Modern C++ Programming

9. Object-Oriented Programming II

Polymorphism and Operator Overloading

Federico Busato

2025-01-22

Table of Contents

Table of Contents

3 Operator Overloading

Table of Contents

Polymorphism

Polymorphism

Polymorphism

Polymorphism (meaning "having multiple forms") is the capability of an entity of *mutating* its behavior in accordance with the specific usage *context*

Polymorphism dispatch can be implemented at

- **Compile-time** (`static polymorphism`): when the called instance is known before the program start
- **Run-time** (`dynamic polymorphism`): when the called instance is known only during the execution, i.e. depends on run-time values

In C++, the term **polymorphic** is strongly associated with dynamic polymorphism (*overriding*)

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
  - the program can jump directly to the function address
- In **Late Binding** or *Dynamic Binding* or *Run-time binding*, the run-time identifies the type of object at execution-time and *then* matches the function call with the correct function definition
  - 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)

C++ achieves **late binding** by declaring a `virtual` function

Polymorphism Forms

- **Ad-hoc polymorphism**: when it involves to a set of individually specified types, e.g. function overloading

```cpp
void f(int);
void f(double);
```

- *Parametric polymorphism*: when it involves generic types, e.g. templates

```cpp
template<typename T>
void f(T);
```

- *Subtyping*: when it operates on elements of subtypes, e.g. virtual functions

```cpp
// B : A
void f(A*); // also works for B if the called function are virtual
```

C++ Mechanisms for Polymorphism 1/2

- *Preprocessing*

```cpp
#define ADD(x, y) x + y  // ADD(3, 4) or ADD(3.0, 4.0)
```

- *Function/Operator overloading*

```cpp
void f(int);
void f(double);
```

- *Templates*

```cpp
template<typename T>
void f(T);            // f(3) or f(3.0)
```

- *Virtual functions* (see next slides)

C++ Mechanisms for Polymorphism 2/2

Mechanism	Implementation	Form
Preprocessing	`static`	Parametric
Function/Operator overloading	`static`	Ad-hoc
Template	`static`	Parametric
Virtual function	`dynamic`	Subtyping

Dynamic Polymorphism in C++

- At <u>run-time</u>, objects of a *base class* behave as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

```
struct A {
    void f()  { cout << "A"; }
};
struct B : A {
    void f() { cout << "B"; }
};
void g(A& a) { a.f(); } // accepts A and B
                        // note: g(B&) would only accept B
A a; B b;
g(a);    // print "A"
g(b);    // print "A" not "B"!!!
```

Polymorphism - `virtual` method

```
struct A {
    virtual void f() { cout << "A"; }
}; // now "f()" is virtual, evaluated at run-time

struct B : A {
   void f() override { cout << "B"; }
   // now B::f() overrides A::f(), run-time dispatch
   // 'virtual void f()' is also valid
}; // 'override' is a c++11 feature, more details in the next slides

void g(A& a) { a.f(); } // accepts A and B

A a;
B b;
g(a); // print "A"
g(b); // NOW, print "B"!!!
```

When `virtual` works

```
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() override { cout << "B"; }
};

void f(A& a) { a.f();  } // ok, print "B"
void g(A* a) { a->f(); } // ok, print "B"
void h(A  a) { a.f();  } // does not work with pass-by value!! print "A"

B b;
f(b);  // print "B"
g(&b); // print "B"
h(b);  // print "A" (cast to A)
```

Polymorphism Dynamic Behavior

```cpp
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() override { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

get_object(true)->f();  // print "A"
get_object(false)->f(); // print "B"
```

Virtual Table

vtable

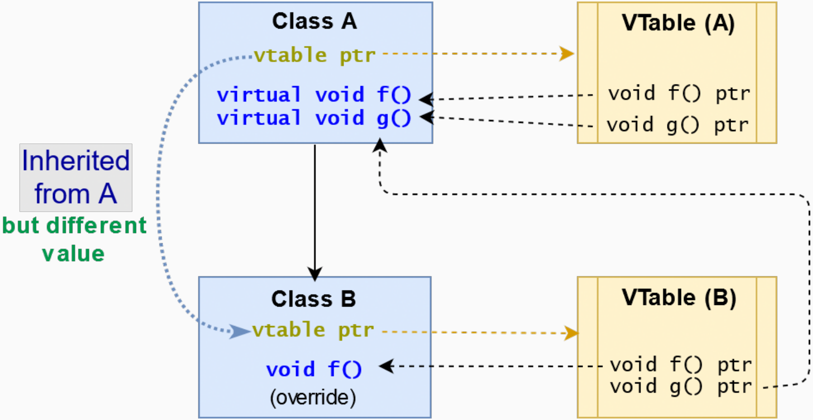
The **virtual table** (`vtable`) is a lookup table of functions used to resolve function calls and support *dynamic dispatch* (late binding)

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)

```
struct A {
    virtual void f();
    virtual void g();
};

struct B : A {
    void f() override;
};
```



Does the `vtable` really exist? (answer: YES)

```
struct A {
    int x = 3;
    virtual void f() { cout << "abc"; }
};

A* a1 = new A;
A* a2 = (A*) malloc(sizeof(A));

cout << a1->x;  // print "3"
cout << a2->x;  // undefined value!!
a1->f();        // print "abc"
a2->f();        // segmentation fault ☠
```

Lesson learned: Never use `malloc` in C++

Virtual Method Notes

`virtual` classes allocate one extra pointer (hidden)

```
struct A {
    virtual void f1();
    virtual void f2();
};

class B : A {};

cout << sizeof(A); // 8 bytes (vtable pointer)
cout << sizeof(B); // 8 bytes (vtable pointer)
```

`override` Keyword 1/2

`override` Keyword (C++11)

The `override` keyword ensures that the function is `virtual` and is <u>overriding</u> a `virtual` function from a base class

- It forces the compiler to check the base class to see if there is a `virtual` function with this <u>exact</u> signature
- `override` clearly expresses the intent of the function, making the code easier to understand

`override` implies `virtual` (`virtual` should be omitted)

`override` Keyword

```
struct A {
    virtual void f(int a);              // a "float" value is casted to "int"
};                                      // ***

struct B : A {
    void f(int   a) override;           // ok
    void f(float a);                    // (still) very dangerous!!
                                        // ***

//  void f(float a) override;           // compile error not safe
//  void f(int   a) const override; // compile error not safe
};

// *** f(3.3f) has a different behavior between A and B
```

final Keyword (C++11)

The `final` keyword prevents inheriting from classes or 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() { 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;           }
};
//--------------------------------------------------------------
void destroy(A* a) {
    delete a;    // call ~A()
}
B* b = new B;
destroy(b); // without virtual, ~B() is not called
            // destroy() prints only "A" -> huge memory leak!!
```

Virtual Methods (Common Error 2)

Do not call virtual methods in constructor and destructor

- *Constructor*: The derived class is not ready until constructor is completed
- *Destructor*: The derived class is already destroyed

```cpp
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() { cout << "Explosion"; }
};
struct B : A {
    B() = default; // call A(). Note: A() may be also implicit

    void f() override { cout << "Safe"; }
};

B b; // call B(), print "Explosion", not "Safe"!!
```

Virtual Methods (Common Error 3)

Do not use default parameters in virtual methods

Default parameters are not inherited

```
struct A {
    virtual void f(int i = 5) { cout << "A::" << i << "\n"; }
    virtual void g(int i = 5) { cout << "A::" << i << "\n"; }
};
struct B : A {
    void f(int i = 3) override { cout << "B::" << i << "\n"; }
    void g(int i)     override { cout << "B::" << i << "\n"; }
};
A a; B b;
a.f();      // ok, print "A::5"
b.f();      // ok, print "B::3"

A& ab = b;
ab.f();     // !!! print "B::5"  // the virtual table of A
                                 // contains f(int i = 5) and
ab.g();     // !!! print "B::5"  // g(int i = 5) but it points
                                 // to B implementations
```

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() = 0; // pure virtual without body
    virtual void g() = 0; // pure virtual with body
};
void A::g() {} // pure virtual implementation (body) for g()

struct B : A {
    void f() override {} // must be implemented
    void g() override {} // must be implemented
};
```

Pure Virtual Method

A class with one *pure virtual function* cannot be instantiated

```cpp
struct A {
    virtual void f() = 0;
};

struct B1 : A {
//  virtual void f() = 0; // implicitly declared
};

struct B2 : A {
    void f() override {}
};

// A  a;  // "A"  has a pure virtual method
// B1 b1; // "B1" has a pure virtual method
B2 b2;    //  ok
```

Abstract Class and Interface

- A class is **interface** if it has only *pure virtual* functions and optionally (*suggested*) a virtual destructor. Interfaces do not have implementation or data
- A class is **abstract** if it has at least one *pure virtual* function

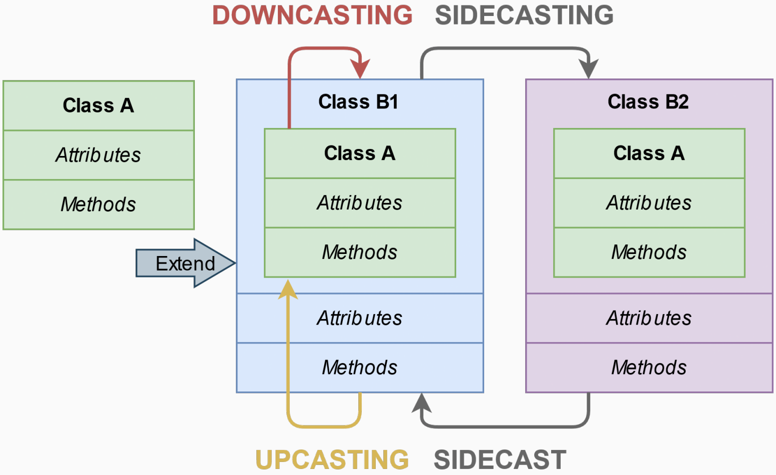
```
struct A {          // INTERFACE
    virtual ~A();   // to implement
    virtual void f() = 0;
};

struct B {          // ABSTRACT CLASS
    B() {}          // abstract classes may have a contructor
    virtual void g() = 0; // at least one pure virtual
protected:
    int x;          // additional data
};
```

Inheritance Casting and Run-time Type Identification ⋆

Hierarchy Casting

Class-casting allows implicit or explicit conversion of a class into another one across its hierarchy



Hierarchy Casting

Upcasting Conversion between a derived class reference or pointer to a base class

- It can be *implicit* or *explicit*
- It is safe
- `static_cast` or `dynamic_cast` // see next slides

Downcasting Conversion between a base class reference or pointer to a derived class

- It is only *explicit*
- It can be dangerous
- `static_cast` or `dynamic_cast`

Sidecasting (*Cross-cast*) Conversion between a class reference or pointer to another class of the same hierarchy level

- It is only *explicit*
- It can be dangerous
- `dynamic_cast`

Upcasting and Downcasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};
struct B : A {
   int var = 3;
   void f() override { cout << "B"; }
};

A  a;
B  b;
A& a1 = b; // implicit cast upcasting

static_cast<A&>(b).f();         // print "B" upcasting
static_cast<B&>(a).f();         // print "A" downcasting
cout << b.var;                  // print 3 (no cast)
cout << static_cast<B&>(a).var; // potential segfault!!! downcasting
                                // "var" does not exist in "A"
```

Sidecasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B1 : A {
   void f() override { cout << "B1"; }
};
struct B2 : A {
   void f() override { cout << "B2"; }
};

B1 b1;
B2 b2;
dynamic_cast<B2&>(b1).f();    // sidecasting, throw std::bad_cast
dynamic_cast<B1&>(b2).f();    // sidecasting, throw std::bad_cast
// static_cast<B1&>(b2).f(); // compile error
```

Run-time Type Identification

RTTI

Run-Time Type Information (RTTI) is a mechanism that allows the type of object to be *determined at runtime*

C++ expresses RTTI through three features:

- `dynamic_cast` keyword: conversion of polymorphic types
- `typeid` keyword: identifying the exact type of object
- `type_info` class: type information returned by the `typeid` operator

RTTI is available only for classes that are *polymorphic*, which means they have *at least one* virtual method

type_info and typeid

`type_info` class has the method `name()` which returns the name of the type

```
struct A {
   virtual void f() {}
};

struct B : A {};

A  a;
B  b;
A& a1 = b; // implicit upcasting
cout << typeid(a).name();  // print "1A"
cout << typeid(b).name();  // print "1B"
cout << typeid(a1).name(); // print "1B"
```

`dynamic_cast`

`dynamic_cast`, differently from `static_cast`, uses *RTTI* for deducing the correctness of the output type

This operation happens at run-time and it is expensive

`dynamic_cast<New>(Obj)` has the following properties:

- Convert between a derived class `Obj` to a base class `New` $\rightarrow$ *upcasting*. `New/Obj` are both pointers or references

- Throw `std::bad_cast` if `New/Obj` are *references* and `New/Obj` cannot be converted

- Returns `NULL` if `New/Obj` are *pointers* and `New/Obj` cannot be converted

dynamic_cast Example 1

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B : A {
   void f() override { cout << "B"; }
};

A a;
B b;
dynamic_cast<A&>(b).f();    // print "B" upcasting

// dynamic_cast<B&>(a).f(); // throw std::bad_cast
                            // wrong downcasting

dynamic_cast<B*>(&a);       // returns nullptr
                            // wrong downcasting
```

dynamic_cast Example 2

```cpp
struct A {
    virtual void f() { cout << "A"; }
};
struct B : A {
    void f() override { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

void g(bool value) {
    A* a = get_object(value);
    B* b = dynamic_cast<B*>(a); // downcasting + check
    if (b != nullptr)
        b->f();       // exectuted only when it is safe
}
```

Operator Overloading

Operator Overloading

Operator Overloading

Operator overloading is a special case of polymorphism in which some *operators* are treated as polymorphic functions and have different behaviors depending on the type of its arguments

```cpp
struct Point {
    int x, y;

    Point operator+(const Point& p) const {
        return {x + p.x, y + p.y};
    }
};

Point a{1, 2};
Point b{5, 3};
Point c = a + b; // "c" is (6, 5)
```

Operator Overloading

Category	Operators
Arithmetic	`+ - * / % ++ --`
Comparison	`== != < <= > >= <=>`
Bitwise	`| & ^ ~ << >>`
Logical	`! && ||`
Compound Assignment Arithmetic	`+= -= *= /= %=`
Compound Assignment Bitwise	`>>= <<= |= &= ^=`
Subscript	`[]`
Function call	`()`
Address-of, Reference, Dereferencing	`& -> ->* *`
Memory	`new new[] delete delete[]`
Comma	`,`

- Categories not in bold are rarely used in practice
- Operators that cannot be overloaded: `? . .* :: sizeof typeid`

Comparison Operator `operator<`

Relational and comparison operators `operator<, <=, ==, >= >` are used for comparing two objects

In particular, the `operator<` is used to determine the ordering of a set of objects (e.g. sort)

```
#include <algorithm>
struct A {
    int x;

    bool operator<(A a) const {
        return x * x < a.x * a.x;
    }
};
A array[] = {5, -1, 4, -7};
std::sort(array, array + 4);
// array: {-1, 4, 5, -7}
```

C++20 allows overloading the **spaceship operator** `<=>` (also called *three-way comparison*) for replacing all comparison operators `operator<, <=, ==, >= >`

```
struct A {
    bool operator==(const A&) const; // *** equal comparison is special,
    bool operator!=(const A&) const; //     see next slides
    bool operator<(const A&)  const;
    bool operator<=(const A&) const;
    bool operator>(const A&)  const;
    bool operator>=(const A&) const;
};

// replaced by
struct B {
    auto operator<=>(const B&) const;
};
```

Spaceship Operator `operator<=>` 2/4

```cpp
struct Obj {
    int x;

    auto operator<=>(const Obj& other) const {
        return x - other.x; // or even better "x <=> other.x"
    }
};

Obj a{3};
Obj b{5};
a < b;          // true, operator< is generated
(a <=> b) < 0; // true
```

Note: a non-defaulted `operator<=>` doesn't generate the operators `==` and `!=` (see next slide)

The compiler can also generate the code for the *spaceship operator* `= default`, even for multiple fields and arrays, by using the default comparison semantic of its members

```
struct Obj {
    int  x;
    char y;
    short z[2];

    auto operator<=>(const Obj&) const = default;
    // if x == other.x, then compare y
    // if y == other.y, then compare z
    // if z[0] == other.z[0], then compare z[1]
};

Obj a{3}, b{5};
a == b; // false, operator== is generated (= default)
a != b; // true,  operator!= is generated (= default)
```

The *spaceship operator* returns one of following ordering (classes) `<compare>`:

`std::strong_ordering`

- If `a` is equivalent to `b`, `f(a)` is also equivalent to `f(b)`
- Exactly one of `<`, `==`, or `>` must be true
  - e.g., integral types (`int`, `char`)

`std::weak_ordering`

- If `a` is equivalent to `b`, `f(a)` may not be equivalent to `f(b)`
- Exactly one of `<`, `==`, or `>` must be true
  - e.g., rectangles `R{2, 5} == R{5, 2}`

`std::partial_ordering`

- If `a` is equivalent to `b`, `f(a)` may not be equivalent to `f(b)`
- `<`, `==`, or `>` may all be false
  - e.g., floating-point (`float` with `NaN`)

Subscript Operator `operator[]`

The **array subscript** `operator[]` allows accessing to an object in an array-like fashion

The operator accepts everything as parameter, not just integers

```
struct A {
    char permutation[] {'c', 'b', 'd', 'a', 'h', 'y'};

    char& operator[](char c) { // read/write
        return permutation[c - 'a'];
    }
    char operator[](char c) const { // read only
        return permutation[c - 'a'];
    }
};

A a;
a['d'] = 't';
```

Multidimensional Subscript Operator `operator[]`

C++23 introduces the *multidimensional subscript operator* and replaces the standard behavior of the *comma operator*

```
struct A {
    int operator[](int x) { return x; }
};
struct B {
    int operator[](int x, int y) { return x * y; } // not allowed before C++23
};

int main() {
    A a;
    cout << a[3, 4]; // return 4 (bug)
    B b;
    cout << b[3, 4]; // return 12, C++23
}
```

Function Call Operator `operator()`

The **function call operator** `operator()` is generally overloaded to create objects which behave like functions, or for classes that have a primary operation (see `Basic Concepts IV` lecture)

```
#include <numeric>  // for std::accumulate

struct Multiply {
    int operator()(int a, int b) const {
        return a * b;
    }
};

int array[]   = { 2, 3, 4 };
int factorial = std::accumulate(array, array + 3, 1, Multiply{});
cout << factorial;  // 24
```

static operator() and static operator[]

C++23 introduces the `static` version of the *function call operator* `operator()` and the *subscript operator* `operator[]` to avoid passing the `this` pointer

```
#include <numeric>  // for std::accumulate

struct Multiply {
//  int         operator()(int a, int b); // declaration only
    static int operator()(int a, int b); // best efficiency, no need to access
};                                       // internal data members

struct MyArray {
//  int         operator[](int x);
    static int operator[](int x);        // best efficiency
};
int array[]   = { 2, 3, 4 };
int factorial = std::accumulate(array, array + 3, 1, Multiply{});
```

The **conversion operator** `operator T()` allows objects to be either implicitly or explicitly (casting) converted to another type

```
class MyBool {
    int x;
public:
    MyBool(int x1) : x{x1} {}

    operator bool() const {  // implicit return type
        return x == 0;
    }
};

MyBool my_bool{3};
bool b = my_bool;  // b = false, call operator bool()
```

C++11 **Conversion operators** can be marked `explicit` to prevent implicit conversions. It is a good practice as for class constructors

```
struct A {
    operator bool() { return true; }
};

struct B {
    explicit operator bool() { return true; }
};

A a;
B b;
bool    c1 = a;
// bool c2 = b;  // compile error: explicit
bool    c3 = static_cast<bool>(b);
```

Return Type Overloading Resolution ★

```cpp
struct A {
    operator float() { return 3.0f; }
    operator int()   { return 2;    }
};

auto f() {
    return A{};
}

float x = f();
int   y = f();
cout << x << " " << y; // x=3.0f, y=2
```

Increment and Decrement Operators `operator++/--`

The increment and decrement operators `operator++`, `operator--` are used to update the value of a variable by one unit

```
struct A {
    int* ptr;
    int  pos;
    A& operator++() {      // Prefix notation (++var):
        ++ptr;             //  returns the new copy of the object by-reference
        ++pos;
        return *this;
    }
    A operator++(int a) {  // Postfix notation (var++):
        A tmp = *this;     //  returns the old copy of the object by-value
        ++ptr;
        ++pos;
        return tmp;
    }
};
```

The **assignment operator** `operator=` is used to copy values from one object to another *already existing* object

```
#include <algorithm> //std::fill, std::copy
struct Array {
    char* array;
    int   size;

    Array(int size1, char value) : size{size1} {
        array = new char[size];
        std::fill(array, array + size, value);
    }
    ~Array() { delete[] array; }

    Array& operator=(const Array& x) { .... } // --> see next slide
};
Array a{5, 'o'}; // ["ooooo"]
Array b{3, 'b'}; // ["bbb"]
```

Assignment Operator operator= 2/3

- First option:

```
Array& operator=(const Array& x) {
    if (this == &x)            // (1) Check for self assignment
        return *this;
    delete[] array;            // (2) Release class resources
    size  = x.size;            // (3) Re-initialize class resources
    array = new int[x.size];
    std::copy(x.array, x.array + size, array); // (4) deep copy
    return *this;
}
```

- Second option (less intuitive):

```
Array& operator=(Array x) { // pass by-value
    swap(*this, x);          // now we need a swap function for A
    return *this;            // x is destroyed at the end
}                            // --> see next slide
```

`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

stackoverflow.com/questions/3279543
stackoverflow.com/questions/5695548

Stream Operator `operator<<`

The **stream operation** `operator<<` can be overloaded to perform input and output for user-defined types

```cpp
#include <iostream>

struct Point {
    int x, y;

    friend std::ostream& operator<<(std::ostream& stream,
                                    const Point&  point) {
        stream << "(" << point.x << "," << point.y << ")";
        return stream;
    }
    // operator<< is a member of std::ostream -> need friend
};  // implementation and definition can be splitted (not suggested for operator<<)
Point point{1, 2};
std::cout << point;  // print "(1, 2)"
```

Operators Precedence

Operators preserve **precedence** and **short-circuit** properties

```cpp
struct MyInt {
    int x;

    int operator^(int exp) { // exponential
        int ret = 1;
        for (int i = 0; i < exp; i++)
            ret *= x;
        return ret;
    }
};
MyInt x{3};
int   y = x^2;
cout << y;  // 9
int   z = x^2 + 2;
cout << z;  // 81 !!!
```

Binary Operators Note

Binary operators should be implemented as `friend` methods

```
struct A {}; struct C {};

struct B : A {
    bool operator==(const A& x) { return true; }
};
struct D : C {
    friend bool operator==(const C& x, const C& y) { return true; } // inline
};
// bool operator==(const C& x, const C& y) { return true; }  // out-of-line

A a;  B b;  C c; D d;
b == a;    // ok
// a == b; // compile error // "A" does not have == operator
c == d;    // ok, use operator==(const C&, const C&)
d == c;    // ok, use operator==(const C&, const C&)
```

C++ Object Layout ★

Overview

The term **layout** refers to how an object is arranged in memory

C++ defines four types of *layouts*:

- `aggregate`
- `trivial copyable`
- `standard layout`
- `plain-old data (POD)`

Such *layouts* are important to understand how the C++ objects interact with pure C API and for optimization purposes, e.g. pass in registers, `memcpy`, and serialization

Aggregate

An **aggregate** is an `array`, `struct`, or `class` which supports *aggregate initialization* (form of list-initialization) through curly braces syntax {}

- No *user-provided* constructors
- No `private` / `protected` *non-* `static` data members and *base* class
- No `virtual` functions

* No base classes, until C++17

* No *brace-or-equal-initializers* for non-static data members, until C++14

R Apply recursively to *base* classes *non-* `static` data members

No restrictions:

- *Non-* `static` uninitialized (until C++14) data and function members
- `static` data and function members

stackoverflow.com/questions/4178175

Aggregate - Examples


```cpp
struct Aggregate {
    int x;        // ok, public member
    int y[3];     // ok, arrays are also fine
    int z { 3 }; // only C++14

    Aggregate() = default;                        // ok, defaulted constructor
    Aggregate& operator=(const& Aggregate);  // ok, function
private:                                          // copy-assignment
    void f() {}                                   // ok, private function
};

struct NotAggregate1 {
    NotAggregate1();   // !! user-provided constructor
    virtual void f();  // !! virtual function
};
class NotAggregate2 : NotAggregate1 { // !! the base class is not an aggregate
    int           x;   // !! x is private
    NotAggregate1 y;   // !! y is not an aggregate (recursive property)
};
```

Aggregate - Examples


```
struct Aggregate1 {
    int x;
    struct Aggregate2 {
        int a;
        int b[3];
    } y;
};

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

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

Trivial copyable:

- No *user-provided* copy/move/default *constructors*, *destructor*, and copy/move *assignment* operators
- No `virtual` functions

R Apply recursively to *base* classes and *non-* `static` data members

No restrictions:

- *User-declared* constructors different from copy/move/default
- Functions or `static` ,*non-* `static` data members initialization
- `protected` / `private` members

```cpp
struct NonTrivial {
    NonTrivial();      // !! user-provided constructor
    virtual void f(); // !! virtual function
};

struct Trivial1 {
    Trivial1() = default;     // ok, defaulted constructor
    Trivial1(int) {}          // ok, user-default constructor
    static int x;             // ok, static member
    void f();                 // ok, function
private:
    int z { 3 }               // ok, private and initialized
};
struct Trivial2 : Trivial1 { // ok, base class is trivial
    int Trivial1[3];          // ok, array of trivials is trivial
};
```

Standard-Layout

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)

- No `virtual` functions
- Only one control access (`public`/`protected`/`private`) for all *non*-`static` data members
- No base classes with *non*-`static` data members
- No base classes of the same type as the first *non*-`static` data member

R Apply recursively to *base* classes and *non*-`static` data members

Standard-Layout Class (examples) 2/2

```cpp
struct StandardLayout1 {
    StandardLayout1(); // ok, user-provided contructor
    void f();          // ok, non-virtual function
};

class StandardLayout2 : StandardLayout1 {
    int x, y;          // ok, both are private
    StandardLayout1 y; // ok, 'y' is not the first data member
};

struct StandardLayout4 : StandardLayout1, StandardLayout2 {
    // ok, can use multiple inheritance as long as only
    // one class in the hierarchy has non-static data members
};
```

Plain Old Data (POD)

Plain Old Data (POD): Trivial copyable (**T**) + Standard-Layout (**S**)

(**T**) No *user-provided* copy/move/default *constructors*, *destructor*, and copy/move *assignment* operators

(**S**) Only one control access (`public` / `protected` / `private`) for all *non-* `static` data members

(**S**) No base classes with *non-* `static` data members

(**S**) No base classes of the same type as the first *non-* `static` data member

(**T, S**) No `virtual` functions

R. Apply recursively to *base* classes and *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, deprecated in C++20
- `std::is_trivially_copyable` checks for trivial copyable
- `std::is_standard_layout` checks for standard-layout

```cpp
#include <type_traits>

struct A {
    int x;
private:
    int y;
};

cout << std::is_trivially_copyable_v<A>; // true
cout << std::is_standard_layout_v<A>;    // false
cout << std::is_pod_v<A>;                // false
```

Object Layout Hierarchy