Contents

[Difference between a Virtual Function and a Pure Virtual Function 2](#_Toc392699174)

[Abstract Classes VS Interfaces 3](#_Toc392699175)

[Aggregation VS Composition 6](#_Toc392699176)

[All Casts Explained 8](#_Toc392699177)

[Up-Down Casting 13](#_Toc392699178)

[Copy And Swap Idiom 16](#_Toc392699179)

[Values In C++ 18](#_Toc392699180)

# Difference between a Virtual Function and a Pure Virtual Function

#include <iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Difference between a virtual function

and a pure virtual function

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Base1 // Is \*\*NOT\*\* an abstract class, can be instantiated.

{

  public:

void Method1 ()  {  cout << "Base1::Method1" << endl;  }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- A virtual function makes its class a polymorphic base class. Derived

classes can override virtual functions. Virtual functions called through

base class pointers or references will be resolved at run-time.

- A virtual function or virtual method is a function or method whose

behavior can be overridden within an inheriting class by a function

with the same signature.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

virtual void Method2 ()  {  cout << "Base1::Method2" << endl;  }

};

class Derived1 : public Base1

{

  public:

void Method1 ()  {  cout << "Derived1::Method1" << endl;  }

void Method2 ()  {  cout << "Derived1::Method2" << endl;  }

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Base2PureVirtual // Is an abstract class, can't be instantiated.

{

  public:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- A pure virtual function or pure virtual method is a virtual function

that is required to be implemented by a derived class that is not abstract.

The virtual function can be overridden

and the pure virtual must be implemented.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

virtual void Method1 () = 0;

virtual void Method2 () { cout << "Base2PureVirtual::Method2" << endl; }

};

class Derived2 : public Base2PureVirtual

{

  public:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- Derived classes need to override/implement all inherited pure

virtual functions. If they do not, they too will become abstract.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Method1 ()  {  cout << "Derived1::Method1" << endl;  }

//void Method2 ()  {  cout << "Derived1::Method2" << endl;  }

};

int main(void)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// We can create/instantiate an object of this. This class contains

// only virtual function which can be overridden in the derived class.

Base1 b1;

Derived1 d1;

Base1& obj1 = d1;

obj1.Method1();  // Prints "Base1::Method1"

obj1.Method2();  // Prints "Derived1::Method2"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Derived2 d2;

Base2PureVirtual& obj2 =  d2;

obj2.Method1();  // Prints "Derived1::Method1"

obj2.Method2();  // Prints "Base2PureVirtual::Method2" because

 // Derived2 doesn't implement Method2

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

return 0;

}

# Abstract Classes VS Interfaces

#include <iostream>

#include <vector>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

When to create Abstract classes:

- Are there many classes that can be "grouped together" and described

by one noun? If so, have an abstract class by the name of this noun,

and inherit the classes from it. A key decider is that these classes

share functionality, and we would never instantiate just an Animal...

We would always instantiate a certain kind of Animal: an implementation

of our Animal base class.

- Cat and Dog can both inherit from abstract class Animal, and this

abstract base class will implement a method void Breathe() which all

animals will thus do in exactly the same fashion.

- We can make this method virtual so that we can override it for certain

animals, like Fish, which does not breath.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Animal

{

public:

virtual void Breath() = 0;

// Some code....

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

When to create Interface:

What kinds of verbs can be applied to my class, that might in general

also be applied to others?

Create an interface for each of these verbs.

- All animals can be fed, so I will create an interface called

IFeedable and have Animal implement that.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class IFeedable

{

public:

virtual void FeedTheAnimal(void) = 0;

};

class Dog: public Animal, public IFeedable

{

public:

void Breath()

{

cout<<"Dog is breathing..."<<endl;

}

void FeedTheAnimal(void)

{

cout<<"Feeding the dog..."<<endl;

}

};

class Fish: public Animal, public IFeedable

{

public:

void Breath()

{

// Fish can't breath.

}

void FeedTheAnimal(void)

{

cout<<"Feeding the fish..."<<endl;

}

};

class Horse: public Animal, public IFeedable

{

public:

void Breath()

{

cout<<"Horse is breathing..."<<endl;

}

void FeedTheAnimal(void)

{

cout<<"Feeding the horse..."<<endl;

}

};

class Lion: public Animal // I am not implementing IFeedable, cos I can't feed Lion

{

public:

void Breath()

{

cout<<"Lion is breathing..."<<endl;

}

};

int main(void)

{

Dog aDog;

Fish aFish;

Horse aHorse;

Lion aLion;

vector<Animal\*> allAnimals;

allAnimals.push\_back(&aDog);

allAnimals.push\_back(&aFish);

allAnimals.push\_back(&aHorse);

allAnimals.push\_back(&aLion);

for (std::vector<Animal\*>::iterator it = allAnimals.begin(); it != allAnimals.end(); ++it)

(\*it)->Breath();

aDog.FeedTheAnimal();

aFish.FeedTheAnimal();

aHorse.FeedTheAnimal();

return 0;

}

# Aggregation VS Composition

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Aggregation vs Composition

Simple rules:

A "owns" B = Composition : B has no meaning or purpose in the system without A

A "uses" B = Aggregation : B exists independently (conceptually) from A

Example 1:

A Company is an aggregation of People. A Company is a composition of Accounts.

When a Company ceases to do business its Accounts cease to exist but its People

continue to exist.

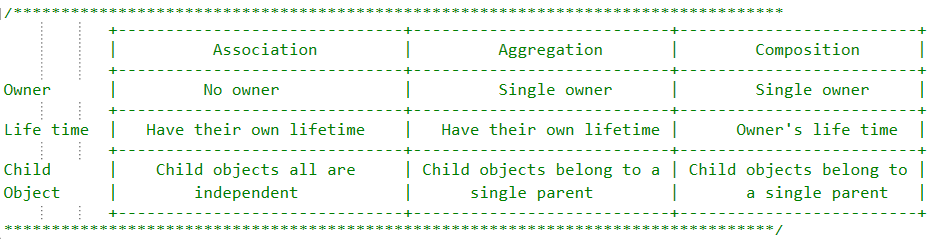
Example 2: (very simplified)

A Text Editor owns a Buffer (composition). A Text Editor uses a File (aggregation).

When the Text Editor is closed, the Buffer is destroyed but the File itself is

not destroyed.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/



class BClass

{

public:

BClass(int id)

{

this->ID = id;

cout<<"Constructing BClass object: "<<id<<endl;

}

~BClass()

{

cout<<"Destructing the BClass class object: "<<this->ID<<endl;

}

void print(void)

{

cout<<"This is just a print method which is called via AClass"<<endl;

}

private:

int ID;

};

class CClass

{

public:

CClass()

{

cout<<"Constructing CClass object"<<endl;

}

~CClass()

{

cout<<"Destructing CClass object"<<endl;

}

private:

};

class AClass

{

public:

AClass(BClass\* bsptr): bPtr(bsptr)

{

cout<<"Constructing AClass object"<<endl;

}

~AClass()

{

cout<<"Destructing AClass object"<<endl;

}

void Aggregation()

{

bPtr->print();

}

private:

BClass\* bPtr; // Aggregation

CClass composition; // Composition

};

int main(void)

{

BClass b(10);

cout<<"Demo for class aggregation relationship"<<endl;

{

AClass a(&b);

a.Aggregation();

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Constructing BClass object: 10

Demo for class aggregation relationship

Constructing CClass object

Constructing AClass object

This is just a print method which is called via AClass

Destructing AClass object

Destructing CClass object --------------------> Break point at return 0;

Destructing the BClass class object: 10 ------> Break point at }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# All Casts Explained

#include <iostream>

#include <thread>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

C Style casting

static\_cast<>()

dynamic\_cast<>()

reinterpret\_cast<>()

const\_cast<>()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma region Polymorphic Class

// Polymorphic classes

class Base

{

private:

int \_x;

public:

Base() { };

virtual ~Base() { };

};

class Derived : public Base

{

private:

int \_y;

public:

Derived() { };

~Derived() { };

};

#pragma endregion Polymorphic Class

#pragma region Non-polymorphic Class

// Non-polymorphic classes

class AnotherBase

{

public:

~AnotherBase() { }; // Not virtual

};

class AnotherDerived : public AnotherBase

{

public:

void f() { }; // Not virtual either

};

#pragma endregion Non-polymorphic Class

#pragma region Dynamic Cast Helper

typedef enum MyEnum

{

MSG\_DEBUG\_OUT, MSG\_XYZ

}MyEnum;

class Message

{

public:

MyEnum message\_id;

Message() : message\_id(MSG\_DEBUG\_OUT){}

virtual ~Message(){}

};

class DebugMsgData : public Message

{

public:

DebugMsgData(){}

~DebugMsgData(){}

};

class XYZMsgData : public Message

{

public:

XYZMsgData(){}

~XYZMsgData(){}

};

#pragma endregion Dynamic Cast Helper

#pragma region Helper Functions

void threadFunction(void \*data)

{

cout << "Inside threadFunction" << endl;

AnotherBase \*c = static\_cast<AnotherBase\*>(data);

}

int add10ToInteger(int\* ptr)

{

\*ptr = \*ptr + 10;

return (\*ptr);

}

void OnMessage(Message \*msg)

{

switch (msg->message\_id)

{

case MSG\_DEBUG\_OUT:

{

DebugMsgData \*data = dynamic\_cast<DebugMsgData\*>(msg);

if (data)

cout << "DebugMsgData:: data is not null" << endl;

else

cout << "DebugMsgData:: data is null" << endl;

break;

}

case MSG\_XYZ:

{

XYZMsgData \*data = dynamic\_cast<XYZMsgData\*>(msg);

if (data)

cout << "XYZMsgData:: data is not null" << endl;

else

cout << "XYZMsgData:: data is null" << endl;

break;

}

}

}

#pragma endregion Helper Functions

int main(int argc, char\* argv[])

{

#pragma region C-Style

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- static\_cast<>() gives you a compile time checking ability, C-Style cast doesn't.

- static\_cast<>() is more readable and can be spotted easily anywhere inside a C++

source code, C-Style cast isn't.

- Intentions are conveyed much better using C++ casts.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

// The static cast performs conversions between compatible types.

// It is similar to the C-style cast, but is more restrictive.

// For example, the C-style cast would allow an integer pointer to point to a char.

char c = 10; // 1 byte

int \*p = (int\*)&c; // 4 bytes

// BOOM!!!

//\*p = 5; // Run-time error: stack corruption

//int \*q = static\_cast<int\*>(&c); // Compile-time error

}

#pragma endregion C-Style

#pragma region static\_cast

{

AnotherBase c;

// Conversion from AnotherBase\* -> void\* is implicit

thread t1(&threadFunction, &c);  // threadFunction(&c) will be called

t1.join();

}

#pragma endregion static\_cast

#pragma region dynamic\_cast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

dynamic\_cast works ONLY with polymorphic class. And polymorphic class

is that which has at least one virtual function, even be it the destructor.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

Base\* b = new Derived(); // Creating derived class object and assign it to base class pointer

Derived\* d1 = static\_cast<Derived\*>(b); // Valid! d1 is a valid and correct pointer to a BClass

Derived\* d2 = dynamic\_cast<Derived\*>(b);// Valid! d2 is a valid and correct

pointer to a BClass

}

{

Base\* b = new Base();

Derived\* d1 = static\_cast<Derived\*>(b); // Invalid!

Derived\* d2 = dynamic\_cast<Derived\*>(b);// Valid, but d2 is now a null pointer

if (d2)

cout << "d2 is not null" << endl;

else

cout << "d2 is null" << endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Note that in dynamic\_cast, only the source type need to be polymorphic

in order to compile. If the destination isn't polymorphic, then

dynamic\_cast will return null pointer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

Base\* b = new Derived(); // Creating derived class object and assign it to base class pointer

AnotherDerived \*pD = dynamic\_cast<AnotherDerived\*>(b);

// OK - source (AClass) is polymorphic

if (pD)

cout << "pD is not null" << endl;

else

cout << "pD is null" << endl;

}

{

Message \*ptr = new DebugMsgData;

ptr->message\_id = MSG\_DEBUG\_OUT;

OnMessage(ptr);

delete ptr;

Message \*ptr1 = new XYZMsgData;

ptr1->message\_id = MSG\_XYZ;

OnMessage(ptr1);

delete ptr1;

}

#pragma endregion dynamic\_cast

#pragma region reinterpret\_cast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is the ultimate cast, which disregards all kind of type safety,

allowing you to cast anything to anything else, basically reassigning

the type information of the bit pattern.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

int i = 12345;

AnotherBase\* p = reinterpret\_cast<AnotherBase\*> (i); // DANGEROUS!!!

}

{

int i = 0;

void \*v = 0;

int c = (int)v; // is valid

//int d = static\_cast<int>(v); // is not valid, different types

int e = reinterpret\_cast<int>(v); // is valid, but very dangerous

}

#pragma endregion reinterpret\_cast

#pragma region const\_cast

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const\_cast is used to cast away the constness of variables.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

int val = 10;

const int \*ptr = &val;

int \*ptr1 = const\_cast <int \*>(ptr);

add10ToInteger(ptr1);

cout << val;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const\_cast can be used to change non-const class members inside a const

member function.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

class student

{

private:

int roll;

public:

// constructor

student(int r) :roll(r) {}

// A const function that changes roll with the help of const\_cast

void fun() const

{

(const\_cast <student\*> (this))->roll = 5;

}

int getRoll()  { return roll; }

};

student s(3);

cout << "Old roll number: " << s.getRoll() << endl;

s.fun();

cout << "New roll number: " << s.getRoll() << endl;

}

#pragma endregion const\_cast

return 0;

}

# Up-Down Casting

#include <iostream>

using namespace std;

class Shape

{

public:

Shape() {}

virtual ~Shape() {}

virtual void draw(void) { cout << "Shape: Draw Method" << endl; }

virtual void move(void) { cout << "Shape: Move Method" << endl; }

virtual void shrink(void) { cout << "Shape: Shrink Method" << endl; }

};

class Circle : public Shape

{

public:

Circle(){}

~Circle(){}

void draw(void) { cout << "Circle: Draw Method" << endl; }

void move(void) { cout << "Circle: Move Method" << endl; }

void shrink(void) { cout << "Circle: Shrink Method" << endl; }

void display(void) { cout << "Circle: Only CIRCLE has this" << endl; }

};

class Triangle : public Shape

{

public:

Triangle(){}

~Triangle(){}

void draw(void) { cout << "Triangle: Draw Method" << endl; }

void move(void) { cout << "Triangle: Move Method" << endl; }

void shrink(void) { cout << "Triangle: Shrink Method" << endl; }

};

class Square : public Shape

{

public:

Square(){}

~Square(){}

void draw(void) { cout << "Square: Draw Method" << endl; }

void move(void) { cout << "Square: Move Method" << endl; }

void shrink(void) { cout << "Square: Shrink Method" << endl; }

};

void play(Shape& s)

{

s.draw();

s.move();

s.shrink();

}

int main(void)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

UP-CASTING

The process of converting a derived-class reference or pointer to a

base-class. In other words, up-casting allows us to treat a derived

type as though it were its base type. It is always allowed for public

inheritance, without an explicit type cast. This is a result of the

is-a relationship between the base and derived classes.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

Circle c;

Triangle t;

Square sq;

// Without an explicit type cast we are converting derived

// class pointer to base class

play(c);

// Without an explicit type cast we are converting derived

// class pointer to base class

play(t);

// Without an explicit type cast we are converting derived

// class pointer to base class

play(sq);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DOWN-CASTING

The process of converting a base-class pointer (reference) to a derived-class

pointer (reference) is called down-casting.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

{

{

Shape newShape;

// Down-casting is not allowed without an explicit type cast.

// Circle \*ptrCircle = &newShape;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Shape\* pa =new Shape();

Circle\* pb =new Circle();

Shape\* paUpcast= new Circle();

Shape \*pa --> +-+   new Shape()

|a|

+-+

Circle\* pb --> +-+-+ new Circle()

|a|b|

+-+-+

Shape\* paUpcast-->  +-+-+ new Circle()

|a|b|

+-+-+

Circle\* pbDowncast -->  +-+ new Shape()

|a|

+-+

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Circle \*ptrCircle1 = (Circle \*)&newShape;

ptrCircle1->draw();

ptrCircle1->move();

ptrCircle1->shrink();

/\*\*\*\*Undefined Behavior\*\*\*\*/

ptrCircle1->display();

ptrCircle1 = nullptr;

ptrCircle1->display();

/\*\*\*\*Undefined Behavior\*\*\*\*/

}

{

Shape newShape;

Circle \*ptrCircle = dynamic\_cast<Circle \*>(&newShape);

if (ptrCircle)

ptrCircle->draw();

else

cout << "Can't be casted!!!" << endl;

}

}

return EXIT\_SUCCESS;

}

# Copy And Swap Idiom

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The rule of three/The Big Three

If you need to explicitly declare either the destructor, copy constructor

or copy assignment operator yourself, you probably need to explicitly

declare all three of them.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Copy And Swap Idiom

Why do we need it?

Any class that manages a resource needs to implement The Big Three.

While the goals and implementation of the copy-constructor and destructor

are straightforward, the copy-assignment operator is arguably the most nuanced

and difficult. How should it be done? What pitfalls need to be avoided?

The copy-and-swap idiom is the solution, and elegantly assists the

assignment operator in achieving two things: avoiding code duplication,

and providing a strong exception guarantee.

How does it work?

Conceptually, it works by using the copy-constructor's functionality to

create a local copy of the data, then takes the copied data with a swap function,

swapping the old data with the new data. The temporary copy then destructs,

taking the old data with it. We are left with a copy of the new data.

In order to use the copy-and-swap idiom, we need three things: a working

copy-constructor, a working destructor and a swap function.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <algorithm> // copy

#include <cstddef> // size\_t

using namespace std;

class myArray

{

private:

size\_t mSize;

int\* mArray;

public:

// (default) constructor

myArray(int value, size\_t size = 0) :

mSize(size), mArray(mSize ? new int[mSize]() : 0)

{

for (int i = 0; i < mSize; i++)

{

mArray[i] = value;

}

}

// copy-constructor

myArray(const myArray& other) :

mSize(other.mSize), mArray(mSize ? new int[mSize] : 0)

{

// note that this is non-throwing, because of the data

// types being used; more attention to detail with regards

// to exceptions must be given in a more general case, however

copy(other.mArray, other.mArray + mSize, mArray);

}

// destructor

~myArray() { delete[] mArray; }

friend void swap(myArray& first, myArray& second) // nothrow

{

// enable ADL (not necessary in our case, but good practice)

using swap;

// by swapping the members of two classes,

// the two classes are effectively swapped

swap(first.mSize, second.mSize);

swap(first.mArray, second.mArray);

}

myArray& operator=(myArray other) // (1)

{

swap(\*this, other); // (2)

return \*this;

}

void display(void)

{

cout << "------" << endl;

for (int i = 0; i < mSize; i++) cout << mArray[i] << endl;

cout << "------" << endl;

}

};

int main(void)

{

myArray temp(555, 5);

myArray temp1(777, 7);

myArray temp2(666, 6);

temp.display();

temp1.display();

temp2.display();

temp = temp1 = temp2;

temp.display();

temp1.display();

temp2.display();

return 0;

}

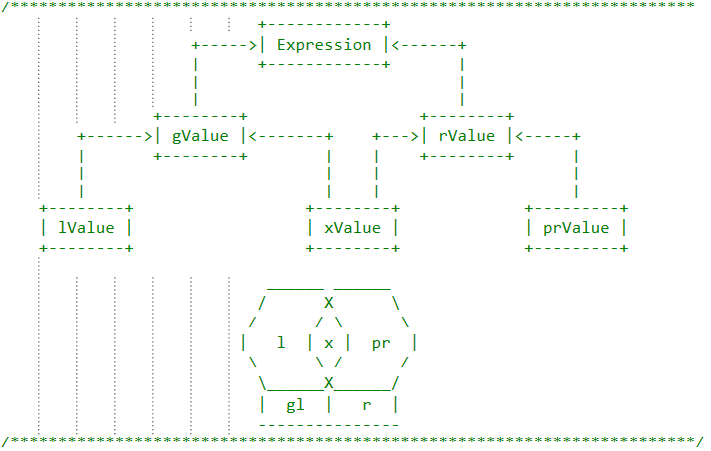
# Values In C++

#include <iostream>

#include <string>

using namespace std;

int globalI;



class MyClass

{

string name;

public:

// The expression move(some\_name)  is an xvalue, because move returns

// an rvalue reference (string&&).

// Every xvalue is also both a glvalue and an rvalue.

MyClass(string some\_name) : name(move(some\_name)) { }

string& originalName() { return name; }

string copyOfName() const { return name; }

};

int foobar(void) { return 10; }

int& foo(void) { return globalI; }

int main(void)

{

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

LVALUES:

An lvalue is an expression that refers to a memory location and allows

us to take the address of that memory location via the & operator.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int i = 42;

i = 43; // Ok, i is an lvalue

int\* p = &i; // Ok, i is an lvalue

int& foo();

foo() = 42; // Ok, foo() is an lvalue

int\* p1 = &foo(); // Ok, foo() is an lvalue

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RVALUES:

An rvalue is an expression that is not an lvalue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int foobar();

int j = 0;

j = foobar(); // Ok, foobar() is an rvalue

//int\* p2 = &foobar(); // ERROR, cannot take the address of an rvalue

j = 42; // Ok, 42 is an rvalue

}

{

class A

{

public:

A(){}

~A(){}

private:

};

A a;

A& a\_ref1 = a;  // an lvalue reference

// An rvalue reference is formed by placing an && after some type.

A a1;

//A&& a\_ref2 = a1;  // an rvalue reference

// An rvalue reference behaves just like an lvalue reference except that

// it can bind to a temporary(an rvalue), whereas you can not bind

// a(non const) lvalue reference to an rvalue.

A&  a\_ref3 = A();  // Error!

A&& a\_ref4 = A();  // Ok

}

{

MyClass temp("Nishith");

// Below expression is is an prvalue. copyOfName() returns an object

// (string), not a reference.

// Every prvalue is also an rvalue. (Rvalues are more general.)

string aString = temp.copyOfName();

// Below expression is is an lvalue. originalName() returns an

// lvalue reference (string&).

// Every lvalue is also a glvalue. (Glvalues are more general.)

temp.originalName() = "Jain";

}

}