Disclaimer: This material is a study notes, so for any technical issues please contact me to resolve it periodically.

A C++ Concepts

Contents

Listings

1 Tips and Tricks

1.1 Pointer V.s. references

The references are only introduced in C++, and it refers to an **"alias"** name for a certain existing variable. it's not an actual memory location.
It differs from pointers that the pointers are actual address holders but reference are not, and references can't be null and should be assigned to value in declaration. A reference cannot be re-assigned, and must be assigned at initialization.

Listing 1: Pointer V.s. references

```
#include <cstdio>
#include <string>


int main( int argc, char ** argv ) {
    int i = 3;

    // A pointer to variable i (or stores
    // address of i)
    int *ptr = &i;

    // A reference (or alias) for i.
    int &ref = i;


    int a = 5;
    int b = 6;
    int &ref = a;

    int &ref = b; //references can't be re-aasigned, At this line it
        will show error as "multiple declaration is not all
    return 0;
}
```

Observation:

```
In D 20 55
In B 4196784 0

...Program finished with exit code 0
Press ENTER to exit console.
```

Observation:

```
main.cpp: In function ‘int main(int, char**)’:
main.cpp:21:10: error: redeclaration of ‘int& p’
     int &p = b;  //references can't be re-aasigned, At this line it will show error as "multiple declaration is not al
          ^
main.cpp:20:10: note: ‘int& p’ previously declared here
     int &p = a;
          ^
```

1.2 Forward declarations in C++

1.3 Object Slice off

In C++, a Base class object can only be assigned to a Derived class object, but the other way is not possible. 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.

Listing 2: Object Slice-off example

```
#include <iostream>
#include <string>

class Employee {
private:
    std::string eName;
    float salary;

public:
    Employee(std::string _name, int _salary) : eName(_name),
        salary(_salary) { }
    virtual void Display(){
        std::cout << this->eName << std::endl;
        std::cout << this->salary << std::endl;
    }

};
```

```
class Programmer : public Employee {
private:
    std::string skills;

public:
    Programmer(std::string name, int salary , std::string _skill) :
        Employee(name, salary), skills(_skill){ }
    void Display(){
        Employee::Display();
        std::cout << this->skills << std::endl;
    }
};

// Global method, Base class object is passed by value
void someFunc (Employee obj)
{
    obj.Display();
}


int main(int argc, char** argv){
    Programmer* pProgrammer = nullptr;
    Employee* pEmp = nullptr;

    Employee e = Employee("Ahmed", 5000);
    //e.Display();

    Programmer p = Programmer("Mohamed", 7000, "C++");
    //p.Display();

    // Setting a Derived class object to Base class object is allowed
    p = e;
    e.Display();

    //The reverse is not allowed, as "A Programmer is an Employee, but
        not every Employee is a Programmmer"
    //std::cout << "P of size: " << sizeof(p) << " Bytes can't be
        assigned to e of size: " << sizeof(e) << " Bytes\n";
    e = p; //this is not allowed, slice off !, in some compliers it
        causes error
    someFunc(p); // Slice-off also happens

    //Upcasting: Assign Derived class pointer to Base class pointer
    pProgrammer = &p;
    pEmp = pProgrammer;
    pEmp->Display();

    //Downcasting (is not allowed in C++), you can achieve using the
        dynamic cast
```

```
    //pProgrammer = pEmp; //Error: downcast is not allowed, you can
        achieve using the dynamic_cast

    return 0;
}
```

1.4 C++ Constructors Types

In C++ there are different types of constructors:

- Default Constructor A default constructor is a contractor that takes no parameters, or one or all of its parameters has a default value

  For each class the default constructor is there, unless it's been explicitly deleted, and it can be deleted for any of the following reasons:

  –

Listing 3: The Default constructors

```
class A
{
private:
    int x;

public:
    A(int x = 1): x(x) {} // user-defined default constructor
};

class B: A
{
    // B::B() is implicitly-defined, calls A::A()
};

class C
{
    A a;
    // C::C() is implicitly-defined, calls A::A()
};

class D: A
{
public:
    D(int y): A(y) {}
    // D::D() is not declared because another constructor exists
};

class E: A
{
```

```
public:
    E(int y): A(y) {}
    E() = default; // explicitly defaulted, calls A::A()
};

class F
{
    int& ref; // reference member
    const int c; // const member
    // F::F() is implicitly defined as deleted
};

int main()
{
    A a;
    B b;
    C c;
//  D d; // compilation error
    E e;
//  F f; // compilation error
}
```

- Parameterized Constructor

  Please refer also to 1.4.1 the explicit constructors

- Copy Constructor The copy constructor is the constructor which its parameters is an object from the same class, and it's purpose is to copy this object to a brand-new object. please also refer to 1.7 the shallow copying section.

Listing 4: The copy constructors

```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
public:

//paramterized constructor
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &otherPoint) {
        x = otherPoint.x;
        y = otherPoint.y;
    }
```

```
    int getX() { return x; }
    int getY() { return y; }
};

int main(int argc, char** argv)
{
    Point p1(10, 15); //paramterized constructor is called here
    Point p2 = p1;  // Copy constructor is called here
    Point p3(p2);   //Also the copy constructor is called here

    return 0;
}
```

1.4.1 The explicit constructors (the explicit specifier)

In C++ an implicit type conversion happens if a class have a constructor with at least one parameter. but the *explicit* keyword is used to prevent that.

Listing 5: The implicit type conversion

```
#include <iostream>

class Type{
private:
    int num;

public:
    //We have here two constructors
    Type(int i) { std::cout << i; }
    Type(char* c) { std::cout << c; }
    Type(std::string s){ std::cout << s; }

};

void foo(const Type& local){

}

int main(int argc, char** argv){

    Type typeObj = 1; //here 1 is implicitly converted to Type object,
        equivalent to Type(1)
    Type anotherObj = 'D'; // the same here the character D is converted
        to A('D') class
    foo('C'); //The same here, C is converted to Type class

    return 0;
}
```

If we tagged a constructor as explicit, then an implicit conversion won't happen. so if you need to have it work, you need to call the constructor explicitly.

Listing 6: The explicit construction

```
#include <iostream>
#include <string>

class Type{
private:
    int num;

public:
    //We have here two constructors
    explicit Type(int i) { std::cout << i << std::endl; }
    explicit Type(char* c) { std::cout << c << std::endl; }
    explicit Type(std::string s){ std::cout << s << std::endl; }

};

void foo(const Type& local){

}

int main(int argc, char** argv){

    //Type typeObj = 1; //Compilation error, no type conversion happens
    //Type anotherObj = 'D'; //Compilation error
    //foo('C'); //Compilation error

    //In order to make it work, you should call the construtor explicitly
    Type type1 = Type(1);
    Type type2 = static_cast<Type>("string");
    Type type3('c');

    return 0;
}
```

1.5 C++ Objects Construction Order

During the construction, the following construction order is been executed:

- During Construction
    1. Constructors of virtual base classes are executed, in the order that they appear in the initialization list.
    2. Constructors of non-virtual base classes are executed, in the declaration order.

3. Constructors of class private members are executed in the declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

- During Destruction

1. In a reverse order, the destructors of the derived classes will be called first, then the base classes destructors a reverse order to the initialization.
2. The body of the destructor are called

Let's have a simple example

Listing 7: Object Construction Order

```
#include <iostream>

template<class T>
void Display (T);

class A
{
public:
    A(int n = 2) : m_i(n) {
        Display("A Construction");
    }

    ~A() {
        Display("A Destruction");
        Display(m_i);
     }

protected:
    int m_i;
};

class B : public A
{
public:
    B(int n) : m_a1(m_i + 1), m_a2(n) {
        Display("B Construction");
    }

public:
    ~B()
    {
        Display("B Destruction");
        Display(m_i);
        --m_i;
    }
```

```
private:
    A m_a1;
    A m_a2;
};

class C : public virtual A {


};

template<class T>
void Display (T msg) {
    std::cout << msg << std::endl;
}

int main(int argc, char** argv)
{
    { B b(5); }

    std::cout << std::endl;

    return 0;
}

/* Output:
 * A Construction
 * A Construction
 * A Construction
 * B Construction
 * B Destruction
 * 2
 * A Destruction
 * 5
 * A Destruction
 * 3
 * A Destruction
 * 1
*/
```

1.6 The constructor V.s. the equal assignment operator

In C++, you need to understand when exactly a constructor is called and on the other hand when the equal assignment operator is called, let's have the following example:

Listing 8: Constructors V.s. equal operators

```
#include<iostream>
```

```
class MyClass
{
    public:
    MyClass() {
        std::cout << "The construtor is called" << std::endl;
    }

    MyClass(const MyClass &t){
        std::cout << "The construtor (copy) is called" << std::endl;
    }

    MyClass& operator= (const MyClass &t)
    {
        std:: cout<<"Assignment operator is called "<< std::endl;
        return *this;
    }
};

// Driver code
int main()
{
    MyClass t1, t2; //For sure normal constructor os called

    t2 = t1; //Of course the assignment operator will be called here

    //Don't be confused, the = operator will be called ONLY if we had
        already
    // a constructed copy of t3, otherwise the normal constructor is
        called.
    MyClass t3 = t1;

    return 0;
}
```

The point here is that the equal operator (=) will ONLY be called when we a have a constructed objects.

1.7 Shallow Copying (THE RULE OF THREE)

When creating copies of arrays or objects one can make a deep copy or a shallow copy. A shallow copy can be made by simply copying the reference, but in deep copying you make it by copying every element of the object or an array to another location. let's have an example to understand:
In this example, we create normal two objects from the Point class, and when we copy the P2 = P1, P1 is copied to an independent copy to P2. then this is deep copying.

Listing 9: Object Deep Copy

```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
public:

    Point(int x1, int y1) :x(x1), y(y1){ }
    int getX() { return x; }
    int getY() { return y; }

    void setX(int val) { x = val; }
    void setY(int val) { y = val; }

};

int main(int argc, char** argv)
{
    Point p1(10, 15);
    Point p2 =p1; //A deep copy is taken from P1 set to P2

    std::cout << "Changing X:" << std::endl;
    p1.setX(30); //When we modify x in P1, X in P2 won't affected
    cout << "X in P2: "<<p2.getX() << std::endl;
    cout << "X in P1: "<< p1.getX() << std::endl;

    return 0;
}

//Output
/*
Changing X:
X in P2: 10
X in P1: 30
*/
```

What if we modified the Point class to have one member **"pointer"** variable.

Listing 10: Object Shallow copy

```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
```

```
    int* p;
public:
    Point(){}
    Point(int x1, int y1, int val) {
        x = x1;
        y = y1;
        //Init p
        p = new int;
        *p = val;
     }

    int getX() { return x; }
    int getY() { return y; }

    void setX(int val) { x = val; }
    void setY(int val) { y = val; }

    void setP(int val) { *p = val; }
    int getP() { return *p; }
};

int main(int argc, char** argv)
{
    Point p1(10, 15, 20);
    Point p2 =p1;

    std::cout << "Changing X:" << std::endl;
    p1.setX(30); //When we modify x in P1, X in P2 won't affected
    cout << "X in P2: "<<p2.getX() << std::endl;
    cout << "X in P1: "<< p1.getX() << std::endl;

    std::cout << "Changing Poiner P:" << std::endl;
    p1.setP(30);
    cout << "P Pointer in P2: "<< p2.getP() << std::endl;
    cout << "P Pointer in P1: "<< p1.getP() << std::endl;

    return 0;
}

//Output
/*
Changing X:
X in P2: 10
X in P1: 30

Changing Poiner P:
P Pointer in P2: 30
P Pointer in P1: 30
*/
```

As you can see that the pointer P is referring to the same value in both independent objects P1, and P2 !, and this is called *"shallow copy"*; as in a shallow copy can be made by simply copying the reference
You might need to override this behavior by defining a copy constructor to allows make a new copy of the pointers in your class.

Listing 11: Define copy constructor for deep copy

```
#include<iostream>
using namespace std;

class Point
{
private:
    int x, y;
    int* p;
public:
    Point(){}
    Point(int x1, int y1, int val) {
        x = x1;
        y = y1;
        //Init p
        p = new int;
        *p = val;
     }

    Point(const Point &otherPoint) {
        x = otherPoint.x;
        y = otherPoint.y;

        //Init P
        p = new int;
        *p = *(otherPoint.p);
    }

    int getX() { return x; }
    int getY() { return y; }

    void setX(int val) { x = val; }
    void setY(int val) { y = val; }

    void setP(int val) { *p = val; }
    int getP() { return *p; }
};

int main(int argc, char** argv)
{
    Point p1(10, 15, 20);
    Point p2 =p1;
```

```
    std::cout << "Changing X:" << std::endl;
    p1.setX(30); //When we modify x in P1, X in P2 won't affected
    cout << "X in P2: "<<p2.getX() << std::endl;
    cout << "X in P1: "<< p1.getX() << std::endl;

    std::cout << "Changing Poiner P:" << std::endl;
    p1.setP(30);
    cout << "P Pointer in P2: "<< p2.getP() << std::endl;
    cout << "P Pointer in P1: "<< p1.getP() << std::endl;

    return 0;
}

//Output
/*
Changing X:
X in P2: 10
X in P1: 30

Changing Poiner P:
P Pointer in P2: 20
P Pointer in P1: 30
*/
```

1.7.1 Deep Copying needs the rule of three

In C++, The rule of three and rule of five (in C++11) are rules of thumb for the building of exception-safe code and for formalizing rules on resource management. It accomplishes this by prescribing how the default members of a class should be used to accomplish this task in a systematic manner.

The rule of three/five says: If a class defines one (or more) of the following it should probably explicitly define all five (and of course a deep copying needs that).

- destructor
- copy constructor
- copy assignment operator
- move constructor (C++ 11)
- move assignment operator (C++ 11)

1.8 C++ Exception Class

In C++ you can inherit the *exception* class and define your own exceptions, but you need to be caution when passing this exception to a catch block, **You**

shouldn't catch an exception by value!.
Let's see if we catch an exception by value:

Listing 12: Catching excpetions by value

```
#include <iostream>
#include <Exception>

class MyException : public std::exception {
private:
    int errorCode;
    std::string errorMsg;
    std::string message;

void ConstructMessage()
{
    message = std::to_string(errorCode) + " " + errorMsg;
}

public:
explicit MyException(int code ,const std::string& msg):
        errorMsg(msg),errorCode(code) { ConstructMessage(); }

const char *what() const throw(){
    return this->message.c_str();
}

~MyException() throw () {}

};


int main(int argc, char** argv){
    try {
        throw MyException(404, "Page not found!" );
    } catch (const std::exception e) { //Slicing of for the MyExcpetion
         object
        std::cout << "Caught " << typeid(e).name() << std::endl;
        std::cout << e.what() << std::endl;
    }
}

//Output
// => Caught Std exception
```

The exception object is sliced off, and the message and code will not be available.

The solution to that is to catch the exception by reference. not by value.

Listing 13: Catching excpetions by reference

```
#include <iostream>
#include <string>
#include <exception>

class MyException : public std::exception {
private:
    int errorCode;
    std::string errorMsg;
    std::string message;

void ConstructMessage()
{
    message = std::to_string(errorCode) + " " + errorMsg;
}

public:
explicit MyException(int code ,const std::string& msg):
        errorMsg(msg),errorCode(code) { ConstructMessage(); }

const char *what() const throw(){
    return this->message.c_str();
}

~MyException() throw () {}

};


int main(int argc, char** argv){
    try {
        throw MyException(404, "Page not found!" );
    } catch (const std::exception& e) {
        std::cout << "Caught " << typeid(e).name() << std::endl;
        std::cout << e.what() << std::endl;
    }
}

//
// Caught MyException
```

Note: always throw by value (don't throw by pointer) and catch by reference.

1.8.1 difference between throw, and throw e

If you are planning to re-throw the exception again you need to understand the difference between *throw* and throw e within the catch context.

Listing 14: Re-throw an exception

```
#include <iostream>
#include <string>
#include <exception>

class MyException : public std::exception {
private:
    int errorCode;
    std::string errorMsg;
    std::string message;

void ConstructMessage()
{
    message = std::to_string(errorCode) + " " + errorMsg;
}

public:
explicit MyException(int code ,const std::string& msg):
        errorMsg(msg),errorCode(code) { ConstructMessage(); }

const char *what() const throw(){
    return this->message.c_str();
}

~MyException() throw () {}

};


int main(int argc, char** argv){
    try {
        try {
            throw MyException(404, "Page not found!" );
        } catch (const std::exception& e) {
            std::cout << "Caught " << typeid(e).name() << std::endl;
            std::cout << e.what() << std::endl;
            // /throw e; //Slicing
            //the solution is to use throw only!, try the following
            throw;
        }
    }catch(const std::exception& e){
        std::cout << "Caught " << typeid(e).name() << std::endl;
        std::cout << e.what() << std::endl;
    }
}
```

1.8.2 Stack Un-winding

When you throw an exception from a function that doesn't have the catch clause, then this function will be destroyed from the stack, and the exception handler searches on the caller function for the exception handling, if it still doesn't exists, then it searches again for the caller in the call stack and this is called **stack unwinding**.
sometimes this can cause issues like memory leaks, let's have an example:

Listing 15: Stack Un-winding

```
#include <iostream>


void func1( int x )
{
    char* pleak = new char[1024];
    std::cout << "func1 start" << std::endl;

    if ( x ) throw std::runtime_error( "error" );

    delete [] pleak; //possible memory leadk
    std::cout << "func1 end" << std::endl; //not reached
}

void func2()
{
    char* pleak = new char[1024];
    std::cout << "func2 start" << std::endl;

    func1( 1 );

    delete [] pleak; //Possible memory leak
    std::cout << "func2 end" << std::endl;//not reached
}

int main()
{
    try
    {
        func2();
    }
    catch ( const std::exception& e )
    {
        std::cout << "Catch clause";
    }

    return 0;
}
```

```
//Output
/*

*/
```

So how can we solve that ? Is by using RAII techniques!, Simply you can used smart pointer in this condition as one of RAII techniques.

RAII - Mutex that locks itself!

From CppReference: **Resource Acquisition Is Initialization** or RAII, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use **to the lifetime of an object.**
Which means if the object is gone the resource will be released and will be available for others to used.

Listing 16: RAII - the lock that releases itself

```
std::mutex m; //Normal

void bad()
{
    m.lock();                   // acquire the mutex
    f();                        // if f() throws an exception, the
        mutex is never released
    if(!everything_ok()) return; // early return, the mutex is
        never released
    m.unlock();                 // if bad() reaches this statement,
        the mutex is released
}


//The good approach is to used a lock_gurad, and make it local
//The lock_gurad will manage to release itself (like smart
    pointers)
//if any exception happend

void good()
{
    std::lock_guard<std::mutex> lk(m); // RAII class: mutex
        acquisition is initialization
    f();                            // if f() throws an exception,
        the mutex is released
    if(!everything_ok()) return;    // early return, the mutex is
        released
}
```

C++ 11 provide some new features for this purpose, like lock_gurad, smart pointers, ... and so one. For more info: please visit this

1.8.3 Some tips of exception safety

- *throw e*, will make a new instance of the exception call and throw it again. while *throw* only will re-throw the current exception and to be catch in

the next clause.

- Don't throw exceptions while objects construction failures, the object will be thrown away and the resources might not be released. Alternative: Use a flag to inform other members that construction failed
- Don't throw exceptions on the object destruction
- Don't throw exception in overloaded delete or delete[] operators, it might cause a memory leaks.

You might have a look also at this resources: Exception Safety

1.9 The const safe member functions

Normally we use the const keyword to keep the values unmodified, that's true and in C++ we use also to provide a const safe functions that are not allowed to modify the members of a certain object.

- Const Safe Functions which NOT allowed to alter the class members variables
- Sending parameters as const

Listing 17: Const Safe Memeber functions

```
#include <iostream>
#include <vector>

class MyClass{
private:
    int size;
    std::vector<int> vec;
MyClass(){
    this->size = 3;
    this->vec.push_back( 3 );
    this->vec.push_back( 4 );
    this->vec.push_back( 8 );
}

int getSize() const { //By having the const after the function operator,
     you assure that getInt won't modify any of the members
    return size;
}

void alterSize() const{
    this->size = 4; //Compliation error
}

int getValue(int index)const{
```

```
    return vec[index];
}

//Use const iterators with const safe function as a good practice
void alterVector() const{
    for ( std::vector<int>::const_iterator itr = vec.begin(), end =
        vec.end(); itr != end; ++itr ){
           *itr = 0; //compliation error
           // just print out the values...
           std::cout<< *itr <<std::endl;
    }
}

//FYI: If you are passing const vectors, you can access them only with
    const iterators
static void printOtherConstVector (const std::vector<std::int>& _vec){
       for ( std::vector<std::int>::const_iterator itr = _vec.begin(),
            end = _vec.end(); itr != end; ++itr ){
           // just print out the values...
           std::cout<< *itr <<std::endl;
    }
}

};

int main(int argc, char** argv){

}
```

1.9.1 Using const_cast¡¿ to allow altering in const safe functions

If your application needs to alter the class variables within a const safe functions (you might not go this way), you might use a const_cast() yo alter the values. or to explicitly make a certain variable as *mutable*, and let's see the next example:

Listing 18: Using const_cast to update inside a const member function

```
#include <iostream>
#include <vector>

class MyClass{
private:
    int size;
    mutable int value;

int getSize() const { //By having the const after the function operator,
    you assure that getInt won't modify any of the members
    return size;
}
```

```
int alterValues() const{
    value = 50; //it's alerable as we use the word mutable
    const_cast<MyClass*>(this)->size = 5; // also you can alter size
        using a const_cast()
}

};

int main(int argc, char** argv){

}
```

1.10 Using the STL algorithms

1.10.1 Define you own algorithm behavior

1.10.2 template algorithms doesn't change the object size

1.10.3 C++ Iterators

A C++ iterators are a generalization of the C++ pointers for the STL Containers, we have five types of iterators, each one can be used with some STL containers while others can not:

- Input Iterators
- Output Iterators
- Forward Iterator
- Bidirectional Iterators
- Random-Access Iterators

For all the list of algorithms refer to this link, and please note that some algorithms are used only for C++11 standard.

1.11 C++ sub-objects layout

1.11.1 The virtual table "vtable", and virtual pointer "__vptr"

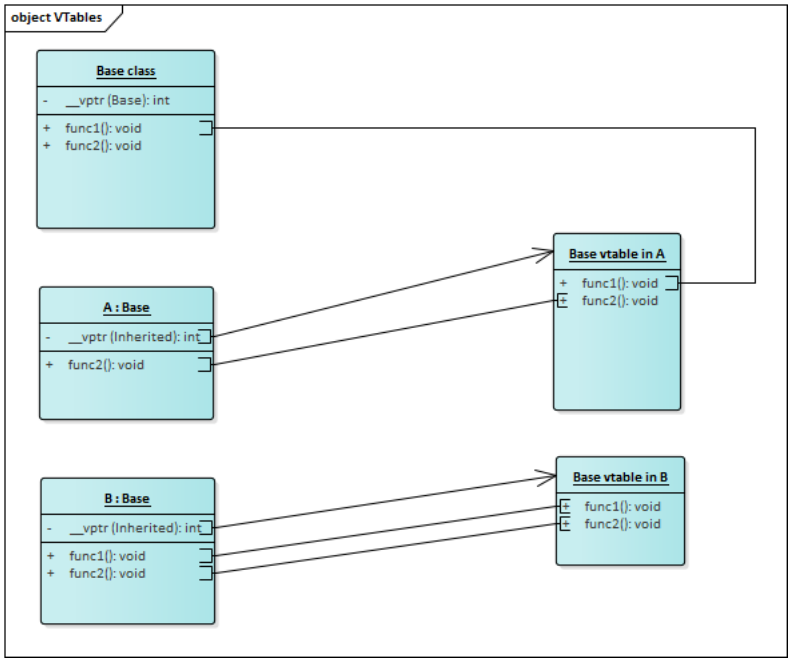
In C++, if a vtable and __vptr are a hidden fields to achieve a polymorphic objects, if we have a class with at least one virtual function you should expect to have a virtual table constructed for it, and let's start with the following to clarify that.

Listing 19: polymorphic class V.s. Normal C++ Class

```
#include<iostream>
```

```cpp
class NonVirtualBase{
    void foo(){}
};

class VirtualBase{
    virtual void foo(){}
};


int main(int argc, char** argv){

    NonVirtualBase b1;
    VirtualBase b2;

    std::cout << "The size of the a normal classs: " << sizeof(b1) << "
        Bytes" << std::endl;
    std::cout << "The size of a class has at least one virtual function:
        " << sizeof(b2) << " Bytes" << std::endl;

    return 0;
}

//Output
// The size of the a normal classs: 1 Bytes
// The size of a class has at least one virtual function: 8 Bytes
```

As you can see that the class with a virtual function more size reserved for the virtual, and let's explore more how the virtual functions can be handled during the inheritance.



In a base class with at least one virtual function is there, the virtual table constructed, and as you can see from the previous figure, in the base class a virtual point *__vptr* is used to point of this virtual table *vtable* for each class will inherit from the base, and you might noticed that the virtual table is just an array of pointer to functions which will refer to a certain function in the base or in the Derived class.
if the Derived class is inheriting for the Base, and then the Derived is overriding this feature of the Base, this is called a Polymorphism which we have features are already inherited with no change but there are others will be overridden by a new feature/properties/behaviors in the Derived classes, and this is also referring to a "Late Binding Concept", as in the runtime this binding is happened by the help of the virtual tables unlike the Early Binding by the complier for the overloaded functions at the compile time.

To understand more, let's have a real example and then to look at a memory layout of a Base and Derived classes:

Listing 20: Changing the behavior of a Base class

```
class Base {
protected:
  int foo;
public:
  int method(int p) {
    return foo + p;
  }
```

```
  virtual void method2(){ }
};

struct Point {
  double cx, cy;
};

class Derived : public Base {
public:
  int method(int p) {
    return foo + bar + p;
  }

protected:
  int bar, baz;
  Point a_point;
  char c;
};


int main(int argc, char** argv) {
  return sizeof(Derived);
}
```

And by using clang AST, we can explore the class layout dump as following:

Listing 21: Derived Class layout

```
$ clang -cc1 -fdump-record-layouts virtual-object-layout.cpp

*** Dumping AST Record Layout
         0 | class Base
         0 |   (Base vtable pointer)
         8 |   int foo
           | [sizeof=16, dsize=12, align=8,
           |  nvsize=12, nvalign=8]

*** Dumping AST Record Layout
         0 | struct Point
         0 |   double cx
         8 |   double cy
           | [sizeof=16, dsize=16, align=8,
           |  nvsize=16, nvalign=8]

*** Dumping AST Record Layout
         0 | class Derived
         0 |   class Base (primary base)
         0 |     (Base vtable pointer)
         8 |     int foo
```

```
12 |   int bar
16 |   int baz
24 |   struct Point a_point
24 |     double cx
32 |     double cy
40 |   char c
   | [sizeof=48, dsize=41, align=8,
   |  nvsize=41, nvalign=8]
```

Please refer also to this interesting article to do the same but using GDB.

1.11.2 The use of virtual, override, and final specifiers

With the override keyword; you will force the compiler to check the base class to see if there is a virtual function with this **exact signature**. And if there is not, the compiler will a compilation error.

Note: The virtual keyword is not necessary in the derived class. However it makes code clearer. Also in C++11 override keyword is introduced which allows the source code to clearly specify that a member function is intended to override a base class method.

1.11.3 The sub-objects layout

Follow this links: Cpp Sub-Objects

1.12 Writing a template class

You might need to implement a generic class, and a sake of organization you might also need to implement it in separate .hpp and .cpp files, if you faced some problems while doing that, we might go together to understand how it works.

1.12.1 template template parameters

1.12.2 Variable template C++14

1.13 The Cpp Run-time Type Information (RTTI) Support

RTTI stands for Run-Time Type Information. It is used to retrieve information about an object at runtime (as opposed to compile time). In non-polymorphic languages there is no need for this information, because the type is always known at compile time and so in runtime. In a polymorphic language like C++, there are situations where the concrete type is not known at compile time.

Normally you can achieve that by using:

- *typeid* operator

- and *typeinfo* class

typeid returns *const std::type_info* which can be accessed to get the *type-info::name*, *typeinfo::hash_code* C++ 11 only, and you need to note that in C++ "The copy and assignment operators of type_info are deleted: objects of this type cannot be copied."

Listing 22: RTTI using the typeid operator

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };

struct C { };
struct D : C { };

int main() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;

  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
}

//Output
/*
ap: B
ar: B
cp: C
cr: C
*/
```

Also you need to know that the comparison operators "==" and "!=" are overloaded in this class which means that you can use to compare between object types at runtime.

Listing 23: references v.s. Pointers

```
#include <iostream>
#include <typeinfo>
```

```cpp
typedef int MyInt_t;

int main() {
    // the == operator
    (typeid(int) == typeid(MyInt_t)) ? std::cout << "same type\n" :
        std::cout << "differnet type\n";

    // the != operator
    (typeid(int) != typeid(int*)) ? std::cout << "differnet type" :
        std::cout << "same type";
}
```

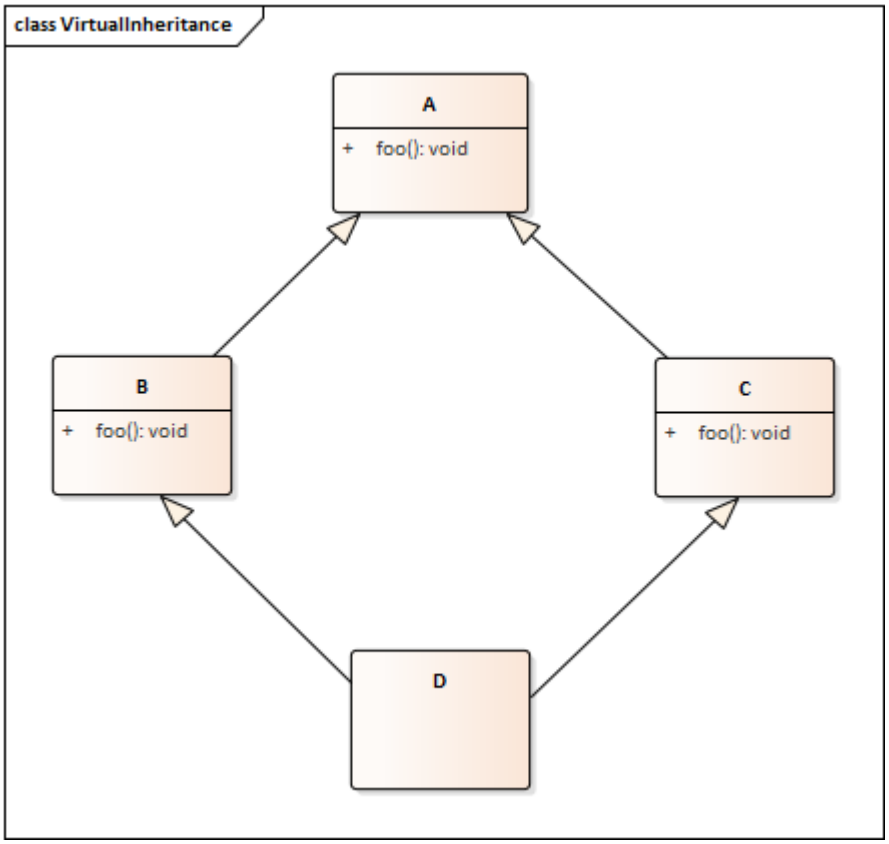
1.14 Virtual inheritance

In general there are different types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multi-level Inheritance
- Hybrid (Virtual) Inheritance

We will discuss here an issue with the virtual inheritance, and the Diamond Problem in the hybrid inheritance. As you can see in the figure below, the hybrid inheritance might consist of different kinds of inheritance, let's say **Multiple Inheritance** as per "D" inherits from both "B" and "C", and **Hierarchical Inheritance** as "B" and "C" inherits from "A", and **Multilevel Inheritance** as "D" inherits from "B" or "C" and both inherits from "A (This forms a diamond!).

The problem here, as "A" class has a member function called *foo()*, then it will be inherited in both "B" and "C", the both will be having this function, when "D" tries to inherit from both, then which *foo()* function will be there !.



To be honest, the complier will throw an error: Ambiguous Type!

Listing 24: The Virtual Inheritance Problem

```
#include <iostream>

class A {
public:
    void foo();
};

class B : public A { };
class C : public A { };

class D : public B, public C { };

int main(int argc, char** argv){

    D dObj;
    dObj.foo(); //D::foo is ambiguous!
```

```
    return 0;
}
```

```
aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/MyLaTex/Cpp-TipsAndTricks/cpp_code/inheritance
$ g++ virtual-inheritance-problem.cpp
virtual-inheritance-problem.cpp: In function 'int main(int, char**)':
virtual-inheritance-problem.cpp:16:10: error: request for member 'foo' is ambiguous
     dObj.foo();
          ^~~
virtual-inheritance-problem.cpp:5:10: note: candidates are: 'void A::foo()'
     void foo();
          ^~~
virtual-inheritance-problem.cpp:5:10: note:                 'void A::foo()'
```

but after we defined the inheritance of "B" and "C" to "A" as virtual, then it's been solved and you can compile normally!.

Listing 25: The Virtual Inheritance Solution

```
#include <iostream>

class A {
public:
    void foo() { std::cout << "Hello from A!"; }
};

class B : public virtual A { };
class C : public virtual A { };

class D : public B, public C { };

int main(int argc, char** argv){

    D dObj;
    dObj.foo(); //D::foo is not ambiguous anymore

    return 0;
}
```

```
aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/MyLaTex/Cpp-TipsAndTricks/cpp_code/inheritance
$ g++ virtual-inheritance-solution.cpp

aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/MyLaTex/Cpp-TipsAndTricks/cpp_code/inheritance
$ ./a.exe
Hello from A!
```

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

1.15 Dynamic Dispatch and late binding (Polymorphism)

In C++ the Polymorphism can be achieved in two ways:

- Compile time Polymorphism (Function Overloading)
- Runtime Polymorphism (Late Binding)

2 Advanced Topics

2.1 Lambda Expression

The lambda expression, anonymous functions or inline function pointers are used in the code were a complete function prototype is not needed and the function will be used only in this location.
You can define a lambda expression as follow:

Listing 26: Using Lambda Expression Notation

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

As a simple usage of inline lambda expression:

Listing 27: Using Lambda Expression Example

```
#include <iostream>
using namespace std;

int main() {
      auto sum = [](int x, int y) { return x + y; };
      //You can also define the previous line as following:
      //auto sum = [](int x, int y) -> int { return x + y; };
      cout << sum(5, 2) << endl;
      cout << sum(10, 5) << endl;
  }
```

You can use lambda expression to define a behavior of STL algorithm

Listing 28: Use lambda expression to define a STL algorithm behaviour

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool is_greater_than_5(int value)
{
```

```
    return (value > 5);
}

int main()
{
    vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };
    auto greater_than_5_count = count_if(numbers.begin(), numbers.end(),
        is_greater_than_5);

    cout << "The number of elements greater than 5 is: "
        << greater_than_5_count << "." << endl;
}

//Or you can use a similar version of the
// count_if() using the lambda expression as follow

#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
    vector<int> numbers { 1, 2, 3, 4, 5, 10, 15, 20, 25, 35, 45, 50 };
    auto great_than_5_count = count_if(numbers.begin(), numbers.end(),
        [](int x) { return (x > 5); });
    cout << "The number of elements greater than 5 is: "
        << greater_than_5_count << "." << endl;
}
```

2.1.1 Using the STL algorithms

The following code example used count_if() with defined lambda expression, the function defines a count if greater than 5:

Listing 29: C++ Example

```
#include <iostream>
using namespace std;

class A { int x; };
class B { int y; };
class C : public A, public B { int z; };

int main() {
cout << "sizeof(A) == " << sizeof(A) << endl;
cout << "sizeof(B) == " << sizeof(B) << endl;
cout << "sizeof(C) == " << sizeof(C) << endl;
C c;
```

```
cout << "&c == " << &c << endl;
A* ap = &c;
B* bp = &c;
cout << "ap == " << static_cast<void*>(ap) << endl;
cout << "bp == " << static_cast<void*>(bp) << endl;
C* cp = static_cast<C*>(bp);
cout << "cp == " << static_cast<void*>(cp) << endl;
cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
cp = 0;
bp = cp;
cout << bp << endl;
}
```

2.2 The C++ Casting Techniques

Along with the C-Style casting, the C++ offers some other four casting techniques. you can use these techniques to cast from two in-comparable types if needed:

- static_cast¡T¿()

  if you tried to cast from float to int, your compiler might promote a warning to you that you will loss precision. but if you need to make this cast happen, you might think of using the static cast to cast between two in-completable types.

Listing 30: Static Cast example

```
#include <iostream>

int main(int argc, char** argv){


    int inumber = 0;

    // In some compliers, a warning for the incompatible types will
        raised
    float fnumber = 3.557f;
    //Then use static casting
    inumber = static_cast<int>(fnumber);

    //Here is another example
    int  iNum = 85;
    char c = iNum; // this should give warning as well
    c = static_cast<char>(iNum);

    return 0;
}
```

- const_cast¡T¿() if you tried to modify a constant value intentionally, the your complier will throw a compilation error that this is a wrong conversion operation.

```
aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/TheDocumentationProject/Cpp-TipsAndTricks/cpp_code/cpp-casting
$ g++ const_cast.cpp
const_cast.cpp: In member function 'void A::Display() const':
const_cast.cpp:9:16: error: assignment of member 'A::iNum' in read-only object
        iNum = 50; //This is not allowed
               ^~
```

You can use a const_cast() to force the conversion from a const types to mutable type.

Listing 31: const cast

```
#include <iostream>

class A {
    int iNum = 0;
    mutable int iNum_m = 0;

public:
    void Display() const{
        iNum = 50; //This is not allowed, iNum is readonly in this
            function
        // You can change the value because we used const_cast
        (const_cast<A*>(this))->iNum = 50;

        //You can change the value as it's tagged mutable
        iNum_m = 50;

        std::cout << this->iNum << ", " << this->iNum_m;
    }
};

int main(int argc, char** argv){

    A a;
    a.Display();

    return 0;
}
```

- reinterpret_cast¡T¿() If you tried to cast one pointer type to another, the compiler might not allow that, but you can force that using the reinterpret cast.

```
aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/TheDocumentationProject/Cpp-TipsAndTricks/cpp_code/cpp-casting
$ g++ reinterpret_cast.cpp
reinterpret_cast.cpp: In function 'int main(int, char**)':
reinterpret_cast.cpp:8:16: error: invalid conversion from 'int' to 'char*' [-fpermissive]
     char* pc = num; //It's not allowed to make this conversion implicitly
                ^~~
reinterpret_cast.cpp:13:19: error: cannot convert 'int*' to 'char*' in initialization
     char* pChar = pInt; //It's not allowed to make this conversion
                   ^~~~
```

reinterpret cast is a powerful casting method, it can cast between any two types, it's mainly used to convert between pointers types (be careful when you use it).

Listing 32: reinterpret cast

```cpp
#include <iostream>

int main(int argc, char** argv){

    //It's little bit dangerous to use the reinterpret
    //As you can cast any type to any other type.
    int num = 1000000;
    char* pc = num; //It's not allowed to make this conversion
        implicitly
    char* pc = reinterpret_cast<char*>(num); //you can achieve
        using reinterpret_cast

    //the reinterpret_cast, can convert from any pointer to any
        other pointer type
    int* pInt = new int(65);
    char* pChar = pInt; //It's not allowed to make this conversion
    char* pChar = reinterpret_cast<char*>(pInt);

    return 0;
}
```

- dynamic_cast¡T¿() You might use the dynamic cast in the down-casting, as by default you can't cast pointer of Derived class to pointer of a base class, so the dynamic cast is used to achieve that.

Listing 33: Dynamic Cast

```cpp
#include <iostream>

class Base {
public:
    virtual void bFunc(){
        std::cout << "Hello I am the Base!" << std::endl;
    }
};
```

```cpp
class Derived : public Base {
    void Display(){
        std::cout << "Hey from Dervied !" << std::endl;
    }
};

int main(int argc, char** argv){
    Derived derivedObj;

    Derived* pDerived = &derivedObj;;
    Base* pBase;

    // This is up casting: Assing pointer of derived to base pointer
    // The upcasting is supported by default in Cpp
    pBase = pDerived;
    pBase->bFunc();

    // In Down Casting, the base class pointer is assigned to
        Derived call pointer
    //Error from the complier
    // This is not supported you need to make a Dynamic cast
    pDerived = pBase; //Not valid
    pDerived->bFunc();

    //pDerived = dynamic_cast<Derived*>(pBase); //This can be
        achived using dynamic_cast
    //pDerived->bFunc();

    return 0;
}
```

Compiling the above code without dynamic cast, the complier will throw an error.

```
aramadan@CAI1-L11666  /c/Users/aramadan/Desktop/TheDocumentationProject/Cpp-TipsAndTricks/cpp_code/cpp-casting
$ g++ dynamic_cast.cpp
dynamic_cast.cpp: In function 'int main(int, char**)':
dynamic_cast.cpp:31:16: error: invalid conversion from 'Base*' to 'Derived*' [-fpermissive]
     pDerived = pBase; //Not valid
                ^~~~~
```

Upcasting V.s. Downcasting

By default in C++, the upcasting is allowed, as you can assign a "Derived" class pointer to a "Base" class pointer, but the Down-casting is not allowed as you can't assign a "Base" class pointer to a "Derived" class pointer, but you need to notice that you can achieve a down-casting using the C++ *dynamic_casting¡T¿()* technique.

2.3 Cpp Perfect forwarding and universal references

2.4 decltype specifier

2.4.1 decltype, what is new in C++14

2.5 using keyword, using-declaration and type aliasing

2.5.1 type aliasing with "sing V.s. typedef

2.6 The Smart Pointers (C++11)

What are smart pointer ?
It's a wrapper around the normal C++ pointers by using a template classes that uses operator overloads to provide a pointer functionality and provide a way more better for memory management. For example: this function returns a pointer, but I have no idea after using this pointer what to do, which might cause a memory leaks. T * foo();

but by using this for example: unique_pointer¡T¿ foo();
You know now that we should have only one copy of the this pointer, then after we finish with it, either to destroy it or to change its ownership.

2.7 Cpp Async Programming, futures and promises

2.7.1 Types of Smart Pointers

- Unique pointer: unique_pointer¡T¿ foo(); Only one copy of the pointer should be there, and it will be released automatically if it's no longer used.
- Shared Pointer: shared_pointer¡T¿ foo(); We might have multiple copies of the pointer, and it will be all released after all the copies no longer used.
- Weak Pointer: is a special kind of the shared pointers, and it used when you need avoid a circular references (objects depends on others so you can't destroy one before destroying the other one and vice versa).

Note: In order to use the smart pointers, you need to include the memory.h header file *#include¡memory¿*

2.7.2 Unique Pointers

It can't be copied, there is only one copy of the pointer. and need to use the unique_pointer if a resources is only available to only one object at a time.

Listing 34: Unique Pointers

```
#include <iostream>
using namespace std;
```

```
class A { int x; };
class B { int y; };
class C : public A, public B { int z; };

int main() {
cout << "sizeof(A) == " << sizeof(A) << endl;
cout << "sizeof(B) == " << sizeof(B) << endl;
cout << "sizeof(C) == " << sizeof(C) << endl;
C c;
cout << "&c == " << &c << endl;
A* ap = &c;
B* bp = &c;
cout << "ap == " << static_cast<void*>(ap) << endl;
cout << "bp == " << static_cast<void*>(bp) << endl;
C* cp = static_cast<C*>(bp);
cout << "cp == " << static_cast<void*>(cp) << endl;
cout << "bp == cp? " << boolalpha << (bp == cp) << endl;
cp = 0;
bp = cp;
cout << bp << endl;
}
```

- unique_pointer::move() function
- unique_pointer::reset() function
- unique_pointer::release() function
- make_unique¡¿() function (non std)

Note: You can't pass a unique_pointer to a function, because it's unique which can't be copied, so don't pass it by value, instead pass it by reference.

2.7.3 Shared Pointer

In the shared pointers, you may taking copies of the pointer. and the shared pointer class will keep a counter of the copies by using the shared pointer constructors and destructors.

The counter is called the reference count and can be accessed using *shared_pointer.use_count()* function.

2.7.4 Custom Deleters

You can specify a custom deleter to be called with smart pointer reset function.

2.8 Move Semantics (C++ 11)

In move semantics, we move an object by re-associate it to a new object instead of copying it. suppose the following

T f(T o) return o;
T b = f(b);

so here in the previous example, the object's copy constructor will be called twice on at the function call, and other one at the return type which creates a temporary not used object to be destructed later and this might consume an amount of time and memory usage till the temporary object is deleted.

Moving the data is taken place by using something called ***rvalue reference***, and you need to notice that it's different from the normal reference types which is called ***lvalue reference***

- T & x - lvalue reference (can't be moved as it's in left side of an assignment operator!)
- T && x - rvalue reference (can be moved)

Listing 35: Using move semantics

```
#include <cstdio>
#include <vector>
#include <string>
#include <utility>

void message(const std::string & s) {
    puts(s.c_str());
    fflush(stdout);
}

void disp_vector(const std::vector<std::string> & v) {
    size_t size = v.size();
    printf("vector size: %ld\n", size);
    if(size) {
        for( std::string s : v ) {
            printf("[%s]", s.c_str());
        }
        puts("");
    }
    fflush(stdout);
}

int main( int argc, char ** argv ) {
    std::vector<std::string> v1 = { "one", "two", "three", "four",
        "five" };
    std::vector<std::string> v2 = { "six", "seven", "eight", "nine",
        "ten" };
```

```
    message("v1");
    disp_vector(v1);
    message("v2");
    disp_vector(v2);

    // v1=v2; // here we copying
    v1 = std::move(v2);
    message("v1");
    disp_vector(v1);
    message("v2");
    disp_vector(v2);

    return 0;
}
```

Output:

```
v1 Original
vector size: 5
[one][two][three][four][five]
v2 Original
vector size: 5
[six][seven][eight][nine][ten]

After Move
v1
vector size: 5
[six][seven][eight][nine][ten]
v2
vector size: 0
```

2.8.1 Creating Move Constructors

2.8.2 Creating Move Assignment Operator