OBJECT ORIENTED PROGRAMMING USING C++

Overview of C++ Polymorphism

- Two main kinds of types in C++: native and user-defined
  - User-defined types: declared classes, structs, unions (including those provided by C++ libraries)
  - Native types are “built in” to the C++ language itself: int, long, float, …
  - A typedef creates new type names for other types
- Public inheritance creates sub-types
  - Inheritance only applies to user-defined classes (and structs)
  - A publicly derived class is-a subtype of its base class
  - Liskov Substitution Principle: if S is a subtype of T, then wherever you need a T you can use an S

Overview of C++ Polymorphism

- Polymorphism depends on virtual member functions in C++
  - Base class declares a member function virtual
  - Derived class overrides the base class's definition of the function

- Private inheritance creates a form of encapsulation
  - Class form of the Adapter Pattern
  - A privately derived class wraps its base class

Public, Protected, Private Inheritance

```
class A {
public:
  int i;
protected:
  int j;
private:
  int k;
};

Class B : public A {
// ...
};
Class C : protected A {
// ...
};
Class D : private A {
// ...
};
```

- Class A declares 3 variables
  - i is **`public`** to all users of class A
  - j is **`protected.`** It can only be used by methods in class A or its derived classes
  - k is **`private.`** It can only be used by methods in class A
- Class B inherits publicly from A
  - i is again **`public`** to all users of class B
  - j is again **`protected.`** It can be used by methods in class B or its derived classes
- Class C uses protected inheritance from A
  - i is now **`protected`** in C, so the only users of class C that can access i are the methods of class C
  - j is again **`protected.`** It can be used by methods in class C or its derived classes
- Class D uses private inheritance from A
  - i and j are **`private`** in D, so users of D cannot access them, only methods of D itself

Inheritance & Constructor Ordering

```
class A {
public:
  A(int i) :m_i(i) {
    cout << "A“ << endl;}
  ~A() {cout<<"~A"<<endl;}
private:
  int m_i;
};
class B : public A {
public:
  B(int i, int j) :A(i), m_j(j) {
    cout << “B” << endl;}
  ~B() {cout << “~B” << endl;}
private:
   int m_j_;
};
int main (int, char *[]) {
   B b(2,3);
   return 0;
};
```

- Class and member construction order
  - B constructor called on object b in main
    - Passes integer values 2 and 3
  - B constructor calls A constructor
    - passes value 2 to A constructor via initializer list
  - A constructor initializes member **`m_i`**
    - with passed value 2
  - Body of A constructor runs
    - outputs “A”
  - B constructor initializes member **`m_j`**
    - with passed value 3
  - Body of B constructor runs
    - outputs “B”

Inheritance & Destructor Ordering

```
class A {
public:
  A(int i) :m_i(i) {
    cout << "A“ << endl;}
  ~A() {cout<<"~A"<<endl;}
private:
  int m_i;
};
class B : public A {
public:
  B(int i, int j) :A(i), m_j(j) {
    cout << “B” << endl;}
  ~B() {cout << “~B” << endl;}
private:
   int m_j_;
};
int main (int, char *[]) {
   B b(2,3);
   return 0;
};
```

- Class and member destructor order:
  - B destructor called on object b in main
  - Body of B destructor runs
    - outputs “~B”
  - B destructor calls member **`m_j`** “destructor”
    - **`int`** is a built-in type, so it’s a no-op
  - B destructor calls A destructor
  - Body of A destructor runs
    - outputs “~A”
  - A destructor calls member **`m_i`** destructor
    - again a no-op
- Compare dtor and ctor order
  - at the level of each class, the order of steps is reversed in ctor vs. dtor
  - ctor: base class, members, body
  - dtor: body, members, base class

Virtual Functions

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

class B : public A {
public:
  B () :A() {cout<<" B";}
  virtual ~B() {cout<<" ~B";}
};

int main (int, char *[]) {
   // prints "A B"
   A *ap = new B;

   // prints "~B ~A" : would only
   // print "~A" if non-virtual
   delete ap;

   return 0;
};
```

- Used to support polymorphism with pointers and references
- Declared virtual in a base class
- Can be overridden in derived class
  - Overriding only happens when signatures are the same
  - Otherwise it just overloads the function or operator name
- Ensures derived class function definition is resolved dynamically
  - E.g., that destructors farther down the hierarchy get called

Virtual Functions

```
class A {
public:
  void x() {cout<<"A:x";};
  virtual void y() {cout<<"A:y";};
};

class B : public A {
public:
  void x() {cout<<"B:x";};
  virtual void y() {cout<<"B:y";};
};

int main () {
   B b;
   A *ap = &b; B *bp = &b;
   b.x (); // prints "B:x"
   b.y (); // prints "B:y"
   bp->x (); // prints "B:x"
   bp->y (); // prints "B:y"
   ap.x (); // prints "A:x"
   ap.y (); // prints "B:y"
   return 0;
};
```

- Only matter with pointer or reference
  - Calls on object itself resolved statically
  - E.g., **`b.y();`**
- Look first at pointer/reference type
  - If non-virtual there, resolve statically
    - E.g., **`ap->x();`**
  - If virtual there, resolve dynamically
    - E.g., **`ap->y();`**
- Note that virtual keyword need not be repeated in derived classes
  - But it's good style to do so
- Caller can force static resolution of a virtual function via scope operator
  - E.g., **`ap->A::y();`** prints "A::y"

Pure Virtual Functions

```
class A {
public:
  virtual void x() = 0;
  virtual void y() = 0;
};

class B : public A {
public:
  virtual void x();
};

class C : public B {
public:
  virtual void y();
};

int main () {
  A * ap = new C;
  ap->x ();
  ap->y ();
  delete ap;
  return 0;
};
```

- A is an Abstract Base Class
  - Similar to an interface in Java
  - Declares pure virtual functions (**=0**)
- Derived classes override pure virtual methods
  - B overrides **`x()`**, C overrides **`y()`**
- Can't instantiate class with declared or inherited pure virtual functions
  - A and B are abstract, can create a C
- Can still have a pointer to an abstract class type
  - Useful for polymorphism

Summary: Tips on Polymorphism

- Push common code and variables up into base classes
- Use public inheritance for polymorphism
- Polymorphism depends on dynamic typing
  - Use a base-class pointer or reference if you want polymorphism
  - Use virtual member functions for dynamic overriding
- Use private inheritance only for encapsulation
- Use abstract base classes to declare interfaces
- Even though you don’t have to, label each virtual method (and pure virtual method) in derived classes