

# Modern C++ Programming

## 7. C++ OBJECT ORIENTED PROGRAMMING II

---

*Federico Busato*

University of Verona, Dept. of Computer Science  
2021, v3.07



## 1 Polymorphism

- virtual Methods
- Virtual Table
- override Keyword
- final Keyword
- Common Errors
- Pure Virtual Method
- Abstract Class and Interface

## 2 Inheritance Casting and Run-time Type Identification ★

## 3 Operator Overloading

- Overview
- Subscript Operator `operator[]`
- Comparison Operator `operator<`
- Spaceship Operator `operator<=>`
- Function Call Operator `operator()`
- Conversion Operator `operator T()`
- Increment and Decrement Operators `operator++`
- Assignment Operator `operator type=`
- Stream Operator `operator<<`
- Operator Notes

## 4 C++ Special Objects ★

- Aggregate
- Trivial Class
- Standard-Layout Class
- Plain Old Data (POD)
- Hierarchy

# Polymorphism

---

# Polymorphism

## Polymorphism

In Object-Oriented Programming (OOP), **polymorphism** (meaning “having multiple forms”) is the capability of an object of *mutating* its behavior in accordance with the 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, invoked at run-time depending on the context

# Polymorphism - The problem

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

# Polymorphism vs. Overloading

**Overloading** is a form of static polymorphism (compile-time polymorphism)

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

```
// overloading example
void f(int a)    {}

void f(double b) {}

f(3);           // calls f(int)
f(3.3);         // calls f(double)
```



# 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 (virtual method)

```
struct A {  
    virtual void f() { cout << "A"; }  
}; // now "f()" is virtual, evaluated at run-time  
  
struct B : A {  
    void f() { cout << "B"; }  
}; // now "B::f()" overrides "A::f()", evaluated at run-time  
  
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() { 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!! print "A"  
  
B b;  
f(b); // print "B"  
g(&b); // print "B"  
h(b); // print "A" (cast to A)
```

# Polymorphism Dynamic Behavior

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

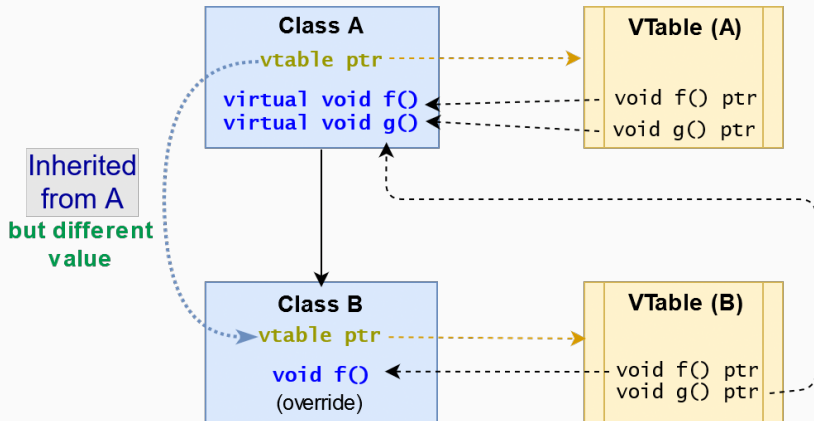
## **vtable**

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();  
};
```



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

```
#include <iostream>
using namespace std;
struct A {
    int x { 3 };
    virtual void f() { cout << "abc"; }
};

int main() {
    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)

```
class A {  
    virtual void f1();  
    virtual void f2();  
}  
  
class B : A {  
    virtual void f1();  
}  
  
cout << sizeof(A); // 8 bytes (vtable pointer)  
cout << sizeof(B); // 8 bytes (vtable 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



### override Keyword (C++11)

The `override` keyword ensures that the function is virtual and is overriding 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 exact signature

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

```

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

## final Keyword (C++11)

The `final` keyword prevents inheriting from classes or prevents 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

```
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() : A() {}    // 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  
                                   // g(int i = 5) but it points  
ab.g();     // !!! print "B::5" // 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  
};
```

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

```
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;
};

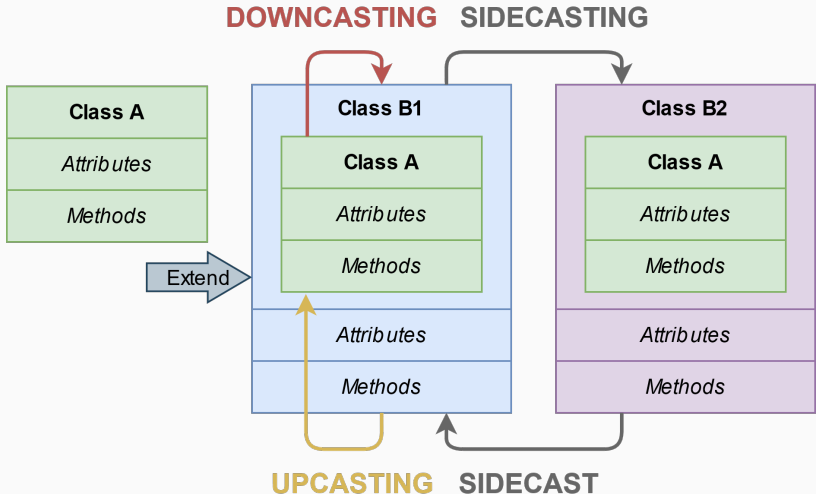
struct B {                // ABSTRACT CLASS
    B() {}                // abstract classes may have a constructor
    virtual void g() = 0; // at least one pure virtual
protected:
    int x;                // additional data
};
```

# 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 an other 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;  
    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&>(a1).f(); // print "A" downcasting
```

```
cout << b.var; // print 3
```

```
cout << static_cast<B&>(a1).var; // potential segfault!!!
```

```
// "var" does not exist in "A" 26/60
```

## 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();    // print "B2", sidecasting  
dynamic_cast<B1&>(b2).f();    // print "B1", sidecasting  
// 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 an 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 an 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 f() {}  
};  
  
struct B : A {};  
  
A a;  
B b;  
A& a1 = b;  
cout << typeid(a).name(); // print "1A"  
cout << typeid(b).name(); // print "1B"  
cout << typeid(a1).name(); // print "1A"
```



## 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` → *upcasting*. `New`, `Obj` are both pointers or references
- Throw `std::bad_cast` if `New`, `Obj` is a reference (`T&`) and `New`, `Obj` cannot be converted
- Returns `NULL` if `New`, `Obj` are pointers (`T*`) 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

```
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(); // executed 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

```
struct Point {  
    int x, y;  
  
    Point operator+(const Point& p) const {  
        return {x + p.x, y + p.x};  
    }  
};  
  
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 typedef` 34/60

## Subscript Operator `operator[]`

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

The operator accepts anything 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'];  
    }  
    const char& operator[](char c) const { // read only  
        return permutation[c - 'a'];  
    }  
};  
  
A a;  
a['d'] = 't';
```

# 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** `<=>` for replacing all comparison operators `operator<`, `<=`, `==`, `>=`, `>` at one time

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

```
#include <compare>

struct Obj {
    Obj(int x1) : x{x1} {}

    auto operator<=>(const Obj& obj) {
        return x - obj.x; // or "x <=> obj.x"
    }
private:
    int x;
};

Obj a{3};
Obj b{5};

a < b;           // true, even if the operator< is not defined
a == b;          // false
a <=> b < 0;      // true
```

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

```
#include <compare>

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

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

The *spaceship operator* can use one of the following ordering:

## strong ordering

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

## weak ordering

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

## partial ordering

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

## 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
```

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 {  
        return x == 0;    // implicit return type  
    }  
};  
  
MyBool my_bool{3};  
bool b = my_bool;    // b = false, call operator bool()
```

**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    c2 = static_cast<bool>(b);
```

# 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  
        ++pos;                 // by-reference  
        return *this;  
    }  
    A operator++(A& a) {        // Postfix notation (var++):  
        A tmp = *this;         // returns the old copy of the object  
        ++ptr;                 // by-value  
        ++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"]
a = b;           // a = ["bbb"] <-- goal
```

- First option:

```
Array& operator=(const Array& x) {  
    if (this == &x)           // Check for self assignment  
        return *this;  
    delete[] array;           // delete everything from this  
    size = x.size;            // copy "x.size"  
    array = new int[x.size];  
    std::copy(x.array, x.array + size, array); // copy "x.array"  
    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;          // see next slide  
}                           // x is destroyed at the end
```

- 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

# Stream Operator operator<<

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

```
#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
};

Point point{1, 2};
std::cout << point; // print "(1, 2)"
```

# Operators Precedence

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

```
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;  
int z = x^2 + 2;  
cout << y;    // 9  
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);  
};  
bool operator==(const C& x, const C& y); { return true; }  
  
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++ 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 };
}
```

## Trivial Class

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

```
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  
};
```

## 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)

### 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

