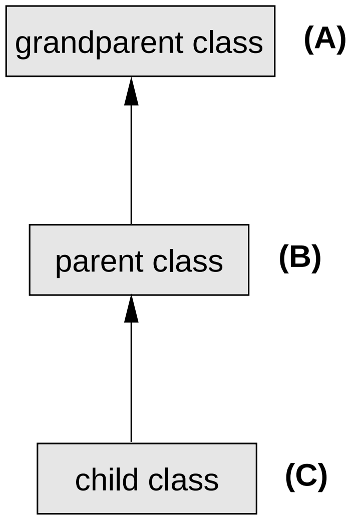
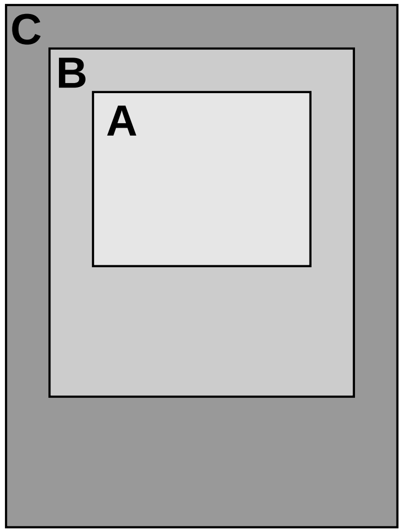
# Comp151

# Inheritance: Initialization & Substitution Principle

# Initializing Base Class Objects

If class C is derived from class B which is in turn derived from class A, then C will contain data members of both B and A.




Class C's constructor can only call class B's constructor; and, class B's constructor can only call class A's constructor. i.e. it is the *responsibility* of each derived class to initialize its direct base class correctly.

# Example: Initializing Base Class Objects

Before a **Student** object can come into existence, we have to create its **Person** part. This has to be done using one of the constructors of **Person**. We use the same "colon syntax" as for initializing data members:

```
Student::Student(string n, string a, Department d)
        : Person(n, a, d), enrolled(NULL), num_courses(0) { }
```

Similarly, **PG_Student** has to create its **Student** part before it can be created; but, it does NOT need to create its **Person** part by calling **Person**'s constructor. In fact, its **Person** part should have been created by **Student**.

```
PG_Student(string n, string a, Department d) :
    Student(n, a, d), research_topic(NONE) { }
```

# Order of Construction/Destruction

```
#include <iostream.h>

class Address { public:
    Address() { cout << "Address's constructor" << endl; }
    ~Address() { cout << "Address's destructor" << endl; }
};

class Person { public:
    Person() { cout << "Person's constructor" << endl; }
    ~Person() { cout << "Person's destructor" << endl; }
};

class Student : public Person {
  private: Address address;
  public:
    Student() { cout << "Student's constructor" << endl; }
    ~Student() { cout << "Student's destructor" << endl; }
};

int main() { Student x; }
```

# Order of Construction/Destruction ..



```
Person's constructor
Address's constructor
Student's constructor
Student's destructor
Address's destructor
Person's destructor
```

## Calling Constructors of Derived Classes



```
//In_Cons1.cpp
//This works fine
# include <iostream.h>

class B {
    private: int x;
    public:
        B(): x(10){};
        void BDisplay() { cout << "x = " << x << endl;}
};
class D: public B {
    private: int y;
    public:
        D(): y(20){};                       // Default Construcor used for B
        void DDisplay() { cout << "y = " << y << endl ;}
};

main()
{ D derive;
  derive.BDisplay(); derive.DDisplay();
}
```

# Calling Constructors of Derived Classes



```
//In_Cons2.cpp
//This works fine
# include <iostream.h>

class B {
    private: int x;
    public:
        B(int x_val): x(x_val){};
        void BDisplay() { cout << "x = " << x << endl;}
};
class D: public B {
    private: int y;
    public:
        D(int x_val, int y_val ): B(x_val), y(y_val){};
        void DDisplay() { cout << "y = " << y << endl ;}
};

main()
{ D derive(10,20);
  derive.BDisplay(); derive.DDisplay();
}
```

# Calling Constructors of Derived Classes

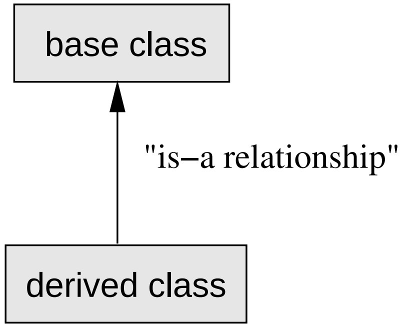


```
//In_Cons3.cpp
//This does not compile.  WHY?
# include <iostream.h>

class B {
   private: int x;
   public:
      B(int x_val): x(x_val){};
      void BDisplay() { cout << "x = " << x << endl;}
};

class D: public B {
   private: int y;
   public:
       D(){};
      void DDisplay() { cout << "y = " << y << endl ;}
};

main()
{ D derive; }
```

# Polymorphic Substitution Principle

The single most important rule in OOP with C++ is:

*Inheritance means "is a."*




If class $\mathbf{D}$ (the derived class) inherits from class $\mathbf{B}$ (the base class) $\Rightarrow$

- Every object of type $\mathbf{D}$ is also an object of type $\mathbf{B}$, but NOT *vice-versa*.
- $\mathbf{B}$ is a more general concept, $\mathbf{D}$ is a more special concept.
- Where an object of type $\mathbf{B}$ is needed, an object of type $\mathbf{D}$ can be used instead.

## Polymorphic Substitution Principle ...



In C++, using our university administration example, where **Student** is derived from **Person**, this means:

| Any Function That Expects an Argument of Type | Will Also Accepts |
|---|---|
| **Person** | **Student** |
| pointer to **Person** | pointer to **Student** |
| **Person** reference | **Student** reference |

It is also known as "Liskov Substitution Principle".

# Example: Substitution in Arguments

```
void dance(const Person &p);        // Anyone can dance
void study(const Student &s);       // Only students study

void dance(Person *p);              // Anyone can dance
void study(Student *s);             // Only students study

int main()
{
    Person p; Student s;

    dance(p); dance(s);

    study(s); study(p);

    dance(&p); dance(&s);

    study(&s); study(&p);
}
```

# Extending class hierarchies



We can easily add classes to our existing class hierarchy of **Person**, **Student**, and **Teacher**.

- New classes can immediately benefit from all functions that are available for their base classes.
- e.g. **bool** print_mailing_label(**const** Person& person) will work immediately for a new class type **Research_Scholar**, even though this type of object was unknown when **print_mailing_label()** was designed and written.
- In fact, it is not even necessary to recompile the existing code: It is enough to link the new class with the object code for **Person** and **print_mailing_label()**.

# Slicing



An assignment from derived class to base class does “slicing”. This is rarely desirable. Once slicing has happened, there is no trace of the fact that we started with a student.

```
Student student("Snoopy", "HKUST", math);
Person *pp = &student;
Person *pp2 = new Student("Mickey", "HKUST", math);

Person person;
person = student;                    // What does "person" have?
```

# Example: Name Conflicts?

```
//In_Cons4.cpp
//two different Display()
# include <iostream.h>

class B {
    private: int x;
    public:
        B(): x(10){};
        void Display() { cout << "x = " << x << endl;}
};
class D: public B {
    private: int y;
    public:
        D(): y(20){};
        void Display() { cout << "y = " << y << endl ;}
};

main()
{ D derive;
 derive.Display();
 derive.B::Display();                    //Distinguish using B::
}
```

# Example: Name Conflicts?

```
//In_Cons5.cpp
//two different x
# include <iostream.h>

class B {
    public:
        int x;
        B(): x(10){};
        void Display() { cout << "B::x = " << x << endl;}
};
class D: public B {
    public:
        int x;
        D(): x(20){};
        void Display() { cout << "D::x = " << x << endl ;}
};

main()
{ D derive;
 derive.Display();
 derive.B::Display();
}
```

# Example: Resolving Name Conflicts



Derived classes can have new, uninherited, members (data and functions) with the same name as those in the base class. These members *are totally distinct* from the ones inherited from the base class.

In cases in which this occurs, i.e., in which both base class and derived class have identically named members, the compiler defaults to choosing the derived class member. In order to override the defaults and access the base class member the member's *type* must be specified as well.

Example: If **B** is the base class, **D** the derived class and they both have an **int x** data member, then we must specify **B::x** or **d.B::x** to specify the **x** that is a base class member. The first is used inside member function definitions, the second when object **D d** is being used to access the member.

# Example: More on Name Conflicts

```
class B {
    int x, y;
  public:
    B() : x(1), y(2) { cout << "Base class constructor" << endl; }
    void f() { cout << "Base class:  " << x << " , " << y << endl; }
};

class D : public B {
    float x, y;
  public:
    D() : x(10.0), y(20.0) { cout << "Derived class constructor" << endl; }
    void f() { cout << "Derived class:  " << x << " , " << y << "\t"; B::f(); }
};

void smart(B* z) { cout << "Inside smart():  "; z->f(); }

int main()
{
    B base; B* b = &base;
    D derive; D* d = &derive;

    base.f(); derive.f();
    b = &derive; b->f();
    smart(b); smart(d);
}
```

# Example Output

```
Base class constructor
Base class constructor
Derived class constructor
Base class: 1 , 2
Derived class: 10 , 20  Base class: 1 , 2
Base class: 1 , 2
Inside smart():  Base class: 1 , 2
Inside smart():  Base class: 1 , 2
```

# Example: Design of Bird Class

```
class Bird
{
    ...
  public:
    ...
    void hatch_eggs();                      // Birds lay eggs
    void lay_egg(int n);
    void spread_wings();                    // Birds have wings
    void fly();                             // Birds can fly
    int altitude() const;                   // return current altitude
};
```

We can reuse **Bird** to implement some special cases:

```
class Swallow : public Bird { ... };
class Eagle : public Bird { public: void hunt_prey(Bird *prey); };
```

## Example: Design of Penguin Class (1)



Now we need a penguin object, and we would like to reuse all the code we have for hatching and laying eggs, spreading wings, etc.

```
class Penguin : public Bird
{
      ...
   public:
      ...
      void swim();
      void catch_fish();
};
```

Oops! Penguins cannot fly!
What can we do?

# Example: Design of Penguin Class (2)



Some people try to solve the problem like this:

```
void Penguin::fly()
{
    cerr << "Penguins cannot fly!" << endl;
    exit(999);
}
```

But this doesn't *really* say "Penguins cannot fly".

It says: "Penguins can fly, but they are forbidden!"

# Example: Design of Penguin Class (3)

Some people try to solve the problem like this:

Penguins can fly, but the altitude is zero:

```
class Penguin : public Bird
{
    ...
  public:
    ...
    void swim();
    void catch_fish();
    void fly() { }
    int altitude() const { return 0; }
};
```

## Penguin Example: What's Wrong?



Declaring **Penguin** as a derived class of **Bird** violates the substitution principle.

It is not possible to use a **Penguin** in *some* functions that work for **Bird** objects:

```
void find_food(Bird *b)
{
    b->fly();                      // visibility decreases with altitude
    double visibility = 10.0 / b->altitude();
    ...
}
```

The only solution is: REDESIGN!

# Summary

- Behavior and structure of the base class is inherited by the derived class.
- However, constructors and destructor are an exception. They are *never* inherited.
- There is a kind of contract between base class and derived class:
  - The base class provides functionality and structure (methods and data members).
  - The derived class guarantees that the base class is initialized in a consistent state by calling an appropriate constructor.
- A base class is constructed *before* the derived class.
- A base class is destructed *after* the derived class.