**25 Essential C++**

**1. What will the line of code below print out and why?**

#include <iostream>

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

{

std::cout << 25u - 50;

return 0;

}

The answer is not -25. Rather, the answer (which will surprise many) is 4294967271, assuming 32 bit integers. Why?

In C++, if the types of two operands differ from one another, then the operand with the “lower type” will be promoted to the type of the “higher type” operand, using the following type hierarchy (listed here from highest type to lowest type): long double, double, float, unsigned long int, long int, unsigned int, int (lowest).

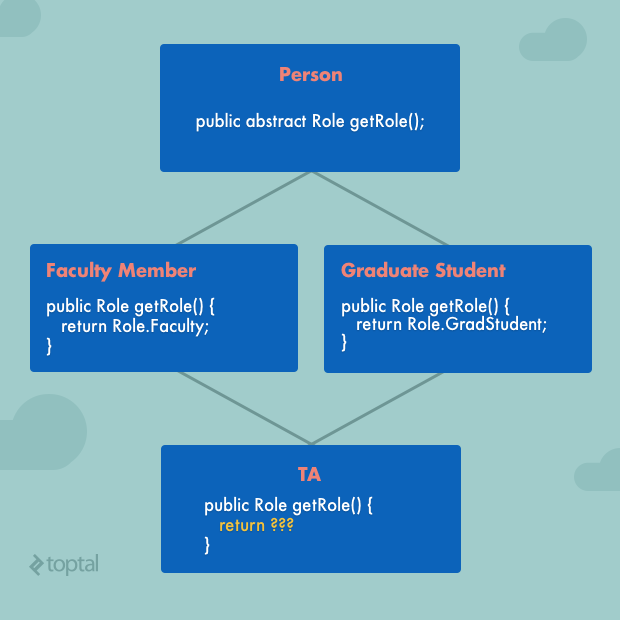
So when the two operands are, as in our example, 25u (unsigned int) and 50 (int), the 50 is promoted to also being an unsigned integer (i.e., 50u). Moreover, the result of the operation will be of the type of the operands. Therefore, the result of 25u - 50u will itself be an unsigned integer as well. So the result of -25 converts to 4294967271 when promoted to being an unsigned integer.

**2. C++ supports multiple inheritance. What is the “diamond problem” that can occur with multiple inheritance? Give an example.**

It means that we cannot create hybrid inheritance using multiple and hierarchical inheritance.

Let’s consider a simple example. A university has people who are affiliated with it. Some are students, some are faculty members, some are administrators, and so on. So a simple inheritance scheme might have different types of people in different roles, all of whom inherit from one common “Person” class. The Person class could define an abstract getRole() method which would then be overridden by its subclasses to return the correct role type.

But now what happens if we want to model a the role of a Teaching Assistant (TA)? Typically, a TA is both a grad student and a faculty member. This yields the classic diamond problem of multiple inheritance and the resulting ambiguity regarding the TA’s getRole() method:

Which getRole() implementation should the TA inherit? That of the Faculty Member or that of the Grad Student? The simple answer might be to have the TA classoverride the getRole() method and return newly-defined role called “TA”. But that answer is also imperfect as it would hide the fact that a TA is, in fact, both a faculty member and a grad student.

**3. What is the error in the code below and how should it be corrected?**

my\_struct\_t \*bar;

/\* ... do stuff, including setting bar to point to a defined my\_struct\_t object ... \*/

memset(bar, 0, sizeof(bar));

The last argument to memset should be **sizeof(\*bar)**, not sizeof(bar). sizeof(bar) calculates the size of bar (i.e., the pointer itself) rather than the size of the structure pointed to by bar.

The code can therefore be corrected by using sizeof(\*bar) as the last argument in the call to memset.

A sharp candidate might point out that using **\*bar** will cause a dereferencing error if bar has not been assigned. Therefore an even safer solution would be to use **sizeof(my\_struct\_t)**. However, an even sharper candidate must know that in this case using \*bar is absolutely safe within the call to sizeof, even if bar has not been initialized yet, since **sizeof** is **a compile time construct**.

**4. What will i and j equal after the code below is executed? Explain your answer.**

int i = 5;

int j = i++;

After the above code executes, i will equal 6, but j will equal 5. Understanding the reason for this is fundamental to understanding how the unary increment (++) and decrement (--) operators work in C++.

When these operators precede a variable, the value of the variable is modified first and then the modified value is used. For example, if we modified the above code snippet to instead say int j = ++i;, i would be incremented to 6 and then j would be set to that modified value, so both would end up being equal to 6.

However, when these operators follow a variable, the unmodified value of the variable is used and then it is incremented or decremented. That’s why, in the statement int j = i++; in the above code snippet, j is first set to the unmodified value of i (i.e., 5) and then i is incremented to 6.

**5. Assuming buf is a valid pointer, what is the problem in the code below? What would be an alternate way of implementing this that would avoid the problem?**

size\_t sz = buf->size();

while ( --sz >= 0 )

{

/\* do something \*/

}

The problem in the above code is that **--sz >= 0** will always be true so you’ll never exit the while loop (so you’ll probably end up corrupting memory or causing some sort of memory violation or having some other program failure, depending on what you’re doing inside the loop).

The reasons that --sz >= 0 will always be true is that the type of sz is size\_t. **size\_t** is really just an alias to one of the fundamental unsigned integer types. Therefore, since sz is unsigned, it can never be less than zero (so the condition can never be true).

One example of an alternative implementation that would avoid this problem would be to instead use a for loop as follows:

for (size\_t i = 0; i < sz; i++)

{

/\* do something \*/

}

**6. Consider the two code snippets below for printing a vector. Is there any advantage of one vs. the other? Explain.**

Option 1:

vector vec;

/\* ... .. ... \*/

for (auto itr = vec.begin(); itr != vec.end(); itr++) {

itr->print();

}

Option 2:

vector vec;

/\* ... .. ... \*/

for (auto itr = vec.begin(); itr != vec.end(); **++itr**) {

itr->print();

}

Although both options will accomplish precisely the same thing, the second option is better from a performance standpoint. This is because the post-increment operator (i.e., itr++) is more expensive than pre-increment operator (i.e., ++itr). The underlying implementation of the post-increment operator makes a copy of the element before incrementing it and then returns the copy.

That said, many compilers will automatically optimize the first option by converting it into the second.

**7. Implement a template function IsDerivedFrom() that takes class C and class P as template parameters. It should return true when class C is derived from class P and false otherwise.**

This question tests understanding of C++ templates. An experienced developer will know that this is already a part of the C++11 std library **(std::is\_base\_of)** or part of the boost library for C++ **(boost::is\_base\_of)**. Even an interviewee with only passing knowledge should write something similar to this, mostly likely involving a helper class:

template<typename D, typename B>

class IsDerivedFromHelper

{

class No { };

class Yes { No no[3]; };

static Yes Test( B\* );

static No Test( ... );

public:

enum { Is = sizeof(Test(static\_cast<D\*>(0))) == sizeof(Yes) };

};

template <class C, class P>

bool IsDerivedFrom() {

return IsDerivedFromHelper<C, P>::Is;

}

**8. Implement a template boolean IsSameClass() that takes class A and B as template parameters. It should compare class A and B and return false when they are different classes and true if they are the same class.**

template <typename T, typename U>

struct is\_same

{

static const bool value = false;

};

template <typename T>

struct is\_same<T, T>

{

static const bool value = true;

};

template <class A, class B>

bool IsSameClass() {

return is\_same<A, B>::value;

}

**9. Is it possible to have a recursive inline function?**

Although you can call an inline function from within itself, the compiler may not generate inline code since the compiler cannot determine the depth of recursion at compile time. A compiler with a good optimizer can inline recursive calls till some depth fixed at compile-time (say three or five recursive calls), and insert non-recursive calls at compile time for cases when the actual depth gets exceeded at run time.

**10. What is the output of the following code:**

#include <iostream>

class A {

public:

A() {}

~A() {

throw 42;

}

};

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

try {

A a;

throw 32;

} catch(int a) {

std::cout << a;

}

}

This program will terminate abnormally. throw 32 will start unwinding the stack and destroy class A. The class A destructor will throw another exception during the exception handling, which will cause program to crash. This question is testing if developer has experience working with exceptions.

**11. You are given library class Something as follows:**

class Something {

public:

Something() {

topSecretValue = 42;

}

bool somePublicBool;

int somePublicInt;

std::string somePublicString;

private:

int topSecretValue;

};

**12. Implement a method to get topSecretValue for any given Something\* object. The method should be cross-platform compatible and not depend on sizeof (int, bool, string).**

Create another class which has all the members of Something in the same order, but has additional public method which returns the value. Your replica Something class should look like:

class SomethingReplica {

public:

int getTopSecretValue() { return topSecretValue; }

bool somePublicBool;

int somePublicInt;

std::string somePublicString;

private:

int topSecretValue;

};

Then, to get the value:

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

Something a;

SomethingReplica\* b = reinterpret\_cast<SomethingReplica\*>(&a);

std::cout << b->getTopSecretValue();

}

It’s important to avoid code like this in a final product, but it’s nevertheless a good technique when dealing with legacy code, as it can be used to extract intermediate calculation values from a library class. (Note: If it turns out that the alignment of the external library is mismatched to your code, you can resolve this using **#pragma pack**.)

**13. Implement a void function F that takes pointers to two arrays of integers (A and B) and a size N as parameters. It then populates B where B[i] is the product of all A[j] where j != i.**

**For example: If A = {2, 1, 5, 9}, then B would be {45, 90, 18, 10}.**

This problem seems easy at first glance so a careless developer might write something like this:

void F(int\* A, int\* B, int N) {

int m = 1;

for (int i = 0; i < N; ++i) {

m \*= A[i];

}

for (int i = 0; i < N; ++i) {

B[i] = m / A[i];

}

}

This will work for the given example, but when you add a 0 into input array A, the program will crash because of division by zero. The correct answer should take that edge case into account and look like this:

void F(int\* A, int\* B, int N) {

// Set prod to the neutral multiplication element

int prod = 1;

for (int i = 0; i < N; ++i) {

// For element "i" set B[i] (i>0) to A[0] \* ... \* A[i - 1], B[0]=1

B[i] = prod;

// Multiply with A[i] to set prod to A[0] \* ... \* A[i]

prod \*= A[i];

}

// Reset prod and use it for the right elements

prod = 1;

for (int i = N - 1; i >= 0; --i) {

// For element "i" multiply B[i] with A[i + 1] \* ... \* A[N - 1]

B[i] \*= prod;

// Multiply with A[i] to set prod to A[i] \* ... \* A[N - 1]

prod \*= A[i];

}

}

The presented solution above has a Big O complexity of O(N). While there are simpler solutions available (ones that would ignore the need to take 0 into account), that simplicity has a price of complexity, generally running significantly slower.

**14. When you should use virtual inheritance?**

While it’s ideal to avoid virtual inheritance altogether (you should know how your class is going to be used) having a solid understanding of how virtual inheritance works is still important:

So when you have a class (class A) which inherits from 2 parents (B and C), both of which share a parent (class D), as demonstrated below:

#include <iostream>

class D {

public:

void foo() { std::cout << "Foooooo" << std::endl; }

};

class C: public D { };

class B: public D { };

class A: public B, public C { };

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

A a;

a.foo();

}

If you don’t use virtual inheritance in this case, you will get two copies of D in class A: one from B and one from C. To fix this you need to change the declarations of

classes C and B to be virtual, as follows:

class C: virtual public D { };

class B: virtual public D { };

**15. Is there a difference between class and struct?**

The only difference between a class and struct are the access modifiers. Struct members are public by default; class members are private. It is good practice to use classes when you need an object that has methods and structs when you have a simple data object.

**16. What is the output of the following code:**

#include <iostream>

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

int a[] = {1, 2, 3, 4, 5, 6};

std::cout << (1 + 3)[a] - a[0] + (a + 1)[2];

}

The above will output 8, since:

(1+3)[a] is the same as a[1+3] == 5

a[0] == 1

(a + 1)[2] is the same as a[3] == 4

This question is testing **pointer arithmetic knowledge**, and **the magic behind square brackets with pointers**.

While some might argue that this isn’t a valuable question as it appears to only test the capability of reading C constructs, it’s still important for a candidate to be able to work through it mentally; it’s not an answer they’re expected to know off the top of their head, but one where they talk about what conclusion they reach and how.

**17. What is the output of the following code:**

#include <iostream>

class Base {

virtual void method() {std::cout << "from Base" << std::endl;}

public:

virtual ~Base() {method();}

void baseMethod() {method();}

};

class A : public Base {

void method() {std::cout << "from A" << std::endl;}

public:

~A() {method();}

};

int main(void) {

Base\* base = new A;

base->baseMethod();

delete base;

return 0;

}

The above will output:

from A

from A

from Base

The important thing to note here is the order of destruction of classes and how Base’s method reverts back to its own implementation once A has been destroyed.

**18. Explain the volatile and mutable keywords.**

The **volatile** keyword informs the compiler that a variable may change without the compiler knowing it. Variables that are declared as volatile will not be cached by the compiler, and will thus always be read from memory.

The **mutable** keyword can be used for class member variables. Mutable variables are allowed to change from within const member functions of the class.

**19. How many times will this loop execute? Explain your answer.**

unsigned char half\_limit = 150;

for (unsigned char i = 0; i < 2 \* half\_limit; ++i)

{

// do something;

}

If you said 300, you would have been correct if i had been declared as an int. However, since i was declared as an unsigned char, the corrct answer is that this code will result in an infinite loop.

Here’s why:

The expression 2 \* half\_limit will get promoted to an int (based on C++ conversion rules) and will have a value of 300. However, since i is an unsigned char, it is rerepsented by an 8-bit value which, after reaching 255, will overflow (so it will go back to 0) and the loop will therefore go on forever.

**20. How can you make sure a C++ function can be called as e.g. void foo(int, int) but not as any other type like void foo(long, long)?**

Implement foo(int, int)…

void foo(int a, int b) {

// whatever

}

…and delete all others through a template:

template <typename T1, typename T2> void foo(T1 a, T2 b) = delete;

Or without the delete keyword:

template <class T, class U>

void f(T arg1, U arg2);

template <>

void f(int arg1, int arg2)

{

//...

}

**21. What is the problem with the following code?**

class A

{

public:

A() {}

~A(){}

};

class B: public A

{

public:

B():A(){}

~B(){}

};

int main(void)

{

A\* a = new B();

delete a;

}

The behavior is **undefined** because A’s destructor is not virtual. From the spec:

( C++11 §5.3.5/3 ) if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined.

**22. Are you allowed to have a static const member function? Explain your answer.**

A const member function is one which isn’t allowed to modify the members of the object for which it is called. A static member function is one which can’t be called for a specific object.

Thus, the const modifier for a static member function is meaningless, because there is no object associated with the call.

A more detailed explanation of this reason comes from the C programming language. In C, there were no classes and member functions, so all functions were global. A member function call is translated to a global function call. Consider a member function like this:

void foo(int i);

A call like this:

obj.foo(10);

…is translated to this:

foo(&obj, 10);

This means that the member function foo has a hidden first argument of type T\*:

void foo(T\* const this, int i);

If a member function is const, this is of type const T\* const this:

void foo(const T\* const this, int i);

Static member functions don’t have such a hidden argument, so there is no this pointer to be const or not.

**23. What is a storage class?**

A class that specifies the life and scope of its variables and functions is called a storage class.

In C++ following the storage classes are supported: auto, static, register, extern, and mutable.

Note, however, that the keyword register was deprecated in C++11. In C++17, it was removed and reserved for future use.

**24. How can a C function be called in a C++ program?**

Using an extern "C" declaration:

//C code

void func(int i)

{

//code

}

void print(int i)

{

//code

}

//C++ code

extern "C"{

void func(int i);

void print(int i);

}

void myfunc(int i)

{

func(i);

print(i);

}

**25. What will be the output of the following program?**

#include <iostream>

struct A

{

int data[2];

A(int x, int y) : data{x, y} {}

virtual void f() {}

};

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

{

A a(22, 33);

int \*arr = (int \*) &a;

std::cout << arr[2] << std::endl;

return 0;

}

In the main function the instance of struct A is treated as an array of integer values. On 32-bit architectures the output will be 33, and on 64-bit architectures it will be 22.

This is because there is virtual method f() in the struct which makes compiler insert a vptr pointer that points to vtable (a table of pointers to virtual functions of class or struct). On 32-bit architectures the vptr takes 4 bytes of the struct instance and the rest is the data array, so arr[2] represents access to second element of the data array, which holds value 33. On 64-bit architectures the vptr takes 8 bytes so arr[2] represents access to the first element of the data array, which holds 22.

This question is testing knowledge of virtual functions internals, and knowledge of C++11-specific syntax as well, because the constructor of A uses the extended initializer list of the C++11 standard.

Compiled with:

g++ question\_vptr.cpp -m32 -std=c++11

g++ question\_vptr.cpp -std=c++11