layout: post title: "C++ CheatSheet" categories: Interview

Polymorphism

Compiled-time polymorphism

Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

Runtime polymorphism: achieved through Virtual Function

Virtual function is member function declared in base class, and is required to be redefined in derived class. When using a **Pointer or Reference** of the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve **Runtime polymorphism**
- Functions are declared with a virtual keyword in base class.
- The resolving of function call is done at Run-time.
- They are always defined in base class and overridden in derived class. It is **not mandatory for derived class to override** (or re-define the virtual function), in that case base class version of function is used.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

Implementation

g4g

Pure Virtual Functions and Abstract Classes in C++

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we **only declare** it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

But a pure virtual function can have a body.

All pure virtual means is that you can't call the function using an object that has declared or has inherited the pure virtual function. Because of this, you cannot create objects of classes with pure virtual functions.

The compiler enforces this by not allowing objects to be created using classes that have pure virtual funcitons or that have inherited pure virtual functions.

```
// An abstract class
class Test
{
    // Data members of class
```

```
public:
    // Pure Virtual Function
    virtual void show() = 0;

   /* Other members */
};
```

- A class is abstract if it has at least one pure virtual function.
- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

---

diff between C and C++

link

Class access control

Friend class

Friend Class A friend class can access private and protected members of other class in which it is declared as friend.

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be: a) A method of another class b) A global function

- Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.
- Friendship is not inherited (See this for more details)

Static

static members are only declared in class declaration, not defined. They must be explicitly defined outside the class using scope resolution operator. Also a static member can't contain any non-static field. Inside a static member we can't use this. They are initialized out side of the class because they won't be intialized when any objects of that class is created. It can only be declared once.

If we try to access static member ‘a’ without explicit definition of it, we will get compilation error. For example, following program fails in compilation.

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

class A
{
```

```
        int x;
public:
        A() { cout << "A's constructor called " << endl; }
};

class B
{
        static A a;
public:
        B() { cout << "B's constructor called " << endl; }
        static A getA() { return a; }
};

int main()
{
        B b;
        A a = b.getA();
        return 0;
}
```

static

Memory

New and malloc

Following are the differences between malloc() and operator new.

new is an operator, while malloc() is a function. new returns exact data type, while malloc() returns void *. new calls constructors(class instances are initalized and deinitialized automatically), while malloc() does not(classes won't get initalized or deinitialized automatically Syntax: int *n = new int(10); // initialization with new() str = (char *) malloc(15); //malloc() free( ) is used on resources allocated by malloc( ), or calloc( ) in C

Delete is used on resources allocated by new in C++

Dynamic Memory and Smart Pointers

The dynamic memory management is done throught `new` and `delete`, also there are smart pointers used to manage memories.

- shared_ptr类（The shared_ptr Class） Share_ptr is also a template, when it is initialized, we need to specify the type of it.

```
shared_ptr<string> p1;      // shared_ptr that can point at a string
shared_ptr<list<int>> p2;   // shared_ptr that can point at a list of ints
```

make_shared函数（定义在头文件memory中）在动态内存中分配一个对象并初始化它，返回指向此对象的shared_ptr。

Every shared_ptr has a reference count,, when shared_ptr is copied, it is incremented. Once it is zero, the object will be deleterd automatically

- unique_ptr Different from shared_ptr, unique_ptr can only point to one object at the same time. When unique pointer is destroied, the object it points to will be destroied.

```
make_unique函数（C++14新增，定义在头文件memory中）在动态内存中分配一个对象并初始化它，返回指向此对象的unique_ptr。

unique_ptr<int> p1(new int(42));
// C++14
unique_ptr<int> p2 = make_unique<int>(42);
```

- weak_ptr（weak_ptr） weak pointer is a smart pointer that will not control the life time of the object, it points to a object managed by shared_ptr, it will not affect the reference count of the shared pointer. 如果shared_ptr被销毁，即使有weak_ptr指向对象，对象仍然有可能被释放。

```
使用weak_ptr访问对象时，必须先调用lock函数。该函数检查weak_ptr指向的对象是否仍然存在。如果存在，则返回指向共享对象的shared_ptr，否则返回空指针。

if (shared_ptr<int> np = wp.lock())
{
    // true if np is not null
    // inside the if, np shares its object with p
}
```

New

The objects allocated by new is default initialzed(un-initialized for non-global variables). if we want to initialize those variables, we need to append a parenthesis such that it will be regarded as calling the constructor of that object.

```
int *pia = new int[10];     // block of ten uninitialized ints
int *pia2 = new int[10]();    // block of ten ints value initialized to 0
string *psa = new string[10];    // block of ten empty strings
string *psa2 = new string[10]();    // block of ten empty strings
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// block of ten strings; the first four are initialized from the given
initializers
// remaining elements are value initialized
string *psa3 = new string[10] { "a", "an", "the", string(3,'x') };
```

Delete

brace: {} parenthesis: () brackets:[]

使用delete[]释放动态数组。

delete p; // p must point to a dynamically allocated object or be null delete [] pa; // pa must point to a dynamically allocated array or be null 如果在delete数组指针时忘记添加方括号，或者在delete单一对象时使用了方括号，编译器很可能不会给出任何警告，程序可能会在执行过程中行为异常。

Allocator

```
// C++ program for illustration
// of std::allocator() function
#include <iostream>
#include <memory>
#include <string>

int main()
{

    // allocator for integer values
    std::allocator<std::string> myAllocator;

    // allocate space for three strings
    std::string* str = myAllocator.allocate(3);

    // construct these 3 strings
    myAllocator.construct(str, "Geeks");
    myAllocator.construct(str + 1, "for");
    myAllocator.construct(str + 2, "Geeks");

    std::cout << str[0] << str[1] << str[2];

    // destroy these 3 strings
    myAllocator.destroy(str);
    myAllocator.destroy(str + 1);
    myAllocator.destroy(str + 2);

    // deallocate space for 3 strings
    myAllocator.deallocate(str, 3);
}
```

Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype: ClassName (const ClassName &old_obj);

Synthesized copy-constructor

如果类未定义自己的拷贝构造函数，编译器会为类合成一个。一般情况下，合成拷贝构造函数（synthesized copy constructor）会将其参数的非static成员逐个拷贝到正在创建的对象中。

发生拷贝初始化的情况：

- 用=定义变量。
- 将对象作为实参传递给非引用类型的形参。
- 从返回类型为非引用类型的函数返回对象。
- 用花括号列表初始化数组中的元素或聚合类中的成员。

The Copy-Assignment Operator

重载运算符（overloaded operator）的参数表示运算符的运算对象。

如果一个运算符是成员函数，则其左侧运算对象会绑定到隐式的this参数上。

赋值运算符通常应该返回一个指向其左侧运算对象的引用。

```
class Foo
{
public:
    Foo& operator=(const Foo&);  // assignment operator
    // ...
};
```

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

// Copy constructor

Point(const Point &p2) {x = p2.x; y = p2.y; }

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

When an object of the class is returned by value. When an object of the class is passed (to a function) by value as an argument. When an object is constructed based on another object of the same class. When compiler generates a temporary object.

https://www.geeksforgeeks.org/commonly-asked-oop-interview-questions/

Virtual Destructor

https://www.geeksforgeeks.org/virtual-destructor/

Deleting a derived class object **using a pointer to a base class that has a non-virtual destructor results in undefined behavior**. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

Pure Virtual destructor

https://www.geeksforgeeks.org/pure-virtual-destructor-c/

Can a destructor be pure virtual in C++? Yes, it is possible to have pure virtual destructor. Pure virtual destructors are legal in standard C++ and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor. You may be wondering why a pure virtual function requires a function body. The reason is because destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation. This means that a derived class' destructor will be invoked first, then base class destructor will be called. If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore the compiler and linker enforce the existence of a function body for pure virtual destructors.

=default

使用=default（Using =default） 可以通过将拷贝控制成员定义为=default来显式地要求编译器生成合成版本。

```
class Sales_data
{
public:
    // copy control; use defaults
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    ~Sales_data() = default;
    // other members as before
};
```

Preventing Copies

大多数类应该定义默认构造函数、拷贝构造函数和拷贝赋值运算符，无论是显式地还是隐式地。

在C++11新标准中，将拷贝构造函数和拷贝赋值运算符定义为删除的函数（deleted function）可以阻止类对象的拷贝。删除的函数是一种虽然进行了声明，但是却不能以任何方式使用的函数。定义删除函数的方式是在函数的形参列表后面添加=delete

```
struct NoCopy
{
    NoCopy() = default; // use the synthesized default constructor
    NoCopy(const NoCopy&) = delete; // no copy
    NoCopy &operator=(const NoCopy&) = delete; // no assignment
    ~NoCopy() = default; // use the synthesized destructor
    // other members
};
```

Lvalue and RValue

Lvalue: an Object that occupies some identifiable location in memory Rvalue: any object that is not Rvalue

```
int i
int &r = i
// r is a lvalue reference
```

Moving Objects

某些情况下，一个对象拷贝后就立即被销毁了，此时移动而非拷贝对象会大幅度提高性能。

在旧版本的标准库中，容器所能保存的类型必须是可拷贝的。但在新标准中，可以用容器保存不可拷贝，但可移动的类型。

标准库容器、string和shared_ptr类既支持移动也支持拷贝。IO类和unique_ptr类可以移动但不能拷贝。

Move Constructor and Move Assignment

When are the move constructor and move assignment called?

The move constructor and move assignment are called when those functions have been defined, and the argument for construction or assignment is an **r-value**. Most typically, this r-value will be a literal or temporary value.

Move Constructor

MS doc

```
// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
   // Create a vector object and add a few elements to it.
   vector<MemoryBlock> v;
   v.push_back(MemoryBlock(25));
   v.push_back(MemoryBlock(75));

   // Insert a new element into the second position of the vector. Using move
constructor, 50 is a temporary created Rvalue and the move constructor will fetch
its rvalue reference
   v.insert(v.begin() + 1, MemoryBlock(50));
}
```

Conversion Operators

STL

Associative container: Map & Set

For Map and set, everything is sorted, both use Red Black Tree (A self balancing BST). Note that the time complexities of search, insert and delete are O(Log n).

While for unordered_set and unordered_Map, everything is stored in hashed map, thus they are not sorted, and time complexity for insert and fecth is O(1).

Black-Red Tree In a hash table, a value is stored by calling a hash function on a key.

Values are not stored in sorted order.

A hash table is traditionally implemented with an **array of linked lists**. When we want to insert a key/Value pair, we map the key to an index in the array using the hash function. The value is then inserted into the linked list at that position.

- Thread safe: STL Maps are not thread safe whereas Hashmaps are thread safe and can be shared with many threads.
- Multi-Map:multimaps have certain runtime complexity (O(lg n) for the interesting operations) and other guarantees, and can be implemented as red-black trees. This is how they are implemented in the GNU standard C++ library.

Sequential Container:

- Vector dynamically allocated array
- Stack & Queue Basically still vector
- List

Stack and Heap Memory

stackoverflow

Function Pointers

link

Type casting:

static_cast

This is the simplest type of cast which can be used. It is a compile time cast.It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions (or implicit ones).

```
//From base to derived
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

reinterpret_cast

It is used to convert one pointer of another pointer of any type, no matter either the class is related to each other or not. It does not check if the pointer type and data pointed by the pointer is same or not.

reinterpret_cast is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable.

const_cast

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet. Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer. const_cast changes the type of 'this' pointer to 'student* const this'.

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

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; }
};

int main(void)
{
        student s(3);
        cout << "Old roll number: " << s.getRoll() << endl;

        s.fun();

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

        return 0;
}
```

dynamic_cast

---

questions

https://www.softwaretestinghelp.com/cpp-interview-questions/

what are static, hash table, virtual, STL

Details

- When there is a Global variable and Local variable with the same name, how will you access the global variable?

Answer: When there are two variables with the same name but different scope, i.e. one is a local variable and the other is a global variable, the compiler will give preference to a local variable.

- A function becomes const when const keyword is used in function's declaration. The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.
- C++11中，在类名后面添加final关键字可以禁止其他类继承它。
- OOP concepts geeksforgeeks