四川大學
SICHUAN UNIVERSITY

Object Oriented Programming—C++
Lecture9 Class Keywords & Polymorphism

Qijun Zhao
College of Computer Science
Sichuan University

Spring 2023

- Class Keywords
  - this
  - static
  - const
  - mutable
  - using
  - friend
  - delete

Class Keywords——this

this

Every object has access to its own address through the const pointer this

Explicit usage is not mandatory (and not suggested)

this is necessary when:

- The name of a local variable is equal to some member name
- Return reference to the calling object

```
struct A {
    int x;
    void f(int x) {
        this->x = x; // without "this" has no effect
    }
    const A& g() {
        return *this;
    }
};
```

Class Keywords——static

static

The keyword static declares members (fields or methods) that are not bound to class instances. A static member is shared by <u>all</u> objects of the class

- A static member function can <u>only</u> access static class members
- A non- static member function can access static class members
- Non-const static data members <u>cannot</u> be directly initialized inline...before C++17

Mutable static members

```
// "static" means the same value for all instances
struct A {
// static int          a = 4;    // compiler error
   static int          a;        // ok, (declaration)
   static inline int b = 4;      // from C++17
};
int A::a = 4; // ok, without definition -> undefined reference
```

Constant static members

```
struct A {
   static const int          c = 4;    // also C++03
// static const float        d = 4.2f; // only GNU extension (GCC)
   static constexpr float e = 4.2f; // ok, C++11
};
```

Class Keywords——static-Example

```
struct A {
    int        y = 2;
    static int x; // declaration

    static int f() { return x * 2; }
//  static int f() { return y;     } // compile error "y" is non-static
    int h()        { return x;     } // ok, "x" is static
};
int A::x = 3;     // definition

//----------------------------------------------------------------------
A a;
a.h();            // return 3
A::x++;
cout << A::x;     // print 4
cout << A::f();   // print 8
```

Class Keywords——const

Const member functions

Const member functions (inspectors or observer) are functions marked with const that are not allowed to change the object state

Member functions without a const suffix are called non-const member functions or **mutators**. The compiler prevents from inadvertently mutating/changing the data members of observer functions

```
struct A {
    int x = 3;

    int get() const {
     // x = 2;     // compile error class variables cannot be modified
        return x;
    }
};
```

Class Keywords——const Overloading

The const keyword is part of the functions signature. Therefore a class can implement two similar methods, one which is called when the object is const , and one that is not

```
class A {
    int x = 3;
public:
    int& get1()       { return x; } // read and write
    int  get1() const { return x; } // read only
    int& get2()       { return x; } // read and write
};

A a1;
cout << a1.get1();    // ok
cout << a1.get2();    // ok
a1.get1() = 4;        // ok
const A a2;
cout << a2.get1();    // ok
// cout << a2.get2(); // compile error "a2" is const
//a2.get1() = 5;      // compile error only "get1() const" is available
```

Class Keywords——mutable

mutable

mutable members of *const* class instances are modifiable

Constant references or pointers to objects cannot modify objects in any way, except for data members marked mutable

- It is particularly useful if most of the members should be constant but a few need to be modified
- *Conceptually, mutable members should not change anything that can be retrieved from the class interface*

```cpp
struct A {
    int         x = 3;
    mutable int y = 5;
};
const A a;
// a.x = 3; // compiler error const
a.y = 5;    // ok
```

Class Keywords——using for type declaration

The using keyword is used to declare a *type alias* tied to a specific class

```
struct A {
    using type = int;
};

typename A::type x = 3; // "typename" keyword is needed when we refer to types

struct B : A {};

typename B::type x = 4; // B can use "type" as it is public in A
```

Class Keywords——using for Inheritance

The using keyword can be also used to change the inheritance attribute of member data or functions

```
struct A {
protected:
    int x = 3;
};

struct B : A {
public:
    using A::x;
};

B b;
b.x = 3;  // ok, "b.x" is public
```

Class Keywords——friend

friend Class

A friend class can access the private and protected members of the class in which it is declared as a friend

Friendship properties:

- **Not Symmetric**: if class A is a friend of class B, class B is not automatically a friend of class A
- **Not Transitive**: if class A is a friend of class B, and class B is a friend of class C, class A is not automatically a friend of class C
- **Not Inherited**: if class Base is a friend of class X, subclass Derived is not automatically a friend of class X; and if class X is a friend of class Base, class X is not automatically a friend of subclass Derived

Class Keywords——friend

```cpp
class B;    // class declaration

class A {
    friend class B;
    int x;        // private
};

class B {
    int f(A a) { return a.x; } // ok, B is friend of A
};

class C : B {
//  int f(A a) { return a.x; } // compile error not inherited
};
```

Class Keywords——friend

friend Method

A ***non-member* function** **can access the private and protected members of a class if it is declared a friend of that class**

```
class A {
    int x = 3;  // private

    friend int f(A a); // friendship declaration, no implementation
};

//'f' is not a member function of any class
int f(A a) {
    return a.x;  // A is friend of f(A)
}
```

friend methods are commonly used for implementing the stream operator operator<<

delete Keyword (C++11)

The delete keyword explicitly marks a member function as deleted and any use results in a compiler error. When it is applied to *copy/move constructor* or *assignment*, it prevents the compiler from implicitly generating these functions

The default copy/move functions for a class can produce unexpected results. The keyword delete prevents these errors

```
struct A {
    A()         = default;
    A(const A&) = delete; // e.g. deleted because unsafe or expensive
};
void f(A a) {}  // implicit call to copy constructor

A a;
// f(a);        // compile error marked as deleted
```

- Polymorphism
  - virtual methods
  - Virtual Table
  - override keyword
  - final keyword
  - Common Errors
  - Pure Virtual Method
  - Abstract Class and Interface

Polymorphism

In Object-Oriented Programming (OOP), 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 *base class* behaves as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

Polymorphism vs. Overloading

Overloading is a form of static polymorphism (compile-time polymorphism)

In C++, the term **polymorphic** is strongly associated with dynamic polymorphism (*overriding*)

```
// overloading example
void f(int a)    {}

void f(double b) {}

f(3);          // calls f(int)
f(3.3);        // calls f(double)
```

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 compile-time

  - the program can jump directly to the function address

- In **Late Binding** or Dynamic Binding or Run-time binding, the run-time identifies the type of object at execution-time and then matches the function call with the correct function definition

  - 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)

C++ achieves **late binding** by declaring a virtual function

Polymorphism - The problem

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

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

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

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

A a;
B b;
g(a);    // print "A"
g(b);    // print "A" not "B"!!!
```

Polymorphism - virtual method

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

struct B : A {
    void f() { cout << "B"; }
}; // now "B::f()" overrides "A::f()", evaluated at run-time

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

A a;
B b;
g(a); // print "A"
g(b); // NOW, print "B"!!!
```

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

When virtual works

```
struct A {
    virtual void f() { cout << "A"; }
};

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

void f(A& a) { a.f();  } // ok, print "B"
void g(A* a) { a->f(); } // ok, print "B"
void h(A  a) { a.f();  } // does not work!! print "A"

B b;
f(b);  // print "B"
g(&b); // print "B"
h(b);  // print "A" (cast to A)
```

Polymorphism Dynamic Behavior

```
struct A {
    virtual void f() { cout << "A"; }
};

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

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

get_object(true)->f();  // print "A"
get_object(false)->f(); // print "B"
```

vtable

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

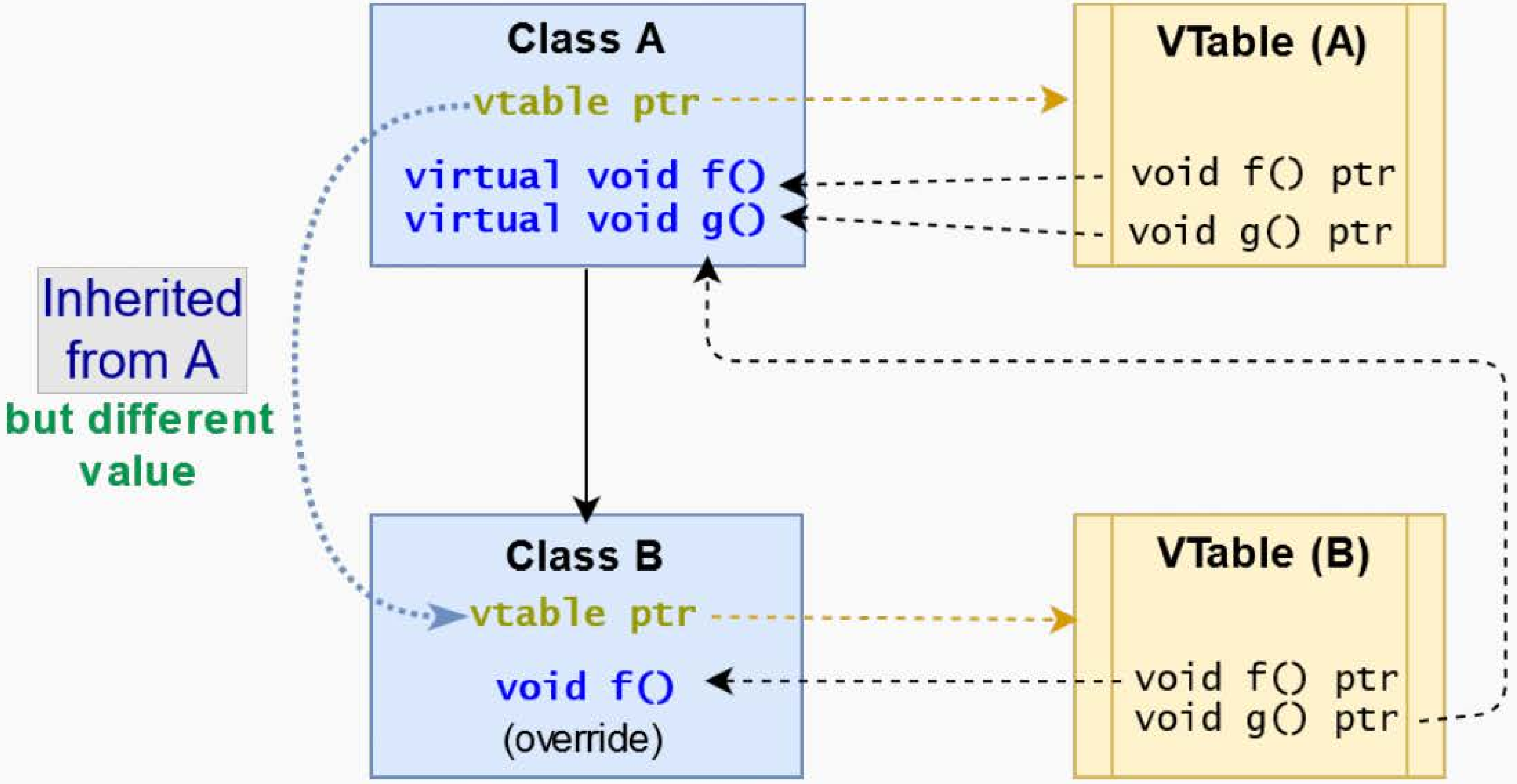
A *virtual table* contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the *most-derived* function accessible by that class

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

Virtual Table

```
struct A {
    virtual void f();
    virtual void g();
};

struct B : A {
    void f();
};
```



Does the vtable really exist? (answer: YES)

```
struct A {
    int x = 3;
    virtual void f() { cout << "abc"; }
};

A* a1 = new A;
A* a2 = (A*) malloc(sizeof(A));

cout << a1->x;  // print "3"
cout << a2->x;  // undefined value!!
a1->f();        // print "abc"
a2->f();        // segmentation fault ☠
```

Lesson learned: Never use malloc in C++

Virtual Method Notes

virtual classes allocate one extra pointer (hidden)

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

class B : A {};

cout << sizeof(A); // 8 bytes (vtable pointer)
cout << sizeof(B); // 8 bytes (vtable pointer)
```

override Keyword

override Keyword (C++11)

The override keyword ensures that the function is virtual and is <u>overriding</u> a virtual function from a base class

It forces the compiler to check the base class to see if there is a virtual function

with this <u>exact</u> signature

override implies virtual (virtual should be omitted)

override Keyword

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

struct B : A {
    void f(int   a) override;       // ok
    void f(float a);                // (still) very dangerous!!
                                    // ***

//  void f(float a) override;       // compile error not safe
//  void f(int   a) const override; // compile error not safe
};

// *** f(3.3f) has a different behavior between A and B
```

final Keyword (C++11)

The final keyword prevents inheriting from classes or 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() { 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;            }
};
//---------------------------------------------------------------------
void destroy(A* a) {
    delete a;    // call ~A()
}
B* b = new B;
destroy(b); // without virtual, ~B() is not called
            // destroy() 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 is 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() { cout << "Explosion"; }
};
struct B : A {
    B() = default; // call A(). Note: A() may be also implicit

    void f() override { cout << "Safe"; }
};

B b; // call B(), print "Explosion", not "Safe"!!
```

Virtual Methods (Common Error 3)

Do not use default parameters in virtual methods

Default parameters are <u>not</u> inherited

```cpp
struct A {
    virtual void f(int i = 5) { cout << "A::" << i << "\n"; }
    virtual void g(int i = 5) { cout << "A::" << i << "\n"; }
};
struct B : A {
    void f(int i = 3) override { cout << "B::" << i << "\n"; }
    void g(int i)     override { cout << "B::" << i << "\n"; }
};
A a; B b;
a.f();     // ok, print "A::5"
b.f();     // ok, print "B::3"

A& ab = b;
ab.f();    // !!! print "B::5"  // the virtual table of A
                                // contains f(int i = 5) and
ab.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() = 0; // pure virtual without body
    virtual void g() = 0; // pure virtual with body
};
void A::g() {} // pure virtual implementation (body) for g()

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

Pure Virtual Method

A class with one *pure virtual function* cannot be instantiated

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

struct B1 : A {
//  virtual void f() = 0; // implicitly declared
};

struct B2 : A {
    void f() override {}
};

// A  a;  // "A"  has a pure virtual method
// B1 b1; // "B1" has a pure virtual method
B2 b2;    //  ok
```

Abstract Class and Interface

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

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

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

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn