
C++

C++ might be the most difficult programming language in many software engineers' judgment. Many interviewers believe a candidate who is proficient on C++ has the abilities needed to master other technical skills, so questions about C++ are quite popular during interviews.

Interview questions about the C++ programming language can be divided into three categories: C++ concepts, tricky C++ coding problems, and implementing a class or member function.

C++ Concepts

Questions in the first category are about C++ concepts, especially about C++ keywords. For example, what are the four keywords for type casting, and under what are scenarios would you use each of them?

The most popular questions in this category concern the keyword `sizeof`.

For instance, the following dialog is repeated again and again in many interviews:

Interviewer: There is an empty class, without any member fields and functions inside its definition. What is the result of `sizeof` for this class?

Candidate: It is 1.

Interviewer: Why not 0?

Candidate: The size seems to be 0 since the class is empty without any data. However, it should occupy some memory; otherwise, it cannot be accessed. The size of memory is decided by compilers. It occupies 1 byte in Visual Studio.

Interviewer: What is the result of `sizeof` if we add the constructor and destructor functions in the class?

Candidate: It is also 1. The addresses for functions are irrelevant to instances, and compilers do not add any data in instances of this class.

Interviewer: How about declaring the destructor function as a virtual function?

Candidate: When a C++ compiler sees a virtual function inside a class, it creates a virtual function table for the class and adds a pointer to the table in each instance. A pointer in a 32-bit machine occupies 4 bytes, so the result of `sizeof` is 4.

The result on a 64-bit machine is 8 because a pointer occupies 8 bytes there.

Analyzing Execution of C++ Code

The second type of interview questions in C++ concerns analyzing execution results of some sample code. Candidates without deep understanding of C++ are prone to make mistakes because the code to be analyzed is usually quite tricky.

For example, an interviewer hands a piece of paper printed with the code in Listing 2-8 to a candidate and asks the candidate what the result is if we try to compile and execute the code: (A) Compiling error; (B) It compiles well, but crashes in execution; or (C) It compiles well, and executes smoothly with an output of 10.

Listing 2-8. *C++ Code about Copy Constructor*

```
   class A {

private:

    int value;


   public:

    A(int n) {

        value = n;

    }

    A(A other) {

        value = other.value;

    }

    void Print() {

         std::cout << value << std::endl;

    }

};


int main(int argc, _TCHAR* argv[]) {

    A a = 10;

    A b = a;

    b.Print();


    return 0;

}
```

The parameter in the copy constructor `A(A other)` is an instance of type `A`. When the copy constructor is executed, it calls the copy constructor itself

because of the pass-by-value parameter. Since endless recursive calls cause call stack overflow, a pass-by-value parameter is not allowed in the C++ standard, and both Visual Studio and GCC report an error during compiling time. We have to modify the constructor as `A(const A& other)` to fix this problem. That is to say, the parameter in a copy constructor should be passed by a reference.

Implementing a Class or Member Function in C++

The third type of C++ interview questions is based on implementing a class or some member functions. Usually it is more difficult to write code than to read and analyze sample code. Many C++ coding interview questions focus on constructor or destructor functions as well as overloading operators. For instance, the following problem about an assignment operator is such an example.

Assignment Operator

Question 2 The declaration of class `CMyString` is found in Listing 2-9. Please add an assignment operator to it.

Listing 2-9. *C++ Code for Declaration of* `CMyString`

```
class CMyString {
public:
 CMyString(char* pData = NULL);
 CMyString(const CMyString& str);
 ~CMyString(void);

private:
 char* m_pData;
};
```

Classic, but Only Suitable for Newbie Developers

When an interviewer asks a candidate to overload the assignment operator, he or she asks the following questions to check the candidate's code:

- Does it return a reference? Before the assignment operator function ends, it should return the instance itself `(*this)` as a reference. If an assignment operator is overloaded as a `void` function, it has no chance to chain multiple assignments together. Suppose that there are three instances of the `CMyString` type: `str1`, `str2`, and `str3`. They cause a compiling error at `str1=str2=str3` if the assignment operator is overloaded as a `void` function.
- Is the argument passed by a constant reference? If the argument is passed by value, a copy constructor is called, and it is a waste of time. The call of the copy constructor is avoided if the argument is passed by reference. Additionally, it should not modify the status of the input instance during assignment, so the reference should be declared as `const`.
- Is the existing memory freed? If the old memory for `m_pData` is not deallocated before it allocates new memory, memory leaks occur.
- Does it protect against self-assignment? It returns directly and does nothing if the assignment source (input instance) is identical to the assignment target `(*this)`. Otherwise if `*this` is the same as the input instance, its memory will be freed and its content cannot be gotten back any more.

Listing 2-10 is a piece of C++ code, covering all four items.

Listing 2-10. *C++ Code for Assignment Operator (Version 1)*

```
CMyString& CMyString::operator =(const CMyString &str) {
 if(this == &str)
     return *this;

 delete []m_pData;
 m_pData = NULL;

 m_pData = new char[strlen(str.m_pData) + 1];
 strcpy(m_pData, str.m_pData);

 return *this;
}
```

This is a classic solution in textbooks. A junior candidate can pass this round of interview with the code above. However, it is far from perfection. An

interviewer will have more stringent requirements if the candidate is a senior C++ programmer.

Exception Safety

In the previous code, the old `m_pData` is deleted before it is renewed. If an exception is thrown while reallocating memory due to insufficient memory, `m_pData` is a `NULL` pointer and it is prone to crash the whole application. In other words, when an exception is thrown in the assignment operator function, the status of a `CMyString` instance is invalid. It breaks the requirement of exception safety: If an exception is thrown, everything in the program must remain in a valid state.

Two approaches are available to achieve the goal of exception safety. The first one is to allocate memory with the `new` operator before deleting. It makes sure the old memory is deleted only after it allocates memory successfully, and the status of `CMyString` instances is valid if it fails to allocate memory.

A better choice is to create a temporary instance to copy the input data and then to swap it with the target. The code in Listing 2-11 is based on the copy-and-swap solution.

Listing 2-11. *C++ Code for Assignment Operator (Version 2)*

```
CMyString& CMyString::operator =(const CMyString &str) {

if(this != &str) {

    CMyString strTemp(str);


    char* pTemp = strTemp.m_pData;

    strTemp.m_pData = m_pData;

    m_pData = pTemp;

}


return *this;

}
```

In this code, a temporary instance `strTemp` is constructed first, copying data from the input `str`. Next, it swaps `strTemp.m_pData` with `this->m_pData`. Because `strTemp` is a local variable, its destructor will be called automatically when the execution flow exits this function. The memory pointed to by `strTemp.m_pData` is what was pointed to by `this->m_pData` before, so the memory of the old instance is deleted when the destructor of `strTemp` is invoked.

It allocates memory with the `new` operator in the constructor of `CMyString`. Supposing that a `bad_alloc` exception is thrown due to insufficient memory, the old instance has not been modified, and its status is still valid. Therefore, it is an exception-safe solution.

Source Code:

```
002_AssignmentOperator.cpp
```

Test Cases:

- Assign an instance to another
- Assign an instance to itself
- Chain multiple assignments together
- Stress tests to check whether the code contains memory leaks