Inheritance - κληρονομικότητα

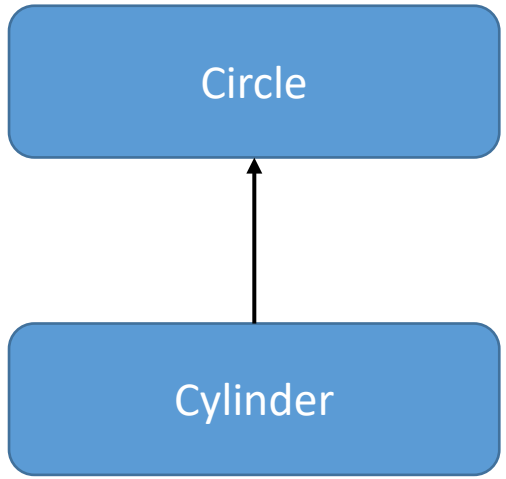
Βασικές έννοιες κληρονομικότητας

Ιεραρχία κλάσεων / Βασική – Παράγωγη κλάση

- Η κληρονομικότητα είναι μια τεχνική από την οποία παράγουμε μια καινούργια κλάση (***παράγωγη***) από μία κλάση που υπάρχει ήδη (***βασική***)
- Η παράγωγη κλάση
  a) κληρονομεί όλα τα μέλη της βασικής (εκτός από constructors/destructors)
  b) μπορεί δε να τροποποιήσει κάποια από αυτά
  c) μπορεί να προσθέσει κάποια καινούργια
- Η βασική ονομάζεται και κλάση (**γονέας - base**) ενώ η παραγόμενη (**παιδί – child - derived**)

Παράδειγμα ιεραρχίας

- Έστω ότι έχουμε μια κλάση που περιγράφει έναν κύκλο
- Μπορούμε να δημιουργήσουμε μια καινούργια κλάση που περιγράφει έναν κύλινδρο
- Η κλάση κύλινδρος
  - ***Προσθέτει*** καινούργια μεταβλητή μέλος ipsos
  - ***Προσθέτει*** μια καινούργια μέθοδο ipol_ogko
  - ***Διαφοροποιεί*** την μέθοδο ipol_epifaneia



Μπορούμε να πούμε ότι η Cylinder ***isa*** Circle

Δήλωση βασικής κλάσης

- Στην βασική κλάση μπορούμε να ορίσουμε μια member function σαν virtual
- Αυτό σημαίνει ότι περιμένουμε από τις derived classes να τις ορίσουν ξανά για τον εαυτό τους

```
class Circle
{
public:
    Circle() :aktina{ 0 } {}
    Circle(double a) : aktina{ a } {}
    void set_aktina(double a);
    virtual double ipol_epifaneia() const;
private:
    double aktina;
};
```

Παράδειγμα παράγωγης κλάσης

- Στην παράγωγη κλάση
- Δηλώνουμε από ποια/ποιες κλάσεις κληρονομούμε στο ***derivation list***
- Δηλώνουμε τα καινούργια member variables / functions
- Δηλώνουμε αυτά που αλλάζουμε από την πατρική (override)

```
class Cylinder : public Circle
{
public:
    Cylinder();
    Cylinder(double a, double i);
    void set_ipsos();
    double ipol_epifaneia() const override;
    double ipol_ogko() const;
private:
    double ipsos;
};
```

Περιεχόμενα Cylinder

Όνομα	Είδος	Που ορίστηκε	Access Spec	Έχει πρόσβαση η Cylinder?
aktina	variable	Circle	Private	ΟΧΙ
Set_aktina	method	Circle	Public	ΝΑΙ
Ipsos	Variable	Cylinder	Private	ΝΑΙ
Ipol_epifaneia	Method	Cylinder	Public	ΝΑΙ
Ipol_ogo	Method	Cylinder	Public	ΝΑΙ
Set_Ipsos	Method	Cylinder	Public	ΝΑΙ
Constructors	Cons	Cylinder	Public	ΝΑΙ

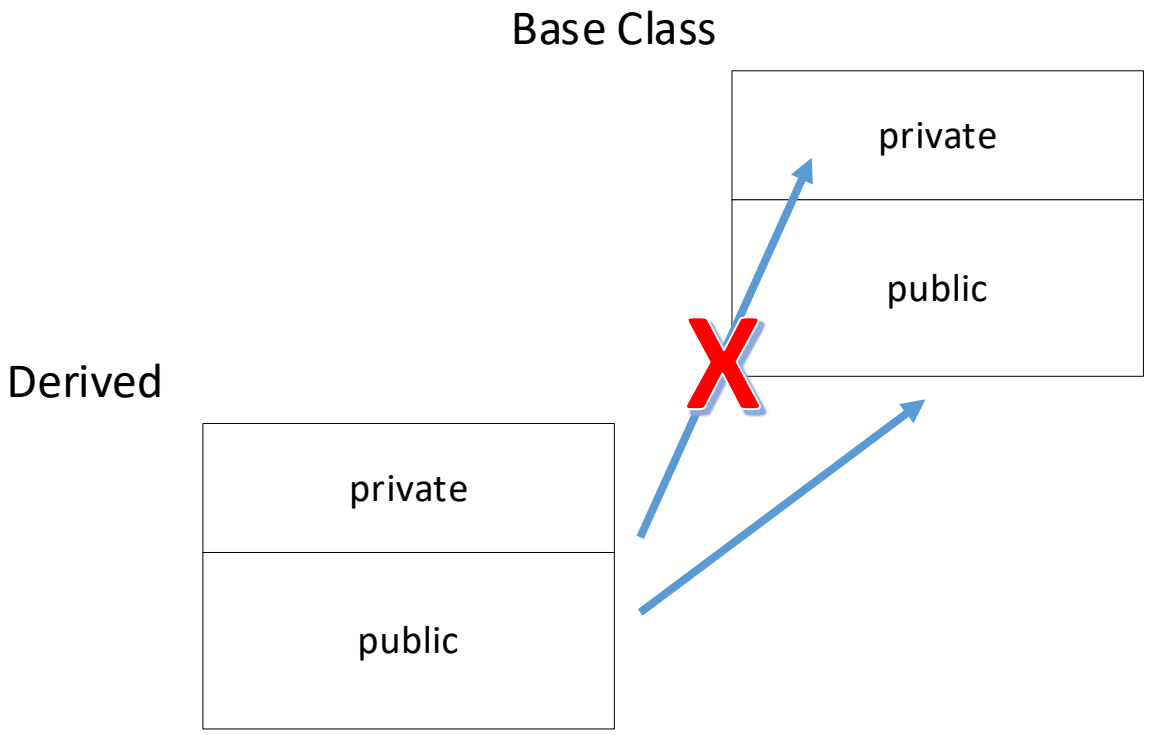
Redefining vs Overloading

- Η μέθοδος ***ipol_epifaneia*** που βρίσκεται στην κλάση Cylinder στην ουσία «επανακαθορίζει» την μέθοδο ***ipol_epifaneia*** που έχει ήδη οριστεί στην πατρική κλάση Circle
- Θα μπορούσε όμως να γίνει και **υπερφόρτωση** της αρχικής μεθόδου
- Για να γίνει αυτό θα πρέπει η μέθοδος ipol_epifaneia που θα οριστεί στην κλάση απόγονο (Cylinder) να διαφέρει στον αριθμό ή τον τύπο των παραμέτρων της
- Σε αυτή την περίπτωση η κλάση απόγονος (Cylinder) θα έχει δύο διαφορετικές μεθόδους ipol_epifaneia. Μία που κληρονομεί από τον γονέα και μία που ορίζει αυτή
- Για να μας «προστατέψει» η C++ ορίζει το επίθεμα override

Protected access specifier

Private μέλη γονικής κλάσης

- ΠΡΟΣΟΧΗ! Η πρόσβαση **στα ιδιωτικά μέλη της γονικής κλάσης επιτρέπεται ΜΟΝΟ στα αντικείμενα της ίδιας κλάσης ΌΧΙ στα αντικείμενα της κλάσης παιδί**!!!



Private μέλη γονικής κλάσης

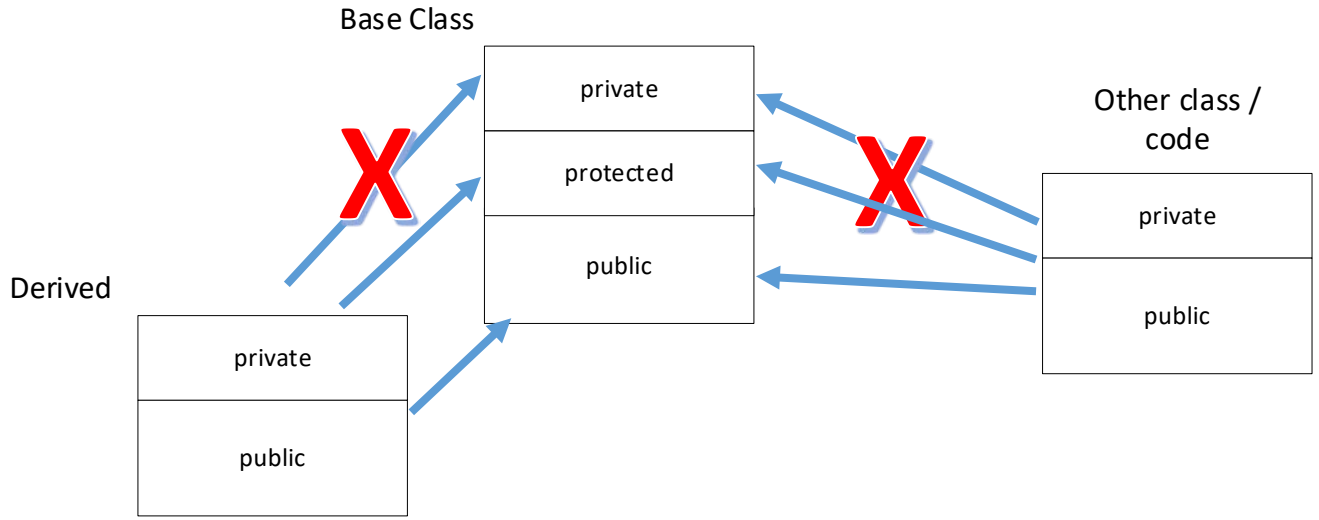
- Για παράδειγμα οι μέθοδοι της *Cylinder* ΔΕΝ έχουν πρόσβαση στα ιδιωτικά μέλη της *Circle* (*aktina*)

```
double Cylinder::ipol_epifaneia() const
{
    double epif;
    epif = 2 * M_PI * aktina*(aktina + ipsos);
    return epif;
}
```

Το επίπεδο πρόσβασης Protected

Επίπεδο πρόσβασης protected.

- Επιτρέπει πρόσβαση στις derived κλάσεις
- Δεν επιτρέπει σε άλλες κλάσεις / κώδικα



Το επίπεδο πρόσβασης Protected

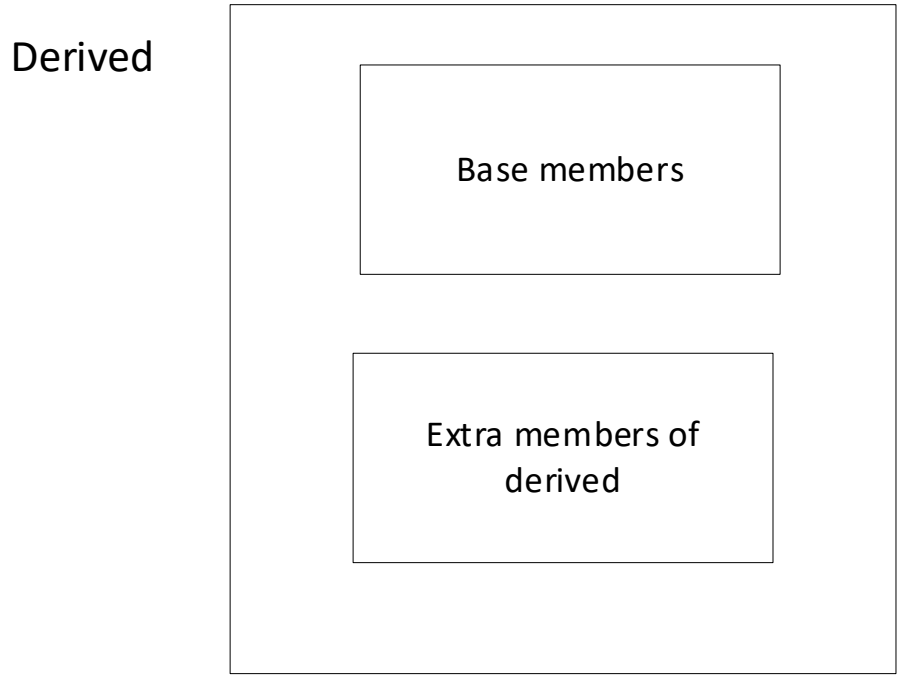
- Επομένως αν χαρακτηρίσουμε protected την *aktina* στην *Circle* τότε η *Cylinder* που παράγετε από αυτή μπορεί να έχει πρόσβαση σε αυτή

```
    Circle();
protected:
    int aktina;
};
```

```
double Cylinder::ipol_epifaneia() const
{
    double epif;
    epif = 2 * M_PI * aktina*(aktina + ipsos);
    return epif;
}
```

Derived-to-Base Conversion

- Κάθε αντικείμενο της derived κλάσης (cylinder) περιέχει μέσα του τόσο τα δικά του members όσο και αυτά της base
- Μπορούμε να πούμε ότι ένα αντικείμενο της derived περιέχει μέσα του και ένα ένα υπο-αντικείμενο της base κλάσης (circle)



Derived-to-Base Conversion

- Το γεγονός ότι ένα derived αντικείμενο περιέχει μέσα του και ένα base υπο-αντικείμενο μας επιτρέπει να χρησιμοποιούμε derived αντικείμενα στην θέση των base
- Συγκεκριμένα μπορούμε να αναθέσουμε pointer και reference του base σε derived objects

```
Circle base;
Cylinder der;

Circle* p = &base;
p = &der;

Circle& r = der;
```

Χρήση base members από την παράγωγη

- Η δημιουργία και η χρήση των αντικειμένων της παράγωγης κλάσης γίνεται όπως και κάθε άλλης.
- Μια προσοχή στην μέθοδο ipol_epifaneia
- Αν καλεστεί από αντικείμενο τύπου Cylinder τότε θα καλεστεί η καινούργια εκδοχή της. Υπάρχει ο scope resolution για να καθορίσουμε ποια θέλουμε να εκτελεστεί

```
int main()
{
    Cylinder a;

    cout << a.ipol_epifaneia() << endl;
    cout << a.Circle::ipol_epifaneia() << endl;
}
```

Διαφορετική πρόσβαση κληρονομικότητας

- Στο προηγούμενο παράδειγμα η κλάση ***Cylinder*** κληρονόμησε την κλάση ***Circle*** σαν ***public***
- Αυτό σημαίνει ότι όλα τα πεδία της Circle θα έχουν στην Cylinder το ίδιο επίπεδο πρόσβασης με αυτό που είχαν αρχικά
- Μπορούμε όμως μια κλάση να την κληρονομήσουμε σαν ***private***
- Αυτό θα κάνει ΌΛΑ τα στοιχεία της κλάσης που κληρονομούμε private ακόμα και αυτά που δεν ήταν.
- Αν δεν γραφεί θεωρείται private

Παράδειγμα private κληρονομιάς

- Τα αντικείμενα της cylinder πλέον δεν μπορούν να χρησιμοποιούν ΟΥΤΕ τα public τμήματα της Circle
- Αντίθετα τα αντικείμενα της Circle εξακολουθούν κανονικά

```
class Cylinder : private Circle
{
public:
    //Cylinder();
```

```
int main()
{
    Cylinder a;

    cout << a.ipol_epifaneia() << endl;
    cout << a.Circle::ipol_epifaneia() << endl;

    a.set_aktina(4);
}
```

Constructors / destructors και παράγωγες κλάσεις

Default Constructor

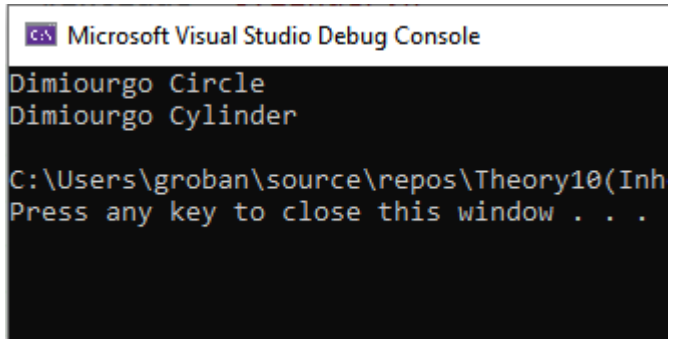
- Κάθε φορά που δημιουργείτε ένα αντικείμενο μιας κλάσης χωρίς αρχικοποίηση στην ουσία καλείται ο default constructor
- Όταν δημιουργείται αντικείμενο παράγωγης κλάσης στην πραγματικότητα δημιουργείται πρώτα ένα αντικείμενο της βασικής κλάσης και μετά ένα της παράγωγης
- Επομένως καλείται πρώτα ο constructor της βασικής κλάσης και στην συνέχεια ο constructor της παράγωγης
- Το ίδιο ισχύει αν έχουμε και κληρονομικότητα σε παραπάνω από ένα επίπεδα
- Το αντίστροφο ισχύει για τον destructor

Παράδειγμα δημιουργίας αντικειμένου παράγωγης κλάσης

- Παρατηρούμε ότι καλείται ο Circle() χωρίς να αναφέρεται από τον Cylinder

```
Circle::Circle(): aktina{0}
{
    cout << "Dimiourgo Circle\n";
}
```

```
Cylinder::Cylinder():ipsos{0}
{
    cout << "Dimiourgo Cylinder\n";
}
```



Δημιουργία αντικειμένου παράγωγης κλάσης

- Η σωστή πρακτική είναι να καλούμε τον base constructor στην initialization list του derived constructor
- Με αυτόν τον τρόπο δημιουργείται το υπο-αντικείμενο και ο derived constructor αρχικοποιεί μόνο τα καινούργια member variables

```
Cylinder::Cylinder():Circle(), ipsos{0}
{
    cout << "Dimiourgo Cylinder\n";
}
```

Μη Default constructor

- Κατά την δημιουργία ενός αντικειμένου της παράγωγης κλάσης με την χρήση μη default constructor στην πραγματικότητα καλείται ο default constructor της πατρικής πρώτα
- Αυτό μπορεί να παραληφθεί αν ο constructor της παράγωγης καλεί από μόνος του έναν άλλο constructor

```
Cylinder::Cylinder(double a, double i): Circle(a), ipsos{i}
{
}
```

Πολυμορφισμός

Χρήση παράγωγης στην θέση γονικής

- Αυτό δίνει την δυνατότητα να χρησιμοποιηθεί σε περιπτώσεις όπου απαιτείται ένα αντικείμενο της πατρικής
- Σε αυτή την περίπτωση όμως θα έχει πρόσβαση μόνο στα στοιχεία της πατρικής.

```
int main()
{
    Cylinder a{ 3,5 };

    Circle* p;

    p = &a;

    p->
}
```

```
ipol_epifaneia
set_aktina
```

Έχει πρόσβαση ΜΟΝΟ στα μέλη της Circle
Παρόλο που είναι Cylinder

Virtual method

- ***virtual μέθοδος***: ο compiler επιλέγει τον ορισμό που θα εκτελέσει ανάλογα με τον τύπο του αντικειμένου όχι την ώρα του compile (compile time) αλλά την ώρα της εκτέλεσης (run time)

```
int main()
{
    Circle a{ 3 };
    Cylinder b{ 3,5 };

    Circle* p;
    p = &a;

    cout << "epifaneia kiklou: " << p->ipol_epifaneia() << endl;

    p = &b;

    cout << "epifaneia kilindrou: " << p->ipol_epifaneia() << endl;
```

```
Microsoft Visual Studio Debug Console
epifaneia kiklou: 28.2743
epifaneia kilindrou: 150.796

C:\Users\groban\source\repos\Theory10(Inheritan
Press any key to close this window . . .
```

Πολυμορφισμός

- Πολυμορφισμός στον αντικειμενοστραφή προγραμματισμό είναι η τεχνική με την οποία αντικείμενα διαφορετικών τύπων χρησιμοποιούνται με τον ίδιο τρόπο
- Compile type:
  - Η υπερφόρτωση συναρτήσεων και τελεστών είναι ένα παράδειγμα πολυμορφισμού που συμβαίνει κατά την μεταγλώττιση του προγράμματος
- Run time:
  - Η χρήση virtual μεθόδων είναι ένα παράδειγμα πολυμορφισμού που συμβαίνει κατά την διάρκεια της εκτέλεσης. Η πατρική κλάση μπορεί να ορίσει μία μέθοδο virtual που στην πραγματικότητα θα εκτελεί τον κώδικα της μεθόδου ανάλογα με τον τύπο της παράγωγης κλάσης από την οποία θα καλείται

Πολυμορφιμός

- Κάθε αντικείμενο της παράγωγης κλάσης είναι ένα «επαυξημένο» αντικείμενο της παράγωγης κλάσης
- Αυτό μας δίνει την δυνατότητα να χρησιμοποιήσουμε αντικείμενα της «παράγωγης» κλάσης αντί για αντικείμενα της βασικής
- Αυτή η ιδιότητα μας δίνει την δυνατότητα να δημιουργήσουμε κώδικα που να «χειρίζεται» αντικείμενα είτε της βασικής είτε της «παράγωγης» κλάσης

Παράδειγμα πολυμορφισμού

- Ας υποθέσουμε ότι θέλουμε να δημιουργήσουμε μια συνάρτηση που θα υπολογίζει πόσο θα κοστίσει να χρωματίσει ένα σχήμα
- Τα σχήματα μας είναι τα Circle και Cylinder
- Για να υπολογίσει θέλει το σχήμα και το κόστος του χρώματος ανά τμ
- Λύσεις
  - Χωρίς πολυμορφισμό – μία συνάρτηση για κάθε σχήμα
  - Με πολυμορφισμό – μία συνάρτηση και virtual

Λύση χωρίς πολυμορφισμό

```
double calc_paint(const Circle& c, double price)
{
    return c.ipol_epifaneia() * price;
}

double calc_paint(const Cylinder& cyl, double price)
{
    return cyl.ipol_epifaneia() * price;
}


int main()
{
    Circle a{ 3 };
    Cylinder b{ 3,5 };

    cout << "Kostos gia kiklou: " << calc_paint(a,10) << endl;
    cout << "epifaneia kilindrou: " << calc_paint(b,10) << endl;
}
```

Η ipol_epifaneia ΔΕΝ είναι virtual

```
Kostos gia kiklou: 282.743
epifaneia kilindrou: 1507.96
```

Λύση με πολυμορφισμό

```
double calc_paint(const Circle& c, double price);

int main()
{
    Circle a{ 3 };
    Cylinder b{ 3,5 };

    cout << "Kostos gia kiklou: " << calc_paint(a,10) << endl;
    cout << "epifaneia kilindrou: " << calc_paint(b,10) << endl;
}

double calc_paint(const Circle& c, double price)
{
    return c.ipol_epifaneia() * price;
}
```

```
Kostos gia kiklou: 282.743
epifaneia kilindrou: 1507.96
```

Συνάρτηση που να δέχεται παράμετρο μόνο Base (σαν reference ή pointer)

Δίνουμε όρισμα είτε base είτε derived

Η ipol_epifaneia είναι virtual επομένως εκτελεί την εκδοχή της ipol_epifaneia ανάλογα με τον τύπο του αντικειμένου at run time

Το προσδιοριστικό final

- Μπορούμε να χρησιμοποιήσουμε το προσδιοριστικό final για να καθορίσουμε ότι δεν θέλουμε
  - Να μπορεί να χρησιμοποιηθεί σαν base μια κλάση
  - Να γίνει override μια virtual μέθοδος στις derived classes

```
class Circle final
{
public:
```

```
Circle();
Circle(double a) : aktina{ a } {}
virtual double ipol_epifaneia() const final;
```

Μερικά θέματα

Υπερφόρτωση τελεστή <<

- Ο τελεστής εκτύπωσης υλοποιείται με την χρήση friend function
- Επομένως δεν μπορεί να γίνει virtual
- Αυτό που μπορεί να γίνει είναι να χρησιμοποιηθεί μια βοηθητική virtual μέθοδος print

Υπερφόρτωση τελεστή <<

- Ο τελεστής δηλώνεται μόνο για την base κλάση
- Αυτό που κάνει είναι να καλεί μία virtual member function που να πραγματοποιεί εκτύπωση των member variables
- Στην κάθε derived κάνουμε override την virtual member function

```
void Circle::print(ostream& out) const
{
    out << "Kiklos aktina:" << aktina;
}

ostream& operator<<(ostream& out, const Circle& r)
{
    r.print(out);
    return out;
}
```

```
void Cylinder::print(ostream& out) const
{
    out << "Kylindros aktina:" << aktina << " ipsos:" << ipsos;
}
```

Αποθήκευση σε δομή

- Μπορούμε να αποθηκεύσουμε σε δομή (π.χ. Vector) τόσο base όσο και derived κλάσεις
- Θα πρέπει όμως τα elements να είναι pointer σε base
- Αλλιώς θα αποθηκεύει μόνο το υπο-αντικείμενο της base

```
vector<Circle *> lista;
Circle a{ 3 };
Cylinder b{ 4,5 };

lista.push_back(&a);
lista.push_back(&b);

for (auto x = lista.begin(); x != lista.end(); ++x)
    cout << **x << endl;
```

Δυναμική ανίχνευση τύπου αντικειμένων

- Μπορούμε να αναγνωρίζουμε at run time αν ο τύπος ενός αντικειμένου είναι base ή derived
- Ένας τρόπος είναι να δημιουργήσουμε μια virtual που να επιστρέφει έναν ακέραιο διαφορετικό για την base και την derived

```
    virtual int what() const;


int Circle::what() const
{
    return 0;
}


for (auto x:lista) {
    auto test = x->what();
    if (test==1) {
        cout << "Einai kylindros\n";
    }
    else {
        cout << "Einai kiklos\n";
    }
}
```

Δυναμική ανίχνευση τύπου αντικειμένων

- Το ίδιο μπορούμε να κάνουμε με την dynamic cast
- Προσπαθεί να κάνει δυναμική μετατροπή σε derived με την χρήση της dynamic cast
- Αν δεν το καταφέρει τότε είναι base

```
for (auto x = lista.begin(); x != lista.end(); ++x) {
    Cylinder* test = dynamic_cast<Cylinder*>(*x);
    if (test) {
        cout << "Einai kylindros\n";
    }
    else {
        cout << "Einai kiklos\n";
    }
}
```