CS32 Summer 2013

Object-Oriented Programming in C++

Inheritance

Victor Amelkin
August 29, 2013
Plan for Today

- Inheritance: State, Implementation, Interface
- Accessing Base
- Construction and Destruction
- Slicing
- Polymorphism and Virtual Functions

Next time:
- Alternatives to Virtual Functions, RTTI
- Abstract Classes
- Multiple Inheritance and Class Hierarchies
- ...
Inheritance of State

- Classes may share a great deal of their internals (their state, in particular)

```cpp
class HighSchoolStudent {
private:
    char *_full_name;
    time_t _dob;
    char *ssn;
};

class UniversityStudent {
private:
    char *_full_name;
    time_t _dob;
    char *ssn;
    char *perm;
    char *major;
    int advisor_id;
};
```

- Can we describe the part classes share only once?
Inheritance of State

• Solution: derive one class from another

```cpp
class HighSchoolStudent {
private:
    char * _full_name;
    time_t _dob;
    char * _ssn;
};

class UniversityStudent : public HighSchoolStudent {
private:
    char * _perm;
    char * _major;
    int advisor_id;
};
```

– “base class for UniversityStudent”

– “class derived from HighSchoolStudent”
Inheritance of State

- Solution: *derive* one class from another

```cpp
class HighSchoolStudent {
private:
    char *_full_name;
    time_t _dob;
    char *_ssn;
};

class UniversityStudent : public HighSchoolStudent {
private:
    char *_perm;
    char * _major;
    int advisor_id;
};
```

<table>
<thead>
<tr>
<th>HighSchoolStudent</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_full_name</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_dob</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_ssn</td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UniversityStudent</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>_full_name</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_dob</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_ssn</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_perm</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_major</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>_advisor_id</td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>
Accessing Base State

- Derived class cannot access private members of its base class

```cpp
class HighSchoolStudent {
private:
    char * _full_name;
    time_t _dob;
    char * _ssn;
public:
    void print() const;
};

class UniversityStudent : public HighSchoolStudent {
private:
    char * _perm;
    char * _major;
    int advisor_id;
public:
    void mymethod() {
        print(); // ok; member print() is public
        _dob = 12345; // error; _dob is private
    }
};
```
Accessing Base State

- Making private members public is a bad idea
- We can make a member protected:

```cpp
class HighSchoolStudent {
private:
  char * _full_name;
  char * _ssn;
protected:
  time_t _dob; // – still not accessible from “the outside”
public:
  void print() const;
};

class UniversityStudent : public HighSchoolStudent {
private:
  char * _perm;
  char * _major;
  int advisor_id;
public:
  void mymethod() {
    _dob = 12345; // ok; _dob is protected
  }
};
```
Accessing Base State

- **Protected** is better than **public** (but not very much)

- Rules for choosing access specifiers:
  - Never give direct access to class' state to anyone; *fields* should always be **private**
  - If you provide public getter and setter for a field, it is not always the same as making such a field **public**
  - If “the outside” needs to access class' internals, provide a **public method**
  - If a derived class needs to access class' internals, provide a **protected method**
  - Never make anything (fields or methods) **public** if it can live fine as **private** (– best) or **protected**

- Be conservative!
Accessing Base State

class HighSchoolStudent {
private:
    char * _full_name;
    time_t dob;
    char * ssn;
protected:
    void set_dob(time_t dob) { ... }
public:
    time_t get_dob() const { ... }
};

class UniversityStudent : public HighSchoolStudent {
private:
    char * _perm;
    char * _major;
    int advisor_id;
public:
    void mymethod() {
        set_dob(12345); // ok; set_dob(time_t) is protected
    }
};

UniversityStudent st;
time_t dob = st.get_dob(); // ok; get_dob() is public
st.set_dob(333); // error; set_dob(time_t) is protected
Inheritance of Implementation

- Behavior ("implementation") is also inherited

```cpp
class HighSchoolStudent {
private:
    time_t dob;
public:
    time_t get_dob() const { ... }
};

class UniversityStudent : public HighSchoolStudent {
    ...
};

HighSchoolStudent s1;
UniversityStudent s2;
bool same_dob = s1.get_dob() == s2.get_dob();
```
Inheritance of Interface

- Objects of derived classes can be treated as objects of base classes

```cpp
class HighSchoolStudent {
public:
    time_t get_dob() const { ... }
};

class UniversityStudent : public HighSchoolStudent {
    ...
};

// pstud will point to an object of class UniversityStudent
HighSchoolStudent *pstud = new UniversityStudent(...);

pstud->get_dob();
```
Public/Protected/Private Inheritance

- Types of inheritance differ in how access specifiers are inherited

```cpp
class Base {
    private: int private_member;
    protected: int protected_member();
    public: int public_member();
};

class DerivedPublic : public Base {
    // private_member is not accessible here
    // protected_member() is protected here
    // public_member() is public here
};

class DerivedProtected : protected Base {
    // private_member is not accessible here
    // protected_member() is protected here
    // public_member() is protected here
};

class DerivedPrivate : private Base {
    // private_member is not accessible here
    // protected_member() is private here
    // public_member() is private here
};
```
Inheritance and Construction

Bjarne Stroustrup draws his class diagrams with derived classes above and base classes below. Hence the name “bottom-up” construction.
Inheritance and Construction


class Class1 {
public:
    Class1() { cout << "Class1 default ctor called.\n"; }  
    Class1(int i) {
        cout << "Class1 ctor(int " << i << ") called.\n";
    }
};

class Class2 : public Class1 {
public:
    Class2() { cout << "Class2 default ctor called.\n"; }  
    Class2(char c) {
        cout << "Class2 ctor(char " << c << ") called.\n";
    }
};

class Class3 : public Class2 {
public:
    Class3() : Class2('x') {
        cout << "Class3 default ctor is called.\n";
    }
};
Inheritance and Construction

```cpp
class Class1 {
public:
    Class1() { cout << "Class1 default ctor called.\n"; }  
    Class1(int i) {
        cout << "Class1 ctor(int " << i << ") called.\n";
    }
};

class Class2 : public Class1 {
public:
    Class2() { cout << "Class2 default ctor called.\n"; }  
    Class2(char c) {
        cout << "Class2 ctor(char '" << c << ") called.\n";
    }
};

class Class3 : public Class2 {
public:
    Class3() : Class2('x') {
        cout << "Class3 default ctor is called.\n";
    }
};
```

Class1 obj;
>> Class1 default ctor is called.
Inheritance and Construction

```cpp
Class1 obj2(123);
>> Class1 ctor(int 123) is called.

class Class1 {
public:
    Class1() { cout << "Class1 default ctor called.\n"; }
    Class1(int i) {
        cout << "Class1 ctor(int " << i << ") called.\n";
    }
};

class Class2 : public Class1 {
public:
    Class2() { cout << "Class2 default ctor called.\n"; }
    Class2(char c) {
        cout << "Class2 ctor(char " << c << ") called.\n";
    }
};

class Class3 : public Class2 {
public:
    Class3() : Class2('x') {
        cout << "Class3 default ctor is called.\n";
    }
};
```
Inheritance and Construction

```cpp
class Class1 {
public:
  Class1() { cout << "Class1 default ctor called.\n"; }
  Class1(int i) {
    cout << "Class1 ctor(int " << i << ") called.\n";
  }
};

class Class2 : public Class1 {
public:
  Class2() { cout << "Class2 default ctor called.\n"; }
  Class2(char c) {
    cout << "Class2 ctor(char '" << c << ") called.\n";
  }
};

class Class3 : public Class2 {
public:
  Class3() : Class2('x') {
    cout << "Class3 default ctor is called.\n";
  }
};
```

Class2 obj3;
>> Class1 default ctor is called.
>> Class2 default ctor is called.
Inheritance and Construction

```cpp
Class2 obj4('z');
>> Class1 default ctor is called.
>> Class2 ctor(char 'z') is called.

class Class1 {
public:
    Class1() { cout << "Class1 default ctor called.\n"; }
    Class1(int i) {
        cout << "Class1 ctor(int " << i << ") called.\n";
    }
};

class Class2 : public Class1 {
public:
    Class2() { cout << "Class2 default ctor called.\n"; }
    Class2(char c) {
        cout << "Class2 ctor(char " << c << ") called.\n";
    }
};

class Class3 : public Class2 {
public:
    Class3() : Class2('x') {
        cout << "Class3 default ctor is called.\n";
    }
};
```
Inheritance and Construction

```cpp
Class3 obj5;
>> Class1 default ctor is called.
>> Class2 ctor(char 'x') is called.
>> Class3 default ctor is called.

class Class1 {
public:
    Class1() { cout << "Class1 default ctor called.\n"; }
    Class1(int i) {
        cout << "Class1 ctor(int " << i << ") called.\n";
    }
};

class Class2 : public Class1 {
public:
    Class2() { cout << "Class2 default ctor called.\n"; }
    Class2(char c) {
        cout << "Class2 ctor(char '" << c << ") called.\n";
    }
};

class Class3 : public Class2 {
public:
    Class3() : Class2('x') {
        cout << "Class3 default ctor is called.\n";
    }
};
```
Inheritance and Destruction

Bjarne Stroustrup draws his class diagrams with derived classes above and base classes below. Hence the name “top-down” destruction.
Inheritance and Destruction


```cpp
class Class1 {
    public:
        ~Class1() { cout << "Class1 destructor called.\n"; }
};

class Class2 : public Class1 {
    public:
        ~Class2() { cout << "Class2 destructor called.\n"; }
};

class Class3 : public Class2 {
    public:
        ~Class3() { cout << "Class3 destructor called.\n"; }
};
```
Inheritance and Destruction

```cpp
Class1 *pobj = new Class1; delete pobj;
>> Class1 destructor called.

class Class1 {
public:
    ~Class1() { cout << "Class1 destructor called.\n"; }
};

class Class2 : public Class1 {
public:
    ~Class2() { cout << "Class2 destructor called.\n"; }
};

class Class3 : public Class2 {
public:
    ~Class3() { cout << "Class3 destructor called.\n"; }
};
```
Inheritance and Destruction

Class2 *pobj = new Class2; delete pobj;
>> Class2 destructor called.
>> Class1 destructor called.

class Class1 {
public:
    ~Class1() { cout << "Class1 destructor called.\n"; } 
};

class Class2 : public Class1 {
public:
    ~Class2() { cout << "Class2 destructor called.\n"; } 
};

class Class3 : public Class2 {
public:
    ~Class3() { cout << "Class3 destructor called.\n"; } 
};
Inheritance and Destruction

Class3 *pobj = new Class3; delete pobj;
>> Class3 destructor called.
>> Class2 destructor called.
>> Class1 destructor called.

class Class1 {
public:
    ~Class1() { cout << "Class1 destructor called.\n"; } }
};

class Class2 : public Class1 {
public:
    ~Class2() { cout << "Class2 destructor called.\n"; } }
};

class Class3 : public Class2 {
public:
    ~Class3() { cout << "Class3 destructor called.\n"; } }
};
Inheritance and Destruction

```cpp
Class1 *pobj = new Class3; delete pobj;
>> Class1 destructor called. // see virtual dtor slide

class Class1 {
public:
    ~Class1() { cout << "Class1 destructor called.\n"; }
};

class Class2 : public Class1 {
public:
    ~Class2() { cout << "Class2 destructor called.\n"; }
};

class Class3 : public Class2 {
public:
    ~Class3() { cout << "Class3 destructor called.\n"; }
};
```
Intermezzo: Initialization Lists vs. Assignment


- Before executing ctor's body, object's fields get initialized

```cpp
class Value {
private:
  int state;
public:
  Value() : state(0) { ... }
  Value(int i) : state(i) { ... }
  Value(const Value& other) : state(other.state) { ... }
  Value& operator=(const Value &other) { ... }
};

class MyClass {
private:
  Value val;
public:
  MyClass(const Value &v) { val = v; }
  MyClass(const Value &v, int dummy) : val(v) { }
  // need dummy since 2'nd ctor must have different signature
};
```
Intermezzo: Initialization Lists vs. Assignment

Value v(123);
>> Value's ctor(int 123) is called.

MyClass obj1(v);
>> Value's default ctor is called.
>> Entered MyClass ctor (assignment version).
>> Value's op=(Value{state=123}) is called.

MyClass obj2(v, 0);
>> Value's ctor(int 123) is called.
>> Entered MyClass ctor (initialization list version).

class MyClass {
private:
    Value val;
public:
    MyClass(const Value &v) { cout ...; val = v; }
    MyClass(const Value &v, int dummy) : val(v) { cout ...; }
};
Inheritance of Overloads (Lack of)

- Method overloading does not work across scopes

```cpp
#include <iostream>

class Base {
public:
    int doit(int n);
};

class Derived {
public:
    int doit(double d); // overloading doit
};

Derived obj;
obj.doit(1); // Derived::doit(double) is called

Base *pobj = &obj;
pobj->doit(1); // Base::doit(int) is called
```
Inheritance of Overloads (Lack of)

- We can explicitly “invite” overloads to the new scope

```cpp
class Base {
public:
    int doit(int n);
};

class Derived : public Base {
public:
    using Base::doit; // import all overloads of doit
    int doit(double d); // overloading doit
};

Derived obj;
obj.doit(1); // Derived::doit(int) is called

Base *pobj = &obj;
pobj->doit(1); // Derived::doit(int) is called
```

Inheritance of Constructors (Lack of)

- Constructors are also not inherited
- We can “invite” constructors from the base like it has been done with overloads

```cpp
class Base { 
public:
    Base(int n) { } 
};

class Derived : public Base { 
public:
    using Base::Base; // imports ctor(int n) 
};
```
Slicing

- Objects of derived classes can be treated as objects of base classes if used through pointers or references

```cpp
class HighSchoolStudent { ... }
class UniversityStudent : public HighSchoolStudent { ... };

HighSchoolStudent *pstud = new UniversityStudent(...);
```

<table>
<thead>
<tr>
<th>UniversityStudent</th>
<th>_full_name</th>
<th>(4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_dob</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td></td>
<td>_ssn</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td></td>
<td>_perm</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td></td>
<td>_major</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td></td>
<td>_advisor_id</td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>
Slicing

- Objects of derived classes can be treated as objects of base classes if used through pointers or references

```cpp
class HighSchoolStudent { ... }
class UniversityStudent : public HighSchoolStudent { ... };

HighSchoolStudent &stud = univ_student;
```
Slicing

- Not using pointers or references, assignment results in *slicing*

```cpp
class HighSchoolStudent { ... };
class UniversityStudent : public HighSchoolStudent { ... };

UniversityStudent ustud;
HighSchoolStudent hstud1(ustud); // ustud gets sliced
HighSchoolStudent hstud2 = ustud; // ustud gets sliced

Either copy ctor or assignment operator are used to initialize fields of hstud1 with field values from ustud.
```
Polymorphism and Virtual Functions

- Inheritance prevents code duplication
- Common functionality is defined in the base class
- Then, it is inherited by derived classes
- What if an inherited method needs to be redefined (“overridden”) in a derived class?

```cpp
class Class1 { void doit() { } };
class Class2 : public Class1 { ... };
class Class3 : public Class2 {
    // void doit() has been inherited from Class1
    // want to override void doit() for Class3
};
```
Polymorphism and Virtual Functions

- Why not to simply define method doit() in Class3?

```cpp
class Class1 {
    void doit() { cout << "hello from Class1"; }
};

class Class2 : public Class1 { ... };

class Class3 : public Class2 {
    void doit() { cout << "hello from Class3"; }
};

Class1 obj1; obj1.doit();
>> hello from Class1
Class2 obj2; obj2.doit();
>> hello from Class1
Class3 obj3; obj3.doit();
>> hello from Class3

Class1 *pobj = &obj3; pobj->doit();
>> hello from Class1
```

Polymorphism and Virtual Functions

- We may want to have a pointer/reference of base type pointing to an object of a derived type

```cpp
class Shape {
public:
    void draw() {
        /* do nothing; do not know what to draw */
    }
};

class Triangle : public Shape { ...draw() is redefined... };
class Sphere : public Shape { ...draw() is redefined... };
class Rect : public Shape { ...draw() is redefined... };

void drawShape(Shape *pshape) {
    // pshape can point to any shape (Triangle, Sphere, Rect)
    pshape->draw();
}

drawShape(&my_triangle_obj); // should draw a triangle
drawShape(&my_sphere_obj); // should draw a sphere
```

- “Polymorphism”: pshape can take many forms (sphere, triangle, ...)
Polymorphism and Virtual Functions

- In C++, polymorphism is implemented through virtual functions (aka virtual methods)

```cpp
class Shape {
public:
    virtual void draw() { }
};

class Triangle : public Shape {
    void draw() { ... draw triangle ... }
};

class Sphere : public Shape {
    void draw() { ... draw sphere ... }
};

Triangle triangle; Sphere sphere;

Shape shape1(triangle); shape1.draw();
>> Nothing is drawn (Shape's draw() is called)

Shape &shape2 = sphere; shape2.draw();
>> Circle is drawn (Circle's draw() is called)

Shape *pshape3 = &triangle; pshape3->draw();
>> Triangle is drawn (Triangle's draw() is called)
```

Polymorphism and Virtual Functions

- Which *regular* function is called depends on the *declared type* of the variable

```cpp
Base obj(my_derived_obj); // my_derived_obj is sliced
obj.regular_method(); // Base's method is called
```

- Which *virtual* function is called depends on *the actual type of the object* a pointer/reference points to

```cpp
Base *pobj = new Derived();
pobj->virtual_method(); // Derived's method is called // (it is overridden in Derived)
```

- If a function is declared *virtual*, it is virtual in all derived classes

- In C++11, we can mark overridden virtual functions with *override*

```cpp
class Shape { public: virtual void draw() { } };

class Triangle : public Shape {
    // the reader sees that draw has been decl'ed virtual
    void draw() override { ... draw triangle ... }
};
```
Virtual Functions: Calling Base

- Virtual methods can call other methods
- In particular, they can call their base implementations

```cpp
class Base {
public:
    virtual int doit() { cout << "Base::doit()\n"; } 
};

class Derived : public Base {
public:
    int doit() {
        cout << "Entered Derived::doit()\n";
        Base::doit();
        cout << "Leaving Derived::doit()\n";
    }
};

Derived derived;
Base *pbase = &derived;
pbase->doit();

>> Entered Derived::doit()
>> Base::doit()
>> Leaving Derived::doit()
Virtual Destructor

- Destructors are methods
- A non-virtual destructor, like any other method, will not be called through a pointer/reference to a base class

```cpp
class Base {  
public:  
  ~Base() { cout << "Base::~Base()\n"; }  
};

class Derived : public Base {  
public:  
  ~Derived() { cout << "Derived::~Derived()\n"; }  
};

Base *pobj = new Derived();
delete pobj; // only Base::~Base() is called
```
Virtual Destructor

- If a chain of destructors should be called (like on the slide with top-down destruction) when operating on pointers/references, destructor needs to be virtual.

```cpp
class Base {
public:
    virtual ~Base() { cout << "Base::~Base()\n"; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "Derived::~Derived()\n"; }
};

Base *pobj = new Derived();
delete pobj;
>> Derived::~Derived()
>> Base::~Base()
```
Bypass of Dynamic Dispatch

• Calling a virtual method through a pointer/reference will be done by the means of *dynamic dispatch* – which implementation to call will be chosen automatically based on the pointed object.

• If needed, *static dispatch* can be enforced by the means of *explicit qualification*, thereby, allowing to call any implementation of a (virtual) method.

```cpp
class Base {
public:
    virtual void doit() { cout << "Base::doit()\n"; }
};

class Derived : public Base {
public:
    void doit() { cout << "Derived::doit()\n"; }
};

Derived obj;
Base *pobj = &obj;
pobj->doit(); // calling Derived::doit() (dynamic dispatch)
pobj->Base::doit(); // calling Base::doit() (static dispatch)
```
Cost of Polymorphism

- Objects of a class that has virtual functions contain a pointer to the **table of virtual functions** (aka **vtbl**) – this is how *dynamic dispatch* knows which implementation to call.

---

**Class Shape**

| int xpos;  
| int ypos; |
| virtual void draw();  
| virtual void resize();  
| void erase(); |

**Class Circle : public Shape**

| int radius; |
| (virtual) void draw();  
| (virtual) void resize(); |

---

**Object of class Circle**

| pointer to vtbl |
| int xpos;  
| int ypos;  
| int radius; |

**vtbl of class Shape**

| draw → Shape::draw()  
| resize → Shape::resize() |

**vtbl of class Circle**

| draw → Circle::draw()  
| resize → Circle::resize() |
Calling Virtual Functions from Ctors/Dtors

- Dynamic dispatch does not work as usually inside constructors/destructors
- Instead of the implementation from the “most derived” class, the current or the closest base implementation is called (– feature; not a bug)
- To avoid problems, you may want to refrain from calling virtual functions from ctors/dtors
~ The End ~

(To be continued…)