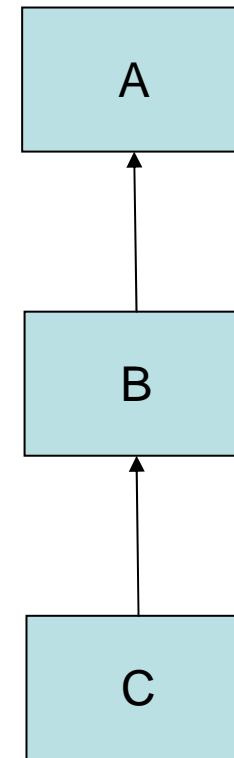


# C++

Inheritance: initialization and  
substitution principle

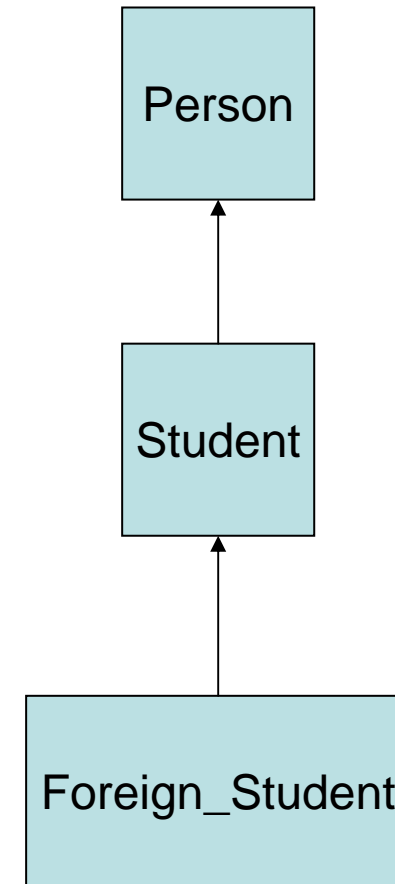
# Inheritance and initialization

- If class C inherits from B which inherits from A, then C can only call its direct base class constructors (i.e. of B)
- It is the responsibility of each derived class to initialize properly its direct base class



# Example

- Before a **Student** object can exist, its **Person** part needs to be created:
  - `Student::Student(string n, int id, int y) : Person(n), student_id(id), year(y) {}`
- Before a **Foreign\_student** object can exist, its **Student** part needs to be created:
  - `Foreign_student::Foreign_student(string n, int id, int y, string c) : Student(n, id, y), country(c) {}`
- **Foreign\_student** object does not need to create its **Person** part. It is the responsibility of **Student** to do so



# Order of construction / destruction

```
class A {  
public:  
    A() {cout << "A()" << endl;}  
    ~A() {cout << "~A()" << endl;}  
};
```

```
class B {  
public:  
    B() {cout << "B()" << endl;}  
    ~B() {cout << "~B()" << endl;}  
};
```

```
class C : public B {  
private:  
    A a;  
public:  
    C() {cout << "C()" << endl;}  
    ~C() {cout << "~C()" << endl;}  
};
```

```
int main() {  
    C c;  
}
```

Result:

```
B()  
A()  
C()  
~C()  
~A()  
~B()
```

# Initialization of base class

```
class A {  
private:  
    int a;  
public:  
    A() : a(1) {};  
    void dispa() {cout << a << endl;}  
};
```

```
class B : public A {  
private:  
    int b;  
public:  
    B() : b(2) {};  
    void dispb() {cout << b << endl;}  
};
```

```
int main(){  
    B b; b.dispa(); b.dispb();  
}
```

← Implicitly call A()

← Prints: 1 then 2

# Initialization of base class

```
class A {  
private:  
    int a;  
public:  
    A(int x) : a(x) {};  
    void dispa() {cout << a << endl;}  
};
```

```
class B : public A {  
private:  
    int b;  
public:  
    B(int x, int y) : A(x), b(y) {};  
    void dispb() {cout << b << endl;}  
};
```

```
int main(){  
    B b(1,2); b.dispa(); b.dispb();  
}
```

← explicitly call A(int x)

← Prints: 1 then 2

# Initialization of base class

```
class A {
private:
    int a;
public:
    A(int x) : a(x) {};
    void dispa() {cout << a << endl;}
};

class B : public A {
private:
    int b;
public:
    B() : b(2) {};
    void dispb() {cout << b << endl;}
};

int main(){
    B b; b.dispa(); b.dispb();
}
```

It does not compile because:  
when a B object is created the compiler will try to create an A object by calling the default constructor of A, which does not exist because we have provided a user-defined constructor for A.

# Inheritance and initialization

- Contrary to other members (data and methods), the following are not inherited:
  - Constructors (including the copy constructors)
  - Assignment operators
  - Destructors
- In the previous slides we illustrated that derived classes are in charge of calling the direct base class constructors if needed
- Same happens for copy constructors and assignment operators

# Example: Inheritance and copy constructor

```
class A {  
public:  
    int a_;  
    A(int a) : a_(a) {}  
    A(const A& an) : a_(an.a_) {}  
    ~A() {}  
};  
  
class B : public A{  
public:  
    B() : A(1) {}  
    B(const B& b) : A(b) {}  
    ~B() {}  
};
```

# Example: Inheritance and assignment operator

```
class A {  
public:  
    int a_;  
    A(int a) : a_(a) {}  
    ~A() {}  
    A& operator= (const A& an) {  
        a_ = an.a_;  
        return *this;  
    }  
};
```

```
class B : public A{  
public:  
    B() : A(1) {}  
    ~B() {}  
    B& operator= (const B& b) {  
        A::operator=(b);  
        return *this;  
    }  
};
```

Explicitly calls the  
assignment operator of the  
base class

# Substitution principle

- **Liskov** substitution principle: if class D inherits from B then:

<b>Any function accepting an arg. of type:</b>	<b>Will also accept an arg. of type:</b>
B	D
Reference to B	Reference to D
Pointer to B	Pointer to D

# Substitution: example

```
void print_name(const Person& p) {
    cout << p.get_name() << endl;
}
void print_name_ptr (Person* p) {
    cout << p->get_name() << endl;
}

// ...

Teacher t("Yamamoto", 123, "CG");
Student s("Yamamoto", 456);
print_name(t);
print_name(s);
print_name_ptr(&t);
print_name_ptr(&s);
```

# Name hiding

```
class B {
private:
    int x;
public:
    B() : x(1) { };
    void display() { cout<< "x = " << x << endl; }
};

class D : public B {
private:
    int y;
public:
    D() : y(2) { };
    void display() { cout<< "y = " << y << endl; }
};

int main() {
    D derived;
    derived.display();           // D::display() called -> prints 2
    derived.B::display();       // force to call B::display() -> prints 1
}
```

# Name hiding - 2

```
class B {
protected:
    int x;
public:
    B() : x(1) { };
    void display() { cout<< "x = " << x << endl; }
};

class D : public B {
private:
    int x;
public:
    D() : x(2) { };
    void display() { cout<< "x = " << B::x << " " << x << endl; }
};

int main() {
    D derived;
    derived.display();           // D::display() called -> prints 1 2
    derived.B::display();       // force to call B::display() -> prints 1
}
```

# Name hiding

```
class B {
public:
    B() { cout << "B()" << endl; }
    void f() { cout << "B::f()" << endl; }
};

class D : public B {
public:
    D() { cout << "D()" << endl; }
    void f() { cout << "D::f()" << endl; }
};

void g (B* b) { b->f(); }

int main() {
    B b1;
    D d1;
    b1.f(); d1.f();
    B* b2 = &d1; b2->f();
    g(&b1); g(&d1);
}
```

Prints B::f() D::f()

Prints B::f()

Prints B::f() B::f()

# Name hiding

- You can define in the derived classes new (un-inherited) members with the same name as members in the base class
  - This is called **name hiding**
- These members are distinct from the members inherited from the base class
- When this occurs, the compiler is selecting the member corresponding to the static type of the object
- In order to override this behavior and access the base class member, the base class member type needs to be specified:
  - Example 1: `d.B::display()` in main
  - Example 2: `B::x` in `D::display()`
  - Or outside of the class D, it would be `d.B::x`

# Slicing

- Slicing: an assignment from a derived class to a base class
- Example:  
    D d; B\* b = &d; // this is from the previous example
- After slicing, information is lost
  - It should be used carefully

# Summary

- Members (data and code) of a base class are inherited by its derived classes (access is defined by access control though)
- Exception is: constructor (including copy constructor), assignment operator and destructor which are never inherited
- Derived classes are in charge of properly initializing their base class (by calling an appropriate constructor)
- Base class is constructed first
- Base class is destructed last
- Substitution principle: a derived class “is-a” base class