Compound Types: References and pointers

C++ supports the following compound types:

- [ ] Functions
- [ ] Arrays
- [ ] Pointer types:
  - Pointer to object
  - Pointer to function
- [ ] Pointer to member types:
  - Pointer to data member
  - Pointer to member function
- [ ] Reference types:
  - L-value references
  - R-value references
- [ ] Enumerated types:
  - Unscoped enumerations
  - Scoped enumerations
- [ ] Class types:
  - Structs
  - Classes
  - Unions

std::optional

Note that `std::optional` has a usage syntax that is essentially identical to a pointer:

Behavior	Pointer	`std::optional`
Hold no value	initialize/assign `{}` or `std::nullptr`	initialize/assign `{}` or `std::nullopt`
Hold a value	initialize/assign an address	initialize/assign a value
Check if has value	implicit conversion to bool	implicit conversion to bool or `has_value()`
Get value	dereference	dereference or `value()`

- A pointer has reference semantics, meaning it references some other object, and assignment copies the pointer, not the object. If we return a pointer by address, the pointer is copied back to the caller, not the object being pointed to. This means we can't return a local object by address, as we'll copy

that object's address back to the caller, and then the object will be destroyed, leaving the returned pointer dangling.

- A std::optional has value semantics, meaning it actually contains its value, and assignment copies the value. If we return a std::optional by value, the std::optional (including the contained value) is copied back to the caller. This means we can return a value from the function back to the caller using std::optional.

Pros and cons of returning a std::optional

Returning a std::optional is nice for a number of reasons:

- Using std::optional effectively documents that a function may return a value or not.
- We don't have to remember which value is being returned as a sentinel.
- The syntax for using std::optional is convenient and intuitive.

Returning a std::optional does come with a few downsides:

- We have to make sure the std::optional contains a value before getting the value. If we dereference a std::optional that does not contain a value, we get undefined behavior.
- std::optional does not provide a way to pass back information about why the function failed.

pointers and references

- References must be initialized, pointers are not required to be initialized (but should be).
- References are not objects, pointers are.
- References can not be reseated (changed to reference something else), pointers can change what they are pointing at.
- References must always be bound to an object, pointers can point to nothing (we'll see an example of this in the next lesson).
- References are "safe" (outside of dangling references), pointers are inherently dangerous (we'll also discuss this in the next lesson).

Initializing an lvalue reference to const with a modifiable lvalue

```
#include <iostream>

int main()
{
    int x { 5 };           // x is a modifiable lvalue
    const int& ref { x }; // okay: we can bind a const reference to a
modifiable lvalue
```

```
    std::cout << ref << '\n'; // okay: we can access the object through our const reference
    ref = 7;                  // error: we can not modify an object through a const reference

    x = 6;                    // okay: x is a modifiable lvalue, we can still modify it through the original identifier

    return 0;
}
```

Favor lvalue references to const over lvalue references to non-const unless you need to modify the object being referenced.

Pass by reference

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

void printValue(std::string& y) // type changed to std::string&
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    std::string x { "Hello, world!" };

    printValue(x); // x is now passed by reference into reference parameter y (inexpensive)

    return 0;
}
```

Pass by reference can only accept modifiable lvalue arguments

Pass by const lvalue reference

Unlike a reference to non-const (which can only bind to modifiable lvalues), a reference to const can bind to modifiable lvalues, non-modifiable lvalues, and rvalues. Therefore, if we make a reference parameter const, then it will be able to bind to any type of argument:

```
#include <iostream>
```

```
void printRef(const int& y) // y is a const reference
{
    std::cout << y << '\n';
}

int main()
{
    int x { 5 };
    printRef(x);   // ok: x is a modifiable lvalue, y binds to x

    const int z { 5 };
    printRef(z);   // ok: z is a non-modifiable lvalue, y binds to z

    printRef(5);   // ok: 5 is rvalue literal, y binds to temporary int
object

    return 0;
}
```

When to pass by (const) reference

As a rule of thumb, pass fundamental types by value, and class (or struct) types by const reference.

Other common types to pass by value: **enumerated types** and **std::string_view**.
Other common types to pass by (const) reference: **std::string**, **std::array**, and **std::vector.**

Prefer pass by value for objects that are cheap to copy, and pass by const reference for objects that are expensive to copy. If you're not sure whether an object is cheap or expensive to copy, favor pass by const reference.

The last question then is, how do we define "cheap to copy"? There is no absolute answer here, as this varies by compiler, use case, and architecture. However, we can formulate a good rule of thumb: An object is cheap to copy if it uses 2 or fewer "words" of memory (where a "word" is approximated by the size of a memory address) and it has no setup costs.

The following program defines a function-like macro that can be used to determine if a type (or object) is cheap to copy accordingly:

```
#include <iostream>

// Function-like macro that evaluates to true if the type (or object) is
equal to or smaller than
// the size of two memory addresses
```

```cpp
#define isSmall(T) (sizeof(T) <= 2 * sizeof(void*))

struct S
{
    double a;
    double b;
    double c;
};

int main()
{
    std::cout << std::boolalpha; // print true or false rather than 1 or 0
    std::cout << isSmall(int) << '\n'; // true
    std::cout << isSmall(double) << '\n'; // true
    std::cout << isSmall(S) << '\n'; // false

    return 0;
}
```

We use a preprocessor function-like macro here so that we can provide either an object OR a type name as a parameter (normal functions disallow this).

Prefer passing strings using **std::string_view** (by value) instead of const std::string&, unless your function calls other functions that require C-style strings or std::string parameters.

Argument Type	std::string_view parameter	const std::string& parameter
std::string	Inexpensive conversion	Inexpensive reference binding
std::string_view	Inexpensive copy	Requires expensive explicit conversion to `std::string`
C-style string / literal	Inexpensive conversion	Expensive conversion

pointers

There are some other differences between pointers and references worth mentioning:

- References must be initialized, pointers are not required to be initialized (but should be).
- References are not objects, pointers are.
- References can not be reseated (changed to reference something else), pointers can change what they are pointing at.
- References must always be bound to an object, pointers can point to nothing (we'll see an example of this in the next lesson).
- References are "safe" (outside of dangling references), pointers are inherently dangerous (we'll also discuss this in the next lesson).

NULL (defined in the <cstddef> header)

Favor references over pointers unless the additional capabilities provided by pointers are needed.

Pointer and const recap

To summarize, you only need to remember 4 rules, and they are pretty logical:

A non-const pointer can be assigned another address to change what it is pointing at.

- A const pointer always points to the same address, and this address can not be changed.
- A pointer to a non-const value can change the value it is pointing to. These can not point to a const value.
- A pointer to a const value treats the value as const when accessed through the pointer, and thus can not change the value it is pointing to. These can be pointed to const or non-const l-values (but not r-values, which don't have an address).

Keeping the declaration syntax straight can be a bit challenging:

- A **const** before the asterisk is associated with the type being pointed to. Therefore, this is a pointer to a const value, and the value cannot be modified through the pointer.
- A **const** after the asterisk is associated with the pointer itself. Therefore, this pointer cannot be assigned a new address.

```
int main()
{
    int v{ 5 };

    int* ptr0 { &v };             // points to an "int" but is not const itself, so this is a normal pointer.
    const int* ptr1 { &v };       // points to a "const int" but is not const itself, so this is a pointer to a const value.
    int* const ptr2 { &v };       // points to an "int" and is const itself, so this is a const pointer (to a non-const value).
    const int* const ptr3 { &v }; // points to a "const int" and is const itself, so this is a const pointer to a const value.

    // if the const is on the left side of the *, the const belongs to the value
    // if the const is on the right side of the *, the const belongs to the pointer
```

```
    return 0;
}
```

In parameters

In most cases, a function parameter is used only to receive an input from the caller. Parameters that are used only for receiving input from the caller are sometimes called in parameters.

Out parameters

A function argument passed by (non-const) reference (or by address) allows the function to modify the value of an object passed as an argument. This provides a way for a function to return data back to the caller in cases where using a return value is not sufficient for some reason.

A function parameter that is used only for the purpose of returning information back to the caller is called an out parameter.

When to pass by non-const reference

If you're going to pass by reference in order to avoid making a copy of the argument, you should almost always pass by const reference.

However, there are two primary cases where pass by non-const reference may be the better choice.

First, use pass by non-const reference when a parameter is an in-out-parameter. Since we're already passing in the object we need back out, it's often more straightforward and performant to just modify that object.

```
void modifyFoo(Foo& inout)
{
    // modify inout
}

int main()
{
    Foo foo{};
    modifyFoo(foo); // foo modified after this call, slightly more obvious

    return 0;
}
```

The alternative is to pass the object by value or const reference instead (as per usual) and return a new object by value, which the caller can then assign back to the original object:

```
Foo someFcn(const Foo& in)
{
    Foo foo { in }; // copy here
    // modify foo
    return foo;
}

int main()
{
    Foo foo{};
    foo = someFcn(foo); // makes it obvious foo is modified, but another copy made here

    return 0;
}
```

This has the benefit of using a more conventional return syntax, but requires making 2 extra copies (sometimes the compiler can optimize one of these copies away).

Second, use pass by non-const reference when a function would otherwise return an object by value to the caller, but making a copy of that object is extremely expensive. Especially if the function is called many times in a performance-critical section of code.

```
void generateExpensiveFoo(Foo& out)
{
    // modify out
}

int main()
{
    Foo foo{};
    generateExpensiveFoo(foo); // foo modified after this call

    return 0;
}
```