Modern C++ Programming

7. C++ Object Oriented Programming II

Federico Busato

University of Verona, Dept. of Computer Science

2019, v2.01

Agenda

Polymorphism

- Function binding
- `virtual` method
- `override`/`final` keywords
- `virtual` - common errors
- Pure virtual methods
- Abstract class and interface

- **Operator Overloading**
  - Operator $\ll$
  - Operator `operator()`
  - Operator `operator=`
- **Special Objects**
  - Aggregate
  - Trivial class
  - Standard-layout class
  - Plain old data type

Polymorphism

Polymorphism

In object-oriented programming, **polymorphism** (meaning "having multiple forms") is the capability of an object of *mutating* its behavior in accordance with the specific usage *context*

- At run-time, objects of a *derived class* may be treated as objects of a *base class*
- **Base** classes may define and implement polymorphic (`virtual`) methods, and **derived** classes can `override` them, which means they provide their own implementations which are invoked at run-time depending on the context

Overloading is a form of static polymorphism (compile-time polymorphism)
In C++ the term *polymorphic* is strongly associated with dynamic polymorphism (overriding)

Polymorphism (the problem)

```cpp
struct A {
    void f()  { std::cout << "A"; }
};

struct B : A { // B extends A (B does something more than A)
    void f() { std::cout << "B"; }
};

void g(A& a) { a.f(); } // accepts A and B

void h(B& b) { b.f(); } // accepts only B

int main() {
    A a; B b;
    g(a);    // print "A"
    g(b);    // print "A" not "B"!!!
//  h(a);    // compile error!!
    h(b);    // print "B"
}
```

Function Binding

Connecting the function call to the function body is called *Binding*

- In **Early Binding** or *Static Binding* or *Compile-time Binding*, the compiler identifies the type of object at <u>compile-time</u>
- In **Late Binding** or *Dynamic Binding* or *Run-time binding*, the compiler identifies the type of object at <u>run-time</u> and *then* matches the function call with the correct function definition

In C++ **late binding** can be can be achieved by declaring a `virtual` function

- *Early binding*: the program can jump directly to the function address
- *Late binding*: the program has to read the address held in the pointer and then jump to that address (less efficient since it involves an extra level of indirection)

Polymorphism (`virtual` method)

```cpp
struct A {
    virtual void f() { std::cout << "A"; }
}; // now "f()" is virtual, evaluated at run-time

struct B : A { // B extends A (B does something more than A)
    void f() { std::cout << "B"; }
}; // now "B::f()" override "A::f()", evaluated at run-time

void g(A& a) { a.f(); } // accepts A and B

void h(B& b) { b.f(); } // accepts only B

int main() {
    A a; B b;
    g(a);    // print "A"
    g(b);    // NOW, print "B"!!!
    h(b);    // print "B"
}
```

When `virtual` works

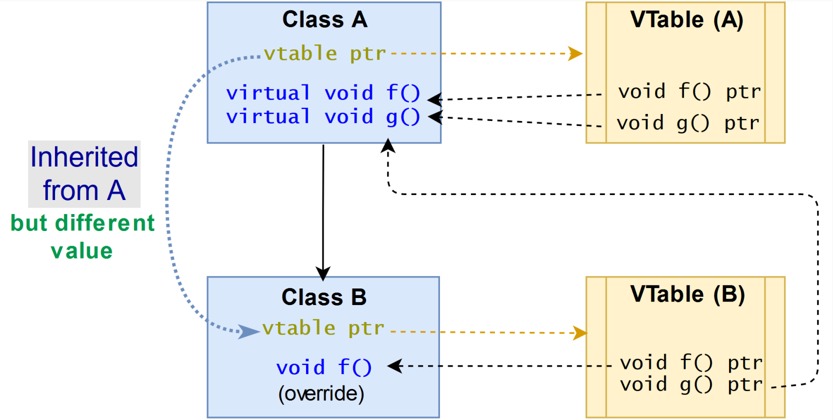
```cpp
struct A {
    virtual void f() { std::cout << "A"; }
    virtual void g() {} // see next slide
};
struct B : A {
    void f() { std::cout << "B"; }
};
void g(A  a) { a.f();  }
void h(A& a) { a.f();  }
void p(A* a) { a->f(); }

int main() {
    A a; B b;
    a.f();          // print "A"
    b.f();          // print "B"
    A* ax1 = &b;    // memory address conversion
    ax1->f();       // print "B"
    g(b);           // print "A" (cast to A)
    h(b);           // print "B"
    p(&b);          // print "B"
}
```

vtable

The **virtual table** (`vtable`) is a lookup table of functions used to resolve function calls and support *dynamic dispatch* (late binding)

A *virtual table* contains one entry for each `virtual` function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the *most-derived* function accessible by that class

The compiler adds a *hidden* pointer to the base class which points to the virtual table for that class (`sizeof` considers the `vtable` pointer)


Class A
vtable ptr
virtual void f()
virtual void g()
VTable (A)
void f() ptr
void g() ptr
Inherited from A
but different value
Class B
vtable ptr
void f()
(override)
VTable (B)
void f() ptr
void g() ptr

Virtual Method Notes

`virtual` classes allocate one extra pointer (hidden)

```
class A {
    double x;
    virtual void f1();
    virtual void f2();
}

class B : A {
    virtual void f1();
}

sizeof(A) = sizeof(double) + 1 * sizeof(pointer) // 16
sizeof(B) = sizeof(A)                             // 16
```

The `virtual` keyword is *not necessary* in derived classes, but it improves *readability* and clearly advertises the fact to the user that the function is `virtual`

override Keyword

override Keyword

The override keyword (C++11) ensures that the function is virtual and is overriding a virtual function from a base class

It force the compiler to check the base class to see if there is a `virtual` function with this exact signature

- `override` implies `virtual` (`virtual` should be omitted)

```
struct A {
    virtual void f(int a);          // a "float" value is casted to "int"
};                                  // see*

struct B : A {
    void f(int a) override;         // ok
    void f(float a);                // (still) very dangerous!! see*
//  void f(float a) override;       // compile error!! not safe
//  void f(int a) const override; // compile error!! not safe
};
// *f(3.3f) has different behavior between A and B
```

final Keyword

The `final` keyword (C++11) prevent inheriting from classes or prevent overriding methods in derived classes

```
struct A {
    virtual void f(int a) final;  // "final" method
};

struct B : A {
//  void f(int a);   // compile error!! f(int) is "final"
    void f(float a); // dangerous!! (still possible)
};                   // "override" prevents these errors

struct C final {   // cannot be extended
};
// struct D : C {  // compile error!! C is "final"
// };
```

Virtual Methods (Common Error 1)

All classes with at least one `virtual` method should declare a `virtual` *destructor*

```
struct A {
    ~A() { std::cout << "A"; }   // <-- here the problem (not virtual)
    virtual void f(int a) {}
};
struct B : A {
    int* array;
    B() { array = new int[1000000]; }
    ~B() {
        delete[] array;
        std::cout << "B";
    }
};
void g(A* a) {
    delete a;    // call ~A()
}
int main() {
    B* b = new B;
    g(b);     // without virtual, ~B() is not called
}             // g() prints only "A" -> huge memory leak!!
```

Virtual Methods (Common Error 2)

Do not call virtual methods in constructor and destructor

- *Constructor*: The derived class is not ready until constructor is completed
- *Destructor*: The derived class could be already destroyed

```
struct A {
    A() { f(); }  // what instance is called? "B" is not ready
                  // it calls A::f(), even though A::f() is virtual
    virtual void f() { std::cout << "A"; }
};

struct B : A {
    B() : A() {}  // call A()     (A() call may be also implicit)

    void f() { std::cout << "B"; }
};

int main() {
    B b;      // call B()
}             // print "A", not "B"!!
```

Virtual Methods (Common Error 3)

Do not use default parameters in virtual methods

Default parameters are not inherited

```
struct A {
    virtual void f(int i = 5) { std::cout << "A::" << i << "\n"; }
    virtual void g(int i = 5) { std::cout << "A::" << i << "\n"; }
};
struct B : A {
    void f(int i = 3) { std::cout << "B::" << i << "\n"; }
    void g(int i)     { std::cout << "B::" << i << "\n"; }
};

int main() {
    A* a = new A();
    a->f();           // ok, print "A::5"
    B* b = new B();
    b->f();           // ok, print "B::3"
    A* bb = new B();
    bb->f();          // !!! print "B::5"  // the virtual table of A
                                           // contains f(int i = 5) and
    bb->g();          // !!! print "B::5"  // g(int i = 5) but it points
}                                          // to B implementations
```

Pure Virtual Method

A **pure virtual method** is a function that must be implemented in derived classes (concrete implementation)

Pure virtual functions can have or not have a body

```
struct A {
    virtual void f(int x) = 0; // pure virtual without body
    virtual void g(int x) = 0; // pure virtual with body
};

void A::g(int x) {} // pure virtual implementation (body) for g()

struct B : A {
    void f(int x) {}  // must be implemented
    void g(int x) {}  // must be implemented
};
```

If a virtual method is not implemented in derived class, it is implicitly declared pure virtual

```
struct A {
    virtual void f(int x) = 0;
};

struct B : A {
//  virtual void f(int x) = 0; // implicitly declared
};

struct C : B {
    void f(int x) override {}  // implemented
};

int main() {
   C c;
   c.f(3); // ok
}
```

Abstract Class and Interface

- A class is **abstract** if it has at least one *pure virtual* function
- A class is **interface** if it has only *pure virtual* functions and optionally (*suggested*) a virtual destructor. Interfaces do not have implementation or data

```
struct A {          // INTERFACE
    virtual ~A();   // to implement
    virtual void f(int x) = 0;
};

struct B {          // ABSTRACT CLASS
   B() {}           // abstract classes may have a contructor
   virtual void g(int x) = 0; // at least one pure virtual
protected:
   int x;           // additional data
};
```

Virtual Methods (Virtual Contructor)

Virtual Constructor is not supported in C++, but can be emulated by using other `virtual` methods

```
struct A {
   virtual ~A() { }                  // A virtual destructor
   virtual A clone()  const = 0; // Uses the copy constructor
   virtual A create() const = 0; // Uses the default constructor
};

struct B : A {
    B clone() const {       // Covariant Return Types
        return B(*this);    // (different from A::clone())
    }

    B create() const {     // Covariant Return Types
        return B();        // (different from A::create())
    }
};

void f(A& a) {
    B b = a.clone();  // ok
}
```

Operator Overloading

Operator Overloading

Operator overloading is a special case of polymorphism in which some *operators* are treated as polymorphic functions and have different behaviors depending on the type of its arguments

```
struct Point {
    int x, y;
    Point(int x1, int y1) : x(x1), y(y1) {}

    Point operator+(const Point& p) const {
        return Point(x + p.x, y + p.x);
    }
};

int main() {
    Point a(1, 2);
    Point b(5, 3);
    Point c = a + b; // "c" is (6, 5)
}
```

Operator Overloading

Syntax: `operator@`

Categories not in bold are rarely used in practice

Arithmetic:	`+ - * \ % ++ --`
Comparison:	`== != < <= > >=`
Bitwise:	`| & ^ ~ << >>`
Logical:	`! && ||`
Compound assignment:	`+= <<= *=`, etc.
Subscript:	`[]`
Address-of, Reference, Dereferencing:	`& -> ->* *`
Memory:	`new new[] delete delete[]`
Comma:	`,`

Operators which cannot be overloaded: `? . .* :: sizeof typeof`

Notes

- **Increment, Decrement**: *Prefix* and *Postfix* notation

```
struct A {
    A& operator++() {  // prefix:  ++obj
        ...
        return *this;
    }
    A operator++(A& a); // postfix:  obj++
};  // NOTE: return the old copy of "this"
```

- **Array subscript** operator accepts anything (not only integer)

```
struct A {
    int&       operator[](char c);       // read/write
    const int& operator[](char c) const; // read, "const A a;"
};
// A a; a['v'] = 3;
```

- Operators preserve **precedence** and **short-circuit** properties (e.g.^)
- `operator<` is used in comparison procedures (`std::sort`)

Binary Operators

Binary Operators should be implemented as friend methods

```cpp
class A {}; class C {};

struct B : public A {
    bool operator==(const A& x) { return true; }
};

class D : public C {
    friend bool operator==(const C& x, const C& y);
};
bool operator==(const C& x, const C& y); { return true; }

int main() {
    A a;  B b;  C c; D d;
    b == a; // ok
//  a == b; // compile error!! // friend is useful to access
                               // private fields
    c == d; // ok
    d == c; // ok
}
```

Special Operators (iostream `operator<<`)

The **stream operations** can be overloaded to perform input and output for user-defined types

```cpp
#include <iostream>
struct Point {
    int x, y;

    // may be also directly defined inside Point
    friend std::ostream& operator<<(std::ostream& stream,
                                    const Point&  point);
};

std::ostream& operator<<(std::ostream& stream,
                         const Point&  point) {
    stream << "(" << point.x << "," << point.y << ")";
    return stream;
}

int main() {
    Point point { 1, 2 };
    std::cout << point;  // print "(1, 2)"
}
```

Special Operators (function call `operator()`)

The **function call operator** is generally overloaded to create objects which behave like functions, or for classes that have a primary operation

Many algorithms (included `std` library) accept objects of such types to customize behavior

```cpp
#include <iostream>
#include <numeric> // for std::accumulate
struct Multiply {
    int operator()(int a, int b) const {
        return a * b;
    }
};
int main() {
    int array[] = { 2, 3 ,4 };
    int mul = std::accumulate(array, array + 3, 1, Multiply());
    std::cout << mul;  // 24
}
```

Special Operators (conversion `operator type()`)

Conversion operators enable objects of a class to be either implicitly (coercion) or explicitly (casting) converted to another type

```cpp
class MyBool {
    int a;
public:
    MyBool(int a1) : a(a1) {}

    operator bool()(const MyBool& b) const {
        return b.a == 0;    // implicit return type
    }
};

int main() {
    MyBool my_bool { 3 };
    bool b = my_bool;  // b = false, call operator bool()
}
```

Special Operators (conversion `operator type()` + `explicit`)

Conversion operators can be marked `explicit` to prevent implicit conversions:

```cpp
struct A {
    operator bool() { return true; }
};

struct B {
    explicit operator bool() { return true; }
};

int main() {
    A a;
    B b;
    bool c = a;
//  bool c = b;  // compile error!! explicit
    bool c = static_cast<bool>(b);
}
```

The **assignment operator** (`operator=`) is used to copy values from one object to another *already existing* object

```
#include <algorithm> //std::fill, std::copy
struct A {
    char* array;
    int   size;

    A(int size1, char value) : size(size1) {
         array = new char[size];
         std::fill(array, array + size, value);
    }
    ~A() { delete[] array; }

    A& operator=(const A& x) { .... } // see next slide
};

int main() {
     A obj(5, 'o');  // ["ooooo"]
     A a(3, 'b');    // ["bbb"]
     obj = a;        // obj = ["bbb"]
}
```

- First option:

```
A& operator=(const A& x) {
    if (this == &x)         // Check for self assignment
        return *this;
    delete[] array;         // delete everything from this
    array = new int[x.size];
    std::copy(x.array, x.array + size, array);  // copy
    return *this;
}
```

- Second option (less intuitive):

```
A& operator=(A x) {  // pass by value: need a copy constructor
    swap(this, x);   // now we need a swap function for A
    return *this;    //    see next slide
}                    // x is destroyed at the end
```

Special Operators (assignment `operator=`) 3/3

- Swap method:

```
friend void swap(A& x, A& y) {
    using std::swap;
    swap(x.size, y.size);
    swap(x.array, y.Array);
}
```

  - **why using `std::swap`?** if `swap(x, y)` finds a better match, it will use that instead of `std::swap`
  - **why `friend`?** it allows the function to be used from outside the structure/class scope

Full story: `stackoverflow.com/questions/5695548`

C++ Special Objects

Aggregate

An **aggregate** is a type which supports *aggregate initialization* (form of list-initialization) through curly braces syntax {}

An aggregate is an *array* or a *class* with

- No user-provided constructors (all)
- No private/protected non-static data members
- No base classes
- No virtual functions (standard functions allowed)

* No *brace-or-equal-initializers* for non-static data members (until C++14)

No restrictions:

- Non-static data member (can be also not aggregate)
- Static data members

Full story: `stackoverflow.com/questions/4178175`

Aggregate (examples) 2/3

```
struct NotAggregate1 {
    NotAggregate1();      // No constructors
    virtual void f();     // No virtual functions
};

class NotAggregate2 : NotAggregate1 { // No base class
    int x;            // x is private
};

struct Aggregate1 {
    int x;
    int y[3];
    int z { 3 };      // only C++14
};

struct Aggregate2 {
    Aggregate1() = default; // ok, defaulted constructor
    NotAggregate2 x;        // ok, public member
    Aggregate2& operator=(const& Aggregate2 obj); // ok
private:                                          // copy-assignment
    void f() {} // ok, private function (no data member)
};
```

Aggregate (examples)

```
struct Aggregate1 {
    int x;
    struct Aggregate2 {
        int a;
        int b[3];
    } y;
};

int main() {
    int array1[3] = { 1, 2, 3 };
    int array2[3]   { 1, 2, 3 };
    Aggregate1 agg1 = { 1, { 2, { 3, 4, 5} } };
    Aggregate1 agg2   { 1, { 2, { 3, 4, 5} } };
    Aggregate1 agg3 = { 1, 2, 3, 4, 5 };
}
```

Trivial Class

A **Trivial Class** is a class *trivial copyable* (supports `memcpy`)

Trivial copyable:

- No user-provided copy/move/default *constructors* and *destructor*
- No user-provided copy/move *assignment* operators
- No virtual functions (standard functions allowed) or virtual base classes
- No *brace-or-equal-initializers* for non-static data members
- All non-static members are trivial (recursively for members)

No restrictions:

- Other user-declared constructors different from default
- Static data members
- Protected/Private members

```
struct NonTrivial1 {
    int y { 3 };       // brace-or-equal-initializers

    NonTrivial1();     // user-provided constructor
    virtual void f(); // virtual function
};

struct Trivial1 {
    Trivial1() = default;    // defaulted constructor
    int x;
    void f();
private:
    int z; // ok, private
};

struct Trivial2 : Trivial1 { // base class is trivial
    int Trivial1[3];          // array of trivials is trivial
};
```

Standard-Layout

A **standard-layout class** is a class with the same memory layout of the equivalent C `struct` or `union` (useful for communicating with other languages)

Standard-layout class

- No virtual functions or virtual base classes
- Recursively on non-static members, base and derived classes
- Only one control access (public/protected/private) for non-static data members
- No base classes of the same type as the first non-static data member

(a) No non-static data members in the *most derived* class and *at most one base* class with non-static data members

(b) No base classes with non-static data members

Standard-Layout Class (examples)

```cpp
struct StandardLayout1 {
    StandardLayout1(); // user-provided contructor
    int x;
    void f();          // non-virtual function
};

class StandardLayout2 : StandardLayout1 {
    int x, y;          // both are private
    StandardLayout1 y; // can have members of base type
                       // if they are not the first
};

struct StandardLayout3 { } // empty

struct StandardLayout4 : StandardLayout2, StandardLayout3 {
    // can use multiple inheritance as long only
    // one class in the hierarchy has non-static data members
};
```

Plain Old Data (POD)

C++11, C++14 Standard-Layout (s) + Trivial copyable (t)

- **(t)** No user-provided copy/move/default constructors and destructor
- **(t)** No user-provided copy/move assignment operators
- **(t)** No virtual functions or virtual base classes
- **(t)** No *brace-or-equal-initializers* for non-static data member
- **(s)** Recursively on non-static members, base and derived classes
- **(s)** Only one control access (public/protected/private) for non-static data members
- **(s)** No base classes of the same type as the first non-static data member
- **(s)a** No non-static data members in the *most derived* class and *at most one base* class with non-static data members
- **(s)b** No base classes with non-static data members

C++ std Utilities

C++11 provides three utilities to check if a type is POD, Trivial Copyable, Standard-Layout

- `std::is_pod` checks for POD
- `std::is_trivially_copyable` checks for trivial copyable
- `std::is_standard_layout` checks for standard-layout

```
#include <type_traits>
struct A {
    int x;
private:
    int y;
};
int main() {
    std::cout << std::is_trivial_copyable<A>::value; // true
    std::cout << std::is_standard_layout<A>::value;  // false
    std::cout << std::is_pod<A>::value;              // false
}
```

Special Objects Hierarchy