12.6 — 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;
}
```

Passing by const reference offers the same primary benefit as pass by reference (avoiding making a copy of the argument), while also guaranteeing that the function can *not* change the value being referenced.

For example, the following is disallowed, because `ref` is const:

```
void addOne(const int& ref)
{
    ++ref; // not allowed: ref is const
}
```

In most cases, we don’t want our functions modifying the value of arguments.

Best practice

Favor passing by const reference over passing by non-const reference unless you have a specific reason to do otherwise (e.g. the function needs to change the value of an argument).

Now we can understand the motivation for allowing const lvalue references to bind to rvalues: without that capability, there would be no way to pass literals (or other rvalues) to functions that used pass by reference!

Passing values of a different type to a const lvalue reference parameter

In lesson 12.4 -- Lvalue references to const, we noted that a const lvalue reference can bind to an value of a different type, as long as that value is convertable to the type of the reference. The primary motivation for allowing this is so that we can pass a value as an argument to either a value parameter or a const reference parameter in exactly the same way:

```
#include <iostream>

void printVal(double d)
{
    std::cout << d << '\n';
}

void printRef(const double& d)
{
    std::cout << d << '\n';
}

int main()
{
    printVal(5); // 5 converted to temporary double, copied to parameter d
    printRef(5); // 5 converted to temporary double, bound to parameter d

    return 0;
}
```

Mixing pass by value and pass by reference

A function with multiple parameters can determine whether each parameter is passed by value or passed by reference individually.

For example:

```
#include <string>

void foo(int a, int& b, const std::string& c)
{
}

int main()
{
    int x { 5 };
    const std::string s { "Hello, world!" };

    foo(5, x, s);

    return 0;
}
```

In the above example, the first argument is passed by value, the second by reference, and the third by const reference.

When to pass by (const) reference

Because class types can be expensive to copy (sometimes significantly so), class types are usually passed by const reference instead of by value to avoid making an expensive copy of the argument. Fundamental types are cheap to copy, so they are typically passed by value.

Best practice

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`.

The cost of pass by value vs pass by reference Advanced

Not all class types need to be passed by reference. And you may be wondering why we don't just pass everything by reference. In this section (which is optional reading), we discuss the cost of pass by value vs pass by reference, and refine our best practice as to when we should use each.

There are two key points that will help us understand when we should pass by value vs pass by reference:

First, the cost of copying an object is generally proportional to two things:

- The size of the object. Objects that use more memory take more time to copy.

- Any additional setup costs. Some class types do additional setup when they are instantiated (e.g. such as opening a file or database, or allocating a certain amount of dynamic memory to hold an object of a variable size). These setup costs must be paid each time an object is copied.

On the other hand, binding a reference to an object is always fast (about the same speed as copying a fundamental type).

Second, accessing an object through a reference is slightly more expensive than accessing an object through a normal variable identifier. With a variable identifier, the running program can just go to the memory address assigned to that variable and access the value directly. With a reference, there usually is an extra step: the program must first access the reference to determine which object is being referenced, and only then can it go to that memory address for that object and access the value. The compiler can also sometimes optimize code using objects passed by value more highly than code using objects passed by reference. This means code generated to access objects passed by reference is typically slower than the code generated for objects passed by value.

We can now answer the question of why we don't pass everything by reference:

- For objects that are cheap to copy, the cost of copying is similar to the cost of binding, so we favor pass by value so the code generated will be faster.
- For objects that are expensive to copy, the cost of the copy dominates, so we favor pass by (const) reference to avoid making a copy.

Best practice

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
#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;
}
```

As an aside…

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).

However, it can be hard to know whether a class type object has setup costs or not. It's best to assume that most standard library classes have setup costs, unless you know otherwise that they don't.

Tip

An object of type T is cheap to copy if `sizeof(T) <= 2 * sizeof(void*)` and has no additional setup costs.

For function parameters, prefer `std::string_view` over `const std::string&` in most cases

One question that comes up often in modern C++: when writing a function that has a string parameter, should the type of the parameter be `const std::string&` or `std::string_view`?

In most cases, `std::string_view` is the better choice, as it can handle a wider range of argument types efficiently.

```
void doSomething(const std::string&);
void doSomething(std::string_view);   // prefer this in most cases
```

There are a few cases where using a `const std::string&` parameter may be more appropriate:

- If you're using C++14 or older, `std::string_view` isn't available.
- If your function needs to call some other function that takes a C-style string or `std::string` parameter, then `const std::string&` may be a better choice, as `std::string_view` is not guaranteed to be null-terminated (something that C-style string functions expect) and does not efficiently convert back to a `std::string`.

Best practice

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.

Why `std::string_view` parameters are more efficient than `const std::string&` Advanced

In C++, a string argument will typically be a `std::string`, a `std::string_view`, or a C-style string/string literal.

As reminders:

- If the type of an argument does not match the type of the corresponding parameter, the compiler will try to implicitly convert the argument to match the type of the parameter.
- Converting a value creates a temporary object of the converted type.
- Creating (or copying) a `std::string_view` is inexpensive, as `std::string_view` does not make a copy of the string it is viewing.
- Creating (or copying) a `std::string` can be expensive, as each `std::string` object makes a copy of the string.

Here's a table showing what happens when we try to pass each type:

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

With a `std::string_view` value parameter:

- If we pass in a `std::string` argument, the compiler will convert the `std::string` to a `std::string_view`, which is inexpensive, so this is fine.
- If we pass in a `std::string_view` argument, the compiler will copy the argument into the parameter, which is inexpensive, so this is fine.
- If we pass in a C-style string or string literal, the compiler will convert these to a `std::string_view`, which is inexpensive, so this is fine.

As you can see, `std::string_view` handles all three cases inexpensively.

With a `const std::string&` reference parameter:

- If we pass in a `std::string` argument, the parameter will reference bind to the argument, which is inexpensive, so this is fine.
- If we pass in a `std::string_view` argument, the compiler will refuse to do an implicit conversion, and produce a compilation error. We can use `static_cast` to do an explicit conversion (to `std::string`), but this conversion is expensive (since `std::string` will make a copy of the string being viewed). Once the conversion is done, the parameter will reference bind to the result, which is inexpensive. But we've made an expensive copy to do the conversion, so this isn't great.
- If we pass in a C-style string or string literal, the compiler will implicitly convert this to a `std::string`, which is expensive. So this isn't great either.

Thus, a `const std::string&` parameter only handles `std::string` arguments inexpensively.

The same, in code form:

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

void printSV(std::string_view sv)
{
    std::cout << sv << '\n';
}

void printS(const std::string& s)
{
    std::cout << s << '\n';
}

int main()
{
    std::string s{ "Hello, world" };
    std::string_view sv { s };

    // Pass to `std::string_view` parameter
    printSV(s);              // ok: inexpensive conversion from std::string to
std::string_view
    printSV(sv);             // ok: inexpensive copy of std::string_view
    printSV("Hello, world"); // ok: inexpensive conversion of C-style string literal
to std::string_view

    // pass to `const std::string&` parameter
    printS(s);              // ok: inexpensive bind to std::string argument
    printS(sv);             // compile error: cannot implicit convert
std::string_view to std::string
    printS(static_cast<std::string>(sv)); // bad: expensive creation of std::string
temporary
    printS("Hello, world"); // bad: expensive creation of std::string temporary

    return 0;
}
```

Additionally, we need to consider the cost of accessing the parameter inside the function. Because a `std::string_view` parameter is a normal object, the string being viewed can be accessed directly. Accessing a `std::string&` parameter requires an additional step to get to the referenced object before the string can be accessed.