουθία από bytes
  - Οργανώνουμε τα bytes se ομάδες ίσες με το sizeof() της δομής που ξέρουμε ότι μας έρχεται
  - Εκτελούμε ένα reinterpret_cast πάνω τους ώστε να τα "δούμε" ως τον τύπο που περιμένουμε να λάβουμε

Ο Τύπος μιας Κλάσης

- Ο compiler δημιουργεί επιπρόσθετη πληροφορία για τις κλάσεις και ανά στιγμιότυπο (RTTI – run-time type information) ώστε να:
- Μπορεί να αποφαίνεται για τον ακριβή τύπο μιας κλάσης
- Αυτό γίνεται με τον built-in τελεστή typeid:

```
Bottle * bs = new SmallBottle();
cout << typeid(bs).name();   // τυπώνει class Bottle *
cout << typeid(*bs).name();  // τυπώνει class SmallBottle
cout << (typeid(Bottle)==typeid(*bs)); // Μπορώ να ζητήσω τον τύπο
                                       // της κλάσης και του instance
                                       // αυτής. Ισχύει ο τελεστής ==
```

typeid

- Προσοχή: Στο παραπάνω παράδειγμα, ο έλεγχος

```
(typeid(Bottle)==typeid(*bs))
```

  δίνει αποτέλεσμα false, καθώς ο ακριβής τύπος των δύο κλάσεων διαφέρει

- Δεν εξετάζεται η κληρονομικότητα. Έλεγχο κληρονομικότητας κάνουμε με dynamic_cast<>().

decltype ή πώς να «κλέψεις» τον τύπο κάποιου (1)

- Χρησιμεύει όταν έχουμε εξαρτημένους τύπους
- Μπορώ να δηλώσω ότι μια μεταβλητή χρησιμοποιεί τον τύπο μιας άλλης:
  - Μεταβλητής
  - Ολόκληρης παράστασης

```
int i = 33;
decltype(i) j = i * 2;
```

final: Απαγορεύοντας τις overridden μεθόδους (1)

Αν θέλουμε μια μέθοδος να μη μπορεί σε παράγωγη κλάση να ξαναοριστεί, μπορούμε να τη δηλώσουμε final:

```
struct Base {
    virtual void foo();
};

struct A : Base {
    void foo() final; // overridden and final
    void bar() final; // Error: non-virtual function
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it's final in A
};

struct C : B // Error: B is final
{
};
```