**Inheritance**

# When do we use Initializer List in C++?

Initializer List is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon.

* Prefer initialization over assignment
* Initialization happens before the body of the constructor is executed

1. For initialization of non-static const data members
2. For initialization of reference members
3. For initialization of member objects which do not have default constructor
4. For initialization of base class members: parameterized constructor of base class can only be called using Initializer List
5. When constructor’s parameter name is same as data member: If constructor’s parameter name is same as data member name then the data member must be initialized either using this pointer or Initializer List
6. For Performance reasons

# What happens when more restrictive access is given to a derived class method in C++?

Unlike Java, C++ allows to give more restrictive access to derived class methods. For example, the following program compiles fine.

In C++ we can set different access specifier for a method in derived class from base class.

Following program works fine because fun() is public in base class.

Access specifiers are checked at compile time and fun() is public in base class. At run time, only the function corresponding to the pointed object is called and access specifier is not checked.

So a private function of derived class is being called through a pointer of base class.

#include<iostream>

using namespace std;

class Base {

public:

virtual void fun(int i) {

cout << "Base fun" << endl;

}

};

class Derived: public Base {

private:

void fun(int x) { // no problem if fun is in private, it's a compiler error in Java

cout << "Derived fun" << endl;

}

};

int main() {

/\* CASE - 1

Derived d;

d.fun(1); // error: 'virtual void Derived::fun(int)' is private

\*/

// CASE - 2

Base \*ptr = new Derived;

ptr->fun(10);

return 0;

}

Output:

Derived fun

# Does overloading work with Inheritance?

If we have a function in base class and a function with same name in derived class, can the base class function be called from derived class object?

Overloading doesn’t work for derived class in C++ programming language. There is no overload resolution between Base and Derived. The compiler looks into the scope of Derived, finds the single function “double f(double)” and calls it. It never disturbs with the (enclosing) scope of Base. In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule.

In Java overloading works across scopes contrary to C++.

#include <iostream>

using namespace std;

class Base {

public:

int f(int i) {

cout << "f(int): ";

return i+3;

}

};

class Derived : public Base {

public:

double f(double d) {

cout << "f(double): ";

return d+3.3;

}

};

int main() {

Derived\* dp = new Derived;

cout << dp->f(3) << endl;

cout << dp->f(3.3) << endl;

delete dp;

return 0;

}

Output:

f(double): 6.3

f(double): 6.6

in Java

f (int): 6

f (double): 6.6

# Is assignment operator inherited?

In C++, like other functions, assignment operator function is inherited in derived class.

In the following program, base class assignment operator function can be accessed using the derived class object.

#include<iostream>

using namespace std;

class A {

public:

A & operator= (A &a) {

cout << "Base class assignment operator called" << endl;

return \*this;

}

};

class B: public A { };

int main() {

B a, b;

a.A::operator=(b); //calling base class assignment operator function // using derived class

return 0;

}

operator= doesn’t inherit because it performs a constructor-like activity. That is, just because you know how to assign all the members of an object on the left-hand side of the = from an object on the right-hand side doesn’t mean that assignment will still have the same meaning after inheritance.

# Hiding of all overloaded methods with same name in base class

In C++, if a derived class redefines base class member method then all the base class methods with same name become hidden in derived class.

For example, the following program doesn’t compile. In the following program, Derived redefines Base’s method fun() and this makes fun(int i) hidden.

Note that the above facts are true for both static and nonstatic methods.

#include<iostream>

using namespace std;

class Base {

public:

void fun() { cout << "Base::fun() called" << endl; }

void fun(int i) {

cout << "Base::fun(int i) called" << endl;

}

};

class Derived: public Base {

public:

void fun() { cout << "Derived::fun() called" << endl; }

};

int main() {

Derived d;

d.fun(5); // CE: no matching function for call to 'Derived::fun(int)'

return 0;

}

# Inheritance and friendship

In C++, friendship is not inherited. If a base class has a friend function, then the function doesn’t become a friend of the derived classes.

#include <iostream>

using namespace std;

class A {

protected:

int x;

public:

A() { x = 0;}

friend void show();

};

class B: public A {

public:

B() : y (0) {}

private:

int y;

};

void show() {

B b;

cout << "The default value of A::x = " << b.x;

// Can't access private member declared in class 'B'

cout << "The default value of B::y = " << b.y;

}

int main() {

show();

return 0;

}

# Object Slicing in C++

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

#include <iostream>

using namespace std;

class Base { int x, y; };

class Derived : public Base { int z, w; };

int main() {

Derived d;

Base b = d; // Object Slicing, z and w of d are sliced off

/\* Compilation error when assigning base class object to derived class object

Base b;

//Derived d1(b); // CE: no matching function for call to 'Derived::Derived(Base&)'

Derived d2;

d2 = b; // CE: no match for 'operator=' (operand types are 'Derived' and 'Base')

\*/

return 0;

}

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

#include <iostream>

using namespace std;

class Base {

protected:

int i;

public:

Base(int a) { i = a; }

virtual void display() { cout << "Base class, i = " << i << endl; }

};

class Derived : public Base

{

int j;

public:

Derived(int a, int b) : Base(a) { j = b; }

virtual void display() { cout << "Derived class, i = " << i << ", j = " << j << endl; }

};

// Global method, Base class object is passed by value

void somefunc (Base obj) {

obj.display();

}

int main() {

Base b(33);

Derived d(45, 54);

somefunc(b);

somefunc(d); // Object Slicing, the member j of d is sliced off

return 0;

}

Output:

Base class, i = 33

Base class, i = 45

# How to avoid Object Slicing in C++

## Using pointers or reference

Object slicing doesn’t occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory.

For example, following method somefunc() will not cause object slicing

void somefunc (Base &obj) {

obj.display();

}

// rest of code is similar to above

Output:

Base class, i = 33

Derived class, i = 45, j = 54

If we use pointers, then also object slicing can be avoided

void somefunc (Base \*objp) {

objp->display();

}

int main() {

Base \*bp = new Base(33) ;

Derived \*dp = new Derived(45, 54);

somefunc(bp);

somefunc(dp); // No Object Slicing

return 0;

}

// rest of code is similar to above

Output:

Base class, i = 33

Derived class, i = 45, j = 54

## Using pure virtual function

Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

# Inheritance and static member functions

Inheritance and static member functions static member functions act the same as non-static member functions:

1. They inherit into the derived class
2. If you redefine a static member, all the other overloaded functions in the base class are hidden
3. If you change the signature of a function in the base class, all the base class versions with that function name are hidden

However, static member functions cannot be virtual.

# Simulating final class in C++

* "final" keyword in Java and C++11 onwards
* "sealed" keyword in C#

final class in C++ can be created by use of private constructor, virtual inheritance and friend class.

In following example, we make the Final class non-inheritable. When a class Derived tries to inherit from it, we get compilation error.

An extra class MakeFinal (whose default constructor is private) is used for our purpose. Constructor of Final can call private constructor of MakeFinal as Final is a friend of MakeFinal .

Note that MakeFinal is also a virtual base class. The reason for this is to call the constructor of MakeFinal through the constructor of Derived, not Final (The constructor of a virtual base class is not called by the class that inherits from it, instead the constructor is called by the constructor of the concrete class).

#include<iostream>

using namespace std;

class Final; // The class to be made final

class MakeFinal { // used to make the Final class final

private:

MakeFinal() { cout << "MakFinal constructor" << endl; }

friend class Final;

};

class Final : virtual MakeFinal {

public:

Final() { cout << "Final constructor" << endl; }

};

class Derived : public Final { // compile error

public:

Derived() { cout << "Derived constructor" << endl; }

};

int main(int argc, char \*argv[]) {

return 0;

}

CE:

G:\coding\test>g++ -Wall -g cppmain.cpp -o out

cppmain.cpp: In constructor 'Derived::Derived()':

cppmain.cpp:8:3: error: 'MakeFinal::MakeFinal()' is private

MakeFinal() { cout << "MakFinal constructor" << endl; }

^

cppmain.cpp:20:13: error: within this context

Derived() { cout << "Derived constructor" << endl; }

^

Derived‘s constructor directly invokes MakeFinal’s constructor, and the constructor of MakeFinal is private, therefore we get the compilation error.

The object of Final class as it is friend class of MakeFinal and has access to its constructor. For example, the following program works fine.

#include<iostream>

using namespace std;

class Final;

class MakeFinal {

private:

MakeFinal() { cout << "MakeFinal constructor" << endl; }

friend class Final;

};

class Final : virtual MakeFinal {

public:

Final() { cout << "Final constructor" << endl; }

};

int main(int argc, char \*argv[]) {

Final f;

return 0;

}

Output: // Compiles and runs fine

MakeFinal constructor

Final constructor

# Aggregation: Classes Within Classes

Aggregation is not directly related to inheritance, both aggregation and inheritance are class relationships that are more specialized than associations.

In Inheritance, if a class B is derived by inheritance from a class A, we can say that “B is a kind of A.” For this reason, **inheritance** is often called a **“is a kind of”** relationship. Example: starling is a kind of bird.

**Aggregation** is called a **“has a”** relationship. Aggregation is also called a “part-whole” relationship. Example: the book is part of the library.

In object-oriented programming, aggregation may occur when one object is an attribute of another. Here’s a case where an object of class A is an attribute of class B:

class A {

};

class B {

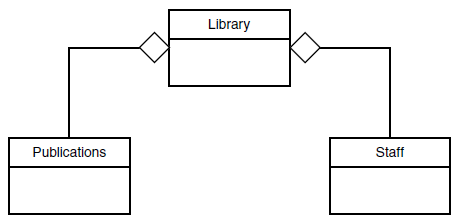
A objA; // define objA as an object of class A

};

In the UML, aggregation is considered a special kind of association.

Sometimes it’s hard to tell when an association is also an aggregation. It’s always safe to call a relationship an association, but if class A contains objects of class B, and is organizationally superior to class B, it’s a good candidate for aggregation.

Aggregation is shown in the same way as association in UML class diagrams, except that the “whole” end of the association line has an open diamond-shaped arrowhead.



UML class diagram showing aggregation.

# Composition: A Stronger Aggregation

Composition is a stronger form of aggregation. It has all the characteristics of aggregation, plus two more:

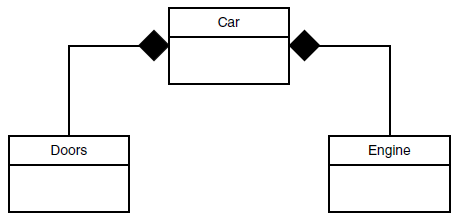
* The part may belong to only one whole.
* The lifetime of the part is the same as the lifetime of the whole.

A car is composed of doors (among other things). The doors can’t belong to some other car, and they are born and die along with the car.

Even a single object can be related to a class by composition. In a car there is only one engine.

While aggregation is a “has a” relationship, composition is a “consists of” relationship.

In UML diagrams, composition is shown in the same way as aggregation, except that the diamond-shaped arrowhead is solid instead of open.



UML class diagram showing composition.

# Choosing Composition vs. Inheritance

The is-a relationship is expressed with inheritance, and the has-a relationship is expressed with composition.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object to implement features of your new class, but the user of your new class sees the interface you’ve defined rather than the interface from the original class. To do this, you follow the typical path of embedding private objects of existing classes inside your new class.

## Subtyping

What if you want everything in the class to come through? This is called subtyping because you’re making a new type from an existing type, and you want your new type to have exactly the same interface as the existing type (plus any other member functions you want to add), so you can use it everywhere you’d use

the existing type. This is where inheritance is essential. You can see that subtyping solves the problem in the preceding example perfectly:

#include "../require.h"

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

class FName2 : public ifstream {

string fileName;

bool named;

public:

FName2() : named(false) {}

FName2(const string& fname): ifstream(fname.c\_str()), fileName(fname) {

assure(\*this, fileName);

named = true;

}

string name() const { return fileName; }

void name(const string& newName) {

if(named) return; // Don't overwrite

fileName = newName;

named = true;

}

};

int main() {

FName2 file("FName2.cpp");

assure(file, "FName2.cpp");

cout << "name: " << file.name() << endl;

string s;

getline(file, s); // These work too! bcoz FName2is a type of ifstream

file.seekg(-200, ios::end);

file.close();

}

## Private Inheritance

When you inherit privately, you’re “**implementing in terms of**;” that is, you’re creating a new class that has all of the data and functionality of the base class, but that functionality is hidden, so it’s only part of the underlying implementation.

There may occasionally be situations where you want to produce part of the same interface as the base class and disallow the treatment of the object as if it were a base-class object. private inheritance provides this ability.

You should think carefully before using private inheritance instead of composition; private inheritance has particular complications when combined with runtime type identification.

## Protected Inheritance

Protected derivation means “**implemented-in-terms-of**” to other classes but “**is-a**” for derived classes and friends.

# Operator overloading & inheritance

Except for the assignment operator, operators are automatically inherited into a derived class.

Check example in book

# Upcasting

The act of converting a derived class reference or pointer into a base class reference or pointer is called upcasting.

## Why “upcasting?”

Upcasting is always safe because you’re going from a more specific type to a more general type – the only thing that can occur to the class interface is that it can lose member functions, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

## Upcasting and the Copy-constructor

If you allow the compiler to synthesize a copy-constructor for a derived class, it will automatically call the base-class copy constructor, and then the copy-constructors for all the member objects (or perform a bitcopy on built-in types) so you’ll get the right behaviour. If you cast to a base-class object instead of a reference you will usually get undesirable results.

You can see that Child has no explicitly-defined copy-constructor. The compiler then synthesizes the copy-constructor by calling the Parent copy-constructor and the Member copy-constructor. If you try to write your own copy-constructor for Child and you make an innocent mistake and do it badly:

Child(const Child& c) : i(c.i), m(c.m) { }

then the default constructor will automatically be called for the base-class part of Child, since that’s what the compiler falls back on when it has no other choice of constructor to call.

You must remember to properly call the base-class copy-constructor (as the compiler does) whenever you write your own copy-constructor. This can seem a little strange looking at first but it’s another example of upcasting:

Child(const Child& c) : Parent(c), i(c.i), m(c.m) {

cout << "Child(Child&)\n";

}

The strange part is where the Parent copy-constructor is called: Parent(c). What does it mean to pass a Child object to a Parent constructor? But Child is inherited from Parent, so a Child reference is a Parent reference. The base-class copy-constructor call upcasts a reference to Child to a reference to Parent and uses it to perform the copy-construction. When you write your own copy constructors you’ll almost always want to do the same thing.

## Composition vs. Inheritance (revisited)

Composition: If you’ll ever need to upcast from your new class

Inheritance:

## Pointer & reference upcasting

Upcasting can also occur during a simple assignment to a pointer or reference:

Wind w;

Instrument\* ip = &w; // Upcast

Instrument& ir = w; // Upcast

Like the function call, neither of these cases requires an explicit cast.