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 ≪
  - Operator `operator()`
  - Operator `operator=`
- **Special Objects**
  - Aggregate
  - Trivial class
  - Standard-layout class
  - Plain old data type

C++ Classes

C++ Classes

Definition (C++ Class)

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

Definition (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.

Definition (`struct` vs. `class`)

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

C++ Classes

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

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

struct A {   // class definition
   int a;    // 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::T obj; // equal to B obj;
}
```

Definition (Child/Derived Class or Subclass)

New class that inherits properties of the base class is called a derived class

Definition (Parent/Base Class)

A parent class is the closest class that we derived from to create the one we are referencing as the child class

Definition (Extend a 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

Class Hierarchy


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

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

struct B : 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 derived;
    C child;
    cout << base.value;   // print 3
    cout << derived.data; // print 4
    cout << child.f();    // print 4
}
```

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

- **`public:`** The public members of a base class can be accessed by members of that base class, members of its derived class as well as the members which are outside the base class and derived class
- **`protected:`** The protected members of a base class can be accessed by members of base class as well as members of its derived class
- **`private:`** The private members of a base class can only be accessed by members of that base class

Inheritance Attributes

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

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

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

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

class B : public A { // without public, B inherits
};                   // "a" and f() as private members

int main() {
    B derived;
    cout << derived.f(); // print 3
    cout << derived.b;   // compile error : private
}
```

Definition (Constructor [ctor])

A **class constructor** is a *special* member function of a class that is executed whenever we create new objects of that class

- A constructor has exact same name as the class
- A constructor does not have any return type
- A constructor is useful for setting initial values for **any** member variables
- 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)

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

class B : A {
    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
    A d {1};   // uniform initialization (C++11)
}
```

Definition (Default Constructor)

A **default constructor** is a constructor which can be called with no arguments

Every class always define an implicit or explicit default constructor.

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

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

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

```
class A {
    // implicit-declared default constructor
};

struct B {
    B() {   // user-defined default constructor
        /* do something */
    }
};

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

int main() {
    A a;      // call the default constructor
    A b();    // interpreted as a function declaration!!
    C c;      // compiler error
}
```

(Any) Member variables should be initialized by constructors with **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 c1) : c(c1), b(1.2f) {}  // direct initilization

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

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

C++11

Definition (Uniform Initialization)

Uniform Initialization expands on the Initializer List syntax, to provide a syntax that allows for fully uniform type initialization that works on any object

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

Full details:
`mbevin.wordpress.com/2012/11/16/uniform-initialization/`

```cpp
struct A {
    int a1, a2;
};

class B {
    int b1, b2;
public:
    B() :                b1(1),  b2(2)  {}
    B(int x1, int x2) : b1(x1), b2(x2) {}
};

A f() {
    return { 1, 2 }; // ok, works also for B
}
B f() {
    return B();  // B() maybe also a function
}                // ``Most Vexing Parse" problem
                 // solved with B{}
struct C {
//  B b (1, 2);   // compile error
    B b { 1, 2 }; // ok, call the constructor
};
```

C++11

Definition

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

Definition (Copy Constructor)

A copy constructor is a special type of constructor used to create a new object as a copy of an existing object.

Every class always define an implicit or explicit copy constructor.

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

The default 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)

```cpp
class A {
public:
    int x;
    A() {}
    A(const A& obj) : x(obj.x) {} // User-defined copy constructor
};

class B : public A {
public:
    int array[3];
    B() : array{1,2,3} {}
};

int main() {
    B c, d;
    c = d;  // call "B" user-declared copy constructor, then
            // call "A" implicitly-declared copy constructor
}
```

Copy Constructor

The copy constructor is used to:

- Initialize one object from another of the same type
  - Direct constructor
  - Assignment operator
- Copy an object to pass it as an argument to a function
- Copy an object to return it from a function

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

void f(A a) {}

void g() { return A(); };
```

```cpp
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, it is possible to use the compiler-generated version of special functions as default/copy constructors, so you don't need to specify a body

```
struct A {
    int a;
    A() : a(1) {}
};

struct B : A {
    B() = default;              // call A()
    B(const B& b) = default; // copy constructor
};
```

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 {
    A(int a) {} // disable implicitly-defined default construtor
    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& b) = default; // copy constructor now is public
};
```

Definition (Destructor [dtor])

A **destructor** is a *special* member function of a class that is executed whenever an object of it's class goes 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 ($\sim$)
- A destructor does not have any return type
- Each object has exactly one destructor
- A destructor is useful for releasing resources before the class instance goes out of scope or it is deleted

Class Destructor

```
struct A {
    int* array;

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

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

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

Class destructor is <u>never</u> 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;
    ~C() { std::cout << "C"; }
};

int main() {
    B 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.

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

RAII Idiom consists in three steps:

- Encapsulate a resource into a class (in the constructor usually).
- 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

Definition

Every object in C++ has access to its own address through a pointer called `this` pointer

The `this` const pointer is a hidden parameter implicitly added to any member function. In general, it is not needed

When `this` is necessary:

- 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;
    }
};
```

Definition (`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

- It can be called/used without an instance of a class
- A *static* member function cannot access non-static class members
- All *static* data is initialized to zero/default 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[];       // declaration (incomplete type)
    static int g();       // declaration

    static int f() { return x * 2; }
    // static int f() { return y; } // compile error (non-static)
};
int A::x   = 3;             // definition
int A::z[] = {1, 2, 3};     // definition
int A::g() { return z[1]; } // definition

int main() {
    A::x++;
    cout << A::x;    // print 4
    cout << A::f();  // print 8
}
```

Constant static members

If a static data member of is declared `const` or `constexpr`, it can be initialized with an initializer in which every expression is a constant expression

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

struct A {
    static const int   x = f(3);        // ok
    static const int   y;               // ok
    static const char* z = "ab";        // ok

    // static constexpr float v;        // compile error
    static constexpr int v[] = {1, 2}; // ok
};
const int A::y = 3;
```

Definition (Const member functions)

Const member functions, or (**inspectors**), should be used to mean the method won't change the object's state.

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

The compiler prevent callers from inadvertently mutating/changing object with functions marked as `const`

```
class A {
    int x = 3;
public:
    int get() const {
        // x = 2;   // compile error
        return x;
    }
};
```

The `const` keyword is part of the functions signature which means that you can implement two similar methods, one which is called when the object is `const`, and one that isn't

```
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

Definition (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;
}
```

Definition (friend Class)

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

Friendship properties:

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

```cpp
class A;    // class declaration

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

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

int B::f(A a) { return a.x; } // ok
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 Keyword

Definition (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;
}
```

Definition (`delete` Keyword)

The `delete` keyword explicitly marks a member function as deleted and any use results in a compiler error. If applied to Copy/Move constructor or assignment prevents the compiler to implicitly generate these functions

Using the default copy/move functions for a class in a hierarchy can produce unexpected results. The keyword `delete` prevents these kind of 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
int main() {
    f(A());     // compile error (marked as deleted)
}
```

Polymorphism

Polymorphism

Definition (Polymorphism)

In object-oriented programming, **polymorphism** (meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in *different contexts* - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

- At run time, objects of a *derived class* may be treated as objects of a *base class*
- Base classes may define and implement `virtual` methods, and derived classes can `override` them, which means they provide their own definition and implementation 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)

Polymorphism (the problem)

```cpp
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 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)

Polymorphism (`virtual` method)

```cpp
struct A {
    virtual 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);    // NOW, print "B"!!!
    h(b);    // print "B"
}
```

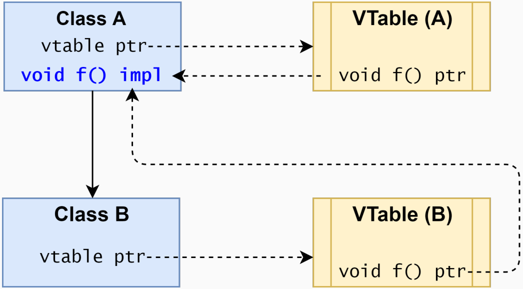
Virtual Table

Definition (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();
}

sizeof(A) = sizeof(double) + 1 * sizeof(pointer)
```

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`

C++11

Definition (`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);
};

struct B : A {
    void f(int a) override;         // ok
    void f(float a);                // (still) very dangerous!!
//  void f(float a) override;      // compile error
//  void f(int a) const override; // compile error
};
// f(3.3f) has different behavior between A and B
```

final Keyword

C++11

Definition (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)
};

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
    virtual void f(int a) {}
};
struct B : A {
    ~B() { std::cout << "B"; }
};

void g(A* a) {
    delete a;
}

int main() {
    B* b = new B;
    g(b);     //without virtual, g() prints only "A"
}
```

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?

    virtual void f() { std::cout << "A"; }
};

struct B : A {
    B() : A() {}

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

int main() {
    B 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 x = 3) {
        std::cout << "A";
    }
};

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

int main() {
    B b;
    b.f(); // print "A", not "B"!!
}
```

Pure Virtual Method

Definition (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() {} // pure virtual implementation 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(); // 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 contructor
   virtual void g(int x) = 0; // at least one pure virtual
protected:
   int x;           // additional data
};
```

Virtual Methods (Virtual Contructor)

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

Definition (Operator Overloading)

Operator overloading is a specific case of polymorphism in which some operators are treated as polymorphic functions and as such have different behaviors depending on the types 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.x);
    }
};

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`

Notes

- Increment, Decrement: *Prefix* and *Postfix* notation

```
struct A {
    A& operator++() {  // prefix:  ++obj
        ...
        return *this;
    }
    A& operator++(const A& a); // postfix:  obj++
};
```

- Array subscript operator accepts anything (not only integer)

```
struct A {
    some_t&       operator[](char a);       // write
    const some_t& operator[](char a) const; // read
};
```

- 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& a) { return true; }
};

class C : public A {
    friend bool operator==(const A& a, const A& b);
};

bool C::operator==(const A& a, const A& b); { return true; }

int main() {
    A a;  B b;  C c;
    b == a; // ok
//  a == b; // compile error // friend is useful to access
    c == a; // ok             // private fields
    a == c; // ok
}
```

Special Operators (iostream `operator<<`)

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

```cpp
#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(arrray, array + 3, 0, 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

Full story: `stackoverflow.com/questions/5695548`

C++ Special Objects

Definition (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

Full story: `stackoverflow.com/questions/4178175`

Aggregate (examples)


```
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)
};
```

Aggregate (examples)


```
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 };
}
```

Trivial Class

Definition

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

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

Trivial Class (examples)

```
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
};
```

Definition

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

```cpp
struct StandardLayout1 {
    StandardLayout2(); // user-provided contructors
    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