9.11 — Return by reference and return by address

ALEX[1]
APRIL 4, 2022

In previous lessons, we discussed that when passing an argument by value, a copy of the argument is made into the function parameter. For fundamental types (which are cheap to copy), this is fine. But copying is typically expensive for class types (such as `std::string`). We can avoid making an expensive copy by utilizing passing by (const) reference (or pass by address) instead.

We encounter a similar situation when returning by value: a copy of the return value is passed back to the caller. If the return type of the function is a class type, this can be expensive.

```
std::string returnByValue(); // returns a copy of a std::string (expensive)
```

Return by reference

In cases where we're passing a class type back to the caller, we may (or may not) want to return by reference instead. **Return by reference** returns a reference that is bound to the object being returned, which avoids making a copy of the return value. To return by reference, we simply define the return value of the function to be a reference type:

```
std::string&       returnByReference(); // returns a reference to an existing std::string (cheap)
const std::string& returnByReferenceToConst(); // returns a const reference to an existing std::string (cheap)
```

Here is an academic program to demonstrate the mechanics of return by reference:

```
#include <iostream>
#include <string>

const std::string& getProgramName() // returns a const reference
{
    static const std::string s_programName { "Calculator" }; // has static duration, destroyed at end of program

    return s_programName;
}

int main()
{
    std::cout << "This program is named " << getProgramName();

    return 0;
}
```

This program prints:

```
This program is named Calculator
```

Because `getProgramName()` returns a const reference, when the line `return s_programName` is executed, `getProgramName()` will return a const reference to `s_programName` (thus avoiding making a copy). That const reference can then be used by the caller to access the value of `s_programName`, which is printed.

The object being returned by reference must exist after the function returns

Using return by reference has one major caveat: *the programmer must* be sure that the object being referenced outlives the function returning the reference. Otherwise, the reference being returned will be left dangling (referencing an object that has been destroyed), and use of that reference will result in undefined behavior.

In the program above, because `s_programName` has static duration, `s_programName` will exist until the end of the program. When `main()` accesses the returned reference, it is actually accessing `s_programName`, which is fine, because `s_programName` won't be destroyed until later.

Now let's modify the above program to show what happens in the case where our function returns a dangling reference:

```
#include <iostream>
#include <string>

const std::string& getProgramName()
{
    const std::string programName { "Calculator" }; // now a local variable, destroyed when function ends

    return programName;
}

int main()
{
    std::cout << "This program is named " << getProgramName();

    return 0;
}
```

The result of this program is undefined. When `getProgramName()` returns, a reference bound to local variable `programName` is returned. Then, because `programName` is a local variable with automatic duration, `programName` is destroyed at the end of the function. That means the returned reference is now dangling, and use of `programName` in the `main()` function results in undefined behavior.

Modern compilers will produce a warning or error if you try to return a local variable by reference (so the above program may not even compile), but compilers sometimes have trouble detecting more complicated cases.

> **Warning**
>
> Objects returned by reference must live beyond the scope of the function returning the reference, or a dangling reference will result. Never return a local variable by reference.

Don't return non-const local static variables by reference

In the original example above, we returned a local static variable by reference to illustrate the mechanics of return by reference in a simple way. However, returning non-const static variables by reference is fairly non-idiomatic, and should generally be avoided. Here's a simplified example that illustrates one such problem that can occur:

```
#include <iostream>
#include <string>

const int& getNextId()
{
    static int s_x{ 0 }; // note: variable is non-const
    ++s_x; // generate the next id
    return s_x; // and return a reference to it
}

int main()
{
    const int& id1 { getNextId() }; // id1 is a reference
    const int& id2 { getNextId() }; // id2 is a reference

    std::cout << id1 << id2;

    return 0;
}
```

This program prints:

```
22
```

This happens because `id1` and `id2` are referencing the same object (the static variable `s_x`), so when anything (e.g. `getNextId()`) modifies that value, all references are now accessing the modified value. Another issue that commonly occurs with programs that return a static local by const reference is that there is no standardized way to reset `s_x` back to the default state. Such programs must either use a non-idiomatic solution (e.g. a reset parameter), or can only be reset by quitting and restarting the program.

While the above example is a bit silly, there are permutations of the above that programmers sometimes try for optimization purposes, and then their programs don't work as expected.

> **Best practice**
>
> Avoid returning references to non-const local static variables.

Returning a const reference to a *const* local static variable is sometimes done if the local variable being returned by reference is expensive to create (so we don't have to recreate the variable every function call). But this is rare.

Returning a const reference to a *const* global variable is also sometimes done as a way to encapsulate access to a global variable. We discuss this in lesson 6.8 – Why (non-const) global variables are evil.[https://www.learncpp.com/cpp-tutorial/why-non-const-global-variables-are-evil/][2]. When used intentionally and carefully, this is also okay.

Assigning/initializing a normal variable with a returned reference makes a copy

If a function returns a reference, and that reference is used to initialize or assign to a non-reference variable, the return value will be copied (as if it had been returned by value).

```
#include <iostream>
#include <string>

const int& getNextId()
{
    static int s_x{ 0 };
    ++s_x;
    return s_x;
}

int main()
{
    const int id1 { getNextId() }; // id1 is a normal variable now and receives a copy of the value returned by reference from getNextId()
    const int id2 { getNextId() }; // id2 is a normal variable now and receives a copy of the value returned by reference from getNextId()

    std::cout << id1 << id2;

    return 0;
}
```

In the above example, `getNextId()` is returning a reference, but `id1` and `id2` are non-reference variables. In such a case, the value of the returned reference is copied into the normal variable. Thus, this program prints:

```
12
```

Of course, this also defeats the purpose of returning a value by reference.

Also note that if a program returns a dangling reference, the reference is left dangling before the copy is made, which will lead to undefined behavior:

```
#include <iostream>
#include <string>

const std::string& getProgramName() // will return a const reference
{
    const std::string programName{ "Calculator" };

    return programName;
}

int main()
{
    std::string name { getProgramName() }; // makes a copy of a dangling reference
    std::cout << "This program is named " << name; // undefined behavior

    return 0;
}
```

It's okay to return reference parameters by reference

There are quite a few cases where returning objects by reference makes sense, and we'll encounter many of those in future lessons. However, there is one useful example that we can show now.

If a parameter is passed into a function by reference, it's safe to return that parameter by reference. This makes sense: in order to pass an argument to a function, the argument must exist in the scope of the caller. When the called function returns, that object must still exist in the scope of the caller.

Here is a simple example of such a function:

```
#include <iostream>
#include <string>

// Takes two std::string objects, returns the one that comes first alphabetically
const std::string& firstAlphabetical(const std::string& a, const std::string& b)
{
    return (a < b) ? a : b; // We can use operator< on std::string to determine which comes first alphabetically
}

int main()
{
    std::string hello { "Hello" };
    std::string world { "World" };

    std::cout << firstAlphabetical(hello, world);
    return 0;
}
```

This prints:

```
Hello
```

In the above function, the caller passes in two std::string objects by const reference, and whichever of these strings comes first alphabetically is passed back by const reference. If we had used pass by value and return by value, we would have made up to 3 copies of std::string (one for each parameter, one for the return value). By using pass by reference/return by reference, we can avoid those copies.

The caller can modify values through the reference

When an argument is passed to a function by non-const reference, the function can use the reference to modify the value of the argument.

Similarly, when a non-const reference is returned from a function, the caller can use the reference to modify the value being returned.

Here's an illustrative example:

```
#include <iostream>

// takes two integers by non-const reference, and returns the greater by reference
int& max(int& x, int& y)
{
    return (x > y) ? x : y;
}

int main()
{
    int x{ 5 };
    int y{ 6 };

    max(x, y) = 7; // sets the greater of x or y to 7

    std::cout << x << y;

    return 0;
}
```

In the above program, `max()` returns by reference whichever parameter has a larger value (in this case, `y`). The caller (`main()`) then uses this reference to modify the value of that object to `7`.

This prints:

```
57
```

Return by address

Return by address works almost identically to return by reference, except a pointer to an object is returned instead of a reference to an object. Return by address has the same primary caveat as return by reference -- the object being returned by address must outlive the scope of the function returning the address, otherwise the caller will receive a dangling pointer.

The major advantage of return by address over return by reference is that we can have the function return `nullptr` if there is no valid object to return. For example, let's say we have a list of students that we want to search. If we find the student we are looking for in the list, we can return a pointer to the object representing the matching student. If we don't find any students matching, we can return `nullptr` to indicate a matching student object was not found.

The major disadvantage of return by address is that the caller has to remember to do a `nullptr` check before dereferencing the return value, otherwise a null pointer dereference may occur and undefined behavior will result. Because of this danger, return by reference should be preferred over return by address unless the ability to return "no object" is needed.

> **Best practice**
>
> Prefer return by reference over return by address unless the ability to return "no object" (using `nullptr`) is important.

Next lesson
9.12 Type deduction with pointers, references, and const

Back to table of contents

Previous lesson
9.10 Pass by address (part 2)