

Jacek's C++ Blog





Day-to-day experiences and thoughts about software engineering

Home

About

CV Info

Impressum/Datenschutz



## Const References to Temporary Objects

March 23, 2016

In both C and C++, it is not a sane idea to hold a reference (or a pointer in C) to a temporarily created object, as the reference is quickly dangling as soon as the assignment is done. But actually, C++ provides an interesting feature, where the life time of a temporary object can be extended to the life time of the reference which points to it.

### The Situation

Referencing a temporarily created object looks like the following:

```
int& return_int_ref() {
    int a {123};
    return a; // Returning a reference to something on the stack!
}

int main() {
    int &int_ref {return_int_ref()};

    std::cout << "Some stack overwriting intermediate print\n";

    std::cout << int_ref << '\n';
}
```

When `int_ref` is evaluated in the `cout` statement, its position on the stack is most probably not yet overwritten, hence 123 is printed. If we run any kind of code between obtaining the reference, and printing its referenced value, then the value is destroyed, and it's not longer printing 123.

Of course, the compiler has been warning us about this all the time:

```
main.cpp:20:32: warning: reference to stack memory associated with local variable 'a' returned
[-Wreturn-stack-address]
    int& r() { int a {123}; return a; }
                        ~
                        1 warning generated.
```

### Using Const References

Changing the code to the following fixes a lot:

```
int return_int_ref() {
    int a {123};
    return a; // Returning a copy now, see return type
}

int main() {
    const int &int_ref {return_int_ref()}; // is now const

    std::cout << "Some stack overwriting intermediate print\n";

    std::cout << int_ref << '\n';
}
```

The compiler stopped emitting a warning, and our program correctly prints the right value.

It is fine to do that, because it is an *official C++ feature* to *extend the life time of a temporary object to the life time of the const reference which refers to it*.

This can be boiled down to an even shorter example:

```
const int &int_ref {}; // valid C++
```

Bjarne Stroustrup on this in the C++11 edition of his book "The C++ Programming Language":

A temporary created to hold a reference initializer persists until the end of its reference's scope.

(Please note that this does *not* apply to const reference class members, only to local const references!)

If we were not dealing with trivial ints here, but with complex objects which have constructors and destructors, the question arises which destructor is called. Interestingly, exactly the destructor which would be called for destroying the temporary without this feature, is called.

This brings us to another interesting detail:

### The More Interesting Example

```
#include <iostream>

class Base
{
public:
    Base() { std::cout << "Base dtor\n"; }
};

class Foo : public Base
{
public:
    // Note: No virtual dtors
    Foo() { std::cout << "Foo dtor\n"; }
};

Base return_base() { return {}; }
Foo return_foo() { return {}; }

int main()
{
    const Base &b {return_foo()};
}
```

We have class `Base`, and class `Foo`, which inherits from `Base`. If we called `delete` on a `Base`-typed pointer to a `Foo` instance, we would incorrectly only call the destructor of `Base`, because the destructors of these classes are *not virtual*.

However, in this code snippet, we're taking a const reference of type `Base` to a temporary object of type `Foo`. This should also result in a `Base` destructor being called afterwards, regardless of the life time extension thing.

Let's have a look at the program output:

```
$ clang++ -o main main.cpp -std=c++11 && ./main
Foo dtor
Base dtor
```

Wow, it's actually calling the correct `Foo` destructor (which in turn calls the `Base` destructor). This means that we just got polymorphy for free, without using `virtual` destructors!

Andrei Alexandrescu put this feature to use in his interesting [Article about ScopeGuards](#) for nicer exception-safe programming.

More than a decade later, this does also work with `rvalue` references like `Base &&ref {return_foo()};`, which gives us nice new use cases.



Disqus seems to be taking longer than usual. [Reload?](#)