Top 10 Most Common C++ Mistakes That Developers Make

View all articles

by **Vatroslav Bodrozic** - Freelance Software Engineer @ Toptal

There are many pitfalls that a C++ developer may encounter. This can make quality programming very hard and maintenance very expensive. Learning the language syntax and having good programming skills in similar languages, like C# and Java, just isn't enough to utilize C++'s full potential. It requires years of experience and great discipline to avoid errors in C++. In this article, we are going to take a look at some of the common mistakes that are made by developers of all levels if they are not careful enough with C++ development.

Common Mistake #1: Using "new" and "delete" Pairs Incorrectly

No matter how much we try, it is very difficult to free all dynamically allocated memory. Even if we can do that, it is often not safe from exceptions. Let us look at a simple example:

```
void SomeMethod()
{
  ClassA *a = new ClassA;
  SomeOtherMethod();      // it can throw an exception
  delete a;
}
```

If an exception is thrown, the "a" object is never deleted. The following example shows a safer and shorter way to do that. It uses auto_ptr which is deprecated in C++11, but the old standard is still widely used. It can be replaced with C++11 unique_ptr or scoped_ptr from Boost if possible.

```
void SomeMethod()
{
  std::auto_ptr<ClassA> a(new ClassA); // deprecated, please check the text
  SomeOtherMethod();      // it can throw an exception
}
```

No matter what happens, after creating the "a" object it will be deleted as soon as the program execution exits from the scope.

However, this was just the simplest example of this C++ problem. There are many examples when deleting should be done at some other place, perhaps in an outer function or another thread. That is why the use of new/delete in pairs should be completely avoided and appropriate smart pointers should be used instead.

Common Mistake #2: Forgotten Virtual Destructor

This is one of the most common errors that leads to memory leaks inside derived classes if there is dynamic memory allocated inside them. There are some cases when virtual destructor is not desirable, i.e. when a class is not intended for inheritance and its size and performance is crucial.

Virtual destructor or any other virtual function introduces additional data inside a class structure, i.e. a pointer to a virtual table which makes the size of any instance of the class bigger.

However, in most cases classes can be inherited even if it is not originally intended. So it is a very good practice to add a virtual destructor when a class is declared. Otherwise, if a class must not contain virtual functions due to performance reasons, it is a good practice to put a comment inside a class declaration file indicating that the class should not be inherited. One of the best options to avoid this issue is to use an IDE that supports virtual destructor creation during a class creation.

One additional point to the subject are classes/templates from the standard library. They are not intended for inheritance and they do not have a virtual destructor. If, for example, we create a new enhanced string class that publicly inherits from std::string there is possibility that somebody will use it incorrectly with a pointer or a reference to std::string and cause a memory leak.

```
class MyString : public std::string
{
	~MyString() {
	// ...
}
};

int main()
{
	std::string *s = new MyString();
	delete s; // May not invoke the destructor defined in MyString
}
```

To avoid such C++ issues, a safer way of reusing of a class/template from the standard library is to use private inheritance or composition.

Common Mistake #3: Deleting an Array With “delete” or Using a Smart Pointer

Creating temporary arrays of dynamic size is often necessary. After they are not required anymore, it is important to free the allocated memory. The big problem here is that C++ requires special delete operator with [] brackets, which is forgotten very easily. The delete[] operator will not just delete the memory allocated for an array, but it will first call destructors of all objects from an array. It is also incorrect to use the delete operator without [] brackets for primitive types, even though there is no destructor for these types. There is no guarantee for every compiler that a pointer to an array will point to the first element of the array, so using delete without [] brackets can result in undefined behaviour too.

Using smart pointers, such as auto_ptr, unique_ptr<T>, shared_ptr, with arrays is also incorrect. When such a smart pointer exits from a scope, it will call a delete operator without [] brackets which results in the same issues described above. If using of a smart pointer is required for an array, it is possible to use scoped_array or shared_array from Boost or a unique_ptr<T[]> specialization.

If functionality of reference counting is not required, which is mostly the case for arrays, the most elegant way is to use STL vectors instead. They don’t just take care of releasing memory, but offer additional functionalities as well.

Common Mistake #4: Returning a Local Object by Reference

This is mostly a beginner's mistake, but it is worth mentioning since there is a lot of legacy code that suffers from this issue. Let's look at the following code where a programmer wanted to do some kind of optimization by avoiding unnecessary copying:

```
Complex& SumComplex(const Complex& a, const Complex& b)
{
    Complex result;
    …..
    return result;
}

Complex& sum = SumComplex(a, b);
```

The object "sum" will now point to the local object "result". But where is the object "result" located after the SumComplex function is executed? Nowhere. It was located on the stack, but after the function returned the stack was unwrapped and all local objects from the function were destructed. This will eventually result in an undefined behaviour, even for primitive types. To avoid performance issues, sometimes it is possible to use return value optimization:

```
Complex SumComplex(const Complex& a, const Complex& b)
{
     return Complex(a.real + b.real, a.imaginar + b.imaginar);
}

Complex sum = SumComplex(a, b);
```

For most of today's compilers, if a return line contains a constructor of an object the code will be optimized to avoid all unnecessary copying - the constructor will be executed directly on the "sum" object.

Common Mistake #5: Using a Reference to a Deleted Resource

These C++ problems happen more often than you may think, and are usually seen in multithreaded applications. Let us consider the following code:

Thread 1:

```
Connection& connection= connections.GetConnection(connectionId);
// ...
```

Thread 2:

```
connections.DeleteConnection(connectionId);
// …
```

Thread 1:

```
connection.send(data);
```

In this example, if both threads used the same connection ID this will result in undefined behavior. Access violation errors are often very hard to find.

In these cases, when more than one thread accesses the same resource it is very risky to keep pointers or references to the resources, because some other thread can delete it. It is much safer to use smart pointers with reference counting, for example shared_ptr from Boost. It uses atomic operations for increasing/decreasing a reference counter, so it is thread safe.

Common Mistake #6: Allowing Exceptions to Leave Destructors

It is not frequently necessary to throw an exception from a destructor. Even then, there is a better way to do that. However, exceptions are mostly not thrown from destructors explicitly. It can happen that a simple command to log a destruction of an object causes an exception throwing. Let's consider following code:

```
class A
{
public:
   A(){}
   ~A()
   {
      writeToLog(); // could cause an exception to be thrown
   }
};

// …

try
{
   A a1;
   A a2;
}
catch (std::exception& e)
{
   std::cout << "exception caught";
}
```

In the code above, if exception occurs twice, such as during the destruction of both objects, the catch statement is never executed. Because there are two exceptions in parallel, no matter whether they are of the same type or different type the C++ runtime environment does not know how to handle it and calls a terminate function which results in termination of a program's execution.

So the general rule is: never allow exceptions to leave destructors. Even if it is ugly, potential exception has to be protected like this:

```
   try
   {
      writeToLog(); // could cause an exception to be thrown
   }
   catch (...) {}
```

The auto_ptr template is deprecated from C++11 because of a number of reasons. It is still widely used, since most projects are still being developed in C++98. It has a certain characteristic that is probably not familiar to all C++ developers, and could cause serious problems for somebody who is not careful. Copying of auto_ptr object will transfer an ownership from one object to another. For example, the following code:

```
auto_ptr<ClassA> a(new ClassA); // deprecated, please check the text
auto_ptr<ClassA> b = a;
a->SomeMethod();    // will result in access violation error
```

… will result in an access violation error. Only object "b" will contain a pointer to the object of Class A, while "a" will be empty. Trying to access a class member of the object "a" will result in

an access violation error. There are many ways of using auto_ptr incorrectly. Four very critical things to remember about them are:

1. Never use auto_ptr inside STL containers. Copying of containers will leave source containers with invalid data. Some STL algorithms can also lead to invalidation of “auto_ptr”s.
2. Never use auto_ptr as a function argument since this will lead to copying, and leave the value passed to the argument invalid after the function call.
3. If auto_ptr is used for data members of a class, be sure to make a proper copy inside a copy constructor and an assignment operator, or disallow these operations by making them private.
4. Whenever possible use some other modern smart pointer instead of auto_ptr.

Common Mistake #8: Using Invalidated Iterators and References

It would be possible to write an entire book on this subject. Every STL container has some specific conditions in which it invalidates iterators and references. It is important to be aware of these details while using any operation. Just like the previous C++ problem, this one can also occur very frequently in multithreaded environments, so it is required to use synchronization mechanisms to avoid it. Lets see the following sequential code as an example:

```
vector<string> v;
v.push_back(“string1”);
string& s1 = v[0];     // assign a reference to the 1st element
vector<string>::iterator iter = v.begin();    // assign an iterator to the 1st
element
v.push_back(“string2”);
cout << s1;     // access to a reference of the 1st element
cout << *iter;  // access to an iterator of the 1st element
```

From a logical point of view the code seems completely fine. However, adding the second element to the vector may result in reallocation of the vector’s memory which will make both the iterator and the reference invalid and result in an access violation error when trying to access them in the last 2 lines.

Common Mistake #9: Passing an Object by Value

You probably know that it is a bad idea to pass objects by value due to its performance impact. Many leave it like that to avoid typing extra characters, or probably think of returning later to do the optimization. It usually never gets done, and as a result leads to lesser performant code and code that is prone to unexpected behavior:

```
class A
{
public:
   virtual std::string GetName() const {return "A";}
    …
};

class B: public A
{
public:
   virtual std::string GetName() const {return "B";}
   ...
```

```
};

void func1(A a)
{
   std::string name = a.GetName();
   ...
}

B b;
func1(b);
```

This code will compile. Calling of the “func1” function will create a partial copy of the object “b”, i.e. it will copy only class “A”’s part of the object “b” to the object “a” (“slicing problem”). So inside the function it will also call a method from the class “A” instead of a method from the class “B” which is most likely not what is expected by somebody who calls the function.

Similar problems occur when attempting to catch exceptions. For example:

```
class ExceptionA: public std::exception;
class ExceptionB: public ExceptionA;

try
{
   func2(); // can throw an ExceptionB exception
}
catch (ExceptionA ex)
{
    writeToLog(ex.GetDescription());
    throw;
}
```

When an exception of type ExceptionB is thrown from the function “func2” it will be caught by the catch block, but because of the slicing problem only a part from the ExceptionA class will be copied, incorrect method will be called and also re-throwing will throw an incorrect exception to an outside try-catch block.

To summarize, always pass objects by reference, not by value.

Common Mistake #10: Using User Defined Conversions by Constructor and Conversion Operators

Even the user defined conversions are very useful sometimes, but they can lead to unpredicted conversions that are very hard to locate. Let’s say somebody created a library that has a string class:

```
class String
{
public:
   String(int n);
   String(const char *s);
   ….
}
```

The first method is intended to create a string of a length n, and the second is intended to create a string containing the given characters. But the problem starts as soon as you have something like this:

```
String s1 = 123;
String s2 = ‘abc’;
```

In the example above, s1 will become a string of size 123, not a string that contains the characters “123”. The second example contains single quotation marks instead of double quotes (which may happen by accident) which will also result in calling of the first constructor and creating a string with a very big size. These are really simple examples, and there are many more complicated cases that lead to confusion and unpredicted conversions that are very hard to find. There are 2 general rules of how to avoid such problems:

1. Define a constructor with explicit keyword to disallow implicit conversions.
2. Instead of using conversion operators, use explicit conversation methods. It requires a little bit more typing, but it is much cleaner to read and can help avoid unpredictable results.

Conclusion

C++ is a powerful language. In fact, many of the applications that you use every day on your computer and have come to love are probably built using C++. As a language, C++ gives a tremendous amount of flexibility to the developer, through some of the most sophisticated features seen in object-oriented programming languages. However, these sophisticated features or flexibilities can often become the cause of confusion and frustration for many developers if not used responsibly. Hopefully this list will help you understand how some of these common mistakes influence what you can achieve with C++.