7.4a — Returning values by value, reference, and address

BY ALEX ON FEBRUARY 25TH, 2008 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

In the three previous lessons, you learned about passing arguments to functions by value, reference, and address. In this section, we'll consider the issue of returning values back to the caller via all three methods.

As it turns out, returning values from a function to its caller by value, address, or reference works almost exactly the same way as passing parameters to a function does. All of the same upsides and downsides for each method are present. The primary difference between the two is simply that the direction of data flow is reversed. However, there is one more added bit of complexity -- because local variables in a function go out of scope and are destroyed when the function returns, we need to consider the effect of this on each return type.

Return by value

Return by value is the simplest and safest return type to use. When a value is returned by value, a copy of that value is returned to the caller. As with pass by value, you can return by value literals (e.g. 5), variables (e.g. x), or expressions (e.g. x+1), which makes return by value very flexible.

Another advantage of return by value is that you can return variables (or expressions) that involve local variables declared within the function without having to worry about scoping issues. Because the variables are evaluated before the function returns, and a copy of the value is returned to the caller, there are no problems when the function's variable goes out of scope at the end of the function.

```
int doubleValue(int x)
{
    int value = x * 2;
    return value; // A copy of value will be returned here
} // value goes out of scope here
```

Return by value is the most appropriate when returning variables that were declared inside the function, or for returning function arguments that were passed by value. However, like pass by value, return by value is slow for structs and large classes.

When to use return by value:

- When returning variables that were declared inside the function
- When returning function arguments that were passed by value

When not to use return by value:

- When returning a built-in array or pointer (use return by address)
- When returning a large struct or class (use return by reference)

Return by address

Returning by address involves returning the address of a variable to the caller. Similar to pass by address, return by address can only return the address of a variable, not a literal or an expression (which don't have addresses). Because return by address just copies an address from the function to the caller, return by address is fast.

However, return by address has one additional downside that return by value doesn't -- if you try to return the address of a variable local to the function, your program will exhibit undefined behavior. Consider the following example:

```
int* doubleValue(int x)
{
    int value = x * 2;
    return &value; // return value by address here
} // value destroyed here
```

As you can see here, value is destroyed just after its address is returned to the caller. The end result is that the caller ends up with the address of non-allocated memory (a dangling pointer), which will cause problems if used. This is a common mistakes that new programmers make. Many newer compilers will give a warning (not an error) if the programmer tries to return a local variable by address -- however, there are quite a few ways to trick the compiler into letting you do something illegal without generating a warning, so the burden is on the programmer to ensure the address they are returning will be to a valid variable after the function returns.

Return by address is often used to return dynamically allocated memory to the caller:

```
int* allocateArray(int size)
{
    return new int[size];
}

int main()
{
    int *array = allocateArray(25);

    // do stuff with array

    delete[] array;
    return 0;
}
```

This works because dynamically allocated memory does not go out of scope at the end of the block in which it is declared, so that memory will still exist when the address is returned back to the caller.

When to use return by address:

- When returning dynamically allocated memory
- When returning function arguments that were passed by address

When not to use return by address:

- When returning variables that were declared inside the function (use return by value)
- When returning a large struct or class that was passed by reference (use return by reference)

Return by reference

Similar to pass by address, values returned by reference must be variables (you can not return a reference to a literal or an expression). When a variable is returned by reference, a reference to the variable is passed back to the caller. The caller can then use this reference to continue modifying the variable, which can be useful at times. Return by reference is also fast, which can be useful when returning structs and classes.

However, just like return by address, you should not return local variables by reference. Consider the following example:

```
int& doubleValue(int x)
{
    int value = x * 2;
    return value; // return a reference to value here
} // value is destroyed here
```

In the above program, the program is returning a reference to a value that will be destroyed when the function returns. This would mean the caller receives a reference to garbage. Fortunately, your compiler will probably give you a warning or error if you try to do this.

Return by reference is typically used to return arguments passed by reference to the function back to the caller. In the following example, we return (by reference) an element of an array that was passed to our function by reference:

```
#include <array>
#include <iostream>

// Returns a reference to the index element of array
int& getElement(std::array<int, 25> &array, int index)
{
    // we know that array[index] will not be destroyed when we return to the caller (since the caller p
assed in the array in the first place!)
    // so it's okay to return it by reference
    return array[index];
}

int main()
{
    std::array<int, 25> array;

    // Set the element of array with index 10 to the value 5
    getElement(array, 10) = 5;

    std::cout << array[10] << '\n';

    return 0;
}
```

This prints:

```
5
```

When we call `getElement(array, 10)`, getElement() returns a reference to the array element with index 10. main() then uses this reference to assign that element the value 5.

Although this is somewhat of a contrived example (because you can access array[10] directly), once you learn about classes you will find a lot more uses for returning values by reference.

When to use return by reference:

- When returning a reference parameter
- When returning an element from an array that was passed into the function
- When returning a large struct or class that will not be destroyed at the end of the function (e.g. one that was passed in)

When not to use return by reference:

- When returning variables that were declared inside the function (use return by value)
- When returning a built-in array or pointer value (use return by address)

Mixing return references and values

Although a function may return a value or a reference, the caller may or may not assign the result to a value or reference accordingly. Let’s look at what happens when we mix value and reference types.

```
int returnByValue()
{
    return 5;
}

int& returnByReference()
{
     static int x = 5; // static ensures x isn't destroyed when it goes out of scope
     return x;
}

int main()
{
    int value = returnByReference(); // case A -- ok, treated as return by value
    int &ref = returnByValue(); // case B -- compile error since the value is an r-value, and an r-value can't bind to a non-const reference
    const int &cref = returnByValue(); // case C -- ok, the lifetime of the return value is extended to the lifetime of cref
}
```

In case A, we’re assigning a reference return value to a non-reference variable. Because value isn’t a reference, the return value is copied into value, as if returnByReference() had returned by value.

In case B, we’re trying to initialize reference ref with the copy of the return value returned by returnByValue(). However, because the value being returned doesn’t have an address (it’s an r-value), this will cause a compile error.

In case C, we’re trying to initialize const reference cref with the copy of the return value returned by returnByValue(). Because const references can bind to r-values, there’s no problem here. Normally, r-values expire at the end of the expression in which they are created -- however, when bound to a const reference, the lifetime of the r-value (in this case, the return value of the function) is extended to match the lifetime of the reference (in this case, cref)

Returning multiple values

C++ doesn't contain a direct method for passing multiple values back to the caller. While you can sometimes restructure your code in such a way that you can pass back each data item separately (e.g. instead of having a single function return two values, have two functions each return a single value), this can be cumbersome and unintuitive.

Fortunately, there are several indirect methods that can be used.

As covered in lesson **7.3 -- Passing arguments by reference**, out parameters provide one method for passing multiple bits of data back to the caller. We don't recommend this method.

A second method involves using a data-only struct:

```
#include <iostream>

struct S
{
    int m_x;
    double m_y;
};

S returnStruct()
{
    S s;
    s.m_x = 5;
    s.m_y = 6.7;
    return s;
}

int main()
{
    S s = returnStruct();
    std::cout << s.m_x << ' ' << s.m_y << '\n';

    return 0;
}
```

A third way (introduced in C++11) is to use std::tuple. A tuple is a sequence of elements that may be different types, where the type of each element must be explicitly specified.

```
#include <tuple>
#include <iostream>

std::tuple<int, double> returnStruct() // return a tuple that contains an int and a double
{
    return std::make_tuple(5, 6.7); // use std::make_tuple() as shortcut to make a tuple to return
}

int main()
{
    std::tuple<int, double> s = returnStruct(); // get our tuple
    std::cout << std::get<0>(s) << ' ' << std::get<1>(s) << '\n'; // use std::get<n> to get the nth element of the tuple

    return 0;
}
```

This works identically to the prior example.

As of C++17, a structured binding declaration can be used to simplify splitting multiple returned values into separate variables:

```
#include <tuple>
#include <iostream>

std::tuple<int, double> returnStruct() // return a tuple that contains an int and a double
{
    return std::make_tuple(5, 6.7); // use std::make_tuple() as shortcut to make a tuple to return
}

int main()
{
    auto [a, b] = returnStruct(); // used structured binding declaration to put results of tuple in var
iables a and b
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

Using a struct is a better option than a tuple if you're using the struct in multiple places. However, for cases where you're just packaging up these values to return and there would be no reuse from defining a new struct, a tuple is a bit cleaner since it doesn't introduce a new user-defined data type.

Conclusion

Most of the time, return by value will be sufficient for your needs. It's also the most flexible and safest way to return information to the caller. However, return by reference or address can also be useful, particularly when working with dynamically allocated classes or structs. When using return by reference or address, make sure you are not returning a reference to, or the address of, a variable that will go out of scope when the function returns!

Quiz time

Write function prototypes for each of the following functions. Use the most appropriate parameter and return types (by value, by address, or by reference), including use of const where appropriate.

1) A function named sumTo() that takes an integer parameter and returns the sum of all the numbers between 1 and the input number.

Hide Solution

```
int sumTo(const int value);
```

2) A function named printEmployeeName() that takes an Employee struct as input.

Hide Solution

```
void printEmployeeName(const Employee &employee);
```

3) A function named minmax() that takes two integers as input and passes back to the caller the smaller and larger number as separate parameters.

Hide Solution

```
void minmax(const int x, const int y, int &minOut, int &maxOut);
```

4) A function named getIndexOfLargestValue() that takes an integer array (as a pointer) and an array size, and returns the index of the largest element in the array.

Hide Solution

```
int getIndexOfLargestValue(const int *array, const int length);
```

5) A function named getElement() that takes an integer array (as a pointer) and an index and returns the array element at that index (not a copy). Assume the index is valid, and the return value is const.

Hide Solution

```
const int& getElement(const int *array, const int index);
```

7.5 -- Inline functions