

# Modern C++ Programming

## 7. C++ OBJECT ORIENTED PROGRAMMING I

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2021, v3.11



## 1 C++ Classes

- RAI

## 2 Class Hierarchy

- Access specifiers
- Inheritance Access Specifiers

## 3 Class Constructor

- Default Constructor
- Initializer List
- Uniform Initialization
- Delegate Constructor
- `explicit` Keyword

## 4 Copy Constructor

## 5 Class Destructor

## 6 Defaulted Members

- Defaulted Constructor = `default`

## 7 Class Keywords

- `this`
- `static`
- `const`
- `mutable`
- `using`
- `friend`
- `delete`

# C++ Classes

---

## C/C++ Structure

A **structure** (struct) is a collection of variables of the same or 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

## struct vs. class

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

# Class Members - Data and Function Members

## Data Member

The data within a class are called **data members** or **class field**

## Function Member

Functions within a class are called **function members** or **methods** of the class

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

**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)

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

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

# struct/class Declaration and Definition

## 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
    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"; // inline definition  
    }  
};  
  
void A::g() {           // function member definition  
    cout << "g";       // out-of-line definition  
}
```

# Class Fields

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

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

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

# Class Hierarchy

---

## 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:

```
class DerivedClass : [<inheritance attribute>] BaseClass {
```

```
struct A {           // base class
    int value = 3;
    void g() {}
};

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

A a;
B b;
a.value;
b.g();
```

The **access specifiers** define the visibility of inherited members of the subsequent base class. The keywords `public`, `private`, and `protected` specify the sections of visibility

The goal of the *access specifiers* is to prevent a direct access to the internal representation of the class for avoiding wrong usage and potential inconsistency (access control)

- **public:** No restriction (*function members, derived classes, outside the class*)
- **protected:** *Function members and derived classes* access
- **private:** *Function members* only access (internal)

`struct` has default `public` members

`class` has default `private` members

```
struct A1 {
    int value;    // public (by default)
protected:
    void f1() {} // protected
private:
    void f2() {} // private
};

class A2 {
    int data;    // private (by default)
};

struct B : A1 {
    void h1() { f1(); } // ok, "f1" is visible in B
    // void h2() { f2(); } // compile error "f2" is private in A1
};

A1 a;
a.value;    // ok
// a.f1() // compile error protected
// a.f2() // compile error private
```

The **access specifiers** are also used for defining how the visibility is propagated from the *base class* to a *specific derived class* in the inheritance

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



```
struct A {  
    int var1; // public  
protected:  
    int var2; // protected  
};  
  
struct B : protected A {  
    int var3; // public  
};  
  
B b;  
// b.var1; // compile error, var1 is protected in B  
// b.var2; // compile error, var2 is protected in B  
b.var3;    // ok, var3 is public in B
```

```
class A {  
public:  
    int var1;  
protected:  
    int var2;  
};
```

```
class B1 : A {};           // private inheritance
```

```
class B2 : public A {};    // public inheritance
```

```
B1 b1;  
// b1.var1; // compile error, var1 is private in B1  
// b1.var2; // compile error, var2 is private in B1
```

```
B2 b2;  
b2.var1;    // ok, var1 is public in B2
```

# 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*

Syntax: `T(...)` same named of the class and no return type

- A *constructor* is supposed to initialize all data members
- We can define *multiple constructors* with different signatures
- Any *constructor* can be `constexpr`

# Default Constructor

## Default Constructor

The **default constructor** `T()` is a constructor with no arguments

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

```
struct A {  
    A()    {} // explicit default constructor  
    A(int) {} // user-defined (non-default) constructor  
};
```

```
struct A {  
    int x = 3; // implicit default constructor  
};  
A a{5}; // ok, but not "A a(5);"
```

# Default Constructor Examples

```
struct A {  
    A() { cout << "A"; } // default constructor  
};  
  
A a1;           // call the default constructor  
// A a2();      // interpreted as a function declaration!!  
A a3{};        // ok, call the default constructor  
               // direct-list initialization (C++11)  
  
A array[3];     // print "A A A"  
  
A* ptr = new A[4]; // print "A A A A"
```

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

- It has any user-defined constructor

```
struct A {  
    A(int x) {}  
};  
  
// A a; // compile error
```

- It has a member of reference/const type

```
struct NoDefault { // deleted default constructor  
    int& x;  
    const int y;  
};
```

- It has a non-static member/base class which has a deleted (or inaccessible) default constructor

```
struct A {  
    NoDefault var;          // deleted default constructor  
};  
struct B : NoDefault {}; // deleted default constructor
```

- It has a Base class with a deleted or inaccessible destructor

```
struct A {  
private:  
    ~A() {}  
};
```



# Initializer List

The **Initializer list** is used for *initializing the data members* of a class or explicitly call the base class constructor before entering in the constructor body

(Not to be confused with `std::initializer_list` )

```
struct A {  
    int x, y;  
  
    A(int x1) : x(x1) {} // ": x(x1)" is the Initializer list  
                  // direct initialization syntax  
  
    A(int x1, int y1) : // ": x{x1}, y{y1}"  
                      // is the Initializer list  
                      x{x1},  
                      y{y1} {} // direct-list initialization syntax  
};                             // (C++11)
```

# Data Member Initialization

**const** and **reference** data members must be initialized by using the *initialization list* or by using *brace-or-equal-initializer* syntax (C++11)

```
struct A {  
    int      x;  
    const char y;  // must be initialized  
    int&      z;  // must be initialized  
    A() : x(3), y('a'), z(x) {}  
};  
  
struct B {  
    int      x = 3;  // equal-initializer (C++11)  
    int      y{4};  // brace initializer (C++11)  
    const char z = 'a'; // equal-initializer (C++11)  
    int&      w = x;  // equal-initializer (C++11)  
};
```

# Initialization Order ★

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

```
struct ArrayWrapper {  
    int* array;  
    int size;  
  
    A(int user_size) :  
        size{user_size},  
        array{new int[size]} {}  
        // wrong!!: "size" is still undefined  
};  
  
ArrayWrapper a(10);  
cout << a.array[4]; // segmentation fault
```

## 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

# 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 {};  
  
struct B {  
    B(A a) {}  
    B(int x, int y) {}  
    void f() {}  
};  
//-----  
  
B b( A() ); // "b" is interpreted as function declaration  
            // with a single argument A (*)() (func. pointer)  
// b.f()    // 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  
};
```

# Constructors and Inheritance

## Class constructors are never inherited

A *Derived* class must call *implicitly* or *explicitly* a *Base* constructor before the current class constructor

Class constructors are called in order from the top Base class to the most Derived class (C++ objects are constructed like onions)

```
struct A {  
    A() { cout << "A" };  
};  
struct B1 : A { // call "A()" implicitly  
    int y = 3; // then, "y = 3"  
};  
struct B2 : A { // call "A()" explicitly  
    B2() : A() { cout << "B"; }  
};  
B1 b1; // print "A"  
B2 b2; // print "A", then print "B"
```

# 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  
    }  
  
    A(int a1, float b1) : A(a1, b1, false) {} // delegate constructor  
    A(float b1)         : A(100, b1, false) {} // delegate constructor  
};
```



# explicit Keyword

## explicit

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

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

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

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

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

# Copy Constructor

---

# Copy Constructor

## Copy Constructor

A **copy constructor** `T(const T&)` is a constructor used to create a new object as a *deep copy* of an existing object

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

- Every class always defines an *implicit* or *explicit* copy constructor
- Even the copy constructor implicitly calls the *default* Base class constructor
- Even the copy constructor is considered a non-default constructor

## Copy Constructor Example

```
struct Array {  
    int size;  
    int* array;  
  
    Array(int size1) : size{size1} {  
        array = new int[size];  
    }  
  
    // copy constructor, ": size{obj.size}" initializer list  
    Array(const Array& obj) : size{obj.size} {  
        array = new int[size];  
        for (int i = 0; i < size; i++)  
            array[i] = obj.array[i];  
    }  
};  
  
Array x{100}; // do something with x.array ...  
Array y{x};   // call "Array::Array(const Array&)"
```

# 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 initialization  
A a3{a1}; // Copy list initialization  
A a3 = a1; // Copy initialization
```

- 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  
}
```

# Copy Constructor Usage Examples

```
struct A {  
    A() {}  
    A(const A& obj) { cout << "copy"; }  
};  
  
void f(A a) {} // pass by-value  
  
A g() { return A(); };  
  
A a;  
A b = a;      // copy constructor (assignment)    "copy"  
A c(b);       // copy constructor (direct)        "copy"  
f(b);         // copy constructor (argument)      "copy"  
g();          // copy constructor (return value)  "copy"  
A d = g();    // * see RVO optimization           (depends)
```

## Pass by-value and Copy Constructor

```
struct A {  
    A() {}  
    A(const A& obj) { cout << "expensive copy"; }  
};  
  
struct B : A {  
    B() {}  
    B(const B& obj) { cout << "cheap copy"; }  
};  
  
void f1(B b) {}  
void f2(A a) {}  
  
B b1;  
f1(b1); // cheap copy  
f2(b1); // expensive copy!! It calls A(const A&) implicitly
```

## Deleted Copy Constructor

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

- It has a member of reference/const type

```
struct NonDefault { int& x; }; // deleted copy constructor
```

- It has a non-static member/base class which has a deleted (or inaccessible) copy constructor

```
struct B { // deleted copy constructor  
    NonDefault a;  
};  
struct B : NonDefault {}; // delete copy constructor
```

- It has a base class with a deleted or inaccessible destructor
- The class has the move constructor (next lectures)



# Class Destructor

---

## Destructor [dtor]

A **destructor** is a special member function 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*

Syntax: `~T()` same name of the class and no return type

- Any object has exactly one *destructor*, which is always *implicitly* or *explicitly* declared
- C++20 The *destructor* can be `constexpr`

```
struct Array {  
    int* array;  
  
    Array() {    // constructor  
        array = new int[10];  
    }  
  
    ~Array() {    // destructor  
        delete[] array;  
    }  
};  
  
int main() {  
    Array a;    // call the constructor  
    for (int i = 0; i < 5; i++)  
        Array b; // call 5 times the constructor + 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.** From the most Derived to the top Base class

```
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"
}
```

# Defaulted Members

---

**C++11** The compiler can generate **default/copy/move constructors** and **copy/more assignment** operators

syntax: `A() = default`

implies `constexpr`

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`)

```
struct A {  
    A(int v1) {}    // delete implicitly-defined default ctor  
                   // because a user-provided constructor is  
                   // defined  
  
    A() = default; // now, A has the default constructor  
};
```

//-----

```
struct B {  
protected:  
    B() = default; // now it is protected  
  
    B(const B&) = default; // now it is protected  
};
```

```
class A {  
    int x = 3;  
    int y;  
public:  
    A(int x1) : x{x1} {}    // user-provided constructor  
                           // disables default constructors  
  
    A() = default;          // initializes its members  
                           // x = 3; y is undefined  
  
    A(const A&) = default;  // copies its members  
};  
  
A a1;    // x = 3;  
a1  = 4; // x = 4  
A a2 = a1; // b2.x = 4
```



# Class Keywords

---

# this Keyword

## 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;  
    }  
};
```

## 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 only access `static` class members
- A non-`static` member function can access `static` class members
- Non-const `static` data members cannot be *directly* initialized inline

## 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
};
```

```
struct A {  
    int y = 2;  
    static int x; // declaration  
  
    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; // definition  
//-----  
  
A a;  
a.h(); // return 3  
A::x++;  
cout << A::x; // print 4  
cout << A::f(); // print 8
```

## 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  
        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 Keyword

## 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;  
};  
const A a;  
// a.x = 3; // 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

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

```
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)  
}
```

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

# delete Keyword

## 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(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
```