Chapter 2: Qualifiers, References, and L/R Values

This chapter covers the tools C++ provides to add constraints and meaning to your variables (`const`, `constexpr`), a safer alternative to pointers (`&`), and the formal system of value categories that is essential for understanding modern C++ features like move semantics.

2.1 The `const` and `constexpr` Qualifiers

`const` Correctness

Why use `const`? The `const` keyword is a tool for expressing intent. It establishes a contract that a piece of data will not be modified. This has three major benefits: 1. **Readability and Safety:** When you see `const`, you know you are only reading data, not changing it. This makes code easier to reason about and prevents accidental modification of variables that shouldn't be changed. 2. **Compiler Enforcement:** The compiler guarantees that the contract is upheld. If you try to modify a `const` object, you get a compile-time error, catching bugs early. 3. **Optimization:** By guaranteeing that a value is immutable, `const` allows the compiler to perform more aggressive optimizations, such as placing `const` data in read-only memory, which is faster and safer.

What can be `const`? Anything: variables, pointers, references, and member functions.

How to read `const` declarations: A good rule of thumb is to read the declaration from right to left.

```
int x = 10;
int y = 20;

const int* p1 = &x;       // "p1 is a pointer to an integer that is constant."
                          // The data pointed to cannot be changed via p1.
                          // *p1 = 5; // ERROR!
                          // p1 = &y; // OK, the pointer itself can be changed.

int* const p2 = &x;       // "p2 is a constant pointer to an integer."
                          // The pointer itself cannot be changed, it must always point to x.
                          // *p2 = 5; // OK, the data can be changed.
                          // p2 = &y; // ERROR!

const int* const p3 = &x; // "p3 is a constant pointer to an integer that is constant."
                          // Neither the pointer nor the data it points to can be changed.
```

`const` Member Functions: A member function declared `const` promises not to change the state of the object it's called on. This allows it to be called on `const` objects.

```
class MyClass {
    int value_;
public:
    void set_value(int v) { value_ = v; } // non-const: modifies state
    int get_value() const { return value_; } // const: does not modify state
};

MyClass obj1;
obj1.set_value(10);

const MyClass obj2 = obj1;
// obj2.set_value(20); // ERROR! Cannot call non-const function on a const object.
int val = obj2.get_value(); // OK, get_value() is const.
```

`constexpr` for Compile-Time Evaluation

Why `constexpr`? A core tenet of modern C++ is to shift work from runtime to compile time whenever possible. This makes programs faster (the work is already done before the program starts) and safer (errors can be caught during compilation).

What is the difference between `const` and `constexpr`? * `const`: "I promise not to change this value." The value might be known at compile time, or it might be calculated at runtime. * `constexpr`: "I am a true compile-time constant." This is a stronger guarantee. A `constexpr` variable *must* be initialized with a value the compiler can determine at compile time. All `constexpr` variables are implicitly `const`.

`constexpr` Functions: A function marked `constexpr` can be run at compile time *if* it is called with compile-time constant arguments. If not, it will simply run at runtime like a normal function. This allows you to write one function for both scenarios.

```
// A constexpr function to calculate factorial recursively
constexpr unsigned long long factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

// Usage:
// 1. At compile time
int my_array[factorial(5)]; // The size of the array is calculated during compilation!
static_assert(factorial(5) == 120, "Factorial calculation failed");

// 2. At runtime
int x = 10;
unsigned long long result = factorial(x); // Executed at runtime like a normal function.
```

2.2 References vs. Pointers

Why were references created? Pointers are powerful but notoriously unsafe. They can be null, they can point to invalid memory, and their syntax can be clumsy. References were introduced as a safer, more convenient alternative for creating an alias to an existing object.

Feature	Pointer (`*`)	Reference (`&`)
Initialization	Can be uninitialized (dangerous).	**Must** be initialized upon creation.
Nullability	Can be `nullptr`.	**Cannot** be null. Represents a valid, existing object.
Reseating	Can be changed to point to another object.	**Cannot** be reseated. It is bound to its initial object for life.
Syntax	Requires explicit dereferencing (`*` or `->`).	Used directly, just like the original object.
Use Case	For dynamic memory, optional objects (can be null), and C-style APIs.	For function parameters (pass-by-reference) and creating simple aliases.

How to use them: The classic example is a `swap` function.

```
// Pointer version: clumsy and requires checks
void swap_ptr(int* a, int* b) {
    if (a == nullptr || b == nullptr) return;
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Reference version: clean, safe, and efficient
void swap_ref(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int x = 5, y = 10;
swap_ptr(&x, &y); // Must pass addresses, syntax is different
swap_ref(x, y);   // Looks like a normal function call, but modifies the originals
```

2.3 Lvalues, Rvalues, and Expression Categories

Why does this exist? This formal classification system is the foundation for **move semantics** (Chapter 13). By categorizing expressions, the compiler can identify temporary values that are about to be destroyed. This allows it to apply powerful

optimizations, like “stealing” the internal data from a temporary instead of performing an expensive copy.

What are the simple definitions? * **lvalue (locator value):** An expression that refers to a persistent memory location. It has a name. You can take its address. Think of variables. * **rvalue (read value):** An expression that is a temporary result and does not have a persistent identity. It often has no name. You cannot take its address.

```
int x = 10; // `x` is an lvalue. `10` is an rvalue.

int y = x;  // `x` is an lvalue expression.

int z = x + y; // The expression `(x + y)` is an rvalue.

&x; // OK, can take address of an lvalue.
// &(x + y); // ERROR! Cannot take address of an rvalue.
```

The Full Picture: The modern C++ standard has a more detailed breakdown: * **`lvalue`**: A traditional lvalue. * **`prvalue` (pure rvalue):** A “true” temporary, like `42` or the object returned by `x + y`. * **`xvalue` (expiring value):** An object that is near the end of its lifetime, which we can treat like an rvalue. The result of `std::move(x)` is an xvalue. * **`glvalue` (generalized lvalue):** An lvalue or an xvalue. * **`rvalue`**: An xvalue or a prvalue.

You don’t need to memorize all these for now, but knowing that `lvalue` and `rvalue` are the two most important categories is crucial for what comes next.

Projects for Chapter 2

Project 1: The `const`-Correct Inspector

- **Problem Statement:** Create a `Registry` class that holds a `std::string`. It should have two “getter” methods: `get_name_copy()` which returns a copy of the string, and `get_name_ref()` which returns a `const` reference to the string. Both of these methods should be `const`. It should also have a non-`const` method `set_name()`. In `main`, create a `const` and a non-`const` instance of `Registry`. Demonstrate which functions can be called on each object and discuss in comments why.
- **Core Concepts to Apply:** `const` objects, `const` member functions, returning by value vs. by `const` reference.
- **Hint:** A `const` object can only call `const` member functions.

Project 2: Compile-Time String Length

- **Problem Statement:** Write a `constexpr` function `str_length(const char* str)` that can calculate the length of a C-style string literal at compile time. Use this function with `static_assert` to verify its correctness for a few different string literals.
- **Core Concepts to Apply:** `constexpr` functions, compile-time evaluation, C-style strings.
- **Hint:** This will likely need to be a recursive function. The base case is when the first character is the null terminator (`'\0'`). `constexpr int len(const char* s) { return *s == '\0' ? 0 : 1 + len(s + 1); }`

Project 3: The Safe Swapper

- **Problem Statement:** Write two functions, `swap_by_ptr(int* a, int* b)` and `swap_by_ref(int& a, int& b)`, that swap the values of two integers. In your `main` function, demonstrate how to call both. Then, add code that tries to call `swap_by_ptr` with a `nullptr`. Add comments explaining why the reference version is generally safer (it forces the caller to provide a valid object) and easier to read.
- **Core Concepts to Apply:** Pointers vs. References, pass-by-pointer, pass-by-reference, null pointer safety.