12.12 — Return by reference and return by address

ALEX[1] JANUARY 3, 2025

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.

```
1 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:

```
1 std::string&       returnByReference(); // returns a reference to an existing
  std::string (cheap)
2 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:

```
1  #include <iostream>
2  #include <string>
3
4  const std::string& getProgramName() // returns a const reference
5  {
6      static const std::string s_programName { "Calculator" }; // has static duration,
   destroyed at end of program
7
8      return s_programName;
9  }
10
11 int main()
12 {
13     std::cout << "This program is named " << getProgramName();
14
15     return 0;
16 }
```

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 non-static local variable, destroyed when function ends

    return programName;
}

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

    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 (non-static) local variable or temporary by reference.

Lifetime extension doesn't work across function boundaries

Let's take a look at an example where we return a temporary by reference:

```
#include <iostream>

const int& returnByConstReference()
{
    return 5; // returns const reference to temporary object
}

int main()
{
    const int& ref { returnByConstReference() };

    std::cout << ref; // undefined behavior

    return 0;
}
```

In the above program, `returnByConstReference()` is returning an integer literal, but the return type of the function is `const int&`. This results in the creation and return of a temporary reference bound to a temporary object holding value 5. This returned reference is copied into a temporary reference in the scope of the caller. The temporary object then goes out of scope, leaving the temporary reference in the scope of the caller dangling.

By the time the temporary reference in the scope of the caller is bound to const reference variable `ref` (in `main()`), it is too late to extend the lifetime of the temporary object -- as it has already been destroyed. Thus `ref` is a dangling reference, and use of the value of `ref` will result in undefined behavior.

Here's a less obvious example that similarly doesn't work:

```
#include <iostream>

const int& returnByConstReference(const int& ref)
{
    return ref;
}

int main()
{
    // case 1: direct binding
    const int& ref1 { 5 }; // extends lifetime
    std::cout << ref1 << '\n'; // okay

    // case 2: indirect binding
    const int& ref2 { returnByConstReference(5) }; // binds to dangling reference
    std::cout << ref2 << '\n'; // undefined behavior

    return 0;
}
```

In case 2, a temporary object is created to hold value `5`, which function parameter `ref` binds to. The function just returns this reference back to the caller, which then uses the reference to initialize `ref2`. Because this is not a direct binding to the temporary object (as the refrence was bounced through a function), lifetime extension doesn't apply. This leaves `ref2` dangling, and its subsequent use is undefined behavior.

Warning

Reference lifetime extension does not work across function boundaries.

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

In the original example above, we returned a const static local variable by reference to illustrate the mechanics of return by reference in a simple way. However, returning non-const static local 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 << '\n';

    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.

This above example can be fixed by making `id1` and `id2` normal variables (rather than references) so they save a copy of the return value rather than a reference to `s_x`.

For advanced readers

Here’s another example with a less obvious version of the same problem:

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

std::string& getName()
{
    static std::string s_name{};
    std::cout << "Enter a name: ";
    std::cin >> s_name;
    return s_name;
}

void printFirstAlphabetical(const std::string& s1, const std::string& s2)
{
    if (s1 < s2)
        std::cout << s1 << " comes before " << s2 << '\n';
    else
        std::cout << s2 << " comes before " << s1 << '\n';
}

int main()
{
    printFirstAlphabetical(getName(), getName());

    return 0;
}
```

Here's the result from one run of this program:

```
Enter a name: Dave
Enter a name: Stan
Stan comes before Stan
```

In this example, `getName()` returns a reference to static local `s_name`. Initializing a `const std::string&` with a reference to `s_name` causes that `std::string&` to bind to `s_name` (not make a copy of it).

Thus, both `s1` and `s2` end up viewing `s_name` (which was assigned the last name we entered).

Note that if we use `std::string_view` parameters instead, the first `std::string_view` parameter will be invalidated when the underlying `std::string` is changed.

Another issue that commonly occurs with programs that return a non-const static local by reference is that there is no standardized way to reset `s_x` back to the default state. Such programs must either use a non-conventional solution (e.g. a reset function parameter), or can only be reset by quitting and restarting the program.

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 and/or initialize (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 7.8 -- 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 << '\n';

    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
```

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 << '\n'; // 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) << '\n';

    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.

It's okay for an rvalue passed by const reference to be returned by const reference

When an argument for a const reference parameter is an rvalue, it's still okay to return that parameter by const reference.

This is because rvalues are not destroyed until the end of the full expression in which they are created.

First, let's look at this example:

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

std::string getHello()
{
    return "Hello"; // implicit conversion to std::string
}

int main()
{
    const std::string s{ getHello() };

    std::cout << s;

    return 0;
}
```

In this case, `getHello()` returns a `std::string` by value, which is an rvalue. This rvalue is then used to initialize `s`. After the initialization of `s`, the expression in which the rvalue was created has finished evaluating, and the rvalue is destroyed.

Now let's take a look at this similar example:

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

const std::string& foo(const std::string& s)
{
    return s;
}

std::string getHello()
{
    return "Hello"; // implicit conversion to std::string
}

int main()
{
    const std::string s{ foo(getHello()) };

    std::cout << s;

    return 0;
}
```

The only difference in this case is that the rvalue is passed by const reference to `foo()` and then returned by const reference back to the caller before it is used to initialize `s`. Everything else works identically.

We discuss a similar case in lesson 14.6 -- Access functions (https://www.learncpp.com/cpp-tutorial/access-functions/)[3].

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:

```cpp
#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 a{ 5 };
    int b{ 6 };

    max(a, b) = 7; // sets the greater of a or b to 7

    std::cout << a << b << '\n';

    return 0;
}
```

In the above program, `max(a, b)` calls the `max()` function with `a` and `b` as arguments. Reference parameter `x` binds to argument `a`, and reference parameter `y` binds to argument `b`. The function then determines which of `x` (`5`) and `y` (`6`) is greater. In this case, that's `y`, so the function returns `y` (which is still bound to `b`) back to the caller. The caller then assigns the value `7` to this returned reference.

Therefore, the expression `max(a, b) = 7` effectively resolves to `b = 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.

Related content

If you need the ability to return "no object" or a value (rather than an object) 12.15 -- std::optional (https://www.learncpp.com/cpp-tutorial/stdoptional/)[4] describes a good alternative.

Related content

See 5.9 -- std::string_view (part 2) (https://www.learncpp.com/cpp-tutorial/stdstring_view-part-2/#stringvsstringview)[5] for a quick guide on when to return `std::string_view` vs `const std::string&` .

Next lesson
12.13 In and out parameters

6

Back to table of contents

7

Previous lesson
12.11 Pass by address (part 2)

8

9

macdonald
June 25, 2025 11:42 pm PDT

quite informative
thanks!

0 Reply

YFN
June 11, 2025 12:09 pm PDT

I'm confused about the following example:

```
#include <iostream>

const int& returnByConstReference(const int& ref)
{
    return ref;
}

int main()
{
    // case 1: direct binding
    const int& ref1 { 5 }; // extends lifetime
    std::cout << ref1 << '\n'; // okay

    // case 2: indirect binding
    const int& ref2 { returnByConstReference(5) }; // binds to dangling reference
    std::cout << ref2 << '\n'; // undefined behavior

    return 0;
}
```

Why does `ref2` bind to a dangling reference? Later on, you say that "it's okay for an rvalue passed by const reference to be returned by const reference", because the temporary object doesn't get destroyed until the end of the expression in which its used. Isn't that just what's happening here? An r-value, `5`, is being passed by const reference and then returned by const reference. So shouldn't this be okay?

2 Reply

Kacper
Reply to YFN [13] July 1, 2025 8:17 am PDT

I might be wrong, but in the example you showed, the reference returned by the function refers to a temporary value (5). That temporary is created during the function call and destroyed when the function returns, because it's not directly bound to a reference in the caller. Since ref2 binds to a reference that points to this already-destroyed temporary, it becomes a dangling reference. And because ref2 doesn't create a copy, it ends up referencing invalid memory.

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

const std::string& foo(const std::string& s)
{
    return s;
}

std::string getHello()
{
    return "Hello"; // implicit conversion to std::string
}

int main()
{
    const std::string s{ foo(getHello()) };

    std::cout << s;

    return 0;
}
```

Here in main he used regular const std::string to save a copy

0 Reply

Felipe
June 2, 2025 2:06 pm PDT

cool stuff

Last edited 29 days ago by Felipe

0 Reply

TryingToLearnCpp
May 8, 2025 12:18 am PDT

hey Alex I have a question about the below code:

```cpp
// scenario 1:

#include <iostream>

const int& foo(const int& ref) { // 5L converted to temporary int at this stack frame
    return ref;
} // temporary int dies here?

int main() {
    std::cout << foo(5L); // UB because the temporary has died once foo() gets popped off the call stack
    return 0;
}
```

```cpp
// scenario 2:

#include <iostream>

const int& foo(const int& ref) { // 5L converted to temporary int at this stack frame
    return ref;
} // temporary is still valid after ref returns

int main() {
    std::cout << foo(5L); // legal
    // temporary int dies here after the expression
    return 0;
}
```

- Earlier in the lessons it's been often said that the temporary doesn't die until the end of the expression it was created in.
- I'm wondering what would happen if a temporary was made in foo(), once it gets popped off the call stack upon returning ref, does the temporary converted int foo() returns become invalid resulting in UB as seen in scenario 1? Or is it valid because the original literal we give foo() isn't going to die until after std::cout << foo(5L)?

Last edited 1 month ago by TryingToLearnCpp

0 Reply

vitrums

April 2, 2025 6:16 am PDT

I swear even 10 minutes later after meticulous attempts to de-cypher the text in the authors article of which temporary reference of which temporary object at which point is going to be copied/destroyed brought nothing but headache. Honestly, the whole "temporary goes out of scope and hence destroyed" part is extremely hard to understand and remember for this level of abstraction. It's like trying to make kids just memorize how sharps and flats of the piano keys work without telling them that between any 2 adjacent keys there's the same half-step interval. So in contrast to the "temporary in scope/out of scope" abstraction when we employ reductionist way of thinking, it would be in terms of stack frames. Hence the idea of "temporary" becomes completely clear, because now all we really have to ask ourselves is whether this temporary was allocated **inside the callee's stack** and therefore it's an invalid case for by-reference/by-address returns, or if it occupies the memory **outside of the callee's stack**.

4 Reply

smart

Reply to vitrums [14] April 8, 2025 7:58 am PDT

I decided to read more about the callee's stack and ended up learning how to use the Memory Window in the debugger. Now I can always check if an address is safe to use without worrying about dangling references. I like this lesson a lot, but it would’ve been nice if this info was included.
(Although it turns out that the Memory Window doesn't always guarantee that the address is good, so you still need to know the stuff from this lesson anyway.)

Last edited 2 months ago by smart

0 Reply

Hari

Reply to smart [15] April 19, 2025 1:20 am PDT

Could you share any links to this reading?

0 Reply

smart

Reply to Hari [16] April 19, 2025 1:41 am PDT

I just ask different AIs:
DeepSeek for questions closely related to coding,
Grok for clear, step-by-step explanations of fundamental concepts (that's where I asked this one),
and ChatGPT for quick answers to lots of simple questions.

0 Reply

Sina

March 7, 2025 10:40 pm PST

Greetings! I've been trying the subjects explained in this part and I tried returning a non-Static pointer by address(expecting to get a nullptr or a dangling one) and it somehow remained intact and I got 13's addrerss printed while being called through my main function... heres my code:

```cpp
#include <iostream>

template <typename T>
void printValue(T value)
{
    std::cout << "value is: " << value << '\n';
}


template <typename T>
void printPointer(T* ptr)
{
    if (nullptr)
        std::cout << "null pointer!" << '\n';
    else
        std::cout << ptr << '\n';
}


const std::string& returnProgram()
{
    static const std::string program{ "program!" };
    return program;
}

const int* ptrReturnInt()
{
    const int a{ 13 };
    const int* program{ &a };
    return program;
}


int main()
{
    printValue(returnProgram());
    printPointer(ptrReturnInt());
    return 0;
}
```

my question is that do pointers outlive their dedicated scope? and reference lifetime in a return by ref value func are not related?

0 Reply

Sina

Reply to Sina [17] March 7, 2025 10:49 pm PST

I get that the programmer must nullptr a pointer but is making a pointer static redundant in this situation?

0 Reply

Nidhi Gupta

February 28, 2025 3:05 pm PST

This is the topic of returning by reference in C++, in which the function returns a reference to the already constructed object and not a copy. This makes it more efficient, particularly for class types, because it maintains unnecessary copies at bay. With returning by reference is stringent expectation: the referent

object must outlive the duration of the call to the function, or else the resulting reference will get left dangling with ensuing undefined behavior. Returning a local variable by reference is dangerous because local variables are destroyed when functions end. Similarly, lifetime extension is not valid over function boundaries so that temporary values returned by reference will return dangling references as well.

1 Reply

benjo

February 13, 2025 3:27 am PST

Hey, I am trying to understand what is happening in this code snippet.
I added some comments describing my thinking.
Does this look right?

```
const int& returnByConstReference()
{
    return 5; // returns const reference to temporary object

    // temp variable is created:
    // int temp { 5 };

    // temp reference bound to temp is created:
    // const int& tempRef { temp };

    // tempRef is returned:
    // return tempRef;
}

int main()
{
    // tempRef from returnByConstReference() is returned
    // temp from returnByConstReference() is destroyed
    // tempRef is now dangling
    // tempRef is bound to ref
    const int& ref { returnByConstReference() };

    std::cout << ref; // undefined behavior

    return 0;
}
```

So the above snippet works something like this one:

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

    return programName;
}

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

    return 0;
}
```

Last edited 4 months ago by benjo

2 Reply

Alex Author

Reply to benjo [18] February 13, 2025 9:20 pm PST

Yes, you have it correct. There are minor differences between the literal case and variable case when things are destroyed (the temporary materialized to hold value 5 dies at the end of the expression, whereas a variable lives until the end of the function), but from the caller's perspective, they are semantically equivalent.

1 Reply

benjo

Reply to Alex [19] February 14, 2025 4:46 am PST

Thanks!

"the temporary materialized to hold value 5 dies at the end of the expression, whereas a variable lives until the end of the function" - does this mean that the following would work (print "5" to the console):

```
1  const int& returnByConstReference()
2  {
3      return 5; // returns const reference to temporary object
4  }
5
6  int main()
7  {
8      std::cout << returnByConstReference();
9  }
```

Last edited 4 months ago by benjo

1 Reply

Alex Author

Reply to benjo [20] February 15, 2025 3:02 pm PST

No. The temporary dies at the end of `return 5;`, not `returnByConstReference()`

1 Reply

ice-shroom

February 12, 2025 10:24 pm PST

Good afternoon once again, can you repeat again why passing arguments to parameters by reference is best done by reference function? and can I pass references as arguments to a parameter, for example

```
int& x{5};
getvalue(x);
```

Last edited 4 months ago by ice-shroom

0 Reply

Alex Author

Reply to ice-shroom [21] February 13, 2025 11:15 am PST

I'm sorry, I don't understand your first question. Are you asking why we prefer pass by reference over pass by address?

As to your second question, yes, you can call `getvalue(x)`. If the function parameter has types `int&` or `const int&`, this will be a pass by reference. If the function parameter has type `int`, it will be a pass by value (and make a copy).

1 Reply

ice-shroom

Reply to Alex [22] February 13, 2025 11:26 am PST

I mean:
`int& function (int& x, int& y) { }`,
function as a reference

Last edited 4 months ago by ice-shroom

0 Reply

Alex Author

Reply to ice-shroom [23] February 13, 2025 9:45 pm PST

I don't understand what you mean by "is best done by reference function?"

Are you asking why it's okay to return a reference parameter by reference?

0 Reply

ice-shroom

Reply to Alex [24] February 14, 2025 3:28 am PST

Yes

0 Reply

Skelemen

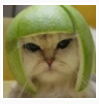
February 6, 2025 8:35 am PST

Hi, I wanted to know how this works:

```cpp
#include <iostream>

const int& foo(const int& s)
{
    return s;
}


int main()
{
    const int& s{ foo(5) };
    std::cout << s;

    return 0;
}
```

does this extend the lifetime of the value returned by foo(5) to match that of the reference s since it's a const reference that is getting initialized with a temporary rvalue?
so basically extendeing the lifetime of 5

1 Reply



Alex Author

Reply to Skelemen [25] February 8, 2025 9:57 pm PST

No. Lifetime extension doesn't work across function boundaries. This program results in undefined behavior. If the function returned by value isntead, that could be extended within the same scope.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/why-non-const-global-variables-are-evil/
3. https://www.learncpp.com/cpp-tutorial/access-functions/
4. https://www.learncpp.com/cpp-tutorial/stdoptional/
5. https://www.learncpp.com/cpp-tutorial/stdstring_view-part-2/#stringvsstringview
6. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/
9. https://www.learncpp.com/return-by-reference-and-return-by-address/
10. https://www.learncpp.com/cpp-tutorial/ellipsis-and-why-to-avoid-them/
11. https://www.learncpp.com/cpp-tutorial/input-and-output-io-streams/
12. https://gravatar.com/
13. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-610849
14. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-608983
15. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-609124

16. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-609379
17. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-608349
18. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607689
19. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607716
20. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607736
21. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607680
22. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607702
23. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607705
24. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607719
25. https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/#comment-607449
26. https://g.ezoic.net/privacy/learncpp.com