Inheritance

Sandra Batista, Mark Redekopp, and David Kempe
Object Oriented Design Components

Encapsulation
• Combine data and operations on that data into a single unit and only expose a desired public interface and prevent modification/alteration of the implementation

Inheritance
• Creating new objects (classes) from existing ones to specify functional relationships and extend behavior

Polymorphism
• Using the same expression to support different types with different behavior for each type
Inheritance

A way of defining interfaces, reusing capabilities and extending capabilities

Allows a new class to inherit all the data members and member functions from a previously defined class

Works from more general objects to more specific objects

- Public inheritance defines an "is-a" relationship
- Square is-a rectangle is-a shape
- Square inherits from Rectangle which inherits from Shape
Derived classes inherit all data members and functions of base class

Student class inherits:
- `get_name()` and `get_id()`
- `name_` and `id_` member variables

```cpp
class Person {
    public:
        Person(string n, int ident);
        string get_name();
        int get_id();
    private:
        string name_; int id_;  
};
class Student : public Person {
    public:
        Student(string n, int ident, int mjr);
        int get_major();
        double get_gpa();
        void set_gpa(double new_gpa);
    private:
        int major_; double gpa_; 
};

int main()
{
    Student s1("Tommy", 1, 9);
    // Student has Person functionality
    // as if it was written as part of
    // Student
    cout << s1.get_name() << endl;
}
```
MEMBER FUNCTIONS AND INHERITANCE
Constructors and Inheritance

How do we initialize base class data members?

class Person {
  public:
    Person(string n, int ident);
    ...  
  private:
    string name_;  
    int id_;  
};
class Student : public Person {
  public:
    Student(string n, int ident, int mjr);
    ...  
  private:
    int major_;  
    double gpa_;  
};
Student::Student(string n, int ident, int mjr)
{
    name_ = n;  
    id_ = ident;  
    major_ = mjr;  
}
Constructors and Inheritance

Constructors are only called when a variable is created and cannot be called directly from another constructor.

To initialize base class private data members or other members:

Use constructor initialization list format instead.
Constructors

- A Derived class will automatically call its Base class constructor **BEFORE** its own constructor executes, either:
  
  * Explicitly calling a specified base class constructor in the initialization list
  
  * Implicitly calling the default base class constructor if no base class constructor is called in the initialization list

- **Constructors get called from base->derived**
Destructors

- The derived class will call the Base class destructor automatically **AFTER** its own destructor executes
- **Destructors get called from derived->base**
Constructor & Destructor Ordering

```cpp
class A {
    int a;
public:
    A() { a=0; cout << "A:" << a << endl; }
    ~A() { cout << "~A" << endl; }
    A(int mya) { a = mya; cout << "A:" << a << endl; }
};

class B : public A {
    int b;
public:
    B() { b = 0; cout << "B:" << b << endl; }
    ~B() { cout << "~B" << endl; }
    B(int myb) { b = myb; cout << "B:" << b << endl; }
};

class C : public B {
    int c;
public:
    C() { c = 0; cout << "C:" << c << endl; }
    ~C() { cout << "~C" << endl; }
    C(int myb, int myc) : B(myb) {
        c = myc;
        cout << "C:" << c << endl;
    }
};

int main() {
    cout << "Allocating a B object" << endl;
    B b1;
    cout << "Allocating 1st C object" << endl;
    C* c1 = new C;
    cout << "Allocating 2nd C object" << endl;
    C c2(4,5);
    cout << "Deleting c1 object" << endl;
    delete c1;
    cout << "Quitting" << endl;
    return 0;
}
```

Test Program

Allocating a B object
A:0
B:0
Allocating 1st C object
A:0
B:0
C:0
Allocating 2nd C object
A:0
B:4
C:5
Deleting c1 object
~C ~B ~A
Quitting
~C ~B ~A
~B ~A

Output

```
Allocating a B object
A:0
B:0
Allocating 1st C object
A:0
B:0
C:0
Allocating 2nd C object
A:0
B:4
C:5
Deleting c1 object
~C ~B ~A
Quitting
~C ~B ~A
~B ~A
```

Sample Classes
Overloading Base Functions

A derived class may overload a base member function

When derived objects call that function the derived version will be executed

When a base objects call that function the base version will be executed
We can still call the base function version by using the scope operator (::)

- `base_class_name::function_name()`

```cpp
class Car{
public:
    double compute_mpg();
private:
    string make; string model;
};
double Car::compute_mpg()
{
    if(speed > 55) return 30.0;
    else return 20.0;
}

class Hybrid : public Car {
public:
    void drive_w_battery();
    double compute_mpg();
private:
    string batteryType;
};
double Hybrid::compute_mpg()
{
    if(speed <= 15) return 45; // hybrid mode
    else return Car::compute_mpg();
}
```
ACCESS: PUBLIC, PRIVATE, PROTECTED
Private and Protected Members

Private members of a base class can not be accessed directly by a derived class member function.

Base class can declare variables with **protected** storage class which means:

- Private to any object or code not inheriting from the base.
- Accessible to any derived class.

```cpp
class Person {
    public:
        ...
    private:
        string name; int id;
};

class Student : public Person {
    public:
        void print_grade_report();
    private:
        int major; double gpa;
};

void Student::print_grade_report()
{
    cout << "Student " << name << ...        X
}
```
Derived class can access base class members using the base class' specification.

1. **Private Base Members**

Base Class

```
private:
  // members
```

X

3\textsuperscript{rd} party class or function

Derived Class

Regardless of public, protected, private inheritance
Protected Access

Derived class access base class members using the base class' specification

2. **Protected Base Members**

Base Class
- `protected:
  // members`

3rd party class or function

Derived Class
- Regardless of public, protected, private inheritance
Derived class access base class members using the base class' specification

3. **Public Base Members**

- **Base Class**
  - `public: // members`

- **3rd party class or function**

- **Derived Class**
  - *Regardless of public, protected, private inheritance*
INHERITANCE: PUBLIC, PRIVATE, PROTECTED
Public Inheritance

Public inheritance before base class indicates how the public base class members are accessed by clients and derived classes.

For **public inheritance**:

- public and protected base class members are accessible to the child class and grandchild classes
- Only public base class members are accessible to 3rd party clients

```cpp
int main(){
    Student s1("Tommy", 73412, 1);
    cout << s1.get_name() << endl; // works
}
```
Private Inheritance

Private inheritance before base class indicates how the public base class members are accessed by clients and derived classes.

For private inheritance:

- public and protected base class members are accessible to the child class.
- No base class members are accessible to grandchild classes or 3rd party clients.

```cpp
class Faculty : private Person {
    public:
    Faculty(string n, int ident, bool tnr);
    bool get_tenure();
    void print_name() {
        cout << get_name() << endl;
    }
    private:
    bool tenure;
};

class Class Visiting : public Faculty {
    public:
    Visiting(int months);
    string get_name() {
        return Faculty::get_name();
    } // will not compile!
    private:
    int duration;
};
```

```cpp
int main(){
    Faculty f1("Brian K.", 123, true);
    cout << f1.get_name() << endl;
}
```
Protected inheritance before base class indicates how the public base class members are accessed by clients and derived classes.

For protected inheritance:

- Public and protected base class members are accessible to the child class and grandchild classes.
- No base class members are accessible to 3rd parties.

```cpp
int main(){
    Student s1("Hannah", 73412, 1);
    HonorsStudent h1("Emily", 53201, 2);
    cout << s1.get_name() << endl;
    cout << h1.get_name() << endl;
}
```

```cpp
class Person {
public:
    Person(string n, int ident);
    string get_name();
    int get_id();
private: // INACCESSIBLE TO DERIVED
    string name; int id;
};

class Student : protected Person {
public:
    Student(string n, int ident, int mjr);
    int get_major();
    double get_gpa();
    void set_gpa(double new_gpa);
private:
    int major; double gpa;
};

class HonorsStudent : public Student {
public:
    HonorsStudent(string n, int ident,int mjr);
    string f1() {return get_name();}//works
private:
    bool thesis;
};
```
Public Inheritance

Base Class

```cpp
public: void f1();
protected: void f2();
private: void f3();
```

How a grandchild class or 3rd party sees what is inherited is the MORE restrictive of the how the base class declared it or how the derived class inherited.

```cpp
class ChildA :
    public Base
{ /* . . . */  };
```
Protected Inheritance

How a grandchild class or 3rd party sees what is inherited is the MORE restrictive of the how the base class declared it or how the derived class inherited.

**Base Class**
```cpp
public: void f1();
protected: void f2();
private: void f3();
```

```cpp
class ChildB :
    protected Base
{ /* . . . */  };
```

```cpp
class GCB :
    public ChildB
{ public:
    void g1()
    { f1(); f2(); f3(); } 
} ✓ ✓ X
```

```cpp
int main()
{ ChildB b;
  b.f1(); b.f2(); b.f3();
} X X X X
```
Private Inheritance

How a grandchild class or 3rd party sees what is inherited is the MORE restrictive of the how the base class declared it or how the derived class inherited.

class ChildC :
    private Base
    { /* . . . */ };

class GCC :
    public ChildC
    { public:
        void g1()
        { f1(); f2(); f3(); } }

int main()
{ ChildC c;
  c.f1(); c.f2(); c.f3();
}
Inheritance and Access Summary

Public Members
Protected Members
Private Members

All users
Derived class’s members
Class’s own members

If a base class inheritance is

1. Public: its public members can be used by all functions
2. Protected: its public and protected members can only be used by derived classes and their derived classes
3. Private: its public and protected members can only be used by the directly derived class

COMPOSITION VS. INHERITANCE
When to Inherit Privately

For protected or private inheritance, "as-a" relationship or "Is-Implemented-In-Terms-Of" (IITO)

- Queue "as-a" List / FIFO "IITO" list

```cpp
class List{
    public:
        List();
        void insert(int loc, const int& val);
        int size();
        int& get(int loc);
        void pop(int loc);
    // private data and function members
};

Base Class

class Queue : private List // or protected
{
    public:
        Queue();
        void push_back(const int& val)
        { insert(size(), val); }
        int& front();
        { return get(0); }
        void pop_front();
        { pop(0); }
};

Derived Class

Queue q1;
q1.push_back(7); q1.push_back(8);
q1.insert(0,9) // not permitted!
```
Composition

Composition defines a "has-a" relationship

- A Queue "has-a" List in its implementation

Some advise to prefer composition rather than inheritance.

Deciding between inheritance and composition requires discernment:

https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

```cpp
class List{
public:
    List();
    void insert(int loc, const int& val);
    int size();
    int& get(int loc);
    void pop(int loc);
    // private data members and functions
}

Base Class

class Queue
{ private:
    List mylist;
public:
    Queue();
    void push_back(const int& val)
    { mylist.insert(size(), val); }
    int& front();
    { return mylist.get(0); }
    void pop_front();
    { mylist.pop(0); }
    int size() // need to create wrapper
    { return mylist.size(); }
};
```
Warning: Multiple Inheritance

C++ allows multiple inheritance but it is not usually recommended

What happens for the following code?

Suppose in main()

- Liger x;
- int wt = x.getWeight();

Example source: https://www.programmerinterview.com/index.php/cplusplus/diamond-problem
Inheritance Summary

- **Public Inheritance =>** "is-a" relationship
- Public inheritance usually for subtype to develop more specialized behavior

- **Composition =>** "has-a" relationship

- **Private/Protected Inheritance =>** "as-a" relationship or "implemented-as" or "implemented-in-terms-of"

```cpp
class List{
    public:
        List();
        void insert(int loc, const int& val);
        int size();
        int& get(int loc);
        void pop(int loc);
    // private data function and members
};
```

**Base Class**

```cpp
class Queue
{
    private:
        List mylist;
    public:
        Queue();
        push_back(const int& val) { mylist.insert(size(), val); }
        int& front(); { return mylist.get(0); }
        void pop_front(); { mylist.pop(0); }
        int size() // need to create wrapper
            { return mylist.size(); }
};
```

**Queue via Composition**