Polymorphism and Virtual Functions

Chapter 11

Implementing Inheritance

And accessing member data

class foo
{
    private:
    int a;
    int b;
    int c;
};

class bar : public foo
{
    private:
    int d;
    int e;
    int f;
};

bar* B = new bar;

Subobject Initialization

Constructor calls

Initialization
  * Inheritance: use class name
  * Composition: use object (attribute) name
  * Built-in: use variable (attribute) name

class foobar : public foo
{
    private:
    bar B;
    int C;

    public:
    foobar(int a, int b, int c) : foo(a), B(b), C(c) {}
};

Overriding Inherited Functions

A bar "is like a" foo – a simile

class foo
{
    public:
    void func1( );
    void func2( );
};

bar B;

B.func1( ); // bar::func1 – overrides foo
B.func2( ); // foo::func2
B.foo::func1(); // foo::func1

class bar : public foo
{
    public:
    void func1( ) {
        foo::func1( );
    }
};

Upcasting

Allowed and disallowed type conversions

Parent Class

Child Class

Safe

Explicit cast

Conversions

class Parent {}

class Child {}

Child Parent* Cptr = &C;
// safe upcast
Child* Child* Cptr = (Child*) Pptr;
// risky down cast

Subobject Location and Casting

Compare with slide 2

bar object

foo object

foo

bar

e = B + sizeof(foo) + d
what is
B1 + sizeof(foo) + d?
Member Functions and Casting

Compile-time binding

```cpp
class foo
{
    public:
    void func();
};

bar B;
bar* barPointer = &B;
foo* fooPointer = &B; // up cast
barPointer->func(); // bar::func
fooPointer->func(); // foo::func
```

Early Binding

Static or compile-time binding

- Compiler identifies the correct function and binds (i.e., generates a “direct call”)
- Applies to
  - “Ordinary functions”
  - Member functions
  - Overloaded functions (member and nonmember)
  - Functions associated with embedded objects
  - Functions belonging to the objects and sub-objects in an inheritance hierarchy

The Early Binding Problem

Dynamic binding

```
Shape* S;

S->draw(); // Which draw method is called?
```

```
// For example – the result of early binding
Circle C;
Shape* S = &C;
S->draw(); // Shape::draw
```

Solution?

The old, functional way

- Add an enum attribute that identifies each object
  ```cpp
  enum { Shape, Circle, Rectangle, Triangle };```
- Use a switch statement to call the correct draw function
  ```cpp
  switch (shape)
  {
      case Shape: ...
      case Circle: ...
      case Rectangle: ...
      case Triangle: ...
  }
  ```
- Adding a new Ellipse shape requires “touching”
  - The enum
  - Every location where the draw function is called (add a case)

Polymorphism

“Many shapes”

- The ability of similar objects to respond differently to the same message
- Selecting the correct function is deferred until runtime and is based on the object
- Implements extensibility and dynamic code reuse
- Requires
  - Inheritance (direct inheritance or common ancestor)
  - Address variables (pointers or references)
  - Upcast to common ancestor
- Activated in C++ with the `virtual` keyword

Late Binding

Run-time or dynamic binding, also dynamic dispatch

- Select the correct function only after the object type is known
- Implements the concept of polymorphism
- Polymorphic functions are declared virtual in the base class
  - Optionally may be declared virtual in base classes (they are virtual even if the “virtual” keyword is not specified)
- virtual functions “expect” to be overridden
Polymorphic Solution

The new way of thinking

```cpp
class Shape {
    public:
        ...  
        virtual void draw();
        
    Circle C;
    Shape* S = &C;
    S->draw(); //Circle::draw
```

Polymorphism and Code Reusability

One benefit

- Adding a new Ellipse shape now requires
  - Declaring the Ellipse class, derived from Shape
  - Defining the member functions and compiling them to object code
  - Re-linking the program
- Does not require changing the function calls
- No existing source code is copied or "soiled" (source may not even be available)

Determining Which Method Is Called

Polymorphism vs non-polymorphism

- **Shape** S = new **Circle**;
- void render(Shape* S);
  - render(new Shape);
  - render(new Circle);
- Non-polymorphic call
  - Function belongs to the class named on the left hand side of the assignment operator
- Polymorphic call
  - Function belongs to the instantiated class on the right hand side of the assignment operator
- Example
  - Non-polymorphic: S->draw() // Shape draw function
  - Polymorphic: S->draw() // Circle draw function

Quick Quiz

Using polymorphism

Which functions are called?

```cpp
int main() {
    class Parent {
        public:
            virtual void funcA();
            virtual void funcB();
            void funcC();
    };
    class Child : public Parent {
        public:
            virtual void funcA();
            virtual void funcB();
            void funcC();
    };
    Parent* P1 = new Parent;
    Parent* P2 = new Child;
    Child* C = new Child;
    P1->funcA(); //_______________
    P1->funcB(); //_______________
    P1->funcC(); //_______________
    P2->funcA(); //_______________
    P2->funcB(); //_______________
    P2->funcC(); //_______________
    C->funcA(); //_______________
    C->funcB(); //_______________
    C->funcC(); //_______________
}
```

Implementing Polymorphism

Done by the compiler

- Addresses of virtual functions put in VTABLE
- One VTABLE per class
- Every object contains a "secret" pointer to the classes' VTABLE
- VPTR is initialized by the constructor

Polymorphism and Constructors

The consequences

- Compiler adds to user-defined constructor or creates a default constructor to initialize VPTR
- Constructors called from the base to most derived class; VPTR updated by each constructor
- Constructors only call local versions of virtual functions
  - Only base-class objects initialized
  - VPTR may not point to the correct VTABLE yet
- Constructors cannot be virtual
Polymorphism and Destructors

The consequences

- Destructors called from most derived to base
- Destructors can be virtual
- If a class has virtual functions, it should have a virtual destructor
- Example
  - Shape* figure = new Circle;
  - delete figure;
- If the destructor is pure virtual (see next slide), it must also have a body
- Destructors only call local versions of virtual functions

Cost of Polymorphism

No free lunches

- VTABLE for every polymorphic class
- VPTR embedded in every polymorphic object
- Indirect function access
- virtual functions cannot be inline
  - virtual and inline keywords are incompatible
  - Functions created inline as part of the class declaration are "flushed" to memory
- Polymorphism is optional in C++

Abstract Base Classes

What does it mean to draw a Shape – what does a Shape look like?

- Abstract class
  - Has one or more pure virtual functions
    - class Shape
      - public:
        - virtual void draw() = 0;
    -

- Concrete classes
  - Inherit Shape position and color
  - Inherit Shape move
  - Must override all pure virtual functions or become abstract

Abstract Classes

Classes that cannot be instantiated

- Concrete classes may be instantiated
- Abstract classes cannot be instantiated
- They only make sense in the context of inheritance
  - Organize features common to many classes
  - Declare an operation (protocol, interface) that each subclass must provide
  - The origin class is the topmost defining class; it defines the protocol
- Abstract (pure virtual) functions must be overridden in concrete classes
- Some abstract classes appear naturally in the problem domain; others are abstractions artificially introduced for code reuse or from implementation requirements

The Copy Constructor

Replacing the compiler-provided default copy constructor

- Copy constructor is used to pass and return objects
  - C++ default copy constructor does pass and return by value (i.e., bit-wise copy)
- Bit-wise copy is easy but has two potential problems
  - Time consuming to copy large objects
  - Generally does not handle pointers correctly
- Big classes:
  - class Big { ...; };
  - Big function(Big var) { ...; return var; }
  - Big a, b;
  a = function(b);      // copies b to var and var to a

Definition of Copy?

What does it mean to copy an object?

- class Employee
  - private:
    - Date hire;
  -
  - class Employee
    - private:
      - Date* hire;
    -
  - 1996/09/03
Overloading the Copy Constructor

Defining the meaning of copy

- Copy constructor must copy all data members

```cpp
class Employee
{
private:
    double salary;
    Date* hire;
    Address* addr;
public:
    Employee(Employee& old)
    {
        salary = old.salary;
        hire = new Date(old.hire);
        addr = new Address(old.addr);
    }
};
```

The Assignment Operator

Replacing the compiler-provided default assignment operator

- Always created for every class
  - The default does a bit-wise copy for each data member
  - When overloaded by the user, the user must define what it means to assign one object to another
- Like the copy constructor, except
  - The destination (left hand side) may have a "value" that must be discarded prior to the new assignment
  - Should test for assigning an object to itself (i.e., A = A;)
  - Can have a return value (NOTE: the return value is irrelevant to the assignment operation and is only needed for chaining assignment: A = B = C;
  - operator= cannot be overloaded outside of a class

Overloading operator=  
Replacing the default assignment operator

```cpp
class Employee
{
private:
    double salary;
    Date* hire;
    Address* addr;
public:
    Employee& operator=(Employee& rhs)
    {
        if (&rhs == this) return *this; // for A = A.
        if (hire != 0) delete hire; // A held an object.
        if (addr != 0) delete addr; // ditto.
        salary = old.salary; // start new
        hire = new Date(old.hire); // assignment.
        addr = new Address(old.addr);
    }
};
```

Quick Quiz

What is an appropriate return value type?

- The overloaded assignment operator returns a reference and has the general form:
  - X& X::operator=(X&)
- operator+ does not return a reference (which is common for arithmetic operators) and has the general form:
  - X X::operator+(X&)
- Why does the assignment operator return a reference while the arithmetic operators return an object?
  - Hint: consider the operands
  - which one exists before the call and one is created by the function?
  - remember that A = B is equivalent operator=(A, B)

Miscellaneous operator= Facts

For the "bar know-it-all"

- The return type of operator= is only used for embedding assignment statements in larger statements:
  - A = B = C is equivalent to operator=(A, operator=(B, C))
  - The expression is executed as A = (B = C); the operation B = C returns the address of B, which becomes the argument to the next assignment operation
- Distinguishing the copy constructor from the assignment operator is not always as straightforward as you might think:
  - foo A;
  - foo B = A; // copy constructor
  - B = A; // assignment operator

Other Cases of Return-By-Reference

Increasing flexibility

```cpp
class Stack
{
private:
    int stackmem[100];
public:
    int& operator[](int index)
    {
        return stackmem[index];
    }
};
Stack S;
S[5] = 10; // can be an lvalue
S[0] = S[5]; // can be an rvalue
```