Προγραμματισμός Υπολογιστών C++

Κληρονομικότητα

Κληρονομικότητα

- Η κληρονομικότητα (*inheritance*) αποτελεί ένα από τα κυριότερα χαρακτηριστικά της C++ και γενικότερα του αντικειμενοστρεφούς προγραμματισμού
- Η κληρονομικότητα επιτρέπει την δημιουργία μίας νέας κλάσης από μία υπάρχουσα. Η νέα κλάση ονομάζεται **παράγωγη** (*derived*), ενώ η υπάρχουσα ονομάζεται **βασική** (*base*)
- Η παράγωγη κλάση **κληρονομεί** τα χαρακτηριστικά της βασικής κλάσης
- Ο προγραμματιστής μπορεί να προσθέσει νέα χαρακτηριστικά στην παράγωγη κλάση, να χρησιμοποιήσει όπως είναι τα χαρακτηριστικά που κληρονομήθηκαν ή να τους αλλάξει την συμπεριφορά και να τα προσαρμόσει στις ανάγκες της παράγωγης κλάσης
- Η γενική μορφή δήλωσης μίας παράγωγης κλάσης είναι:

```
class όνομα_παράγωγης : τύπος_πρόσβασης όνομα_βασικής
```

- Ένα αντικείμενο μίας παράγωγης κλάσης **περιέχει** ένα αντικείμενο της βασικής κλάσης
- Η παράγωγη κλάση μπορεί να χρησιμοποιηθεί σαν βασική για τη δημιουργία μίας νέας παράγωγης κλάσης και αυτή το ίδιο με τη σειρά της, και έτσι να δημιουργηθεί μία ιεραρχική αλυσίδα κλάσεων. Για παράδειγμα, μία κλάση `C` μπορεί να παράγεται από την κλάση `B`, η οποία με τη σειρά της να παράγεται από την `A`, κ.ο.κ.

Κληρονομικότητα

- Ο `τύπος_πρόσβασης` καθορίζει την πρόσβαση στα μέλη της βασικής κλάσης
- Αν λείπει, ο εξ'ορισμού τύπος πρόσβασης είναι ο `private`
- Όταν ο τύπος πρόσβασης είναι `public`, οι συναρτήσεις και τα αντικείμενα της παράγωγης κλάσης, καθώς και οι φιλικές συναρτήσεις της παράγωγης έχουν πρόσβαση στα δημόσια μέλη της βασικής, αλλά όχι στα ιδιωτικά
- Ειδικότερα, η παράγωγη κλάση μπορεί να προσπελάσει τα ιδιωτικά μέλη της βασικής μόνο μέσω των δημόσιων και των προστατευμένων συναρτήσεων της βασικής κλάσης
- Επίσης, οι συναρτήσεις της παράγωγης κλάσης, καθώς και οι φιλικές συναρτήσεις της μπορούν να προσπελάσουν απευθείας τα προστατευμένα μέλη της βασικής, ενώ τα αντικείμενά της όχι
- Η βασική κλάση περιέχει τα γενικά χαρακτηριστικά μίας οντότητας (π.χ. `Product`). Η παράγωγη κλάση αποτελεί συνήθως μία εξειδίκευση της βασικής (π.χ. `Book`), η οποία κληρονομεί τα μέλη της βασικής και προσθέτει τα δικά της χαρακτηριστικά. Έτσι, ένα αντικείμενο μίας παράγωγης κλάσης, εκτός από τα δικά του μέλη, περιέχει ένα υπο-αντικείμενο της βασικής κλάσης. Για παράδειγμα, δείτε το παρακάτω πρόγραμμα:

Παράδειγμα

```
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::string;

class Product
{
public:
      string code;
      float prc;

      Product() {cout << "Base Constructor\n";}
      ~Product() {cout << "Base Destructor\n";}
void show() const {cout << "\nC:" << code << '\n';}
};

class Book : public Product
{
public:
      string title;
      string auth;

      Book() {cout << "Derived Constructor\n";}
      ~Book() {cout << "Derived Destructor\n";}
      void display() const {cout << "T:" << title <<
      " A:" << auth << " P:" << prc << '\n';} /* Η
      συνάρτηση έχει πρόσβαση στα μέλη της βασικής
      κλάσης. */
};
```

```
int main()
{
            Book b;

            cout << "Product Details: ";
            cin >> b.code >> b.prc; /*
Προσπέλαση των μελών της βασικής κλάσης. */
            cout << "Book Details: ";
            cin >> b.title >> b.auth;

            b.show(); /* Κλήση συνάρτησης της
βασικής κλάσης. */
            b.display();
            return 0;
}
```

Παράδειγμα

- Το κύριο πλεονέκτημα της κληρονομικότητας είναι ότι μας επιτρέπει να χρησιμοποιήσουμε υπάρχοντα κώδικα, χωρίς να χρειάζεται να τον ξαναγράψουμε
- Αυτό που έχουμε να κάνουμε είναι να γράψουμε τον κώδικα που υλοποιεί την παράγωγη κλάση
- Στο παράδειγμά μας, η `Book` κληρονόμησε όλα τα μέλη και τις συναρτήσεις της `Product`. Δεν χρειάζεται να τα προσθέσουμε πάλι σε αυτήν. Δηλαδή, ένα `Book` αντικείμενο κληρονομεί τα μέλη `code` και `prc` και τη συνάρτηση `show()`. Επίσης, η `Book` περιέχει τα δικά της μέλη και συναρτήσεις
- Όταν δημιουργείται ένα αντικείμενο της παράγωγης κλάσης, **πρώτα** καλείται ο κατασκευαστής της παράγωγης κλάσης. Αυτός με τη σειρά του **καλεί** κατασκευαστή της βασικής κλάσης για να **κατασκευάσει** το **αντικείμενο** της **βασικής** κλάσης που **περιέχεται** στο **παράγωγο** αντικείμενο

Παράδειγμα

- Δηλαδή, όπως είπαμε, το παράγωγο αντικείμενο περιέχει ένα υπο-αντικείμενο της βασικής κλάσης
- Αυτή η σχέση αναφέρεται και ως «**είναι-ένα**» (is-a), με την έννοια ότι κάθε αντικείμενο μίας παράγωγης κλάσης **είναι** επίσης ένα αντικείμενο της βασικής
- Για παράδειγμα, ένα `Book` αντικείμενο είναι επίσης ένα `Product` αντικείμενο
- Ειδικότερα, **πρώτα** πρέπει να δημιουργηθεί το υπο-αντικείμενο της βασικής κλάσης και **μετά** το τμήμα του παράγωγου αντικειμένου
- Άρα, όταν δημιουργείται το `b` ο κατασκευαστής της `Book` **πρώτα** καλεί τον κατασκευαστή της `Product` για να δημιουργήσει το `Product` υπο-αντικείμενο και **μετά** συνεχίζει με την κατασκευή του `b`, δηλαδή, εκτελείται το σώμα του `Book` κατασκευαστή
- Επομένως, το πρόγραμμα αρχικά εμφανίζει:

```
Base Constructor
Derived Constructor
```

- Γενικά, σε μία ιεραρχία κλάσεων (π.χ. `A`, `B`, `C`, `D`), όταν δημιουργείται ένα αντικείμενο της παράγωγης κλάσης (π.χ. `D`), πρώτα καλείται ο κατασκευαστής της υψηλότερης βασικής κλάσης (π.χ. `A`), μετά ο κατασκευαστής της κλάσης που παράγεται από αυτήν (π.χ. `B`), μετά της επόμενης (π.χ. `C`), κ.ο.κ.. Για παράδειγμα, πρώτα δημιουργούνται τα `A`, `B`, `C` υπο-αντικείμενα που περιέχονται στο `D` αντικείμενο και μετά ο `D` κατασκευαστής ολοκληρώνει την κατασκευή του `D` αντικειμένου

Παράδειγμα

- Επειδή η Book παράγεται με δημόσια πρόσβαση, το b μπορεί να προσπελάσει τα δημόσια μέλη της Product
- Το πρόγραμμα διαβάζει την πληροφορία που εισάγει ο χρήστης, την αποθηκεύει στα πεδία της Product και της Book και καλεί συναρτήσεις και από τις δύο κλάσεις για να την εμφανίσει
- Όταν καταστρέφεται το παράγωγο αντικείμενο η αποδόμηση γίνεται σε **αντίστροφη** σειρά από τη σειρά κατασκευής, δηλαδή, το πρόγραμμα **πρώτα** καλεί τον αποδομητή της παράγωγης κλάσης και **μετά** της βασικής
- Επομένως, το πρόγραμμα εμφανίζει:

```
Derived Destructor
Base Destructor
```

Παράδειγμα

- Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream>

class A
{
private:
        int a;
        void f();
protected:
        int p;
        void g();
};

class B : public A
{
public:
        int c;
        void h();
};
void A::f()
{
        std::cout << c << '\n';
}
```

```
void A::g()
{
        std::cout << a << ' ' << p << '\n';
}
void B::h()
{
        std::cout << p << ' ' << c << '\n';
}
int main()
{
        B b;

        std::cin >> b.a >> b.p;
        b.g();
        b.h();
        return 0;
}
```

Παράδειγμα

- Υπάρχει πρόβλημα που καμία κλάση δεν έχει κατασκευαστή; Όχι βέβαια, θα κληθούν οι εξ'ορισμού κατασκευαστές. Να θυμάστε, **πρώτα** εκτελείται ο εξ'ορισμού κατασκευαστής της βασικής κλάσης και **μετά** της παράγωγης
- Συνεχίζουμε, αφού τα μέλη `a` και `f()` είναι ιδιωτικά, τα αντικείμενα αλλά και οι συναρτήσεις της κλάσης `B` **δεν επιτρέπεται** να έχουν πρόσβαση σε αυτά
- Όσον αφορά τα **προστατευμένα** μέλη, θυμηθείτε ότι για την **ίδια** την κλάση συμπεριφέρονται όπως τα ιδιωτικά. Για **παράγωγη** κλάση, οι συναρτήσεις της **επιτρέπεται** να έχουν πρόσβαση στα προστατευμένα μέλη της βασικής, ενώ τα αντικείμενά της **όχι**. Επομένως, οι ακόλουθες εντολές δεν είναι αποδεκτές:

```
cin >> b.a >> b.p; /* Δεν επιτρέπεται η πρόσβαση στα πεδία a και p. */
b.g();
```

- Η κλάση `A` δεν περιέχει μέλος με όνομα `c`. Το πεδίο `c` έχει δηλωθεί στην παράγωγη κλάση `B`, στην οποία η κλάση `A` δεν έχει πρόσβαση. Άρα, η ακόλουθη εντολή δεν είναι αποδεκτή:

```
cout << c << '\n'; // Στην f().
```

Και αυτό είναι λογικό. Η κληρονομικότητα δεν λειτουργεί ανάποδα, δηλαδή, μία βασική κλάση και τα αντικείμενά της δεν γνωρίζουν τίποτα για κλάσεις που παράγονται από αυτήν και επομένως δεν έχουν πρόσβαση στα μέλη τους

Is-a και Has-a Σχέσεις

- Όπως είπαμε, η κληρονομικότητα κανονικά αναπαριστά στον σχεδιασμό του προγράμματος την σχέση «είναι ένα» (is-a), δηλαδή, ένα αντικείμενο παράγωγης κλάσης θα πρέπει να είναι επίσης ένα αντικείμενο της βασικής κλάσης. Για παράδειγμα, ένα **Book** αντικείμενο είναι επίσης ένα **Product** αντικείμενο
- Θεωρήστε ότι έχουμε ορίσει την κλάση **Engine**, την κλάση **Tires**, και θέλουμε να ορίσουμε την **Car** κλάση. Αφού ένα αυτοκίνητο **έχει** μηχανή και λάστιχα, η **ταιριαστή** επιλογή είναι να χρησιμοποιήσουμε την σχέση «έχει» (has-a) και να γράψουμε:

```
class Car
{
        ...
        Engine eng;
        Tires tir;
};
```

- Τώρα, υποθέστε ότι έχουμε ορίσει την **Shape** κλάση και θέλουμε να ορίσουμε την **Circle** κλάση. Αφού ένας κύκλος **είναι** σχήμα, η ταιριαστή επιλογή είναι να χρησιμοποιήσουμε την is-a σχέση και να γράψουμε:

```
class Circle : public Shape
{
        ...
};
```

Is-a και Has-a Σχέσεις

- Παρόμοια, υποθέστε ότι έχουμε ορίσει την `BankAccount` κλάση και θέλουμε να ορίσουμε την `CheckingAccount` κλάση. Αφού ένας λογαριασμός όψεως είναι ένας ειδικός τύπος τραπεζικού λογαριασμού, ταιριάζει να χρησιμοποιήσουμε κληρονομικότητα και να παράξουμε την `CheckingAccount` κλάση από την `BankAccount`

- Ποια σχέση θα επιλέγαμε αν είχαμε τις `Animal` και `Cat` κλάσεις; Αφού μία γάτα είναι είδος ζώου, ταιριάζει να χρησιμοποιήσουμε κληρονομικότητα και να παράξουμε την `Cat` κλάση από την `Animal`

- Γενικά, όταν πρέπει να αποφασίσετε αν θα προσθέσετε μια κλάση ως μέλος ή θα χρησιμοποιήσετε κληρονομικότητα, ο γενικός κανόνας είναι να δείτε ποια από τις has-a και is-a σχέσεις ταιριάζει καλύτερα και να ενεργήσετε ανάλογα

Εμβέλεια και Κληρονομικότητα

- Όταν ένα όνομα χρησιμοποιείται σε μία κλάση, ο μεταγλωττιστής ψάχνει για τη δήλωσή του **πρώτα** στην εμβέλεια της κλάσης
- Αν δεν το βρει, **συνεχίζει** την αναζήτηση στις εμβέλειες των βασικών της κλάσεων, σε **όλη** την αλυσίδα από την **απευθείας** βασική της κλάση μέχρι την **κορυφή**
- Για παράδειγμα, έστω μία κλάση `C` παράγεται από την κλάση `B`, η οποία παράγεται από την κλάση `A` και μόνο αυτή περιέχει ένα δημόσιο μέλος με το όνομα `p`. Αν γράψουμε:

```
C c;
c.p = 10;
```

αφού το `c` είναι αντικείμενο της κλάσης `C` ο μεταγλωττιστής ψάχνει για την δήλωση του `p` σε αυτήν. Επειδή το `p` δεν δηλώνεται στη `C`, ο μεταγλωττιστής συνεχίζει την αναζήτηση στη βασική της κλάση, στην `B`. Παρόμοια, επειδή το `p` δεν δηλώνεται στην `B` και επειδή η `B` παράγεται από την `A`, η αναζήτηση συνεχίζεται σε αυτήν, όπου και τελικά βρίσκεται. Αν δεν υπήρχε η δήλωση, ο μεταγλωττιστής θα εμφάνιζε σχετικό μήνυμα λάθους

Εμβέλεια και Κληρονομικότητα

- Μία λογική ερώτηση είναι, και τι θα γίνει αν το όνομα p εμφανίζεται και σε άλλη κλάση; Για παράδειγμα:

```
void g() // Καθολική g().
{
        ...
}

class A
{
public:
        int p;
        ...
};

class B : public A
{
public:
        int p;
        ...
};
```

```
class C : public B
{
public:
    int p;
    void f();
    ...
};
void C::f()
{
    g();
}
```

Εμβέλεια και Κληρονομικότητα

- Όπως ξέρουμε από το Κ.11, τα ονόματα που δηλώνονται σε μία ένθετη εμβέλεια **κρύβουν** τα ονόματα έξω από αυτήν
- Επομένως, με την εντολή `c.p` αναφερόμαστε στο μέλος `p` της κλάσης `C`. Επειδή ο μεταγλωττιστής βρίσκει το `p` στην `C` **σταματάει** την αναζήτηση. Αν όμως το `p` δεν είχε δηλωθεί στην `C`, θα **συνέχιζε** την αναζήτηση στη βασική της κλάση και, άρα, θα αναφερόμασταν στο μέλος `p` της κλάσης `B`
- Και τι κάνουμε αν θέλουμε να αναφερθούμε στο μέλος `p` κάποιας άλλης κλάσης; Χρησιμοποιούμε τον τελεστή επίλυσης εμβέλειας `::`. Για παράδειγμα:

```
C c;
c.A::p = 10;
c.B::p = 20;
c.p = 30;
```

- Σημειώστε ότι ο μεταγλωττιστής εφαρμόζει την **ίδια** διαδικασία αναζήτησης για οποιοδήποτε όνομα, δεν έχει σημασία αν είναι μεταβλητή, συνάρτηση ή κάτι άλλο
- Για παράδειγμα, έστω ότι είναι όνομα συνάρτησης. Αν κληθεί η `f()`, ο μεταγλωττιστής ψάχνει για την δήλωση της `g()` στην εμβέλεια της `C`. Αν δεν την βρει συνεχίζει στην εμβέλεια της `B`, και μετά στην εμβέλεια της `A`. Αν δεν την βρει, συνεχίζει την αναζήτηση για κάποια καθολική `g()`, και αν πάλι αποτύχει εμφανίζει μήνυμα λάθους

Παράγωγες Κλάσεις και Κατασκευαστές

- Ας αλλάξουμε το πρώτο πρόγραμμα, να προσθέσουμε παραμέτρους στους δύο κατασκευαστές και να δούμε με ποιο τρόπο ο κατασκευαστής της παράγωγης κλάσης μπορεί να μεταβιβάσει τιμές στον κατασκευαστή της βασικής

```
#include <iostream>
#include <string>
using std::cout;
using std::string;

class Product
{
private:
          string code;
          float prc;
public:
          Product(const string& c, float p);
          void show() const;
};

class Book : public Product
{
private:
          string title;
          string auth;
public:
          Book(const string& c, float p, const string& t, const string& a);
          void display() const;
};
```

Παράγωγες Κλάσεις και Κατασκευαστές

```
Product::Product(const string& c, float p)
{
     code = c;
     prc = p;
}

void Product::show() const
{
     cout << "C:" << code << " P:" << prc << '\n';
}

Book::Book(const string& c, float p, const string& t, const string& a) : Product(c, p) /* Πρώτα
     θα κληθεί ο κατασκευαστής της βασικής κλάσης και μετά θα εκτελεστούν οι παρακάτω εντολές.
     */
{
     title = t;
     auth = a;
}

void Book::display() const
{
     show();
     cout << "T:" << title << " A:" << auth << '\n';
}

int main()
{
     Book b("AB25", 8.5, "Nice", "Many");

     b.display();
     return 0;
}
```

Παράγωγες Κλάσεις και Κατασκευαστές

- Όπως είπαμε, όταν δημιουργείται ένα αντικείμενο μίας παράγωγης κλάσης, πρέπει πρώτα να δημιουργηθεί το υπο-αντικείμενο της βασικής
- Και πώς θα γίνει αυτό; Απλά, η παράγωγη κλάση πρέπει να καλέσει έναν κατασκευαστή της βασικής κλάσης για να αρχικοποιήσει το υπο-αντικείμενο της βασικής κλάσης
- Όταν λοιπόν δημιουργείται το αντικείμενο `b` καλείται ο κατασκευαστής της κλάσης `Book` και μεταβιβάζονται οι τιμές των ορισμάτων στις αντίστοιχες παραμέτρους. Για παράδειγμα, το `c` θα γίνει ίσο με `AB25` και το `p` ίσο με `8.5`
- Μετά την μεταβίβαση των τιμών, εκτελείται το τμήμα του κώδικα:

```
: Product(c, p)
```

το οποίο καλεί τον κατασκευαστή της βασικής κλάσης χρησιμοποιώντας τη σύνταξη αρχικοποίησης μελών που είδαμε στο Κ.19 και του μεταβιβάζει τις τιμές των `c` και `p`

Παρατηρήσεις

- Θυμόμαστε, όταν το πρόγραμμα δημιουργεί ένα αντικείμενο μίας παράγωγης κλάσης πρώτα δημιουργείται το υπο-αντικείμενο της βασικής
- Ο κατασκευαστής της παράγωγης κλάσης καλεί κατασκευαστή της βασικής
- Ο προγραμματιστής μπορεί να επιλέξει τον κατασκευαστή της βασικής κλάσης που θα κληθεί, αλλιώς, αν δεν επιλέξει κατασκευαστή, το πρόγραμμα θα καλέσει τον εξ'ορισμού κατασκευαστή, εφόσον αυτός υπάρχει
- Αν δεν υπάρχει, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους
- Όταν ένα αντικείμενο της παράγωγης κλάσης καταστρέφεται, πρώτα καλείται ο αποδομητής της παράγωγης κλάσης και μετά της βασικής

Ιεραρχία Κλάσεων

- Σε μία αλυσίδα παραγώγων κλάσεων, κάθε κλάση μπορεί να χρησιμοποιήσει τη σύνταξη αρχικοποίησης για να μεταβιβάσει τιμές στον κατασκευαστή της αμέσως προηγούμενης βασικής κλάσης
- Προσέξτε όμως, μόνο της προηγούμενης βασικής κλάσης και όχι κάποιας άλλης σε υψηλότερο επίπεδο
- Εξαίρεση σε αυτόν τον κανόνα αποτελούν οι εικονικές βασικές κλάσεις

Παράδειγμα

```
#include <iostream>
using std::cout;

class A
{
public:
        A(int k) {cout << "A:" << k;}
        ~A() {cout << " ~A\n";}
};

class B : public A
{
public:
        B(int m) : A(m+1) {cout << " B:" << m;}
        ~B() {cout << " ~B";}
};

class C : public B
{
public:
        C(int p) : B(2*p) {cout << " C:" << p << '\n';}
        ~C() {cout << "~C";}
};

int main()
{
        C c(10);
        return 0;
}
```

Ιεραρχία Κλάσεων

- Όταν δημιουργείται το αντικείμενο c καλείται ο κατασκευαστής της C κλάσης, ο οποίος καλεί τον κατασκευαστή της B κλάσης και του μεταβιβάζει την τιμή 20
- Αυτός με τη σειρά του καλεί τον κατασκευαστή της A κλάσης, του μεταβιβάζει την τιμή 21 και εκτελείται το σώμα του. Το πρόγραμμα εμφανίζει A:21
- Μετά, εκτελείται το σώμα του B κατασκευαστή και το πρόγραμμα εμφανίζει B:20
- Πρώτα λοιπόν δημιουργείται το υπο-αντικείμενο της κλάσης A και μετά το υπο-αντικείμενο της κλάσης B
- Δηλαδή, το c αντικείμενο περιέχει δύο υπο-αντικείμενα, ένα της κλάσης A και ένα της κλάσης B. Μετά την κατασκευή αυτών των δύο υπο-αντικειμένων εκτελείται το σώμα του C κατασκευαστή και το πρόγραμμα εμφανίζει C:10
- Γενικά, σε μία ιεραρχία κλάσεων, ένα παράγωγο αντικείμενο περιέχει ένα υπο-αντικείμενο της άμεσης βασικής του κλάσης και ένα υπο-αντικείμενο για κάθε μία από τις έμμεσες βάσεις του
- Οι αποδομητές καλούνται με την αντίστροφη σειρά από αυτήν με την οποία κλήθηκαν οι κατασκευαστές, σε όλη την ιεραρχία των κλάσεων, από την τελευταία παράγωγη κλάση μέχρι τη ρίζα. Τελικά, το πρόγραμμα εμφανίζει:

```
A:21        B:20        C:10
~C            ~B        ~A
```

Ιεραρχία Κλάσεων

- Όπως είπαμε, ο κατασκευαστής της C κλάσης δεν επιτρέπεται να χρησιμοποιήσει την παραπάνω σύνταξη για να καλέσει απευθείας κατασκευαστή της A κλάσης
- Αυτό που επιτρέπεται είναι να καλέσει κατασκευαστή μόνο από την απευθείας βασική του κλάση, δηλαδή, από την B, και ένας B κατασκευαστής να καλέσει κατασκευαστή μόνο από την A κλάση.
- Για παράδειγμα, δεν επιτρέπεται να γράψουμε:

```
C(int p) : A(p) {} // Λάθος μεταγλώττισης.
```

- Αν δεν θέλουμε μία κλάση να κληρονομηθεί χρησιμοποιούμε τη λέξη `final`, η οποία εισήχθηκε με την C++11. Για παράδειγμα:

```
class A final {...}; /* Η κλάση A δεν μπορεί να είναι βασική κλάση */
class B : public A {...}; // Λάθος.
```

Ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους παρόμοιο με «A is not a direct base of B».

Παράδειγμα

- Ποια είναι τα λάθη στο παρακάτω πρόγραμμα;

```
#include <iostream>

class A
{
private:
        int p;
public:
        A(int a) {p = a;}
};

class B : public A
{
public:
        B(int v) {std::cout << v << '\n';}
        void show(int v) const {std::cout << p+v << '\n';}
};

int main()
{
        B b(10);
        A& r = b;

        r.show(20);
        return 0;
}
```

Παράδειγμα

- Απάντηση: Το πρώτο λάθος είναι στον ορισμό της `show()`. Αφού το `p` είναι ιδιωτικό μέλος, η `show()` δεν έχει άμεση πρόσβαση σε αυτό. Το δεύτερο λάθος είναι στη δημιουργία του `b`. Αφού ο κατασκευαστής της `B` δεν καλεί κατασκευαστή της `A` δεν μπορεί να δημιουργηθεί το `A` αντικείμενο. Για παράδειγμα, αν γράφαμε `B(int v):A(v)` ή αν είχαμε ορίσει τον εξ'ορισμού κατασκευαστή στην `A` δεν θα υπήρχε πρόβλημα. Το `r` μπορεί να αναφερθεί σε `B` αντικείμενο, αλλά επειδή έχει δηλωθεί σαν αναφορά στην κλάση `A` και η `A` δεν περιέχει `show()` είναι λάθος να την καλέσουμε

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Μία ειδική σχέση μεταξύ **βασικής** και **παράγωγης** κλάσης είναι ότι ένας δείκτης ή αναφορά σε βασική κλάση **μπορεί** να δείξει ή να αναφερθεί σε αντικείμενο παράγωγης κλάσης
- Σημειώστε ότι **δεν χρειάζεται** προσαρμογή τύπου. Αυτή η μετατροπή συχνά ονομάζεται παράγωγη-σε-βασική (derived-to-base) μετατροπή. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product *p = &b; /* Η μετατροπή δείκτη από παράγωγη κλάση (π.χ. &b) σε δείκτη σε βασική (π.χ. p) επιτρέπεται. Ο p δείχνει στο Product υπο-αντικείμενο του b. */
Product &r = b; /* Το r αναφέρεται στο Product υπο-αντικείμενο του b. */
p->show();
r.show();
```

- Η μετατροπή που εφαρμόζει ο μεταγλωττιστής είναι **ασφαλής**, γιατί, όπως ξέρουμε, ένα `Book` αντικείμενο περιέχει ένα `Product` υπο-αντικείμενο ή, αλλιώς, ένα `Book` αντικείμενο είναι επίσης ένα `Product` αντικείμενο

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Γενικά, ένα αντικείμενο παράγωγης κλάσης ή μία αναφορά σε αυτό μπορεί να χρησιμοποιηθεί σε εκφράσεις όπου μπορεί να χρησιμοποιηθεί αντικείμενο της βασικής του κλάσης
- Παρόμοια, μπορούμε να χρησιμοποιήσουμε ένα δείκτη σε αντικείμενο της παράγωγης κλάσης σε εκφράσεις όπου μπορεί να χρησιμοποιηθεί δείκτης στη βασική κλάση
- Βέβαια, μία αναφορά ή ένας δείκτης σε βασική κλάση **δεν επιτρέπεται** να προσπελάσει μέλη της παράγωγης κλάσης. Για παράδειγμα, δεν επιτρέπεται να γράψουμε:

```
p->display(); // Λάθος.
r.display(); // Λάθος.
```

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Επειδή ένας δείκτης ή μία αναφορά σε βασική κλάση μπορεί να αναφερθεί σε παράγωγη κλάση (όπως είπαμε, αναφέρεται στο υπο-αντικείμενο της βασικής που υπάρχει στην παράγωγη), μία συνάρτηση που δέχεται σαν όρισμα δείκτη ή αναφορά σε αντικείμενο βασικής κλάσης μπορεί να δεχτεί δείκτη ή αναφορά και σε αντικείμενο παράγωγης κλάσης. Για παράδειγμα:

```
void f(Product& r)
{
        r.show();
}
```

- Αφού η παράμετρος `r` είναι αναφορά σε βασική κλάση, μπορεί να αναφερθεί σε αντικείμενο βασικής ή παράγωγης κλάσης χωρίς καμία ανησυχία για τη λειτουργία της συνάρτησης. Έτσι, μπορούμε να γράψουμε:

```
Product p("One", 1);
Book b("AB25", 8.5, "Nice", "Many");
f(p);
f(b); // Το Book αντικείμενο είναι και Product αντικείμενο.
```

- Σημειώστε ότι αυτή η ιδιότητα είναι μεταβατική. Δηλαδή, αν δημιουργήσουμε την κλάση `ProgrammingBook` που κληρονομεί την `Book`, τότε ένας δείκτης ή αναφορά στην κλάση `Product` μπορεί να αναφερθεί σε ένα `Product` ή `Book` ή `ProgrammingBook` αντικείμενο

Μετατροπές από Παραγωγή σε Βασική Κλάση

- Μπορούμε να εκχωρήσουμε ένα αντικείμενο μίας παράγωγης κλάσης σε ένα αντικείμενο βασικής κλάσης και **μόνο** τα μέλη της βασικής κλάσης θα αντιγραφούν. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product p;
p = b;
```

- Η εκχώρηση `p = b;` αντιγράφει τις τιμές του `Product` υπο-αντικειμένου που περιέχεται στο `b` στα πεδία του `p`. Όπως θα δούμε στη συνέχεια, το αντίστροφο, δηλαδή, το `b = p;` **δεν επιτρέπεται** αυτόματα, αφού ένα `Product` αντικείμενο δεν είναι και `Book` αντικείμενο
- Επίσης, σε μία συνάρτηση που δέχεται σαν παράμετρο αντικείμενο της βασικής κλάσης μπορούμε να μεταβιβάσουμε αντικείμενο παράγωγης κλάσης. Για παράδειγμα:

```
void f(Product p)
{
        p.show();
}
Book b("AB25", 8.5, "Nice", "Many");
f(b); // Το Book αντικείμενο είναι και Product αντικείμενο.
```

- Με την κλήση της `f()` ο κατασκευαστής αντιγράφου της `Product` αναλαμβάνει να αντιγράψει το `Product` τμήμα του `b` στο `p`. Παρόμοια με πριν, το αντίστροφο δεν επιτρέπεται, δηλαδή, **δεν επιτρέπεται** να μεταβιβάσουμε αντικείμενο βασικής κλάσης σε συνάρτηση που δέχεται αντικείμενο παράγωγης κλάσης
- Θυμόμαστε, όταν εκχωρούμε ή αρχικοποιούμε ένα αντικείμενο βασικής κλάσης με ένα αντικείμενο παράγωγης κλάσης **μόνο** τα μέλη του βασικού υπο-αντικειμένου του αντιγράφονται. Το υπόλοιπο τμήμα του παράγωγου αντικειμένου **αγνοείται**

Μετατροπές από Βασική σε Παραγωγή Κλάση

- Η ανάποδη μετατροπή (base-to-derived) **δεν επιτρέπεται** με αυτόματο τρόπο, δηλαδή, μία αναφορά ή ένας δείκτης σε παράγωγη κλάση **δεν επιτρέπεται** να δείξει σε αντικείμενο βασικής κλάσης χωρίς προσαρμογή τύπου
- Και αυτό είναι λογικό, γιατί ένα αντικείμενο βασικής κλάσης **δεν περιέχει** υπο-αντικείμενο της παράγωγης κλάσης, στο οποίο να μπορεί να αναφερθεί ή να δείξει ένας δείκτης σε παράγωγη κλάση
- Μόνο το αντίστροφο ισχύει, όπως είδαμε στην προηγούμενη ενότητα. Για παράδειγμα:

```
Product prod("AB25", 8.5);
Book *p = &prod; /* Λάθος, δεν επιτρέπεται μετατροπή από δείκτη σε βασική κλάση (π.χ. &prod) σε δείκτη σε παράγωγη (π.χ. p). Ένα Product αντικείμενο δεν είναι Book αντικείμενο. */
Book &r = prod; // Λάθος.
```

- Όμως, αν χρησιμοποιήσουμε προσαρμογή τύπου ο κώδικας μεταγλωττίζεται. Για παράδειγμα:

```
Book *p = static_cast<Book*>(&prod);
Book &r = static_cast<Book&>(prod);
```

- Ωστόσο, αυτή η λειτουργία **δεν είναι** ασφαλής, γιατί ένα `Product` αντικείμενο δεν περιέχει ένα `Book` υπο-αντικείμενο. Αν δοκιμάσουμε να προσπελάσουμε μέλη της παράγωγης κλάσης, μπορεί να προκύψουν προβλήματα στην εκτέλεση του προγράμματος. Για παράδειγμα, αν γράψουμε κατά λάθος `p->display();` μπορεί να προκύψουν προβλήματα αφού η `display();` δεν είναι μέλος της `Product` κλάσης

Μετατροπές από Βασική σε Παραγωγή Κλάση

- Είδαμε στην προηγούμενη ενότητα ότι μπορούμε να εκχωρήσουμε ένα αντικείμενο παράγωγης κλάσης σε ένα αντικείμενο βασικής κλάσης. Το ανάποδο δεν επιτρέπεται. Για παράδειγμα:

```
Book b("AB25", 8.5, "Nice", "Many");
Product p;
b = p; // Λάθος.
```

Επανορισμός Συνάρτησης

- Όπως ήδη γνωρίζουμε, ένα αντικείμενο μίας παράγωγης κλάσης μπορεί να χρησιμοποιήσει συναρτήσεις της βασικής κλάσης
- Υπάρχει όμως περίπτωση να θέλουμε μία συνάρτηση της βασικής κλάσης να συμπεριφέρεται διαφορετικά στην παράγωγη κλάση, δηλαδή, να θέλουμε να έχουμε **πολλαπλές** μορφές (polymorphism) της **ίδιας** συνάρτησης
- Έτσι, ο τρόπος που μία συγκεκριμένη συνάρτηση συμπεριφέρεται μπορεί να εξαρτάται από το αντικείμενο που την καλεί
- Για να επιτευχθεί ο **πολυμορφισμός** πρέπει να ορίσουμε πάλι την συνάρτηση στην παράγωγη κλάση
- Υπάρχουν δύο περιπτώσεις για να ελέγξουμε. Η πρώτη είναι όταν οι συναρτήσεις έχουν την **ίδια** υπογραφή και η δεύτερη με **διαφορετικές** υπογραφές

Επανορισμός με Ίδια Υπογραφή

- Ως παράδειγμα, ας χρησιμοποιήσουμε το ίδιο πρόγραμμα με τις Product και Book κλάσεις. Ας αλλάξουμε το όνομα της display() στην κλάση Book σε show() και να γράψουμε τον παρακάτω κώδικα:

```
void Book::show() const
{
        cout << "T:" << title << " A:" << auth << '\n';
}
```

- Τώρα, έχουμε δύο εκδόσεις της show(), μία στη βασική κλάση και μία στην παράγωγη. Αν γράψουμε τον παρακάτω κώδικα:

```
Product p("A", 1);
Book b("B", 2, "Nice", "Many");
p.show(); // Καλείται η Product::show().
b.show(); // Καλείται η Book::show().
```

- Ο μεταγλωττιστής ελέγχει τον τύπο του αντικειμένου για να αποφασίσει ποια από τις δύο εκδόσεις να καλέσει
- Ουσιαστικά, όταν μία συνάρτηση βασικής κλάσης επανορίζεται σε μία παράγωγη κλάση, η συνάρτηση της παράγωγης κλάσης **κρύβει** τη συνάρτηση της βασικής κλάσης

Επανορισμός με Ίδια Υπογραφή

- Και αν θέλουμε να καλέσουμε την show() της βασικής κλάσης από την παράγωγη κλάση υπάρχει τρόπος; Ναι, απλά, χρησιμοποιούμε τον τελεστή επίλυσης εμβέλειας με το όνομα της βασικής κλάσης. Για παράδειγμα:

```
void Book::show()
{
        cout << "T:" << title << " A:" << auth << '\n';
        Product::show(); /* Προσοχή, αν γράψουμε μόνο
show() θα δημιουργηθεί μία ατέρμονη αναδρομή. */
}
```

Επανορισμός με Διαφορετικές Υπογραφές

- Ας δούμε τώρα την περίπτωση που η συνάρτηση της βασικής κλάσης δηλώνεται διαφορετικά στην παράγωγη κλάση. Για παράδειγμα:

```
#include <iostream>

class A
{
public:
        void show() const {std::cout << "A\n";}
};
class B : public A
{
public:
        void show(int k) const {std::cout << k << '\n';}
};
class C : public B
{
public:
        void show(const char *s) const {std::cout << s << '\n';}
};
int main()
{
        B b;
        b.show(10); // Μεταγλωττίζεται κανονικά.
        b.show(); // Προκαλεί λάθος μεταγλώττισης.

        C c;
        c.show("Test"); // Μεταγλωττίζεται κανονικά.
        c.show(10); // Προκαλεί λάθος μεταγλώττισης.
        c.show(); // Προκαλεί λάθος μεταγλώττισης.
        return 0;
}
```

Επανορισμός με Διαφορετικές Υπογραφές

- Ο επανορισμός μίας συνάρτησης **κρύβει** τις δηλώσεις όλων των συναρτήσεων με το **ίδιο** όνομα στις βασικές κλάσεις, ανεξάρτητα από τις **υπογραφές** των συναρτήσεων
- Ο επανορισμός μίας συνάρτησης σε παράγωγη κλάση **δεν αποτελεί** υπερφόρτωση συνάρτησης
- Έτσι, με τη δήλωση της `show()` στην κλάση `B` δεν έχουμε δύο υπερφορτωμένες εκδόσεις της `show()`, αλλά αυτή η δήλωση κρύβει την δήλωση στη βασική κλάση `A`. Ειδικότερα, με την εντολή `b.show();` ο μεταγλωττιστής ψάχνει το όνομα `show` στην `B`. Όταν το βρει, **σταματάει** την αναζήτηση
- Μετά, επειδή βλέπει ότι για την κλήση της `show()` πρέπει να μεταβιβαστεί όρισμα, εμφανίζει μήνυμα λάθους
- Το σημαντικό να καταλάβετε είναι ότι ο μεταγλωττιστής **σταματάει** την αναζήτηση, **δεν συνεχίζει** να ψάχνει για κάποια άλλη έκδοση της `show()` στις βασικές κλάσεις της `B` που να ταιριάζει με την κλήση
- Παρόμοια, η δήλωση της `show()` στην παράγωγη κλάση `C` **κρύβει** τις δηλώσεις της στις βασικές κλάσεις `A` και `B`. Επομένως, ο μεταγλωττιστής θα εμφανίσει άλλα δύο μηνύματα λάθους
- Για να καλέσουμε τη `show()` μίας βασικής κλάσης πρέπει να χρησιμοποιήσουμε τον τελεστή εμβέλειας και το όνομα της κλάσης. Για παράδειγμα, `c.B.show(10);`

Επανορισμός με Διαφορετικές Υπογραφές

- Και αν θέλουμε να έχουμε υπερφορτωμένες εκδόσεις μίας συνάρτησης από την βασική κλάση στην παράγωγη κλάση υπάρχει τρόπος;
- Ναι, χρησιμοποιούμε **using** δηλώσεις και γράφουμε το όνομα των συνάρτησης. Η **using** δήλωση εισάγει την συνάρτηση στον χώρο εμβέλειας της παράγωγης κλάσης, άρα, οι εκδόσεις είναι διαθέσιμες στην παράγωγη κλάση
- Ο μεταγλωττιστής ελέγχει τις υπογραφές τους για να τις ξεχωρίζει. Για παράδειγμα:

```
class C : public B
{
public:
        using B::A::show;
        using B::show;
        void show(const char *s) const {std::cout << s << '\n';}
};
int main()
{
        C c;
        c.show("Test"); // Καλείται η C::show().
        c.show(10); // Καλείται η B::show().
        c.show(); // Καλείται η A::show().
        return 0;
}
```

Στατική Σύνδεση

- Θεωρήστε ότι μία συνάρτηση παράγωγης κλάσης επανορίζει με την **ίδια** υπογραφή μία συνάρτηση της βασικής κλάσης. Τι συμβαίνει όταν χρησιμοποιούμε **δείκτες** και **αναφορές** για να καλέσουμε αυτή την συνάρτηση; Για παράδειγμα, ας χρησιμοποιήσουμε πάλι το ίδιο πρόγραμμα με την παρακάτω `show()` αντί της `display()`

```
void Book::show() const
{
        cout << "T:" << title << " A:" << auth << '\n';
}
```

- Τι πιστεύετε ότι θα εμφανίσει ο παρακάτω κώδικας;

```
Product p("A", 1);
Book b("B", 2, "Nice", "Many");

Product *ptr1 = &p;
ptr1->show();
Product *ptr2 = &b;
ptr2->show(); // Ποια show() θα κληθεί;
Product& ref = b;
ref.show(); // Ποια show() θα κληθεί;
```

Στατική Σύνδεση

- Στην περίπτωση της `ptr2->show();` το πιθανότερο είναι να απαντήσατε ότι αφού ο `ptr2` δείχνει στο αντικείμενο `b` θα κληθεί η `show()` της κλάσης `Book`
- **Λάθος**, ο μεταγλωττιστής ελέγχει τον **τύπο** του δείκτη και αποφασίζει ανάλογα
- Έτσι, αφού ο τύπος του `ptr2` είναι δείκτης στην κλάση `Product` θα καλέσει τη δική της `show()`
- Δηλαδή, η απόφαση για το ποια συνάρτηση θα κληθεί λαμβάνεται κατά την **μεταγλώττιση** του προγράμματος, και αυτή η απόφαση **δεν** θα αλλάξει
- Αυτός ο τρόπος επιλογής ονομάζεται **στατική σύνδεση** (static binding)
- Το ίδιο ισχύει και για τις **αναφορές**, δηλαδή, ο μεταγλωττιστής χρησιμοποιεί **στατική σύνδεση** για να επιλέξει την έκδοση της `show()`
- Όπως και με τους δείκτες, η επιλογή γίνεται με βάση τον **τύπο** της αναφοράς και **όχι** τον τύπο του αναφερόμενου αντικειμένου. Άρα, ο κώδικας εμφανίζει:

```
C:A  P:1;
C:B  P:2;
C:B  P:2;
```

Εικονικές Συναρτήσεις

- Ας μιλήσουμε τώρα για τις **εικονικές** συναρτήσεις (virtual functions) και ένα από τα σημαντικότερα χαρακτηριστικά της C++, τον **δυναμικό** πολυμορφισμό
- Αν δηλώσουμε την `show()` σαν εικονική στη βασική κλάση, τα αποτελέσματα είναι διαφορετικά
- Για να την δηλώσουμε εικονική χρησιμοποιούμε τη λέξη `virtual`

```
class Product
{
        ...
        virtual void show() const;
};
class Book : public Product
{
        ...
        virtual void show() const;
};
```

- Όταν μία συνάρτηση σε μία παράγωγη κλάση **επανορίζει** μία εικονική συνάρτηση της βασικής κλάσης με την **ίδια** υπογραφή, ονομάζεται **υποσκέλιση** συνάρτησης (function overriding). Όσον αφορά τον τύπο επιστροφής, γενικά, ο τύπος επιστροφής της εικονικής συνάρτησης στην παράγωγη κλάση θα πρέπει να είναι ο ίδιος με αυτόν στη βασική κλάση

Εικονικές Συναρτήσεις

- Εδώ είναι η ουσία του δυναμικού πολυμορφισμού. Όταν μία εικονική συνάρτηση καλείται μέσω **αναφοράς** ή **δείκτη**, η επιλογή της συνάρτησης που θα κληθεί γίνεται με βάση τον **τύπο** του αντικειμένου στο οποίο **αναφέρεται** η αναφορά ή **δείχνει** ο δείκτης. Άρα, ο προηγούμενος κώδικας εμφανίζει:

```
C:A      P:1;
T:Nice   A:Many;
T:Nice   A:Many;
```

- Δηλαδή, μπορούμε να δηλώσουμε μία αναφορά ή δείκτη σε ένα αντικείμενο βασικής κλάσης, και αν αναφερθεί ή δείξει σε κάποιο παράγωγο αντικείμενο, να κληθεί η συνάρτηση του παράγωγου αντικειμένου, αρκεί αυτή να είναι **εικονική**
- Σκεφτείτε, για παράδειγμα, μία ιεραρχία κλάσεων που να παράγονται από την κλάση `Shape` η οποία να περιέχει μία εικονική συνάρτηση `draw()`. Αφού η `draw()` είναι εικονική μπορούμε να δηλώσουμε μία αναφορά ή δείκτη στη `Shape` και να καλούμε την `draw()` της κάθε κλάσης

Δυναμικός Πολυμορφισμός

- Με τις εικονικές συναρτήσεις η απόφαση για το ποια έκδοση της συνάρτησης θα κληθεί δεν λαμβάνεται κατά τη μεταγλώττιση του προγράμματος, αλλά κατά την **εκτέλεσή** του
- Δηλαδή, με τις εικονικές συναρτήσεις, ο πολυμορφισμός επιτυγχάνεται **δυναμικά** (run-time binding ή run-time polymorphism) με την εκτέλεση του προγράμματος και όχι στατικά με την μεταγλώττιση του προγράμματος
- Αυτό συμβαίνει επειδή ο μεταγλωττιστής **δεν γνωρίζει** όταν μεταγλωττίζει το πρόγραμμα (αναφερόμαστε στη γενική περίπτωση και όχι στον προηγούμενο κώδικα) τον τύπο του αντικειμένου που πρόκειται να αναφερθεί η αναφορά ή να δείχνει ο δείκτης
- Για μη-εικονικές συναρτήσεις η διεύθυνση του κώδικα της συνάρτησης που καλείται είναι γνωστή κατά τη μεταγλώττιση του προγράμματος (στατική σύνδεση), ενώ για εικονικές συναρτήσεις προσδιορίζεται κατά την εκτέλεση του προγράμματος (δυναμική σύνδεση)

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
using std::cout;

class A
{
public:
        void f() const {cout << "A ";}
        virtual void g() const {cout << "A ";}
        void v() const {cout << "A ";}
};
class B : public A
{
public:
        void f() const {cout << "B ";}
        virtual void g() const {cout << "B ";}
        virtual void v() const {cout << "B ";}
};
int main()
{
        B b;
        A &r = b;

        r.f();
        r.g();
        r.v();
        return 0;
}
```

Παράδειγμα

- Όταν ένας δείκτης ή μία αναφορά χρησιμοποιείται για να κληθεί μία συνάρτηση, **ελέγχουμε** αν η συνάρτηση είναι εικονική ή όχι. Αν δεν είναι, η σύνδεση είναι στατική και κοιτάζουμε τον τύπο του δείκτη ή της αναφοράς. Αν είναι εικονική, η σύνδεση είναι δυναμική και κοιτάζουμε τον τύπο του αντικειμένου που δείχνει ο δείκτης ή αναφέρεται η αναφορά. Ας εφαρμόσουμε αυτόν τον **κανόνα** στο πρόγραμμα
- Αφού η `f()` δεν είναι εικονική, ο μεταγλωττιστής χρησιμοποιεί στατική σύνδεση για να επιλέξει ποια έκδοση θα καλέσει. Επειδή ο τύπος της αναφοράς `r` είναι η κλάση `A`, θα γίνει η κλήση της `A.f()`. Αφού η `g()` είναι εικονική, χρησιμοποιείται δυναμική σύνδεση. Επειδή ο τύπος του αντικειμένου `b` που αναφέρεται η αναφορά `r` είναι η κλάση `B` θα γίνει η κλήση της `B.g()`. Η δήλωση της `v()` ως εικονική στην παράγωγη κλάση και όχι στη βασική δεν έχει κάποια επίδραση. Άρα, η σύνδεση είναι στατική και θα κληθεί η `A.v()`
- Επομένως, το πρόγραμμα εμφανίζει: `A B A`

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Μία πολύ συνηθισμένη χρήση των εικονικών συναρτήσεων σε πραγματικές εφαρμογές είναι σε συναρτήσεις που δέχονται σαν παραμέτρο αναφορά σε αντικείμενο της βασικής κλάσης. Όπως ξέρουμε, μπορούμε να μεταβιβάσουμε ένα αντικείμενο βασικής κλάσης ή ένα αντικείμενο παράγωγης κλάσης και να χρησιμοποιήσουμε την αναφορά για να καλέσουμε μία εικονική συνάρτηση. Η εικονική συνάρτηση που θα κληθεί είναι του αντικειμένου το οποίο θα μεταβιβαστεί. Για παράδειγμα:

```
#include <iostream>

class Shape
{
public:
        virtual void draw() const {std::cout << "Shape ";}
};
class Circle : public Shape
{
public:
        virtual void draw() const {std::cout << "Circle ";}
};
void f(const Shape& s)
{
        s.draw();
}
int main()
{
        Circle c;
        f(c);
        return 0;
}
```

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Όπως ξέρουμε, επιτρέπεται στην `f()` να μεταβιβάσουμε αναφορά ή δείκτη σε αντικείμενο παράγωγης κλάσης. Επειδή η `draw()` είναι εικονική και η `s` αναφέρεται σε `Circle` αντικείμενο θα κληθεί η `Circle::draw()`. Άρα, το πρόγραμμα εμφανίζει `Circle`
- Για να αντιληφθείτε την δύναμη και την ευελιξία του δυναμικού πολυμορφισμού, μέσω των εικονικών συναρτήσεων, θεωρήστε μία πραγματική εφαρμογή που σχεδιάζει σχήματα. Μπορούμε να προσθέσουμε όποιο σχήμα θέλουμε το όποιο να παράγεται από την `Shape`, να ορίσουμε την δική του `draw()` και να καλούμε την `f()`
- Δείτε την ευελιξία τώρα, δεν χρειάζεται να αλλάξουμε την `f()`. Η `f()` δεν έχει καμία ιδέα για το ποιος τύπος αντικειμένου της μεταβιβάζεται πραγματικά, ωστόσο, καλείται η κατάλληλη `draw()`
- Δηλαδή, μπορούμε να επεκτείνουμε το πρόγραμμά μας χωρίς να αλλάξουμε υπάρχοντα κώδικα. Χωρίς αμφιβολία, ο μηχανισμός των εικονικών συναρτήσεων είναι μία πολύ ισχυρή τεχνική που παρέχει μεγάλη προγραμματιστική ευελιξία
- Σημειώστε ότι αν μεταβιβάσουμε ένα δείκτη σε `Circle` αντικείμενο, αντί για αναφορά, θα κληθεί πάλι η `Circle::draw()` με τις απαραίτητες αλλαγές στον κώδικα βέβαια

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Συνεχίζουμε, αν η `draw()` δεν ήταν εικονική το πρόγραμμα θα εμφάνιζε `Shape`
- Και αν αλλάξουμε την `f()` ώστε να δέχεται αντικείμενο, δηλαδή, γράψουμε: `void f(Shape s) {s.draw();}` τι θα εμφανίσει το πρόγραμμα;
- Αφού τώρα το `s` είναι `Shape` αντικείμενο, θα κληθεί η `Shape::draw()` και το πρόγραμμα εμφανίζει `Shape`. Με αυτό το παράδειγμα, να γίνει πάλι σαφές ότι όταν οι εικονικές συναρτήσεις καλούνται από αντικείμενα και όχι μέσω δεικτών ή αναφορών, δεν υπάρχει ειδική μεταχείριση. Δηλαδή, θα κληθεί η συνάρτηση της αντίστοιχης κλάσης
- Σε ένα άλλο παράδειγμα, τι θα εμφανίσει το παρακάτω πρόγραμμα;

```
int main()
{
        Shape sh;
        Circle cir;

        sh.draw();
        cir.draw();
        return 0;
}
```

Παράδειγμα Χρήσης Εικονικών Συναρτήσεων

- Σε κάθε κλήση, καλείται η `draw()` της αντίστοιχης κλάσης. Άρα, το πρόγραμμα εμφανίζει: `Shape Circle`
- Η σύνδεση αντικειμένων (π.χ. `sh`) με κλήσεις συναρτήσεων, είτε αυτές είναι εικονικές (π.χ. `draw()`) είτε όχι, γίνεται **στατικά** κατά την μεταγλώττιση του προγράμματος, ανάλογα με τον **τύπο** του αντικειμένου
- Η διαφορετική συμπεριφορά των εικονικών συναρτήσεων συμβαίνει κατά την **εκτέλεση** του προγράμματος όταν αυτές καλούνται μέσω **δεικτών** ή **αναφορών** σε αντικείμενα παραγώγων κλάσεων και **όχι** όταν καλούνται από τα **ίδια** τα αντικείμενα. Όταν ένα αντικείμενο χρησιμοποιείται για την κλήση συνάρτησης, η κλήση επιλύεται **στατικά** κατά την **μεταγλώττιση**

Αφαιρετικότητα

- Όταν σχεδιάζουμε τις κλάσεις μίας εφαρμογής, υπάρχει περίπτωση κάποιες κλάσεις που φαίνονται ασυσχέτιστες μεταξύ τους να έχουν **κοινά** στοιχεία
- Για παράδειγμα, έστω ότι θέλουμε να αναπτύξουμε μία εφαρμογή που να διαχειρίζεται ορθογώνια τρίγωνα και παραλληλόγραμμα
- Για τη διαχείριση των τριγώνων, μπορούμε να δημιουργήσουμε την κλάση `Triangle`, η οποία να περιέχει δύο πεδία για τις δύο κάθετες πλευρές του τριγώνου. Για τη διαχείριση των παραλληλογράμμων, μπορούμε να δημιουργήσουμε την κλάση `Rectangle` με δύο πεδία για τις δύο κάθετες πλευρές του παραλληλογράμμου
- Επίσης, και οι δύο κλάσεις θα θέλαμε να έχουν μία συνάρτηση που να εμφανίζει τις διαστάσεις του σχήματος (π.χ. `show()`), μία συνάρτηση που να υπολογίζει το εμβαδό του (π.χ. `area()`) και άλλη μία για την περίμετρο (π.χ. `perim()`).
- Μία κομψή προγραμματιστική τεχνική είναι να εφαρμόσουμε τη φιλοσοφία της **αφαιρετικότητας** (abstraction)
- Και ποια είναι αυτή; Ορίζουμε μία κλάση (π.χ. `Shape`), η οποία περιέχει τα **κοινά** πεδία (π.χ. τις δύο κάθετες πλευρές) και συναρτήσεις των δύο κλάσεων (π.χ. `show()`)
- Η `Shape` θα αποτελέσει την βασική κλάση και οι άλλες δύο θα την κληρονομούν

Αφαιρετικότητα

- Τις συναρτήσεις που λειτουργούν διαφορετικά σε κάθε κλάση τις δηλώνουμε εικονικές
- Για παράδειγμα, θα κάνουμε εικονική την `area()` αφού το εμβαδό του κάθε σχήματος υπολογίζεται διαφορετικά. Αφού η `Shape` δεν αντιπροσωπεύει κάποιο σχήμα δεν χρειάζεται να υλοποιεί την `area()`
- Επιτρέπεται όμως μία κλάση να περιέχει μία συνάρτηση που δεν ορίζεται;
- Ναι, η C++ μας δίνει αυτή τη δυνατότητα με το να τη δηλώσουμε σαν γνήσια εικονική (pure virtual). Και πώς την κάνουμε γνήσια εικονική;
- Προσθέτουμε το `= 0` στο τέλος της δήλωσής της

Παράδειγμα

```
#include <iostream>
#include <vector>
#include <cmath>
using std::cout;
using std::cin;
using std::vector;

class Shape
{
protected:
     float l;
     float h;
public:
     Shape(float len, float hght) {l = len; h =
     hght;}
     void show() {cout << "L:" << l << " H:" << h
     << '\n';}
     virtual void area() const = 0; /* Γνήσια
     εικονική συνάρτηση */
     virtual void perim() const = 0; /* Γνήσια
     εικονική συνάρτηση */
};

class Triangle : public Shape
{
public:
     Triangle(float l, float h) : Shape(l, h) {} /*
     Καλείται ο κατασκευαστής της βασικής κλάσης.
     */
     virtual void area() const {cout << "A:" <<
     l*h/2 << '\n';}
     virtual void perim() const {cout << "P:" <<
     l+h+sqrt(l*l+h*h) << '\n';} /* Για τον
     υπολογισμό της υποτείνουσας εφαρμόζουμε το
     Πυθαγόρειο θεώρημα. */
};

class Rectangle : public Shape
{
public:
          Rectangle(float l, float h) :
Shape(l, h) {}
          virtual void area() const {cout <<
"A:" << l*h << '\n';}
          virtual void perim() const {cout <<
"P:" << 2*(l+h) << '\n';}
};

int main()
{
          int i;
          vector<Shape*> vec_sh(2);

          vec_sh[0] = new Triangle(1, 2);
          vec_sh[1] = new Rectangle(3, 4);
          for(i = 0; i < 2; i++)
          {
                    vec_sh[i]->show();
                    vec_sh[i]->area();
                    vec_sh[i]->perim();
                    delete vec_sh[i];
          }
          return 0;
}
```

Παράδειγμα

- Για ευκολία, τα πεδία `l` και `h` δηλώθηκαν ως προστατευμένα, ώστε οι παράγωγες κλάσεις να έχουν άμεση πρόσβαση σε αυτά
- Για να διαχειριστούμε αντικείμενα και των δύο κλάσεων `Triangle` και `Rectangle` με μία οντότητα και να εφαρμόσουμε **πολυμορφισμό** ένας τρόπος είναι να χρησιμοποιήσουμε ένα διάνυσμα δεικτών στη βασική κλάση `Shape`
- Όπως φαίνεται, μία γνήσια εικονική συνάρτηση **δεν ορίζεται**
- Ο **σκοπός** της είναι να επιτραπεί στις εκδόσεις της συνάρτησης στις παράγωγες κλάσεις να καλούνται με πολυμορφικό τρόπο

Αφηρημένη Κλάση

- Μία κλάση που περιέχει **μία τουλάχιστον** γνήσια εικονική συνάρτηση (π.χ. η `Shape`) ονομάζεται **αφηρημένη** (abstract class)
- Μία αφηρημένη κλάση μπορεί να περιέχει **μόνο** γνήσιες εικονικές συναρτήσεις
- Σημειώστε ότι **δεν επιτρέπεται** να δημιουργήσουμε **αντικείμενα** μίας αφηρημένης κλάσης. Δηλαδή, αν γράψουμε: `Shape s(1, 2);` ο μεταγλωττιστής θα εμφανίσει μήνυμα **λάθους**
- Η λογική μίας αφηρημένης κλάσης είναι να περιλαμβάνει τις κοινές ιδιότητες των αντίστοιχων οντοτήτων και να χρησιμοποιείται μόνο ως βασική
- Μία γνήσια εικονική συνάρτηση που δεν ορίζεται στην παράγωγη κλάση εξακολουθεί να είναι γνήσια εικονική. Άρα, η παράγωγη κλάση γίνεται επίσης αφηρημένη. Π.χ:

```
class A
{
        ...
        virtual void f() = 0;
        virtual void g() = 0;
};
class B : public A
{
        ...
        virtual void g() {}; // Η f() παραμένει γνήσια εικονική.
};
class C : public B
{
        ...
        virtual void f() {};
};
B b; /* Λάθος, η B είναι αφηρημένη κλάση. Δεν επιτρέπεται η δημιουργία αντικειμένων της. */
C c; // Σωστό.
```

Παρατηρήσεις

- Το προγραμματιστικό μοντέλο που βασίζεται στην έννοια της αφαιρετικότητας είναι αρκετά συνηθισμένο
- Για να επιστρέψουμε σε αυτό, τα **κοινά** στοιχεία που προκύπτουν από τον σχεδιασμό των κλάσεων τοποθετούνται σε μία **βασική** κλάση, ενώ οι συναρτήσεις που υλοποιούνται διαφορετικά σε κάθε παράγωγη κλάση δηλώνονται ως **γνήσιες** εικονικές
- Με αυτόν τον τρόπο σχεδίασης αποφεύγεται η επανάληψη του ίδιου κώδικα και έτσι διευκολύνεται η συντήρηση και αναβάθμιση του προγράμματος
- Γενικά, οι **αφηρημένες** κλάσεις συνηθίζεται να περιέχουν μόνο γνήσιες εικονικές συναρτήσεις (τα πεδία μεταβλητές δηλώνονται στις παράγωγες κλάσεις) και, επομένως, δεν περιέχουν κατασκευαστή (αφού δεν υπάρχουν πεδία μεταβλητές για να αρχικοποιηθούν, είναι απίθανο να χρειάζεται κατασκευαστής)
- Ουσιαστικά, μία αφηρημένη κλάση αποτελεί ένα είδος **συμβολαίου** για το τι θα πρέπει να παρέχει μία παράγωγη κλάση. Η αφηρημένη κλάση καθορίζει το «**τι πρέπει να υλοποιηθεί**», αφήνοντας την υλοποίηση των εικονικών συναρτήσεων να την αναλάβει ο προγραμματιστής της κάθε παράγωγης κλάσης

Ιδιωτική Κληρονομικότητα

- Εκτός από τη δημόσια κληρονομικότητα που είδαμε στο Κ.20, μία κλάση μπορεί να παραχθεί από μία βασική κλάση με **ιδιωτική** και **προστατευμένη** πρόσβαση
- Με την **ιδιωτική** κληρονομικότητα τα **δημόσια** και **προστατευμένα** μέλη της βασικής κλάσης γίνονται **ιδιωτικά** μέλη της παράγωγης κλάσης
- Αυτό σημαίνει ότι είναι προσπελάσιμα από τις συναρτήσεις της παράγωγης κλάσης, αλλά όχι έξω από την παράγωγη κλάση
- Για να δηλώσουμε την ιδιωτική κληρονομικότητα χρησιμοποιούμε τη λέξη `private` στη δήλωση της παράγωγης κλάσης

Παράδειγμα

```
#include <iostream>

class A
{
protected:
        int r;
        void sub(int i) {v -= i;}
public:
        int v;
        A() : v(1), r(2) {};
        void add(int i) {r += i;}
};

class B : private A
{
public:
        void show();
};

void B::show()
{
        /* Όλες οι παρακάτω ενέργειες
είναι επιτρεπτές. */
        add(3);
        sub(4);
        std::cout << v+r << '\n';
}
```

```
int main()
{
    B b;

    b.show();
    b.add(5); // Μη επιτρεπτή ενέργεια.
    b.sub(6); // Μη επιτρεπτή ενέργεια.
    std::cout << b.v << '\n'; /* Μη
    επιτρεπτή ενέργεια. */
    return 0;
}
```

Προστατευμένη Κληρονομικότητα

- Για να παράξουμε μία κλάση με προστατευμένη πρόσβαση χρησιμοποιούμε τη λέξη `protected` στη δήλωσή της
- Τα **δημόσια** και **προστατευμένα** μέλη της βασικής κλάσης γίνονται **προστατευμένα** μέλη της παράγωγης κλάσης
- Επομένως, όπως και στην ιδιωτική πρόσβαση, είναι προσπελάσιμα από τις συναρτήσεις της παράγωγης κλάσης, αλλά όχι έξω αυτήν

Προστατευμένη Κληρονομικότητα

- Η κύρια διαφορά μεταξύ της ιδιωτικής και προστατευμένης κληρονομικότητας είναι όταν παράγεται μία νέα κλάση από την παράγωγη κλάση. Με την ιδιωτική κληρονομικότητα, η δεύτερη παράγωγη κλάση δεν έχει απευθείας πρόσβαση στα μέλη της βασικής κλάσης γιατί έχουν γίνει ιδιωτικά στην πρώτη παράγωγη. Για παράδειγμα, ας προσθέσουμε την κλάση C στο προηγούμενο πρόγραμμα:

```
class C : private B
{
public:
        void f();
};
void C::f() /* Όλες οι παρακάτω ενέργειες είναι μη επιτρεπτές. */
{
        add(1);
        sub(2);
        cout << v+r << '\n';
}
```

- Επειδή τα μέλη και οι συναρτήσεις της βασικής κλάσης A έχουν γίνει ιδιωτικά μέλη της κλάσης B, **δεν επιτρέπεται** η απευθείας πρόσβαση σε αυτά από την κλάση C. Αντίθετα, αν η κληρονομικότητα είναι προστατευμένη, η πρόσβαση **επιτρέπεται**

Κανόνες Κληρονομικότητας

- Τώρα που έχουμε μιλήσει για όλα τα είδη κληρονομικότητας, ας συνοψίσουμε τους κανόνες που ισχύουν για την πρόσβαση στα μέλη της βασικής κλάσης. Έστω ότι η κλάση **B** παράγεται από την κλάση **A**:

α) Αν η **A** είναι **ιδιωτική** βασική κλάση, τα δημόσια και προστατευμένα μέλη της **A** γίνονται ιδιωτικά μέλη της **B** και είναι προσβάσιμα μόνο από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**

β) Αν η **A** είναι **προστατευμένη** βασική κλάση, τα δημόσια και προστατευμένα μέλη της **A** γίνονται προστατευμένα μέλη της **B** και είναι προσβάσιμα μόνο από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**, καθώς και από συναρτήσεις-μέλη και φιλικές συναρτήσεις των κλάσεων που παράγονται από την **B**

γ) Αν η **A** είναι **δημόσια** βασική κλάση, τα δημόσια μέλη της είναι προσβάσιμα από οποιαδήποτε συνάρτηση της **B**. Τα προστατευμένα μέλη της **A** είναι προσβάσιμα από συναρτήσεις-μέλη και φιλικές συναρτήσεις της **B**, καθώς και από συναρτήσεις-μέλη και φιλικές συναρτήσεις των κλάσεων που παράγονται από την **B**

Πολλαπλή Κληρονομικότητα

- Στο Κ.20 μιλήσαμε για την **μονή** κληρονομικότητα, δηλαδή, μία κλάση να παράγεται από μία μόνο βασική κλάση
- Σε αυτήν την ενότητα, θα μιλήσουμε για **πολλαπλή** κληρονομικότητα, δηλαδή, την δυνατότητα μία κλάση να παράγεται από παραπάνω από μία βασικές κλάσεις (multiple inheritance)
- Η πολλαπλή κληρονομικότητα είναι χρήσιμη όταν θέλουμε να **συγχωνεύσουμε** στοιχεία διαφορετικών, συνήθως μη σχετικών κλάσεων, σε μία τρίτη κλάση
- Όμως, επειδή αυξάνει η πολυπλοκότητα του προγράμματος, θα πρέπει να χρησιμοποιείται με σύνεση
- Ας δούμε ένα παράδειγμα και προβλήματα που μπορεί να προκύψουν

Παράδειγμα

```
#include <iostream>

class A
{
public:
        void show() const {...}
        ...
};

class B
{
public:
        void show() const {...}
        ...
};

class C : public A, public B
{
        ...
}

int main()
{
        C c;

        c.show(); /* Ασάφεια, ποια show() θα κληθεί; */
        ...
}
```

Πολλαπλή Κληρονομικότητα

- Η κλάση C παράγεται με δημόσια κληρονομικότητα από τις κλάσεις A και B. Κληρονομεί όλα τα μέλη των A και B κλάσεων
- Όπως φαίνεται στη δήλωση της C, οι κλάσεις διαχωρίζονται με κόμμα και καθορίζουμε το προσδιοριστικό πρόσβασης για κάθε βασική κλάση
- Για την κατασκευή του c, πρώτα κατασκευάζονται τα υπο-αντικείμενα των βασικών του κλάσεων
- Η σειρά κατασκευής είναι, από αριστερά προς τα δεξιά, όπως οι βασικές κλάσεις εμφανίζονται στη δήλωση της κλάσης (π.χ. public A, public B)
- Συγκεκριμένα, όταν δημιουργείται το c καλείται ο εξ'ορισμού κατασκευαστής της C, ο οποίος καλεί τον εξ'ορισμού κατασκευαστή της A, μετά της B, και μετά ολοκληρώνεται η δημιουργία του c
- Δηλαδή, πρώτα κατασκευάζεται το A υπο-αντικείμενο και μετά το B
- Όταν καταστρέφεται το c, οι αποδομητές καλούνται με αντίστροφη σειρά, δηλαδή, πρώτα της C, μετά της B, και τελευταία της A

Πολλαπλή Κληρονομικότητα

- Η υλοποίηση της πολλαπλής κληρονομικότητας μπορεί να είναι πιο δύσκολη και επιρρεπής σε προβλήματα σε σύγκριση με την απλή
- Για παράδειγμα, με την πολλαπλή κληρονομικότητα μπορεί να κληρονομηθούν ίδια ονόματα, και όπως φαίνεται και στον παραπάνω κώδικα, να προκύψουν προβλήματα ασάφειας
- Ποια `show()` θα κληθεί; Της `A` ή της `B`;
- Επειδή ο μεταγλωττιστής δεν ξέρει ποια να καλέσει, θα εμφανίσει μήνυμα λάθους παρόμοιο με «call to show() is ambiguous»
- Για να ξεπεράσουμε το πρόβλημα, μία λύση είναι να χρησιμοποιήσουμε τον τελεστή εμβέλειας `::` και το όνομα της κλάσης που περιέχει τη `show()` που θέλουμε να καλέσουμε. Για παράδειγμα:

```
c.A::show(); // Καλείται η show() της A
```

Κληρονομικότητα και Στατικά Μέλη

- Αν σε μία βασική κλάση ορίζεται ένα `static` μέλος, αυτό δημιουργείται μόνο μία φορά ανεξάρτητα από τον αριθμό των κλάσεων που παράγονται από την βασική. Για κάθε στατικό μέλος ισχύουν οι ίδιοι κανόνες πρόσβασης που ισχύουν και για τα συνηθισμένα μέλη. Για παράδειγμα:

Παράδειγμα

```
#include <iostream>

class A
{
public:
        static inline int v = 10;
};

class B : public A
{
public:
        void f() {v = 20;}
};

class C : public A
{
public:
        void g() {v = 30;}
}

int main()
{
        B b;
        C c;
        b.f();
        c.g();
        std::cout << b.v << ' ' << c.v << '\n'; /* Εναλλακτικά, μπορούμε να
γράψουμε cout << B::v << ' ' << C::v; */
}
```

Κληρονομικότητα και Στατικά Μέλη

- Επειδή υπάρχει μόνο μία υπόσταση του `v` οι συναρτήσεις `f()` και `g()` αλλάζουν διαδοχικά την τιμή του. Έτσι, το πρόγραμμα εμφανίζει `30 30`. Αν δεν δηλώσουμε το `v` ως `static`, το πρόγραμμα θα εμφανίσει `20 30`, αφού η `b.v` είναι διαφορετική μεταβλητή από την `c.v`