Modern C++ Programming

7. C++ Object Oriented Programming I

Federico Busato

University of Verona, Dept. of Computer Science
2020, v3.02

Table of Context

C++ Classes

C++ Classes

C/C++ Structure

A **structure** (`struct`) is a collection of variables of different data types under a single name

C++ Class

A **class** (`class`) extends the concept of structure to hold data members and also functions as members

Class Member/Field

The data within a class are called *data members* or *class field*. Functions within a class are called *function members* or *methods* of the class

`struct` vs. `class`

Structures and classes are *semantically* equivalent. In general, `struct` represents *passive* objects, while `class` *active* objects

Holding a resource is a class invariant, and is tied to object lifetime

Implication 1: C++ programming language does not require the garbage collector!!

Implication 2 :The programmer has the responsibility to manage the resources

RAII Idiom consists in three steps:

- Encapsulate a resource into a class (constructor)
- Use the resource via a local instance of the class
- The resource is automatically releases when the object gets out of scope (destructor)

Struct declaration and definition

```
struct A;       // struct declaration

struct A {      // struct definition
    int x;      // data member
    void f(); // function member
};
```

Class declaration and definition

```
class A;        // class declaration

class A {       // class definition
public:         // visibility attribute
    int x;      // data member
    void f(); // function member
};
```

Struct/Class function declaration and definition

```
struct A {
   void g();          // function member declaration

   void f() {         // function member declaration
       cout << "f";   // and inline definition
   }
};

void A::g() {         // function member definition
    cout << "g";      // (not inline)
}
```

```
struct B {
    void g() { cout << "g"; }
};

struct A {
    int  x;
    B    b;
    void f() { cout << "f"; }
    using T = B;
};

A a;
cout << a.x;
a.f();
a.b.g();
A::T obj; // equal to "B obj"
```

Child/Derived Class or Subclass

A new class that inheriting variables and functions from another class is called a **derived** or **child** class

Parent/Base Class

The *closest* class providing variables and function of a derived class is called **parent** or **base** class

Extend a base class refers to creating a new class which retains characteristics of the base class and *on top it can add* (and never remove) its own members

Syntax:

```
struct DerivedClass : [<inheritance>] BaseClass {
    ...
};
```

Class Hierarchy


```cpp
#include <iostream>

struct A {  // base class
    int value = 3;
};

struct B : A { // B inherits from A (B extends A) (B is child of A)
    int data = 4;
    int f() { return data; }
};

struct C : B { // C extends B (C is child of B)
};

int main() {
    A base;
    B derived1;
    C derived2;
    std::cout << base.value;     // print 3
    std::cout << derived1.data; // print 4
    std::cout << derived2.f();  // print 4
}
```

Inheritance Attributes

`private`, `public`, and `protected` inheritance

- **`public:`** The public members can be accessed without any restriction
- **`protected:`** The protected members of a base class can be accessed by its derived class
- **`private:`** The private members of a class can only be accessed by function members of that class

Inheritance Attributes

Member declaration		Inheritance		Derived classes
public protected private	→	**public**	→	public protected \
public protected private	→	**protected**	→	protected protected \
public protected private	→	**private**	→	private private \

- structs have default **public** members
- classes have default **private** members

```
#include <iostream>
using namespace std;

class A {
public:
    int var1 = 3;
    int f() { return var1; }
protected:
    int b;
};

class B : public A { // without public, B inherits
};                   // the data member "var1" and f()
                     // as private members
int main() {
    B derived;
    cout << derived.f(); // print 3
//  cout << derived.b;   // compile error protected
}
```

Class Constructor

Class Constructor

Constructor [ctor]

A **constructor** is a *special* member function of a class that is executed when a new instance of that class is created.

Goals: *initialization* and *resource acquisition*

- A constructor is always named as the class
- A constructor have no return type
- A constructor is supposed to initialize *all* the data members of a class
- We can define multiple constructors (different signatures)

Class constructors are never inherited. *Derived* class must call a *Base* constructor before the current class constructor

Class constructors are called in order of declaration
(C++ objects are constructed like onions)

Class Constructor (Examples)

```cpp
#include <iostream>
class A {
    int x;
public:
    // constructor
    A(int x1) : x(x1) {  // initialization list syntax
        std::cout << "A";
    }
};

class B : A {
public:
    B(int b1) : A{b1} { std::cout << "B"; } // A{b1} better syntax
};

int main() {
    A a(1);     // print "A"
    B b(2);     // print "A", then print "B"
    A c = {1};  // initialization, print "A"
    A d {1};    // initialization (C++11), print "A"
}
```

Initialization Order

Class members initialization follows the order of declarations and *not* the order in the initialization list

```
struct A {
    int* array;
    int  size;

    A(int user_size) :
        size{user_size},
        array{new int[size]} {}
        // very dangerous: "size" is still undefined
};

A a{10};
cout << a.array[4]; // potential segmentation fault
```

Default Constructor

Default Constructor

The **default constructor** is a constructor with no arguments

Every class has always either an *implicit* or *explicit* default constructor

```
class A {
public:
    A()    {} // default constructor
    A(int) {} // normal user-defined constructor
};
```

if a *user-provided constructor* is defined while the *default constructor* is not, the *default constructor* is marked as deleted

Example

```
struct A {};  // implicit-declared public default constructor

class B {
public:     // <- visibility
    B() { cout << "B"; }  // default constructor
};

struct C {
    int& a; // implicit-deleted default constructor (next slide)
};

A  a1;              // call the default constructor
// A a2();          // interpreted as a function declaration!!
B  b;               // ok, print "B"
B  array[3];        // print three times "B"
B* ptr = new B[4]; // print four times "B"
// C c;             // compile error deleted
```

Deleted Default Constructor

The *implicit* default constructor of a class is marked as **deleted** if (simplified):

- It has a member of reference/`const` type
- It has any user-defined constructor
- It has a member/base class which has a deleted (or inaccessible, or ambiguous) default constructor
- It has a base class which has a deleted (or inaccessible, or ambiguous) destructor

Delegate Constructor

The problem:
Most constructors usually perform identical initialization steps before executing individual operations

A **delegate constructor** (C++11) calls another constructor of the same class to reduce the repetitive code by adding a function that does all of the initialization steps

```
struct A {
    int   a1;
    float b1;
    bool  c1;
    // standard constructor:
    A(int a1, float b1, bool c1) : a(a1), b(b1), c(c1) {
        // do a lot of work
    }
    // delegate construtors:
    A(int a1, float b1) : A(a1, b1, false)    {}
    A(float b1)         : A(100, b1, false)   {}
};
```

explicit Keyword

explicit

The `explicit` keyword specifies that a constructor or conversion function does not allow implicit conversions or copy-initialization

```cpp
struct A {
    A(int) {}
    A(int, int) {}
};

struct B {
    explicit B(int) {}
    explicit B(int, int) {}
};
```

```cpp
int main() {
    A a1 = 1;       // ok (implicit)
    A a2(2);        // ok
    A a3 {4, 5};    // ok. Selected A(int, int)
    A a4 = {4, 5}; // ok. Selected A(int, int)


//  B b1 = 1;       // error implit conversion
    B b2(2);        // ok
    B b3 {4, 5};    // ok. Selected B(int, int)
//  B b4 = {4, 5}; // error implit conversion
    B b5 = (B) 1;  // OK: explicit cast
}
```

Copy Constructor

Copy Constructor

Copy Constructor

A **copy constructor** is a constructor used to create a new object as a *copy* of an existing object

Every class always define an *implicit* or *explicit* copy constructors

```
struct A {
    A()          {} // default constructor
    A(int)       {} // user-provided constructor
    A(const A&) {} // copy constructor
}
```

Note: in `class` the implicit copy constructor is marked as `private`

Example

```
struct A {
    int  size;
    int* array;

    A(int size1) : size{size1} {
        array = new int[size];
    }

    A(const A& obj) : size{obj.size} { // copy constructor
        for (int i = 0; i < size; i++)
            array[i] = obj.array[i];
    }
};

int main() {
    A x{100};
    A y{x};     // call "A::A(const A&)" copy constructor
}
```

Copy Constructor Usage

The copy constructor is used to:

- Initialize one object from another having the same type
  - Direct constructor
  - Assignment operator

```
A a1;
A a2(a1); // Direct copy-constructor
a1 = a2;  // Assignment operator
```

- Copy an object which is *passed-by-value* as input parameter of a function

```
void f(A a);
```

- Copy an object which is returned as result from a function*

```
A f() {
    return A(3); // * see RVO optimization
}
```

Examples

```cpp
#include <iostream>
class A {
public:
    A() {}
    A(const A& obj) { std::cout << "copy" << std::endl; }
};

void f(A a) {}
A    g()    { return A(); };

int main() {
    A a;
    A b = a;    // copy constructor (assignment)
    A c(b);     // copy constructor (direct)
    f(b);       // copy constructor (argument)
    g();        // copy constructor (return value)
    A d = g();  // * see RVO optimization
}
```

Pass by-value and Copy Constructor

```cpp
#include <iostream>
class A {
public:
    A() {}
    A(const A& obj) { std::cout << "expensive copy" << std::endl; }
};

class B : public A {
public:
    B() {}
    B(const B& obj) { std::cout << "cheap copy" << std::endl; }
};

void f1(B b) {}
void f2(A a) {}

int main() {
    B b1;
    f1(b1); // cheap copy
    f2(b1); // expensive copy!! It calls A(const A&) implicitly
}
```

Deleted Copy Constructor

The copy constructor of a class is marked as **deleted** if (simplified):

- Every non-static class type (or array of class type) member has a valid (accessible, not deleted, not ambiguous) copy constructor
- Every base classes has a valid (accessible, not deleted, not ambiguous) copy constructor
- It has a base class with a deleted or inaccessible destructor
- The class has no move constructor (next lectures)

Class Destructor

Destructor [dtor]

A **destructor** is a member function of a class that is executed whenever an object is out-of-scope or whenever the `delete /delete[]` expression is applied to a pointer of that class

Goals: *resources releasing*

- A destructor will have exact same name as the class prefixed with a tilde ($\sim$)
- A destructor does not have any return type
- Each object has exactly one destructor
- A destructor is useful for releasing resources before the class instance goes out of scope or it is deleted

Class Destructor

```
struct A {
    int* array;

    A() {   // constructor
        array = new int[10];
    }

    ~A() {  // destructor
        delete[] array;
    }
};

int main() {
   A a;      // call the constructor
   for (int i = 0; i < 5; i++)
       A b; // call 5 times the constructor and the destructor
   // call the destructor of "a"
}
```

Class destructor is never inherited. *Base* class destructor is invoked *after* the current class destructor.

Class destructors are called in reverse order

```
struct A {
    ~A() { cout << "A"; }
};
struct B {
    ~B() { cout << "B"; }
};
struct C : A {
    B b;              // call ~B()
    ~C() { cout << "C"; }
};

int main() {
    C b;  // print "C", then "B", then "A"
}
```

Initialization and Defaulted Members

Member Initialization

```
struct A {
   int       a = 3;          // not allowed in C++03
   const int b = 3;          // not allowed in C++03

// int c { 3.3 };            // compiler error (narrowing)
                             //   uniform-initilization
                             //   should be preferred
   // "static" means the same value for all instances
   static const int   d = 4; // also C++03

// static int         e = 4; // compiler error (-Wpedantic)

   static const float f = 4.2f; // only GNU extension (GCC)

   static constexpr float g = 4.2f; // ok
};

int A::e = 4; // ok
```

Initialization List

Any data member should be initialized by constructors with the **initialization list** or by using **brace-or-equal-initializer** (C++11) syntax

const and **reference** data members must be initialized by using the *initialization list* or by using *brace-or-equal-initializer*

```
struct A {
    int        x;
    const char y;  // must be initilizated
    int&       z;  // must be initilizated
    A() : x(3), y('a'), z(x) {} // initialization-list, also x{3}
};

struct A {
    int        x = 3;   // brace-or-equal-initializer (C++11), also x{3}
    const char y = 'a'; // brace-or-equal-initializer (C++11)
    int&       z = x;   // brace-or-equal-initializer (C++11)
};
```

Uniform Initialization (C++11)

Uniform Initialization {}, also called *list-initialization*, is a way to fully initialize any object independently from its data type

- **Minimizing Redundant Typenames**
  - In function arguments
  - In function returns
- Solving the **"Most Vexing Parse" problem**
  - Constructor interpreted as function prototype

mbevin.wordpress.com/2012/11/16/uniform-initialization

Minimizing Redundant Typenames

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

C++03

```
Point add(Point a, Point b) {
    return Point(a.x + b.x, a.y + b.y);
}

Point c = add(Point(1, 2), Point(3, 4));
```

C++11

```
Point add(Point a, Point b) {
    return { a.x + b.x, a.y + b.y }; // here
}

auto c = add({1, 2}, {3, 4});        // here
```

"Most Vexing Parse" problem ⋆

```
struct A {
    int a1, a2;
};
class B {
    int b1, b2;
public:
    B(A a) {}
    B(int x1, int x2) : b1(x1), b2(x2) {}
};
//--------------------------------------------------------------------

B g(A a) {      // "b" is interpreted as function declaration
    B b( A() ); //  with a single argument A (*)() (func. pointer)
//  return b;   // compile error "Most Vexing Parse" problem
}               // solved with B b{ A{} };
//--------------------------------------------------------------------

struct C {
// B b (1, 2);   // compile error (struct)! It works in a function scope
   B b { 1, 2 }; // ok, call the constructor
};
```

Defaulted Constructor (= `default`)

In C++11, the compiler can generate default/copy/move constructors and copy/more assignment operators

syntax: `A() = default`

The **defaulted** default constructor has a similar effect as a user-defined constructor with empty body and empty initializer list

When compiler-generated constructor is useful:

- Any user-provided constructor disables implicitly-generated default constructor
- Change the visibility of non-user provided constructors and assignment operators (`public`, `protected`, `private`)

Defaulted Constructor (= default) 2/2

```
struct A {
    int v;

    A(int v1) : v(v1){} // delete implicitly-defined default ctor
                        // because a user-provided constructor is
                        // defined

    A() = default;      // now, A has the default constructor
};

class B : A {   // default/copy constructor marked private
                // because B is a class
public:
     B()          = default; // default constructor is now public

     B(const B&) = default; // default constructor is now public
};
```

Defaulted Constructor and Inheritance

```
struct A {
   int x;
   A(int x1) : x(x1){}
   A() = default;
};

struct B : A {
    int y;
    B()          = default;
    // "B()" initializes its members and calls "A()"
    B(const B&) = default;
};  // "B(const B&)" copies its members and calls "A(const A&)"

B b1, b2;
b1.x = 3;
b1.y = 4;
b2   = b1; // "b2.x" = 3, "b2.y" = 4
```

Defaulted vs. User-Provided Default Constructor

```
struct A {
    int x;
    A() {}  // User-Provided
};

struct B {
    int x;
    B() = default; // Compiler-Provided
};

A a;
cout << a.x; // a.x is undefined

B b;
cout << b.x; // b.x is zero
```

Class Keywords

this Keyword

this

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

The `this` const pointer is an implicit variable added to any member function. In general, it is not needed (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;
    }
};
```

`static` Keyword

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

- A *static* member function can access only *static* class members
- A *non-static* member function can access *static* class members
- All *static* data are initialized to zero/default useless if no user-initialization is provided
- It can be initialized (defined) only once
- Non-const static data members cannot be `inline` initialized

```cpp
#include <iostream>
struct A {
    int        y = 2;
    static int x; // declaration (= 3 -> compile error)

    static int f() { return x * 2; }
//  static int f() { return y;     } // error "y" is non-static
    int h()        { return x;     } // ok, ("x" is static)
};

int A::x = 3; // static variable definition

int main() {
    A a;
    a.h();                 // return 3
    A::x++;
    std::cout << A::x;     // print 4
    std::cout << A::f();   // print 8
}
```

Const member functions

Const member functions, or **inspectors**, do not change the object state

Member functions without a `const` suffix are called *non-const member functions* or *mutators*

The compiler prevents callers from inadvertently mutating/changing the object data members with functions marked as `const`

```
class A {
    int x = 3;
public:
    int get() const {
     // x = 2;     // compile error class variables cannot
        return x; //               be modified
    }
};
```

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
```

mutable

`mutable` members of *const* class instances are modifiable

Constant references or pointers to objects cannot modify that object 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*

```
struct A {
    int         x = 3;
    mutable int y = 5;
};
int main() {
    const A a;
//  a.x = 3;   // |\underline{compiler error} const
    a.y = 5;   // ok
}
```

`using` Keyword

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

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

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

int main() {
    B b;
    b.x = 3;  // ok, "b.x" is public
}
```

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`

```cpp
class A;   // class declaration

class B {
    int y = 3;  // private
    int f(A a) { return a.x; } // ok, B is friend of A
};

class A {
    friend class B;
    int x = 3;  // private
//  int f(B b) { return b.y; } // compile error not symmetric
};

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

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);
};

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

delete Keyword

The `delete` keyword (C++11) 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(const A& a) = delete;
};
                // e.g. if a class uses heap memory
void f(A a) {}  // the copy construct should be
                // written by the user -> expensive copy
A a;
// f(a);        // compile error marked as deleted
}
```