Classes

09 June 2021 07:07

1. RAII

2. Layout of class in memory

https://vishalchovatiya.com/posts/memory-layout-of-cpp-object/

3. Class member functions access do not require object creation in memory. Refer Py.

Class member *functions* only exist once in a code segment in memory. At a low level, they are just like normal global functions but they receive a pointer to this
From <https://stackoverflow.com/questions/12378271/what-does-an-object-look-like-in-memory>

4. Class forward declaration/prototype :)

```
Class Aclass;
```

5. Encapsulation (also called information hiding)

- is the process of keeping the details about how an object is implemented hidden away from users of the object. Instead, users of the object access the object through a public interface. In this way, users are able to use the object without having to understand how it is implemented.
  encapsulation done via access specifiers.

6. Class members are private by default

Is no limit to the number of access specifiers you can use in a class.

- Public -> members can be accessed by anybody via obj name
- Protected -> class the member belongs to, friends, and derived classes have DIRECT access to the member
- Private -> members can only be accessed by member functions of the same class (or) its friends.

- Inheritance - implementation and interface

> In his new book, "Programming Principles and Practice Using C++", Stroustrup described the implementation and interface as follows: The **interface** is the part of the class's declaration that its users access directly. The **implementation** is the part of the class's declaration that its users access only indirectly through the interface.

- Inheritance
  - Public Inheritance - everything => public are inherited as public, remaining are as is
    - public members of the base class become public members of the derived class
    - **protected members of the base class become protected members of the derived class.**
      - **Outside derived class, no one is aware of the inheritance. Its as if no inheritance exist.**
    - **Private inheritance:**
    - **Inside derived class, a base class's**
      - **private members are never accessible directly from inside a derived class**
      - **private members can be accessed through calls to the public and protected members of the base class**
      - **Public members can be accessed directly.**
    - **Outside derived class, no one is aware of the inheritance. Its as if no inheritance exist.**
- Protected Inheritance - everything => protected are inherited as protected, remaining are as is
  - public and protected members of the base class become protected members of the derived class.

- Private Inheritance - everything > private are inherited as private, remaining are as is

With public inheritance, the public methods of the base class become public methods of the derived class. In other words, the derived class **inherits** the base-class **interface** (the interface is still visible to outside and can use it). This is the **is-a** relationship. But with the private inheritance, the public methods of the base class become private methods of the derived class, even if they were protected or public in the base class. So, the derived class **does not inherit** the base-class **interface**.

But we should be careful when we talk about private inheritance. Sometimes it is very confusing. The **inherit** does not mean **"own"**. Suppose, a parent gave a child a secret recipe for a candy under the condition of not releasing the recipe. The child can give variety of candies to other people but not the recipe. With private inheritance, the derived class does enjoy(implement) the inherited interface but does not own the method. Therefore, derived class cannot show the interface to outside world. The only thing that they can show off to the outside is the product whose inner secret workings are hidden.

A class **does inherit** the **implementation** with private inheritance.

- **i.e. derived class can use the privately inherited base class's interface but cannot show it outside derived class**

From <https://www.bogotobogo.com/cplusplus/private_inheritance.php>

Below applies to protected & private inheritance

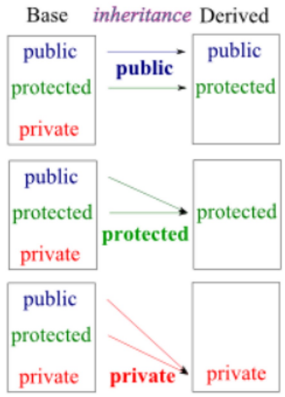
- Public inheritance means a derived class maintains all the capabilities of the base class and potentially adds more besides.
- Private, protected inheritance often means more or less the opposite: that the derived class uses a general base class to implement something with a more restricted interface.
  Cannot implement polymorphism here

https://stackoverflow.com/questions/9661936/inheritance-a-is-an-inaccessible-base-of-b

```
14 class X {
15     int     x;
16     string str;
17 public:
18     X() {std::cout << "X created" << std::endl; }
19     virtual ~X() {std::cout << "X Destroyed" << std::endl; }
20     virtual void printAll() {}
21 };
22 class Y : private X {
23     int     y;
24 public:
25     Y() {std::cout << "Y Created" << std::endl; }
26     ~Y() {std::cout << "Y Destroyed" << std::endl; }
27     void printAll() {}
28 };
29
30 int main()
31 {
32     X* xptr = new Y;
33     delete xptr;
34
35     return 0;
36 }
37
```

input

```
Compilation failed due to following error(s).

main.cpp:32:19: error: 'X' is an inaccessible base of 'Y'
   32 |     X* xptr = new Y;
      |                   ^
```



7. Inheritance memory structure

https://vishalchovatiya.com/posts/memory-layout-of-cpp-object/
Base class object is a sub-object in derived class object

8. Inheritance -> rules
   By restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once
   When a derived class is destroyed, each member destructor is called in the reverse order of construction -> C -> B -> A

```
X created
Y Created
Y Destroyed
X Destroyed
```

9. During inheritance following are not inherited.
   Private members --> either not inherited or not accessible.. Friends is one way
   Constructors --> because of point#8
   Operator overloads --> operator overloads are mostly specific to a class
   Friends --> derived class may have additional functionality which may be exposed to friend class.

10. Default inheritance for class -> private… Default inheritance for struct is public (same as members)

11. In private inheritance -> direct conversion from a derived type to a base type isn't possible

    'type cast': conversion from 'derived *' to 'Base *' exists, but is inaccessible outside derived class scope

    Private inheritance means only the derived type can access conversion.
    But outside the scope of the derived type, this relation is inaccessible and everything happens as if there was no inheritance at all

    From <https://stackoverflow.com/questions/49208173/with-private-inheritance-when-is-it-ok-to-upcast>
    https://www.bogotobogo.com/cplusplus/private_inheritance.php

12. Getters should return by value (or) const reference

13. Default methods given by compiler
    So by default compiler will generate:
    1. default constructor
    2. copy constructor
    3. copy-assign operator
    4. destructor
    5. move constructor
    6. move-assign operator

```
class Thing {
public:
    Thing();                          // default constructor
    Thing(const Thing&);              // copy c'tor
    Thing& operator=(const Thing&); // copy-assign
    ~Thing();                         // d'tor
    // C++11:
    Thing(Thing&&);                   // move c'tor
    Thing& operator=(Thing&&);        // move-assign
};
```

14. Default constructor (with no args) is called when no user-defined constructor. Else default = user-defined constructor

- A constructor which can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter).
- Has the same effect as a user-defined constructor with empty body and empty initializer list. That is, it calls the default constructors of the bases and of the non-static members of this class.
- If some user-declared constructors are present, the user may still force the automatic generation of a default constructor by the compiler that would be implicitly-declared otherwise with the keyword default.
- Inside constructor, member variables are created and initialized.. Initialized in order of declaration in class.
- Default constructor call

```
Aclass a = Aclass(); // fundamental data types aren't initialized automatically,AUTO call default constructors of class members
Aclass a = Aclass{}; // fundamental data types are initialized automatically,AUTO call default constructors of class members

//below calls are converted to above by compiler.. And copy construction is elided

Aclass a;            // fundamental data types arent initialized automatically,AUTO call default constructors of class members
Aclass a{};          // fundamental data types are initialized,AUTO call default constructors of class members
Aclass a();          // invalid.. Confusion with function call
```

- TO Keep both default constructor and user-defined ones if needed

```
// Tell the compiler to create a default constructor, even if
// there are other user-provided constructors.
Date() = default;
```

- Classes containing classes or classes which are derived from other classes, default constructor of member/base is called FIRST unless explicit user-defind constructor called.
- If base class has default & user-defined constructors, default constructor will called by default (PRIORITY).

15. Default constructors --- more properties

Deleted implicitly-declared default constructor

The implicitly-declared or defaulted (since C++11) default constructor for class T is undefined (until C++11)defined as deleted (since C++11) if any of the following is true:

- T has a member of reference type without a default initializer (since C++11).
- T has a non-const-default-constructible const member without a default member initializer (since C++11).
- T has a member (without a default member initializer) (since C++11) which has a deleted default constructor, or its default constructor is ambiguous or inaccessible from this constructor.
- T has a direct or virtual base which has a deleted default constructor, or it is ambiguous or inaccessible from this constructor.
- T has a direct or virtual base or a non-static data member which has a deleted destructor, or a destructor that is inaccessible from this constructor.

16. Trivial constructors - A trivial default constructor is a constructor that performs no action. All data types compatible with the C language (POD types) are trivially default-constructible.

Trivial default constructor

The default constructor for class T is trivial (i.e. performs no action) if all of the following is true:

- The constructor is not user-provided (i.e., is implicitly-defined or defaulted on its first declaration)
- T has no virtual member functions
- T has no virtual base classes
- T has no non-static members with default initializers. (since C++11)
- Every direct base of T has a trivial default constructor
- Every non-static member of class type (or array thereof) has a trivial default constructor

17. In constructors body, member vars are created and then **assigned** value inside constructor body - 2 steps

18. With member initializer lists, member variables are initialized when created - in single step
    - Initialize **const, non-const variables** using initializer list or in-class initialization
    - Initialize **reference variables** using initializer list
    - **Member classes** are initialized with one constructor call

19. Different ways to initialize class

```
Entity e0{7};  //Constructor call. Uniform init is Inplace
Entity e(8);   //Constructor call. Direct init
Entity e2((Entity(10)));      //constructor call. Copy constructor elided(refer 26#) to optimize by compiler. Prefer uniform init
    Equivalent to e2 = (Entity(10))
Entity e2{Entity{10}};        //constructor call. Copy constructor elided(refer 26#) to optimize by compiler. Prefer uniform init
    Equivalent to e2 = {Entity{10}}
Entity e3 = Entity(11);       //constructor call. Copy constructor elided(refer 26#). And prefer uniform init
Entity* e6 = new Entity((13));//constructor call. ??
```

20. Implicit constructor calls

```
Entity e5 = {12};     //implicit conversion and constructor - one conversion only.. Two will be error
Entity e7 = 13;       //implicit conversion and constructor
Entity e8 = int(14.0);//implicit conversion and constructor
```

All calls get converted to

```
AClass aobj = Aclass{};
```

or

```
AClass aobj = Aclass();
```

21. In-Class initialization - Default initialization can be given for non-static/const members directly at declaration. may or may not be used by constructor

    static only **can't** be initialized this way

22. Inline

https://en.cppreference.com/w/cpp/language/inline#:~:text=The%20inline%20specifier%2C%20when%20used%20in%20a%20decl%2Dspecifier%2D,is%20implicitly%20an%20inline%20variable.
error C2864: 'AClass::a': a static data member with an in-class initializer must have non-volatile **const** integral type or be specified as '**inline**'
-- non-volatile? Is it because static const volatile is unpredictable and since its single instance, it may change unpredictably?
-- const cannot be changed
-- inline gives external linkage? Explained below

23. Constructors are allowed to call other constructors->delegating constructors (or constructor chaining).

```
Foo(int value): Foo{} //user-defined constructor calling default constructor
```

Delegating constructor cannot initialize members

24. this pointer - implicitly added to any non-static member function.

```
void setID(int id) { m_id = id; }
void setID(Simple* const this, int id) { this->m_id = id; }
```

25. Doesn't defining member functions in the header violate the one-definition rule?

Member functions defined inside the class definition are considered implicitly inline. Inline functions are exempt from the one definition per program. So including class header in multiple files is okay

Member functions defined outside the class definition are treated like normal functions, and are subject to the one definition per program part of the one-definition rule

Implementations for the classes that belong to the C++ standard library are contained in a precompiled file that is linked in at the link stage.

- It's **faster to link** a precompiled library than to recompile it every time you need it
- a single copy of a precompiled library can be shared by many applications, whereas **compiled code gets compiled into every executable that uses it (inflating file sizes)**
- intellectual property reasons (you don't want people stealing your code).

26. Const member function guarantees it will not modify the object (or) call any non-const member functions (or) return non-const references to members

27. Static member functions cannot access non-static members (or) call non-static methods. Static member variables are one instance shared by all objects of the class

https://en.cppreference.com/w/cpp/language/static

Like Static duration functions

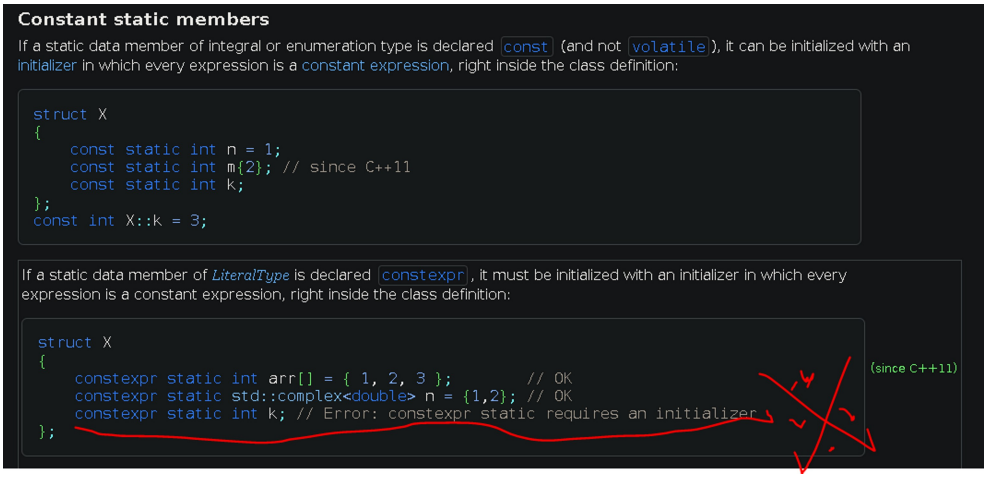
- static members exist even if no objects of the class have been instantiated
  Hence can be accessed directly using the class name and the scope resolution operator
- Not subject to access controls

static member variables, static member functions are not attached to any particular object..
because static member functions don't have *this pointer
hence cannot access non-static members

The `static` keyword is only used with the declaration of a static member, inside the class definition, but not with the definition of that static member:

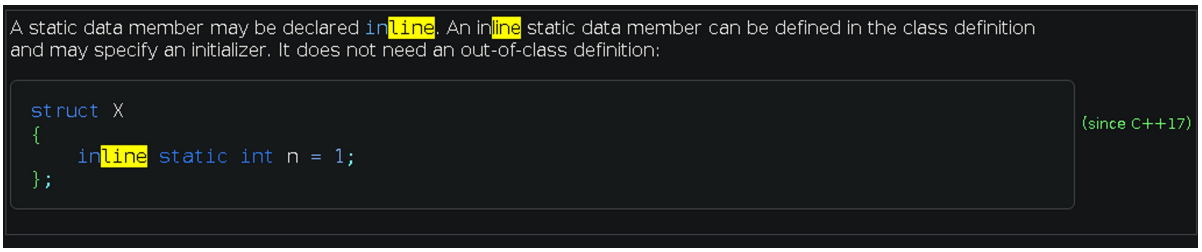
```
class X { static int n; }; // declaration (uses 'static')
int X::n = 1;              // definition (does not use 'static')
```

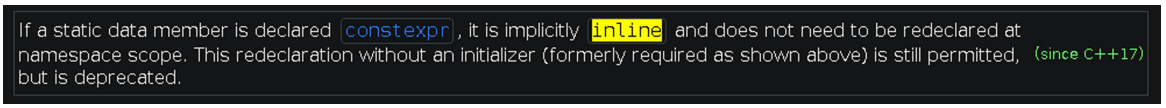
Non-const static vars can be modified by one instance of class without the knowledge of other. So better to have const-static vars?



28. Inline static

https://stackoverflow.com/questions/46874055/why-is-inline-required-on-static-inline-variables
Explicit inlining needed for static variable.. Will break legacy code.. How?
https://en.cppreference.com/w/cpp/language/static





29. Friend function - can access private and protected members of class

- A friend function is a function that can access the private members of a class as though it were a member of that class.
- No implicit this pointer
- Instead of this pointer, it has a reference to object of class as input
- It may be just a normal function or a member of another class
- A function can be a friend of more than one class at the same time.

```
class Accumulator
{
private:
    int m_value;
public:
    ..
    // Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};

// reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator)
{
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}
```

30. Friend classes

- All of the members of the friend class access to the private members of the other class
- no direct access to the *this pointer of class
- Since ClassB is a friend class, we can create objects of ClassA inside of ClassB

```
class Storage
{
private:
    int m_nValue;
public:
    ..
```

```
        friend class Display;
        template<typename, std::size_t> friend class Array;
    };

    class Display
    {..}
```

31. Friend member functions

```
class Storage
{
    private:
    int m_nValue;
    public:
    ..
    // Make the Display::displayItem member function a friend of the Storage class
    friend void Display::displayItem(const Storage& storage); // okay now
};
```

32. Anonymous objects are primarily used either to pass or return values without having to create lots of temporary variables to do so

```
Aclass{1};
```

33. Types can be defined (nested) inside of a class

Classes essentially act as a namespace for any nested types.

34. Copy constructor -> Compiler provides default copy assignment member -> shallow copy

```
    // Copy constructor
    Fraction(const Fraction &fraction) :
        m_numerator(fraction.m_numerator), m_denominator(fraction.m_denominator)
    {..}
```

35. Compiler is allowed to opt-out of calling the copy constructor and just do a direct initialization instead when **anonymous object/R-value are used for initialization**. This process is called **elision.**

Examples

```
Fraction six = Fraction(6);  // copy constructor elided.

Something foo()
{
  Something s;
  return s; // copy constructor elided. Move constructed probably into 'something s'.
}
}
int main()
{
  Something s = foo();
}

Fraction makeNegative(Fraction f) // copy constructor called
{
    f.setNumerator(-f.getNumerator());
    return f; // copy constructor called. Not elided probably because of use in cout
}
std::cout << makeNegative(fiveThirds);
```

36. Implicit conversion

Implicit conversion works for all kinds of initialization (direct, uniform, and copy) --> converting constructors (or conversion constructors).

```
void printFraction(const Fraction &f)
printFraction(6);

class Entity
{
private:
    std::string m_Name;
    int m_Age;
public:
    Entity(const std::string name)
        :m_Name(name), m_Age(-1)
    {..}

    Entity(int age)
```

```
        :m_Name("Unknown"), m_Age(age)
    {..}
};


//Entity c = "Name1"; // cannot convert from 'const char [6]' to 'Entity'---> 2 conversions - char arr to string, string to
object
Entity c = std::string("Name1"); //Only one implicit conversion compiler will allow
Entity d = 28;
```

37. Explicit keyword --> all implicit conversions disallowed

Uniform initialization will not do narrowing conversions, but it will do implicit conversions

Rule: Consider making your constructors and user-defined conversion member functions explicit to prevent implicit conversion errors & delete constructors if required that allows implicit conversions

```
class MyString
{
private:
    std::string m_string;
public:
     // explicit keyword makes this constructor ineligible for implicit conversions
    explicit MyString(int x) // allocate string of size x
    {
        m_string.resize(x);
    }
};

printString('x'); // compile error, since MyString(int) can't be used for implicit conversions
```

38. Use default copy constructors provided by STL libs for eg. Don't try to write your own

39. Overriding access specifier using 'using' - move inherited members under any access specifier you want

Derived class can change access level of inherited members. Doesn't convert private to public or protected.
For eg

```
private:
using Base::print();
```

40. Function overriding and Overriding the function-overriding

The member function of derived class overrides the member function of base class i.e. when same func in derived class is called using derived obj, derived class func is invoked
From <https://www.programiz.com/cpp-programming/inheritance>

```
derived1.print();       // base class function override. Static binding
derived2.Base::print(); // access print() function of the Base class

class Derived : public Base {
   public:
    void print() {
        Base::print();
    }
};
```

41. Polymorphism

```
Base& b = static_cast<Base&>(derived) works
Base& rBase{ derived }; // points to base part of derived
Base* pBase{ &derived }
```

However when you have base class pointer pointing to derived class it will call base class function only. NO OVERRIDING.

```
Base* ptr = &derived1;  // this won't work with private inheritance .. Point#7
ptr->print(); // call function of Base class using ptr
```

Base pointer/reference cannot see Derived::print().. They can only see Base::print()
This is a problem when we try to write generic code - common function or common array for all derived classes

Virtual functions and polymorphism

1. A virtual function, when called, resolves to the most-derived version of the function that exists between the base and derived class. This capability is known as polymorphism.
2. A derived function is considered a match if it has the same signature as the base version of the virtual function. Such functions are called **overrides**.

```
class Base
{
public:
    virtual std::string_view getName() const { return "Base"; } // note addition of virtual keyword
```

```
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; } //Need not be declared virtual. Inherits virtual property from base
};

Base &rBase{ derived };
rBase.getName(); // points to derived method
```

Calling virtual function from constructor or destructor -> would resolve to base class method depending upon derived class is created or destroyed

42. Override keyword (COMPILE TIME)

Override keyword when used with virtual func of derived class, it serves as a **compile-time check** to ensure it is indeed overriding the base class virtual method. Else it wont be known if coding has error.

43. Final keyword (COMPILE TIME)

If the user tries to override a function or inherit from a class that has been specified as final, the compiler will give a compile error.

```
virtual const char* getName() override final
{..}
class B final : public A
{..}
```

44. Co-variant return types - return base* or derived* but still be virtual override match

Because base version of the function returns a Base*, using b->getThis(), eventhough will call derivedclass::getType, will result in the returned Derived* upcast to a Base*. And thus, Base::printType() is called.

```
class Base
{
public:
virtual Base* getThis() { std::cout << "called Base::getThis()\n"; return this; }
void printType(){..}
};

class Derived : public Base
{
public:
// However, because Derived is derived from Base, it's okay to return Derived* instead of Base*
Derived* getThis() override { std::cout << "called Derived::getThis()\n";  return this; }
void printType(){..}
};

Derived d{};
Base* b{ &d };
d.getThis()->printType(); // calls Derived::getThis(), returns a Derived*, calls Derived::printType
b->getThis()->printType(); // calls Derived::getThis(), returns a Base*, calls Base::printType
```

45. Virtual destructor

Same like virtual funcs, if u have a base* b{derived} and you call delete b; base's destructor alone is called.
Make all destructors participating in inheritance virtual.

```
class base {
  public:
    base()
    { cout << "Constructing base\n"; }
    virtual ~base()
    { cout << "Destructing base\n"; }
};
```

46. Binding

Binding refers to the process that is used to convert identifiers (such as variable and function names) into addresses.

1. Static/Early/ binding
   Compiler will replace the add() function call with an instruction that tells the CPU to jump to the address of the add() function.

2. Dynamic/Late/Lazy binding
   Not possible to know which function will be called until runtime (when the program is run). Example -> function pointer

47. Virtual table

https://www.learncpp.com/cpp-tutorial/the-virtual-table/
http://www.vishalchovatiya.com/memory-layout-of-cpp-object/

48. Pure virtual functions, Abstract class, Interface classes

A pure virtual function makes it so the base class cannot be instantiated, and the derived classes are forced to define pure virtual functions before they can be instantiated.
Abstract class has atleast one pure virtual functions. It can have a virtual table and pure virtual function entry will point to null or some error function
An interface class is a class that has no member variables, and where all of the functions are pure virtual

I'd use an *interface* if I want to define a set of rules using which a component can be programmed, without specifying a concrete particular behavior. Classes that implement this interface will provide some concrete behavior themselves.

Instead, I'd use an *abstract class* when I want to provide some default *infrastructure code* and behavior, and make it possible to client code to derive from this abstract class, overriding the pure virtual methods with some custom code, and *complete* this behavior with custom code. Think for

49. Multiple inheritance

Ambiguity resulting from inheriting from two classes both having a common base. Resolved using virtual base class

```
class PoweredDevice
{..};
class Scanner: virtual public PoweredDevice
{..};
class Printer: virtual public PoweredDevice
{..};
class Copier: public Scanner, public Printer
{..};
```

http://www.vishalchovatiya.com/part-2-all-about-virtual-keyword-in-cpp-how-virtual-class-works-internally/

Above link xplains memory layout when atleast one virtual base class in inherited

1. virtual base classes are always created before non-virtual base class
2. Most derived class will call virtual base class function (watever be its level), not its parent!
3. Most derived class is responsible for constructing the virtual base class.
4. All classes inheriting a virtual base class will have a virtual table with offset to shared region (refer to above link).

50. Object slicing

Assigning/Copying derived to base results in slicing of derived and base only gets stored. Happens only when u assign/copy. So always take referen!ce
Another problem

```
std::vector<Base> v{};
v.push_back(Base{ 5 });    // add a Base object to our vector
v.push_back(Derived{ 6 }); // add a Derived object to our vector ---> Slicing

        // Print out all of the elements in our vector
for (const auto& element : v)
std::cout << "I am a " << element.getName() << " with value " << element.getValue() << '\n';
```

Make sure your function parameters are references (or pointers) and try to avoid any kind of pass-by-value when it comes to derived classes.

51. RAII

Im

52. Copy constructors

What

A copy constructor of class T is a non-template constructor whose first parameter is `T&`, `const T&`, `volatile T&`, or `const volatile T&`, and either there are no other parameters, or the rest of the parameters all have default values.

Syntax

class-name (`const` *class-name* &)	(1)	
class-name (`const` *class-name* &) = `default;`	(2)	(since C++11)
class-name (`const` *class-name* &) = `delete;`	(3)	(since C++11)

Where *class-name* must name the current class (or current instantiation of a class template), or, when declared at namespace scope or in a friend declaration, it must be a qualified class name.

When

The copy constructor is called whenever an object is initialized (by direct-initialization or copy-initialization) from another object of the same type (unless overload resolution selects a better match or the call is elided), which includes

- initialization: `T a = b;` or `T a(b);`, where b is of type T;
- function argument passing: `f(a);`, where a is of type T and f is `void f(T t);`
- function return: `return a;` inside a function such as `T f()`, where a is of type T, which has no move constructor.

j

53. Implicity defined copy-constructor

Declared if:

Implicitly-declared copy constructor

If no user-defined copy constructors are provided for a class type (`struct`, `class`, or `union`), the compiler will always declare a copy constructor as a non-explicit `inline public` member of its class. This implicitly-declared copy constructor has the form `T::T(const T&)` if all of the following are true:

- each direct and virtual base B of T has a copy constructor whose parameters are `const B&` or `const volatile B&`;
- each non-static data member M of T of class type or array of class type has a copy constructor whose parameters are `const M&` or `const volatile M&`.

Otherwise, the implicitly-declared copy constructor is `T::T(T&)`. (Note that due to these rules, the implicitly-declared copy constructor cannot bind to a volatile lvalue argument.)

A class can have multiple copy constructors, e.g. both `T::T(const T&)` and `T::T(T&)`.

If some user-defined copy constructors are present, the user may still force the generation of the implicitly declared copy constructor with the keyword `default`. (since C++11)

The implicitly-declared (or defaulted on its first declaration) copy constructor has an exception specification as described in dynamic exception specification (until C++17)noexcept specification (since C++17).

Deleted if:

Deleted implicitly-declared copy constructor

The implicitly-declared copy constructor for class T is undefined if any of the following conditions are true: (until C++11)

The implicitly-declared or defaulted copy constructor for class T is defined as *deleted* if any of the following conditions are true: (since C++11)

- T has non-static data members that cannot be copied (have deleted, inaccessible, or ambiguous copy constructors);
- T has direct or virtual base class that cannot be copied (has deleted, inaccessible, or ambiguous copy constructors);
- T has direct or virtual base class or a non-static data member with a deleted or inaccessible destructor;

Deleted if:

The generation of the implicitly-defined copy constructor is deprecated if T has a user-defined destructor or user-defined copy assignment operator. (since C++11)

54. Move constructor

What

A move constructor of class T is a non-template constructor whose first parameter is `T&&`, `const T&&`, `volatile T&&`, or `const volatile T&&`, and either there are no other parameters, or the rest of the parameters all have default values.

Syntax

class-name **(** *class-name* **&&** **)**	(1)	(since C++11)
class-name **(** *class-name* **&&** **) = default;**	(2)	(since C++11)
class-name **(** *class-name* **&&** **) = delete;**	(3)	(since C++11)

When

The move constructor is typically called when an object is initialized (by direct-initialization or copy-initialization) from rvalue (xvalue or prvalue) (until C++17)xvalue (since C++17) of the same type, including

- initialization: `T a = std::move(b);` or `T a(std::move(b));`, where b is of type T;
- function argument passing: `f(std::move(a));`, where a is of type T and f is `void f(T t)`;
- function return: `return a;` inside a function such as `T f()`, where a is of type T which has a move constructor.

When the initializer is a prvalue, the move constructor call is often optimized out (until C++17)never made (since C++17), see copy elision.

Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc.) rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For example, moving from a `std::string` or from a `std::vector` may result in the argument being left empty. However, this behavior should not be relied upon. For some types, such as `std::unique_ptr`, the moved-from state is fully specified.

55. Implicit Move constructor

Implicitly-declared move constructor

If no user-defined move constructors are provided for a class type (`struct`, `class`, or `union`), and all of the following is true:

- there are no user-declared copy constructors;
- there are no user-declared copy assignment operators;
- there are no user-declared move assignment operators;
- there is no user-declared destructor.

then the compiler will declare a move constructor as a non-explicit `inline public` member of its class with the signature `T::T(T&&)`.

A class can have multiple move constructors, e.g. both `T::T(const T&&)` and `T::T(T&&)`. If some user-defined move constructors are present, the user may still force the generation of the implicitly declared move constructor with the keyword `default`.

Deleted implicitly-declared move constructor

The implicitly-declared or defaulted move constructor for class T is defined as *deleted* if any of the following is true:

- T has non-static data members that cannot be moved (have deleted, inaccessible, or ambiguous move constructors);
- T has direct or virtual base class that cannot be moved (has deleted, inaccessible, or ambiguous move constructors);
- T has direct or virtual base class or a non-static data member with a deleted or inaccessible destructor;

Implicitly-defined move constructor

If the implicitly-declared move constructor is neither deleted nor trivial, it is defined (that is, a function body is generated and compiled) by the compiler if odr-used or needed for constant evaluation. For `union` types, the implicitly-defined move constructor copies the object representation (as by `std::memmove`). For non-union class types (`class` and `struct`), the move constructor performs full member-wise move of the object's bases and non-static members, in their initialization order, using direct initialization with an xvalue argument. If this satisfies the requirements of a constexpr constructor, the generated move constructor is `constexpr`.

56. Move constructor notes

Notes

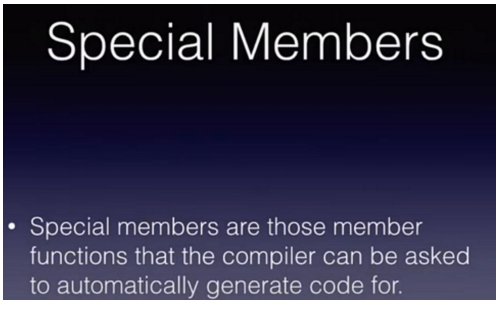
To make the strong exception guarantee possible, user-defined move constructors should not throw exceptions. For example, `std::vector` relies on `std::move_if_noexcept` to choose between move and copy when the elements need to be relocated.

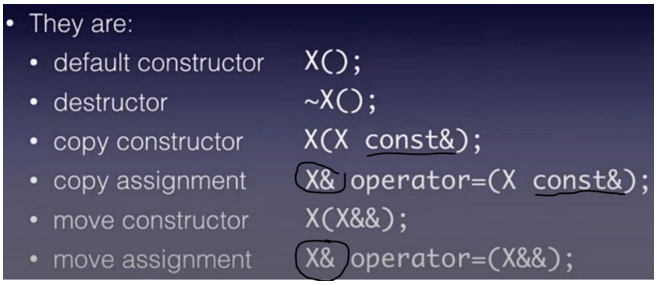
If both copy and move constructors are provided and no other constructors are viable, overload resolution selects the move constructor if the argument is an rvalue of the same type (an xvalue such as the result of `std::move` or a prvalue such as a nameless temporary (until C++17)), and selects the copy constructor if the argument is an lvalue (named object or a function/operator returning lvalue reference). If only the copy constructor is provided, all argument categories select it (as long as it takes a reference to const, since rvalues can bind to const references), which makes copying the fallback for moving, when moving is unavailable.

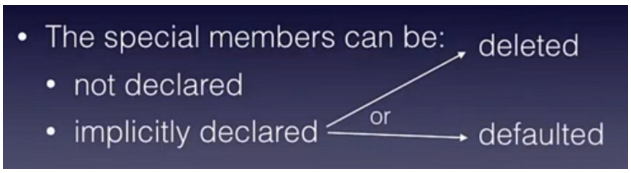
A constructor is called a 'move constructor' when it takes an rvalue reference as a parameter. It is not obligated to move anything, the class is not required to have a resource to be moved and a 'move constructor' may not be able to move a resource as in the allowable (but maybe not sensible) case where the parameter is a const rvalue reference (const T&&).

57. Special members Syntax & Terminology

Engineering Distinguished Speaker Series: Howard Hinnant







+ User declared and be deleted, defaulted or user-defined

```
struct X
{
    X() {}          // user-declared
    X();            // user-declared
    X() = default; // user-declared
    X() = delete;  // user-declared
};
```

=delete means exists but cant be used.

- Deleted members participate in overload resolution.
- Members not-declared do not participate in overload resolution.

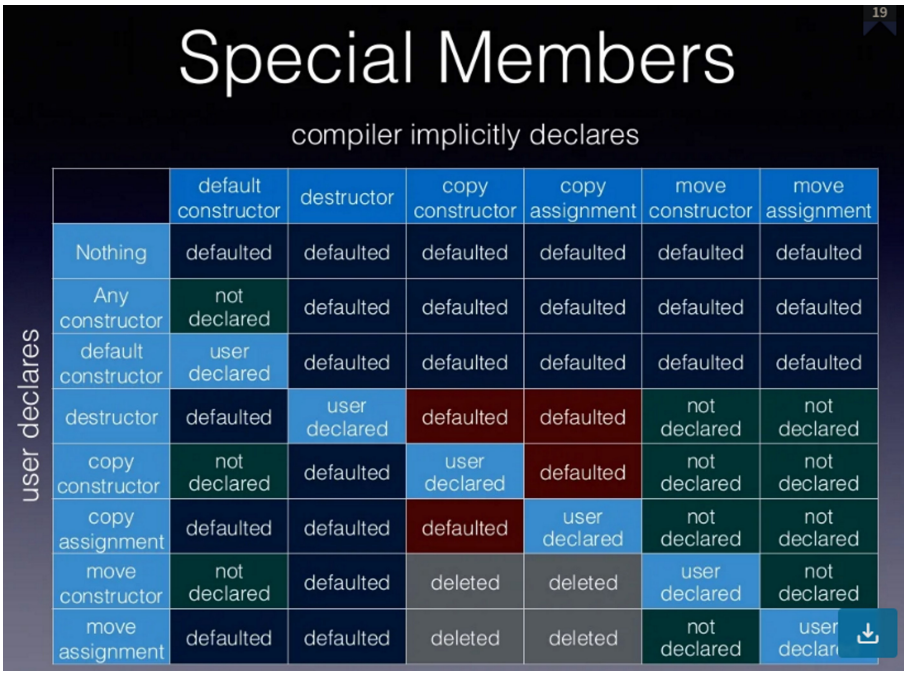
```
struct X
{
    template <class ...Args>
        X(Args&& ...args);

    X() = delete;
};
```

- Now X() binds to the deleted default constructor instead of the variadic constructor.
- X is no longer default constructible.

D

58. Rules

Rule of Zero
3-if u supply any of copy const, copy assign or destr then supply all
5-if u supply any of copy const
Single Responsibility principle
https://stackoverflow.com/questions/33932824/why-does-destructor-disable-generation-of-implicit-move-methods
https://en.cppreference.com/w/cpp/language/rule_of_three

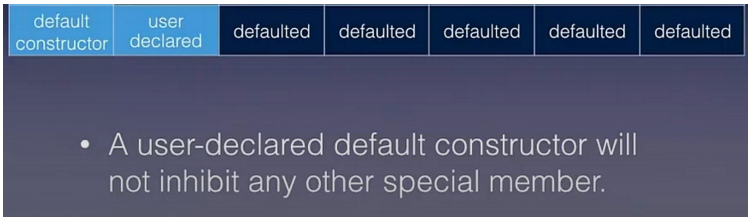


defaulted -> implicitly defaulted
deleted -> implicitly deleted
not declared -> doesn't exist
red defaulted -> implicilty defaulted but deprecated. Make it user declared + defaulted by making = default (user takes call to keep it or supply one)
last two rows -> move only classes

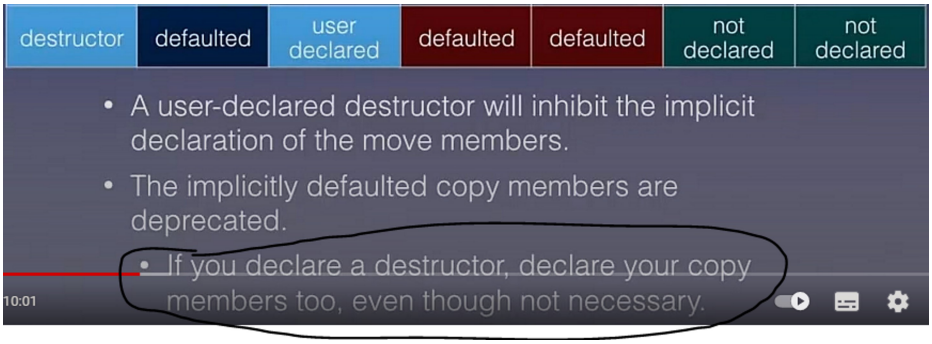
=delete means declared but not defined. Will participate in overload resolution.
Not declared = no declaration also. Doesn't exist
=default means user declared but definition by compiler.
Deprecated not recommended

Implicitly declared & defaulted. When user doesn't supply const.
Implicity declared & deleted. When default const is deleted when user supplies const.
User declared & defaulted. X() = default
User declared & deleted. X() = delete
User declared.

User declares any constructor then default constructor is not declared
User declares default constructor then no difference



Move members not declared: From C++11. move members if defaulted wud give wrong behavior mostly if destructor is user-defined. So not declaring.
Copy members defaulted: Defaulted in red means deprecated i.e. Upto C++11 copy members were declared by compiler by default.
After C++11 if user sets copy=default, then it is user-defined copy member and that is not deprecated. But compiler supplying and declaring is deprecated.?
Another option is to declare a user-defined copy member and then set it default in cpp.
Basically copy member shud be user-defined.



Class can only be moved.
Constructor default is inhibited.
Copy members are there and participate in overload resolution. But cannot be used.

Assignment members do not inhibit default constructor but influence copy/move constructor.

Key Rule in C++ (Post-C++11)If any of copy constructor, copy assignment operator, move constructor, move assignment operator, or destructor is explicitly declared by the user, the default constructor is not declared automatically.However, if the class has no user-declared constructors at all, the compiler will still implicitly declare and default the default constructor.

59. h
I