Αντικειμενοστραφής Προγραμματισμός

Μέρος 4ο: Πολυμορφισμός

Εξάμηνο Σπουδών: 6ο

Κωδικός Μαθήματος: 647

Τμήμα Μαθηματικών
Πανεπιστήμιο Ιωαννίνων

Μιχάλης Α. Μπέκος
bekos@uoi.gr

Περιεχόμενα

- Πολυμορφισμός
  - Εικονικές συναρτήσεις (virtual functions)
  - Καθυστερημένη σύνδεση (late binding)
  - Υλοποίηση εικονικών συναρτήσεων
  - Αφηρημένες κλάσεις (abstract classes)
  - Αμιγώς εικονικές συναρτήσεις (pure virtual functions)

- Δείκτες και εικονικές συναρτήσεις
  - Συμβατότητα τύπων
  - Μετατροπή τύπων (casting)

Πολυμορφισμός και εικονικές συναρτήσεις

- Τι είναι ο πολυμορφισμός;
  - Θεμελιώδης έννοια του αντικειμενοστρεφούς προγραμματισμού
  - Αφορά στο συσχετισμό πολλών αποδόσεων σε μία συνάρτηση
- Ο πολυμορφισμός στη C++ υποστηρίζεται μέσω των εικονικών συναρτήσεων που υποστηρίζουν την παραπάνω δυνατότητα
- Μια εικονική συνάρτηση μπορεί να “χρησιμοποιηθεί πριν οριστεί”
- Ο πολυμορφισμός εξηγείται καλύτερα με ένα παράδειγμα

Το παράδειγμα Shapes

- Υπάρχουν διάφορες κλάσεις για είδη σχημάτων (shapes):
  - Ορθογώνια, κύκλοι, τρίγωνα κλπ.
  - Κάθε σχήμα είναι ένα αντικείμενο διαφορετικής κλάσης
    - Rectangle: `height, width, center point`
    - Circle: `center point, radius`
- Όλες οι κλάσεις αυτές όμως κληρονομούν από τη γονική κλάση: Shape
- Πρόβλημα: Η υποστήριξη της συνάρτησης μέλους `area()`
  - διαφορετική υλοποίηση για κάθε σχήμα
  - όλα τα σχήματα θα πρέπει να την υποστηρίξουν

Το παράδειγμα Shapes

- Υπενθύμιση του προβλήματος: Κάθε κλάση θα πρέπει να υποστηρίζει τη συνάρτηση μέλος `area()` με διαφορετική υλοποίηση

- Μια πιθανή προσέγγιση στο πρόβλημα θα ήταν η εξής:
  - Να υλοποιηθεί η συνάρτηση `area()` σε κάθε παράγωγη κλάση
  - Σε αυτή την περίπτωση ο παρακάτω κώδικας είναι σωστός

```
int main() {
    Rectangle r;  //Create a rectangle
    Circle c;     //and a circle

    cout << r.area() << endl;     //Calls Rectangle class's area()
    cout << c.area() << endl;     //Calls Circle class's area()

    return 0;
}
```

Ένα νέο πρόβλημα που προκύπτει με αυτή την προσέγγιση

- Θεωρήστε τη συνάρτηση `empty()` μέλος της κλάσης `Shape`, η οποία ελέγχει αν ένα σχήμα είναι τετρημένο (σ.σ., έχει μηδενικές διαστάσεις)
  - Αρχικά υπολογίζει το εμβαδόν του σχήματος και αν αυτό είναι μηδέν επιστρέφει `true`
  - Οπότε, η συνάρτηση `Shape::empty()` θα χρησιμοποιούσε τη συνάρτηση `area()` για τον υπολογισμό του εμβαδού
- Δημιουργείται έτσι το εξής πρόβλημα:
  - Ποια συνάρτηση `area()` θα κληθεί;
  - Από ποια κλάση;

Ένα νέο πρόβλημα που προκύπτει με αυτή την προσέγγιση

- Ας υποθέσουμε επιπλέον ότι πρέπει να υλοποιηθεί ένα νέο είδος σχήματος
  - Η κλάση `Triangle` ως παράγωγη της κλάσης `Shape`
- Η συνάρτηση `empty()` κληρονομείται από την `Shape`
  - Θα λειτουργήσει για τρίγωνα;
    - Όχι, η `area()` είναι διαφορετική για κάθε σχήμα
    - Η `Shape::area()` δεν λειτουργεί για τρίγωνα
  - Πράγματι, όταν υλοποιήθηκε η κλάση `Shape`, η κλάση `Triangle` δεν υπήρχε καν

Οι εικονικές συναρτήσεις είναι η λύση στο πρόβλημα

- Οι εικονικές συναρτήσεις ενημερώνουν τον μεταγλωττιστή:
  - “ότι δεν είναι γνωστό πώς υλοποιείται η συνάρτηση”
  - “και ότι η απόφαση θα πρέπει να παρθεί όταν κληθεί η συνάρτηση στο πρόγραμμα”
  - “τότε, θα χρησιμοποιηθεί η υλοποίηση από το κατάλληλο αντικείμενο στιγμιότυπο”
- Η διαδικασία αυτή ονομάζεται καθυστερημένη ή δυναμική σύνδεση (late or dynamic binding)
  - Παρατηρήστε τη λέξη κλειδί `virtual` στη δήλωση της συνάρτησης μέλους `area()`

```
class Shape {
private:
    double center_x, center_y;
public:
    virtual double area() const;
};
double Shape::area() const {
    exit(1);
}
```

```
class Rectangle : public Shape {
private:
    double width, height;
public:
    double area() const;
};
double Rectangle::area() const {
    return width * height;
}
```

Hands-on ενότητα 1

Υλοποίηση της κλάσης Shape και των παράγωγων της Rectangle και Circle

Καθυστερημένη σύνδεση

- Η συνάρτηση μέλος `area()` της `Shape` ως εικονική υλοποιεί την καθυστερημένη σύνδεση
- Οι παράγωγες κλάσεις `Rectangle` και `Circle` ορίζουν νέες εκδόσεις της `area()`
- Οι υπόλοιπες συναρτήσεις, οι οποίες κληρονομούνται στις παράγωγες κλάσεις, θα χρησιμοποιήσουν την έκδοση της `area()` του αντικειμένου που κάνει την κλήση
  - Παράδειγμα: Η συνάρτηση μέλος `empty()`.

```
class Shape {
private:
    double center_x, center_y;
public:
    virtual double area() const;
    bool empty() const();
};
bool Shape::empty() const {
    return (area()==0);
}
```

```
class Rectangle : public Shape {
private:
    double width, height;
public:
    double area() const;
};
double Rectangle::area() const {
    return width * height;
}
```

Καθυστερημένη σύνδεση

- Στην παράγωγη κλάση `Rectangle` η συνάρτηση μέλος `area()` υλοποιείται διαφορετικά από ότι στην κλάση `Shape`
- Η λέξη-κλειδί `virtual` δεν εμφανίζεται ούτε στη δήλωση ούτε στην υλοποίηση της
  - η συνάρτηση μέλος `area()` είναι “αυτομάτως” `virtual` στην παράγωγη κλάση
- Γενικά, στην παράγωγη κλάση δεν απαιτείται η λέξη-κλειδί `virtual`
  - συνίσταται όμως η περίληψη της δήλωσης `virtual` για αναγνωσιμότητα

```
class Shape {
private:
    double center_x, center_y;
public:
    virtual double area() const;
    bool empty() const();
};
bool Shape::empty() const {
    return (area()==0);
}
```

```
class Rectangle : public Shape {
private:
    double width, height;
public:
    double area() const;
};
double Rectangle::area() const {
    return width * height;
}
```

Πως λειτουργούν οι εικονικές συναρτήσεις;

- Κατά την ανάπτυξη προγραμμάτων C++:
  - υποθέτουμε ότι λειτουργούν “ως δια μαγείας”

- Αλλά η εξήγηση σχετίζεται με την καθυστερημένη σύνδεση.
  - Οι εικονικές συναρτήσεις υλοποιούν την καθυστερημένη σύνδεση (late binding)
  - Ο μεταγλωττιστής “περιμένει” μέχρι να χρησιμοποιηθεί η συνάρτηση στο πρόγραμμα
  - Η απόφαση ποια υλοποίηση θα χρησιμοποιηθεί λαμβάνεται βάση του αντικειμένου που κάνει την κλήση

- Η καθυστερημένη σύνδεση αποτελεί σημαντική αρχή του αντικειμενοστρεφούς προγραμματισμού

Η λέξη κλειδί `override`

- Η υλοποίηση μιας εικονικής συνάρτησης αλλάζει σε μια παράγωγη κλάση
  - λέμε ότι η συνάρτηση υπερσκελίζεται (override)
  - υπό την έννοια ότι η υλοποίηση στην βασική κλάση παρακάμπτεται
- Η C++11 υποστηρίζει τη λέξη κλειδί `override`
  - για να καταστεί σαφές ότι μια συνάρτηση υπερσκελίζει την αντίστοιχη της γονικής κλάσης
  - συνίσταται η περίληψη της για λόγους αναγνωσιμότητας

```
class Shape {
private:
    double center_x, center_y;
public:
    virtual double area() const;
};
double Shape::area() const {
    exit(1);
}
```

```
class Rectangle : public Shape {
private:
    double width, height;
public:
    double area() const override;
};
double Rectangle::area() const {
    return width * height;
}
```

Η λέξη κλειδί `final`

- Η C++11 υποστηρίζει τη λέξη κλειδί `final`
  - για την αποτροπή υπερσκελισμού μιας συνάρτησης
  - π.χ. εάν μια συνάρτηση έχει υπερσκελιστεί αλλά δεν θέλουμε να υπερσκελιστεί ξανά
- Μια κλάση μπορεί επίσης να δηλωθεί `final`
  - για την αποτροπή δημιουργίας υποκλάσεων της
  - αποτρέπεται έτσι και ο υπερσκελισμός των εικονικών συναρτήσεων της

```
class Triangle final : public Shape {
public:
    ...
    double area() const override final;
    ...
private:
    double height, base;
};
```

```
class RightTriangle : public Triangle { //error
public:
    ...
    double area() const override; //error
    ...
private:
    double angle_1, angle_2;
};
```

Οι εικονικές συναρτήσεις θα πρέπει να χρησιμοποιούνται με φειδώ

- Οι εικονικές συναρτήσεις σαφώς πλεονεκτούν όπως έχουμε δει
- Ένα σημαντικό μειονέκτημα τους ωστόσο είναι η επιβάρυνση (overhead) που προκαλούν στο πρόγραμμα
  - η καθυστερημένη σύνδεση εκτελείται on the fly
  - επομένως, τα προγράμματα γίνονται πιο αργά
- Καλή πρακτική: Εάν μια εικονική συνάρτηση δεν χρειάζεται πραγματικά, δεν θα πρέπει να υλοποιηθεί ως εικονική

Αμιγώς εικονικές συναρτήσεις και αφηρημένες κλάσεις

- Σε κάποιες βασικές κλάσεις, ίσως "να μην έχει νόημα" ή ακόμη και "να μην είναι εφικτό" να οριστούν κάποιες ή ακόμη και όλες οι συναρτήσεις μέλη
  - ο σκοπός μιας τέτοιας κλάσης είναι απλά για την παραγωγή άλλων κλάσεων με βάση αυτή
  - ουσιαστικά υποχρεώνει τις παράγωγες κλάσεις να ορίσουν τις "δικές τους εκδόσεις"
- Παράδειγμα: Στην κλάση `Shape`, η συνάρτηση μέλος `area()` δεν μπορεί να υλοποιηθεί
  - οπότε θα πρέπει να οριστεί ως αμιγώς εικονική: `virtual void area() = 0;`
  - όλα τα σχήματα είναι αντικείμενα των παράγωγων κλάσεων (ορθογώνια, κύκλοι κ.ο.κ.)

```
class Shape {
public:
    ...
    virtual void area() const = 0;
    ...
private:
    double center_x, center_y;
};
```

```
/**
 * Cannot create objects of an abstract class
 */
int main() {
    Shape s;  //This yields an error since
    s.area(); //this doesn't have a meaning
    return 0;
}
```

Αμιγώς εικονικές συναρτήσεις και αφηρημένες κλάσεις

- Μια κλάση η οποία περιέχει μια ή περισσότερες αμιγώς εικονικές συναρτήσεις ονομάζεται αφηρημένη (abstract class)
  - μια αφηρημένη κλάση μπορεί να χρησιμοποιηθεί μόνο ως βασική
  - δεν μπορούν να δημιουργηθούν αντικείμενα με βάση μια αφηρημένη κλάση
    - δεδομένου ότι δεν έχει πλήρεις “ορισμούς” για όλες τις συναρτήσεις μέλη της
- Παρατήρηση: Αν σε μια παράγωγη κλάση δεν μπορούν να υλοποιηθούν όλες οι αμιγώς εικονικές συναρτήσεις που κληρονομεί, τότε είναι επίσης αφηρημένη

```
class Shape {
public:
    ...
    virtual void area() const = 0;
    ...
private:
    double center_x, center_y;
};
```

```
/**
 * Cannot create objects of an abstract class
 */
int main() {
    Shape s;  //This yields an error since
    s.area(); //this doesn't have a meaning
    return 0;
}
```

Συμβατότητα τύπων

- Ας υποθέσουμε ότι η κλάση `Derived` είναι παράγωγη της κλάσης `Base`. Τότε:
  - Αντικείμενα της κλάσης `Derived` μπορούν να ανατεθούν σε αντικείμενα της κλάσης `Base`
  - Αλλά όχι αντιστρόφως
- Στο προηγούμενο παράδειγμα, θεωρείστε τις δηλώσεις:
  - `Shape vshape;`
  - `Rectangle vrectangle;`
- Η παρακάτω ανάθεση (σε γονικό τύπο, όχι αντιστρόφως) είναι επιτρεπτή:
  - `vshape = vrectangle;`
  - Αντιστρόφως: Ένα σχήμα “δεν είναι” (απαραίτητα) ορθογώνιο

Το πρόβλημα της κατάτμησης (slicing problem)

- Ωστόσο, το αντικείμενο `vshape` “χάνει” τα πεδία `width` και `height`

- Όντως, ο παρακάτω κώδικας παράγει μήνυμα σφάλματος:
  - `cout << vshape.getWidth();`

- Το πρόβλημα αυτό λέγεται πρόβλημα κατάτμησης
  - ένα αντικείμενο τύπου `Shape` δεν έχει όλες τις ιδιότητες ενός αντικειμένου τύπου `Rectangle`
  - όταν ένα αντικείμενο τύπου `Rectangle` εκχωρείται σε ένα τύπου `Shape`, θα πρέπει να αντιμετωπίζεται ως τύπου `Shape`

Το πρόβλημα της κατάτμησης (slicing problem)

- Στη C++, το πρόβλημα της κατάτμησης είναι ενοχλητικό
- Το πρόβλημα παραμένει ακόμη και αν χρησιμοποιηθούν δείκτες (αντί μεταβλητών)
- Στο παρακάτω παράδειγμα, δεν είναι δυνατή η πρόσβαση στο πεδίο `width` του αντικειμένου στο οποίο δείχνει ο δείκτης `pshape`

```
int main() {
  Shape *pshape;
  Rectangle *prectangle;

  prectangle = new Rectangle(1,1,1,1);

  pshape = prectangle;
  cout << pshape->getWidth(); //ILLEGAL!

  return 0;
}
```

Η διόρθωση στο πρόβλημα της κατάτμησης

- Μπορεί να επιτευχθεί:
  - χρησιμοποιώντας δείκτες σε δυναμικές μεταβλητές και
  - καλώντας εικονικές συναρτήσεις
- Στο παράδειγμα: Η κλήση της εικονικής συνάρτησης `print()` από ένα δείκτη τύπου `pshape`
  - ενεργοποιεί την συνάρτηση μέλος `print` της κλάσης `Rectangle`
  - επειδή η συνάρτηση αυτή είναι εικονική
- Η C++ "αναμένει" και "ελέγχει" σε τι είδους αντικείμενο δείχνει ο δείκτης `pshape` πριν κάνει τη σύνδεση με μια συγκεκριμένη συνάρτηση

```
class Shape {
private:
    double center_x, center_y;
public:
    virtual void print() const;
};
```

```
class Rectangle : public Shape {
private:
    double width, height;
public:
    virtual void print() const override;
};
```

Εικονικοί καταστροφείς

- Οι καταστροφείς είναι απαραίτητοι για την αποδέσμευση δυναμικά-δεσμευμένης μνήμης

- Θεωρήστε τον παρακάτω κώδικα:
  - `Base *pBase = new Derived;`
  - `...`
  - `delete pBase;`

- Η κλήση εδώ γίνεται στον καταστροφέα της βασικής κλάσης
  - παρόλο που ο δείχτης δείχνει σε αντικείμενο της παράγωγης κλάσης

- Το πρόβλημα αυτό διορθώνεται ορίζοντας τον καταστροφέα εικονικό
  - Καλή πολιτική: Όλοι οι καταστροφείς να δηλώνονται εικονικοί

Μετατροπή τύπου - Casting

- Θεωρήστε τον παρακάτω κώδικα:
  - `Shape vshape;`
  - `Rectangle vrectangle;`
  - `...`
  - `vrectangle = static_cast<Rectangle>(vshape); //ILLEGAL!`
- Δεν επιτρέπεται η μετατροπή ενός αντικειμένου τύπου `Shape` σε ένα τύπου `Rectangle`.
- Ωστόσο, το αντίστροφο (από τύπο-απογόνου σε τύπο-προγόνου) είναι αποδεκτό:
  - `vshape = vrectangle; // Legal!`
  - `vshape = static_cast<Shape>(vrectangle); //Also legal!`

Σύνοψη

- Η καθυστερημένη σύνδεση αναβάλλει την απόφαση για το ποια συνάρτηση-μέλος θα κληθεί μέχρι την εκτέλεση του προγράμματος
- Οι αμιγώς εικονικές συναρτήσεις δεν έχουν υλοποίηση
  - οι κλάσεις με τουλάχιστον μια τέτοια συνάρτηση είναι
  - δεν μπορούν να δημιουργηθούν αντικείμενα από αφηρημένες κλάσεις
  - χρησιμοποιούνται αυστηρώς ως βάση για την υλοποίηση άλλων κλάσεων
- Αντικείμενα μιας παράγωγης κλάσης μπορούν να ανατεθούν σε αντικείμενα βασικής κλάσης
  - τα μέλη της παράγωγης κλάσης χάνονται
  - το πρόβλημα της κατάτμησης (slicing problem)
- Οι καταστροφείς καλό θα είναι να δηλώνονται εικονικοί
  - προγραμματιστική πρακτική, η οποία εξασφαλίζει τη σωστή αποδέσμευση της μνήμης