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

BY ALEX ON FEBRUARY 25TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 arguments 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
```

```cpp
} // 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 mistake 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 was often used to return dynamically allocated memory to the caller:

```cpp
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. Keeping track of manual allocations can be difficult. Separating the allocation and deletion into different functions makes it even harder to understand who's responsible for deleting the resource or if the resource needs to be deleted at all. Smart pointers (covered later) and types that clean up after themselves should be used instead of manual allocations.

When to use return by address:

- When returning dynamically allocated memory and you can't use a type that handles allocations for you
- 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 or parameters that were passed by value (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 should not return a reference to a literal or an expression that resolves to a temporary value, as those will go out of scope at the end of the function and you'll end up returning a dangling reference). 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:

```cpp
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
    // 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 or parameters that were passed by value (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 the function ends
     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
    const int &cref{ returnByValue() }; // case C -- ok, the lifetime of the return value is e

    return 0;
}
```

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)

Lifetime extension doesn't save dangling references

Consider the following program:

```
const int& returnByReference()
{
     return 5;
}

int main()
{
    const int &ref { returnByReference() }; // runtime error
}
```

In the above program, returnByReference() is returning a const reference to a value that will go out of scope when the function ends. This is normally a no-no, as it will result in a dangling reference. However, we also know that assigning a value to a const reference can extend the lifetime of that value. So which takes precedence here? Does 5 go out of scope first, or does ref extend the lifetime of 5?

The answer is that 5 goes out of scope first, then the reference to 5 is copied back to the caller, and then ref extends the lifetime of the now-dangling reference.

However, the following does work as expected:

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

int main()
```

```
{
    const int &ref { returnByValue() }; // ok, we're extending the lifetime of the copy passed
}
```

In this case, the literal value 5 is first copied back into the scope of the caller (main), and then ref extends the lifetime of that copy.

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.

Here's an example that returns a tuple, and uses std::get to get the nth element of the tuple:

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

std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
{
    return { 5, 6.7 };
}

int main()
{
    std::tuple s{ returnTuple() }; // get our tuple
    std::cout << std::get<0>(s) << ' ' << std::get<1>(s) << '\n'; // use std::get<n> to get th

    return 0;
}
```

This works identically to the prior example.

You can also use std::tie to unpack the tuple into predefined variables, like so:

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

std::tuple<int, double> returnTuple() // return a tuple that contains an int and a double
{
    return { 5, 6.7 };
}

int main()
{
    int a;
    double b;
    std::tie(a, b) = returnTuple(); // put elements of tuple in variables a and b
    std::cout << a << ' ' << b << '\n';

    return 0;
}
```

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> returnTuple() // return a tuple that contains an int and a double
{
    return { 5, 6.7 };
}

int main()
{
    auto [a, b]{ returnTuple() }; // used structured binding declaration to put results of tup
    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.

Show Solution

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

Show Solution

3) A function named minmax() that takes two integers as input and returns back to the caller the smaller and larger number in a `std::pair`. A `std::pair` works identical to a `std::tuple` but stores exactly two elements.

Show Solution

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

Show Solution

5) A function named getElement() that takes an array of `std::string` (as a `std::vector`) 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.

Show Solution

7.5 -- Inline functions

Index

7.4 -- Passing arguments by address

C++ TUTORIAL | PRINT THIS POST

245 comments to 7.4a — Returning values by value, reference, and address

Nguyen
January 16, 2020 at 4:06 pm · Reply

"Return by reference is typically used to return arguments passed by reference to the function back to the caller."

Here is my simple program.

```
#include <iostream>
int& doubleValue(int &y)
{
    return y;
} // Is y destroyed here?
int main()
{
    int x;
    doubleValue(x) = 123;

    std::cout<<x<<std::endl;

    return 0;
}
```

I think the program is returning a reference to y that will be destroyed when the function returns. This would mean the caller receives a reference to garbage. I don't understand why it seems working. Could you please explain?

nascardriver
January 17, 2020 at 1:27 am · Reply

\`y\` is a reference to \`x\`. When \`doubleReference\` returns \`y\`, it's returning a reference to \`x\`. \`x\` lives until the end of \`main\`.

Ged
December 4, 2019 at 2:01 pm · Reply

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.

But in exercise 3 you are using this code.

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

1. Shouldn't we be using this?

```
std::tuple<int,int> minMax(const int n1, const int n2)
```

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.

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

2. What's the difference? It is only one array element. Why should it be returned with const int&.

3. Could you explain the return by reference a bit more? Cause for example if I have a struct and I pass it by reference, why should I return it if the struct is not local and the change applies outside of the function?

The only reason I see now is if I return a struct by reference and assign it to a new variable. It only copies one time instead of two? (if i return struct with no reference it has to make a copy via return and then a copy to assign to the new value) Am I wrong?

```
struct Example
{
    int n1;
    int n2;
    int n3;
};

Example& test(Example& list) // Would make 1 copy.
// Example test(Example& list) // Would make 2 copies.
{
    return list;
}

int main()
{
Example list;
Example er{test(list)};
return 0;
}
```

nascardriver
December 5, 2019 at 1:41 am · Reply

1.
Quiz updated to use `std::pair` rather than reference parameters.

2.
Quiz updated to use an array of `std::string`, as an `int` should be returned by value, not by reference.

Thanks for the improvements!

3.
If a function returns a reference that it got as an argument, that function can be used to chain calls. For example, `std::cout << "hello"` returns `std::cout`. A reference to `std::cout` is implicitly passed to `std::cout <<`, you'll learn more about that later. This allows us to use `<<` again and again

```
std::cout << 123 << "hello" << 321.0 << '\n';

// rather than

std::cout << 123;
std::cout << "hello";
std::cout << 321.0;
std::cout << '\n';
```

Aside from that, the function could return one of multiple parameters that are all of the same type, eg.

```
Car& getFasterCar(Car& a, Car& b);

rev(getFasterCar(zonda, taycan));
```

potterman28wxcv
August 17, 2019 at 7:09 am · Reply

Hello! Thanks again for this post.

I have two questions for the Quizz.

For the getIndexOfLargestValue i was tempted to write the following:

```
int getIndexOfLargestValue(const int const *ar, const int size);
```

It is supposed to tell that `ar` will not be modified. Do you think it's a good thing to write it that way, or it's too much specification?

Another question that I had is on the return by const ref:

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

I don't understand what returning a "const int &" is supposed to mean, compared to returning an "int &". When is the programmer supposed to use a "const T &" instead of a "T &" when it comes to returning?

I would have thought returning a "const T &" could prevent further modifications of the returned value, but the following compiles alright:

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

int main()
{
    int t[50]{};
    int a{getElement(t, 23)};
    int b{};
    b = getElement(t, 12);
    b++;
    return a+b;
}
```

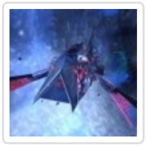
(it fails to link due to the missing implementation, but it does compile)

I can also compile the following, even though i modified ar in the function. So I don't really understand what guarantee the "const T &" is supposed to give in that scenario.

```
const int &getElement(int *ar, const int index){
    ar[index]++;
    const int &ret {ar[index]};
    return ret;
}

int main()
{
    int t[50]{};
    int a{getElement(t, 23)};
    int b{};
    b = getElement(t, 12);
    b++;
    return a+b;
}
```

Thanks!



potterman28wxcv
August 17, 2019 at 7:17 am · Reply

Alright I just figured out the difference. With the const int & declaration, I cannot do the following:

```
    getElement(t, 14) = 42;
```

So returning a (const T &) will prevent you from using it as a l-value.

nascardriver
August 18, 2019 at 12:23 am · Reply

> Do you think it's a good thing to write it that way, or it's too much specification?
The caller won't notice modifications to the pointer, as the pointer is passed by copy. The second `const` doesn't add any restrictions to the caller. To the caller, the function isn't any different than without `const`.

> When is the programmer supposed to use a "const T &" instead of a "T &" when it comes to returning?
Return a `const` reference if it's not supposed to be modified. More often than not, your references will be `const`.

> but the following compiles alright
> I can also compile the following
You're copying the returned value into a non-reference variable. When you modify `a` or `b`, you're modifying the copy, not the returned reference.

```
int& i{ getElement(t, 23) }; // Doesn't work, because @i isn't const
const int& i2{ getElement(t, 23) };
i2 = 9; // Doesn't work
```

Anastasia
August 10, 2019 at 1:53 am · Reply

Hi!
First I want to make sure that I undestood how things work. Is it close to correct?

When returning a variable
by value:
a copy of the variable's value is made which lives untill the end of the expression it's used in after returning (or dies right away if there's no expression, or, in case we pass it to a (const) reference, it's lifetime is extended to the lifetime of the reference). A copy of anything (l-values, r-values (local or not)) can be returned this way.

by address:
a pointer to the address of the variable is returned. In case of (non-static) local variables (declared in the function or passed (by value) to it) a dangling pointer is returned, since the local variable is already destroyed. Only (pointers to) addresses of variables and dynamically allocated memory can be returned.

by reference:
a reference to the value stored at the address of the variable is returned. In case of local variables we are dereferencing a dangling pointer and returning a reference to some (undefined) value it points to. In case of ordinary references only (non-const) things having an actuall address (l-values) can be returned. Const references can refer to pretty much everything, but it is pointless to try to return literals and expressions this way (they will cease to exist by the point of return anyway).

This whole lifetime extension thing is a bit confusing though. I think I got how it works (more or less), but I struggle to figure out in which cases extending the lifetime by a const reference can be normally used (apart from when returning from a function) and be useful vs. just copying the value to a normal variable.

nascardriver
August 10, 2019 at 2:42 am · Reply

> Is it close to correct?
Yes

I can't help you with your struggles, maybe someone else knows of a good example.

Anastasia
August 10, 2019 at 7:04 am · Reply

The only example I've found which I've been able to understand was this (sorry if the formatting is messed up):

```
X f( const X& i)
{
   return i;
}
const T& x = f(1);
```

And the author's explanation: "In this case we pass the value 1 (a temporary) to function f , which is then returned by value (possibly another temporary). It's pretty clear from this example that it's important that the temporary variable containing "1" be extended in order for the variable "i" to be able to take a reference to it. Remember, a reference really is a non-null pointer. Therefore it's important for the temporary's lifetime to not end before the function returns. For this to compile and be standard compliant, which it is, the temporary value containing 1 must have and extended lifetime." (https://marcbeauchesne.com/temporary-objects-lifetime-and-extensions/)

Is it a valid example of when the lifetime extension is necessary?

edit: though it seems that it's an exception, not a case where extending the lifetime was really necessary. The author writes:
"However, the case for temporary objects passed to functions is actually an exception:

The exceptions to this lifetime rule are:

A temporary object bound to a reference parameter in a function call (8.5.1.2) persists until the completion of the full-expression containing the call." [class.temporary / 15.2.6.9]. "

Now I feel absolutely lost, not sure I understand how it is supposed to work at all.

nascardriver
August 10, 2019 at 8:40 am · Reply

This is "when returning from a function".
You already know (I suppose) that you can pass temporaries to functions that want a const reference.
He's returning @i by copy, so we're back at the "when returning from a function", but we already know that that's useful.
`x` is extending the lifetime of of the temporary returned by `f`, not of the `1` that was originally passed.

Anastasia
August 10, 2019 at 8:56 am · Reply

I see now, I guess the author's wording was a bit confusing (or I'm just dumb). Actually I don't remember ever passing a literal to a const reference (I've learned about them only quite recently), so I supposed it may need it's life extended in order not to have a dangling reference in the function.
Thank you!

Nirbhay
August 1, 2019 at 1:32 am · Reply

Hello!

I did not clearly understand why the lifetime extension doesn't work on "value" in section "Lifetime extension doesn’t save dangling references" whereas it works in section "Mixing return references and values".

Especially this line:

"Lifetime extension only works when the object going out of scope is going out of scope in the same block (e.g. because it has expression scope). It does not work across function boundaries."

Can anyone please help?

Thank you

nascardriver
August 1, 2019 at 2:01 am · Reply

```
const int& returnByReference()
{
  return 5;
  // 5 dies here
}

int returnByValue()
{
  return 5;
  // 5 dies here, but we returned by copy. This copy is returned to the caller.
}

int main()
{
  const int &ref{ returnByReference() }; // 5 is already dead.
  // The 5 went out of scope at the end of @returnByReference.

  const int &ref2{ returnByValue() }; // We're binding a reference to the copy of 5.
  // That copy lives as long as ref2.
}
```

Nirbhay
August 1, 2019 at 2:36 am · Reply

Aah! When you used the word "copy"; it made things easier to grasp. Thanks!
However, how would you explain this line "Lifetime extension only works when the object going out of scope is going out of scope in the same block (e.g. because it has expression scope)" using your example above?

nascardriver
August 1, 2019 at 2:50 am · Reply

You can't extend the lifetime of the 5 from `returnByReference`, because it's not in the same scope as `ref`. `ref` is in `main`, `5` is in `returnByReference`.
The same is true for `returnByValue`, but `ref2` doesn't bind to the `5` in `returnByValue`, it binds to the copy of `5`.

That copy is stored in a temporary in `main`. `ref2` can extend the lifetime of that temporary, because they are both in `main`, ie. in the same scope.

Nirbhay
August 1, 2019 at 3:48 am · Reply

I see.

And the expression scope here means after

const int &ref2{ returnByValue() };

line is executed, the copy of 5 goes out of scope (the same scope as 'ref2' i.e. main) and as it is going out of scope in the same scope as 'ref2', the lifetime is extended. Right?

nascardriver
August 1, 2019 at 3:51 am · Reply

The lifetime _can be_ extended because they're in the same scope.
The lifetime _is_ extended, because `ref2` binds to the temporary.

Nirbhay
August 1, 2019 at 3:56 am · Reply

Ok thank you!

Jeffrey Liu
July 28, 2019 at 4:13 pm · Reply

Hi,

I'm kinda struggling to understand the different uses of references and pointers. I understand that references are some sort of implementation of pointers but have a little less functionality, but in most cases I've seen, it doesn't really matter which one is used. How do I know which one should be used in those types of cases?

Thank you!

nascardriver
July 29, 2019 at 4:12 am · Reply

Alias? Reference.
Parameter that's not modified? Reference.
Parameter that's modified? Pointer.
Nullable? Pointer.

Jeffrey Liu
July 29, 2019 at 7:20 am · Reply

Thank you for the response!

« Older Comments 1 2 3 4