

Modern C++ Programming

7. OBJECT-ORIENTED PROGRAMMING I

CLASS CONCEPTS

Federico Busato

2025-01-14

1 C++ Classes

- RAII Idiom

2 Class Hierarchy

3 Access specifiers

- Inheritance Access Specifiers
- When Use `public/protected/private/` for Data Members?

4 Class Constructor

- Default Constructor
- Class Initialization
- Uniform Initialization for Objects
- Delegate Constructor
- `explicit` Keyword

5 Copy Constructor

6 Class Destructor

7 Defaulted Constructors, Destructor, and Operators
(=default)

8 Class Keywords

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

C++ Classes

C Structure

A **C 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 functions as members

struct vs. class in C++

Structures and *classes* are *semantically* equivalent in **C++**. However, the keywords should be used to distinguish between different semantics:

- `struct` represents *passive* objects, namely the *physical state* (set of data)
- `class` represents *active* objects, namely the *logical state* (data abstraction)

Class Members - Data and Function Members

Data Member

Data within a class are called **data members** or **class fields**

Function Member

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

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 released 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  
}
```

struct/class Members

```
struct B {  
    void g() { cout << "g"; } // function member  
};  
  
struct A {  
    int x; // data member  
    B b; // data member  
    void f() { cout << "f"; } // function member  
};  
  
A a;  
a.x;  
a.f();  
a.b.g();
```

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 functions 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();
```

```
struct A {};  
struct B : A {};  
  
void f(A a) {}      // copy  
void g(B b) {}      // copy  
  
void f_ref(A& a) {} // the same for A*  
void g_ref(B& b) {} // the same for B*  
  
A a;  
B b;  
f(a); // ok, also f(b), f_ref(a), g_ref(b)  
g(b); // ok, also g_ref(b), but not g(a), g_ref(a)  
  
A a1 = b;    // ok, also A& a2 = b  
// B b1 = a; // compile error
```


Access specifiers

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 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	→	public	→	public
protected				protected
private				\
public	→	protected	→	protected
protected				protected
private				\
public	→	private	→	private
protected				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
```

When Use `public/protected/private/` for Data Members?

When use `protected/private` data members:

- They are not part of the interface, namely the *logical state* of the object (not useful for the user)
- They must preserve the `const` correctness (e.g. for pointer), see Advanced Concepts I

When use `public` data members:

- They can potentially change any time
- `const` correctness is preserved for values and references, as opposite to pointers. *Data members* should be preferred to *member functions* in this case

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 argument

Every class has always either an *implicit*, *explicit*, or *deleted* 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{}; // call the default constructor, equivalent to: A a;
```

Note: an *implicit* default constructor is `constexpr`

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 "AAA"  
  
A* ptr = new A[4]; // print "AAAA"
```

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 non-static member/base class 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 non-static member/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 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)
```

In-Class Member_INITIALIZER

C++11 In-class non-static data members initialization (NSDMI) allows initializing the data members where they are declared. A user-defined constructor can be used to override their default values

```
struct A {  
    int      x      = 0;          // in-class member initializer  
    const char* str = nullptr;    // in-class member initializer  
  
    A() {} // "x" and "str" are well-defined if  
           // the default constructor is called  
  
    A(const char* str1) : str{str1} {}  
};
```

Data Member Initialization

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

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


Initialization Order

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

```
struct ArrayWrapper {  
    int* array;  
    int size;  
  
    ArrayWrapper(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 for Objects

Uniform Initialization (C++11)

Uniform Initialization `{}`, also called *list-initialization*, is a way to fully initialize any object independently of 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
```

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

```
struct A {};  
  
struct B {  
    B(A a) {}  
    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{} };
```

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

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

```
struct A {  
    int    a;  
    float  b;  
    bool   c;  
    // 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

The `explicit` keyword specifies that a *constructor* or *conversion operator* (C++11) does not allow implicit conversions or copy-initialization from single arguments or braced initializers

The problem:

```
struct MyString {  
    MyString(int n);           // (1) allocates n bytes for the string  
    MyString(const char *p);  // (2) initializes starting from a raw string  
};  
MyString string = 'a';        // calls (1), implicit conversion!!
```

`explicit` cannot be applied to *copy/move-constructors*


```
struct A {  
    A() {}  
    A(int) {}  
    A(int, int) {}  
};  
void f(const A&) {}
```

```
A a1 = {};           // ok  
A a2(2);             // ok  
A a3 = 1;            // ok (implicit)  
A a4{4, 5};          // ok. Selected A(int, int)  
A a5 = {4, 5};       // ok. Selected A(int, int)  
f({});               // ok  
f(1);                 // ok  
f({1});               // ok
```

```
struct B {  
    explicit B() {}  
    explicit B(int) {}  
    explicit B(int, int) {}  
};  
void f(const B&) {}
```

```
// B b1 = {};         // error implicit conversion  
B b2(2);              // ok  
// B b3 = 1;          // error implicit conversion  
B b4{4, 5};           // ok. Selected B(int, int)  
// B b5 = {4, 5};     // error implicit conversion  
B b6 = (B) 1;         // OK: explicit cast  
// f({});             // error implicit conversion  
// f(1);               // error implicit conversion  
// f({1});            // error implicit conversion  
f(B{1});              // ok
```

Copy Constructor

Copy Constructor

Copy Constructor

A **copy constructor** `T(const T&)` creates 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 → direct initialization  
}
```

Copy Constructor Details

- Every class always defines an *implicit* or *explicit* copy constructor, potentially *deleted*
- The copy constructor implicitly calls the *default* Base class constructor
- Even the copy constructor is considered a *user-defined* constructor
- The copy constructor doesn't have template parameters, otherwise it is a standard member function
- The copy constructor must not be confused with the assignment operator

`operator=`

```
MyStruct x;  
MyStruct y{x}; // copy constructor  
y = x;         // call the assignment operator=, not the copy constructor  
               // → copy initialization, see next lecture
```

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 one having the same type
 - Direct constructor
 - Assignment operator

```
A a1;  
A a2(a1);    // Direct copy initialization  
A a3{a1};    // Direct copy initialization  
A a4 = a1;   // Copy initialization  
A a5 = {a1}; // Copy list 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 g1(A& a) { return a; }  
  
A g2()      { return A(); }  
  
A a;  
A b = a;    // copy constructor (assignment)    "copy"  
A c(b);     // copy constructor (direct)         "copy"  
f(b);       // copy constructor (argument)       "copy"  
g1(a);      // copy constructor (return value)    "copy"  
A d = g2(); // * see RVO optimization (Advanced Concepts I)
```

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 non-static member/base class 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 non-static member/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 Constructors, Destructor, and Operators (=default)

C++11 The compiler can automatically generate

- **default/copy/move constructors**

`A() = default`

`A(const A&) = default`

`A(A&&) = default`

- **destructor**

`~A() = default`

- **copy/move assignment operators** `A& operator=(const A&) = default`

`A& operator=(A&&) = default`

- **spaceship operator**

`auto operator<=>(const A&) const = default`

`= default` implies `constexpr`, but not `noexcept` or `explicit`

When the compiler-generated constructors, destructors, and operators are useful:

- Change the visibility of non-user provided constructors and assignment operators (`public`, `protected`, `private`)
 - Make visible the declarations of such members
-

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

When the compiler-generated constructor is useful:

- Any user-provided constructor disables implicitly-generated default constructor
- Force the default values for the class data members


```
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  
};
```

```
struct C {  
    int x;  
    // C() {}          // 'x' is undefined  
    C() = default;    // 'x' is zero  
};
```

Class Keywords

this Keyword

this

Every object has access to its own address through the 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

```
struct A {  
    int x;  
  
    int f() { return x; }  
  
    static int g() { return 3; } // g() cannot access 'x' as it is associated  
};                               // with class instances  
A a{4};  
a.f(); // call the class instance method  
A::g(); // call the static class method  
a.g(); // as an alternative, a class instance can access static class members
```

```
struct A {  
    static const int      a = 4;           // C++03  
    static constexpr float b = 4.2f;       // better, C++11  
    // static const float  c = 4.2f;       // only GNU extension (GCC)  
  
    static constexpr int f() { return 1; } // ok, C++11  
    // static const int    g() { return 1; } // 'const' refers to the return type  
};
```

Non-`const` `static` data members cannot be *directly* initialized “inline” before C++17 (see also Translation Units I lecture)

```
struct A {  
    // static int      a = 4; // compiler error  
    static int      a;      // ok, declaration only  
    static inline int b = 4; // ok from C++17  
  
    static int f() { return 2; }  
    static int g();          // ok, declaration only  
};  
  
int A::a = 4;                // ok  
int A::g() { return 3; }     // ok  
// NOTE: link error (undefined reference) without the two previous definitions
```

```
struct A {  
    static int x; // declaration  
  
    static int f() { return x; }  
  
    static int& g() { return x; }  
};  
int A::x = 3; // definition  
  
//-----  
  
A::f();    // return 3  
A::x++;  
A::f();    // return 4  
A::g() = 7;  
A::f();    // return 7
```

- A `static` member function can only access `static` class members
- A non-`static` member function can access `static` class members

```
struct A {  
    int          x = 3;  
    static inline int y = 4;  
  
    int          f1() { return x; } // ok  
    // static int f2() { return x; } // compiler error, 'x' is not visible  
    int          g1() { return y; } // ok  
    static int g2() { return y; } // ok  
  
    struct B {  
        int h() { return y + g2(); } // ok  
    }; // 'x', 'f1()', 'g1()' are not visible within 'B'  
};
```


Const member functions

Const member functions (**inspectors** or **observers**) are functions marked with `const` that are not allowed to change the object logical state

The compiler prevents from inadvertently mutating/changing the data members of *observer* functions → All data members are marked `const` within an **observer** method, including the `this` pointer

- The *physical state* can still be modified, see `mutable` member functions ~→
- Member functions without a `const` suffix are called *non-const member functions* or **mutators/modifiers**

```
struct A {  
    int x = 3;  
    int* p;  
  
    int get() const {  
        // x = 2;           // compile error class variables cannot be modified  
        // p = nullptr;    // compile error class variables cannot be modified  
        p[0] = 3;          // ok, p is 'int* const' -> its content is  
                           // not protected  
        return x;  
    }  
};
```

A common case where `const` member functions are useful is to enforce const correctness when accessing pointers, see Advanced Concepts I, Const Correctness

The `const` keyword is part of the function 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` data members of *const* class instances are modifiable. They should be part of the object *physical state*, but not of the *logical state*

- 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 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
```

using Keyword for Inheritance

The `using` keyword can be also used to change the *inheritance attribute* of data members and 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 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
};
```


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

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
```