

# Modern C++ Programming

## 5. C++ OBJECT ORIENTED PROGRAMMING

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2018, v1.0



# Agenda

## ■ C++ Classes

- Class hierarchy
- Inheritance attributes
- Class constructor
- Default constructor
- Class initialization
- Copy constructor
- default keyword
- Class destructor

## ■ Class keyword

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

## ■ 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

# C++ Classes

---

## C++ Class

**Classes** extend the concept of data structures: they can contain 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

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

# C++ Classes

```
struct A;      // class declaration (incomplete type)

class B {
    void g() { cout << "g"; } // function member definition
};

struct A {     // class definition
    int x;     // field/variable member
    void f(); // function member (declaration)

    B b;      // b class is a field of A
    using T = B; // alias of B inside A
};

void A::f() { cout << "f"; } // function member definition

int main() {
    A a;
    std::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** class

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

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

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;
    cout << base.value;    // print 3
    cout << derived1.data; // print 4
    cout << derived2.f();  // print 4
}
```

`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



member declaration	inheritance	derived classes
public protected private	public	public protected \ private
public protected private	protected	protected protected \ private
public protected private	private	private private \ public

- 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
}
```

## Constructor [ctor]

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

- 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)

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

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

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

## Default Constructor

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

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

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

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

- It has a member of reference/`const` type
- It has a 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

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

class B {}; // implicit-declared private default constructor

class C {
public:
    C() { // user-defined default constructor
        std::cout << "C";
    }
};

struct D {
    int& a; // implicit-deleted default constructor
};

int main() {
    A a1; // call the default constructor
    // A a2(); // interpreted as a function declaration!!
    // B b; // compile error!! private
    C c; // ok, print "C"
    C array[3]; // print three times "C"
    // D d; // compile error!! deleted
}
```

(Any) Member data should be initialized by constructors with the **initialization lists** or by using **brace-or-equal-initializer** syntax  
const and *reference* data members must be initialized by using the *initialization lists*

```
struct A {
    char      a;
    const float b;
    const int  c = 3;           // default initialization
    int*      ptr { nullptr }; // default initialization(C++11)

    A(char a1) : a(a1), b(1.2f) {} // direct initialization

    A() : a{'x'}, b{1.2f} {} // uniform initialization(C++11)

    // A() : c('a') {}           // compile error!! "b" is const
};
```

## C++11

### Uniform Initialization

**Uniform 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

To not confuse with narrowing conversion

Full details:

[mbevin.wordpress.com/2012/11/16/uniform-initialization/](http://mbevin.wordpress.com/2012/11/16/uniform-initialization/)



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

A f() {
    return { 1, 2 }; // ok, works also for B (call B(int, int))
}

B g(A a) {
    B b( A() ); // interpreted as function declaration
// 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
};
```

## C++11

The `explicit` keyword specifies that a constructor or conversion function doesn't allow implicit conversions or copy-initialization

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

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

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

## 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

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

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 slides)

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

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

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

int main() {
    A x(100), y(10);
    x = y;    // call "A::A(const A&)" copy constructor
}
```

```
struct A {
    int x;
    A(const A& obj) : x(obj.x) {} // user-defined copy constructor
                                // -> delete default ctor

    A() {} // user-defined
};

struct B : A {
    int array[3];
    B() : A(), array{1, 2, 3} {}
// B(const B& obj) ... // implicitly-declared copy constructor
};

int main() {
    B x;
    B y = x; // call "A" user-declared copy constructor, then
            // call "B" implicitly-declared copy constructor
// the value of y.array[0] is 1
    B z = B(); // ok, call "B" implicitly-declared copy ctor
}
```

## The copy constructor is used to:

- Initialize one object from another having the same type
  - Direct constructor
  - Assignment operator
- Copy an object which is passed by value as input parameter of a function
- Copy an object which is returned as result from a function

```
class A {  
public:  
    A() {}  
    A(const A& obj) {}  
};
```

```
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)  
    // copy constructor (return value)  
    A d = g(); // but see RVO optimization  
}
```

In **C++11**, we can use the compiler-generated version of default/copy constructors

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

When compiler-generated constructor is useful:

- Define any constructor different from the default constructor disables implicitly-generated default constructor
- Default/copy constructors from classes are marked `private`

```
struct A {
    int v;
    A(int v1) : v(v1){} // delete implicitly-defined default ctor
    A() = default;      // now A has the default constructor
};

class B { // default/copy constructor marked private
public:
    B()          = default; // default constructor now is public
    B(const B&) = default; // copy constructor now is public
};           // "B() = default" equal to "B() : A() {}"
           // "B(const B&) = default" equal to
int main() { // "B(const B& b) : A(b.x) {}"
    B x, y;
    x.v = 4 ;
    y = x;    // "y.x" has value 3
}
```



# Default vs. Copy Constructor

```
struct A{
    A()          { std::cout << "default"; }
    A(const A&) { std::cout << "copy"; }
};

void f(A a) {}
void g(A& a) {}
A h() { return A(); } // default constructor "default"

int main() {
    A x, y; // default constructor "default"
    A z = x; // copy constructor "copy"
    x = y; // copy assignment operator (see next slides)
    f(x); // copy constructor "copy"
    g(x); // nothing
    A j = h(); // copy constructor, but RVO (copy elision)
}
```

## 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 expression is applied to a pointer to the object of that class

- A destructor will have exact same name as the class prefixed with a tilde (~)
- 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

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

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

# RAII Idiom - Resource Acquisition is Initialization

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

Implication1: C++ programming language does not require the garbage collector!!

Implication2 :The programmer has the responsibility to manage the resources

**RAII Idiom consists in three steps:**

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

# Class Keywords

---

# this Keyword

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

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

`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 is initialized to zero/default useless if no user-initialization is provided
- It can be initialized (defined) only once
- Static data members cannot be `inline` initialized



```
struct A {
    int y = 2;
    // static int x = 3;    // compile error!! inline initialization
    static int x;        // declaration
    static int z[];      // array declaration (incomplete type)
    static int g();      // function 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;           // static variable definition
int A::z[] = {1, 2, 3}; // static array definition
int A::g() { return z[1]; } // static function definition

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

## Constant static members

If a static data member is declared as `const` or `constexpr`, then it can be initialized inline and only through a constant expression

```
constexpr int f(int a) { return a * 2}

struct A {
    static const int x = f(3);           // ok
    static const float y;               // ok, declaration
    // static const char* z = "ab";     // compile error!! "static const"
    // static const int w[] = {1, 2};   // cannot be initialized "inline"
    // static const float v = 3.3f;     // with arrays, pointers,
                                        // and floating-point

    static constexpr char* z = "ab";    // ok, but must be initialized
    static constexpr int w[] = {1, 2};  // "inline"
    static constexpr float v = 3.3f;    //
};

const float A::y = 3.3f;                // ok, definition
```

## 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 prevent 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; }
    int get1() const { return x; }
    int get2()      { return x; }
};

int main() {
    A a1;
    std::cout << a1.get1();    // ok
    std::cout << a1.get2();    // ok
    const A a2;
    std::cout << a2.get1();    // ok
    // std::cout << a2.get2(); // compile error!! "a2" is const
}
```

# 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 your class interface*

```
struct A {  
    int      x = 3;  
    mutable int y = 5;  
};  
  
int main() {  
    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

```
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`

```
class A; // class declaration

class B {
    int y = 3; // private
    int f(A a);
};

class A {
    friend class B;
    int x = 3; // private
    int f(B b);
};

int B::f(A a) { return a.x; } // ok, B is friend of A
int A::f(B b) { return b.y; } // compile error!! (no symmetric)

class C : B {
    int f(A a) { return a.x; } // compile error!! (no 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 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  
  
int main() {  
    // f(A()); // compile error!! (marked as deleted)  
}
```

# Polymorphism

---

# Polymorphism

## Polymorphism

In object-oriented programming, **polymorphism** (meaning “having multiple forms”) is the capability of an object of *mutating* its behavior in accordance with a 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)

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

In C++ **late binding** 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)

```
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

```
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"
    h(b);           // print "B"
    p(&b);          // print "B"
}
```



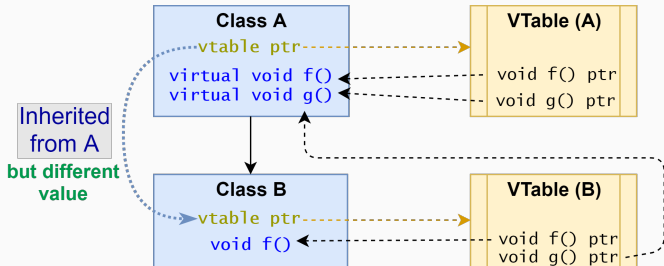
# Virtual Table

## 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 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

## C++11

### override Keyword

The `override` keyword 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

## C++11

### final Keyword

The `final` keyword 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)

### Don't 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)

### Don't 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* zz = new B();
    zz->f();         // !!! print "B::5" // the virtual table of A
                    // contains f(int i = 5) and
    A* ww = new B(); // contains f(int i = 5) and
    ww->g();         // !!! print "B::5" // g(int i = 5) but it points
                    // to B implementations
}
```

# Pure Virtual Method

## 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  
};
```



# Pure Virtual Method

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 don't 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 constructor
    virtual void g(int x) = 0; // at least one pure virtual
protected:
    int x;           // additional data
};
```

## Virtual Methods (Virtual Constructor)

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

**Operator overloading** is a specific 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.y);
    }
};

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`

- 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 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

```
class A {};  
  
class B : public A {  
    bool operator==(const A& x) { return true; }  
};  
  
class C : public A {  
    friend bool operator==(const A& x, const A& y);  
};  
bool C::operator==(const A& x, const A& y); { return true; }  
  
int main() {  
    A a; B b; C c;  
    b == a; // ok  
    // a == b; // compile error!! // friend is useful to access  
    // private fields  
  
    c == a; // ok  
    a == c; // ok  
}
```



## Special Operators (iostream operator<<)

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

```
#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

```
#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

```
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:

```
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);
}
```

## Special Operators (assignment operator=)

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"]
}
```

## Special Operators (assignment operator=)

- 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=)

- 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

# 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

```
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)
};
```

```
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 };
}
```

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

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
};
```

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

```
struct StandardLayout1 {
    StandardLayout1(); // user-provided constructor
    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

