

Ch6: References and Pointers

Lvalue References

Lvalues and Rvalues

Lvalues (left values) expressions are those that evaluate to variables or other identifiable objects that persist beyond the end of execution of the expression. They can be used on an assignment operator's left or right side and come in two subtypes:

- **modifiable lvalue** whose value can be modified,
- **non-modifiable lvalue** whose value cannot be modified.

Rvalues (right values) expressions are those that evaluate to literals or the returned value of functions and operators that are discarded at the end of execution of the expression. They can be used on only an assignment operator's right side, but not vice versa.

Note: An assignment operation requires the left operand of the assignment to be a modifiable lvalue expression, and the right operand to be an rvalue expression, thus, $x = 5$ is valid but $5 = x$ is not.

Lvalues and Rvalues


```
#include <iostream>

int return5() {
    return 5;
}

int main() {
    int x{ 5 }; // 5 is an rvalue expression
    const double d{ 1.2 }; // 1.2 is an rvalue expression

    int y{ x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
    int z{ return5() }; // return5() is an rvalue expression (since the result is returned by value)

    int w{ x + 1 }; // x + 1 is an rvalue expression
    int q{ static_cast<int>(d) }; // the result of static casting d to an int is an rvalue expression
}
```



Lvalue Reference (or Reference)

A **Lvalue Reference** (commonly called a **Reference**) is an alias for an existing object (or variable). Once a reference has been defined, any operation on the reference is applied to the object (or variable) being referenced.

- To declare an lvalue reference type, an ampersand (&) is used in the type declaration.

```
int    // a normal int type
int&   // an lvalue reference to an int object
double& // an lvalue reference to a double object
```

- To create an lvalue reference variable, we simply define a variable with an lvalue reference type to read and modify the value of the object being referenced.

```
#include <iostream>
int main() {
    int x{5}; // x is a normal integer variable
    int& ref{x}; // or "int &ref{x};". ref is an lvalue reference variable that can now be used as an alias for variable x
    std::cout << x << ", " << ref << "\n"; // prints 5, 5 the value of x and the value of x via ref
    x = 6; // x now has value 6
    std::cout << x << ", " << ref << "\n"; // prints 6, 6 the value of x and the value of x via ref
    ref = 7; // the object being referenced (x) now has value 7
    std::cout << x << ", " << ref << "\n"; // prints 7, 7 the value of x and the value of x via ref
}
```

Note: The value of x can be changed through either x or ref.



Lvalue Reference (or Reference)

- Lvalue references must be **bound** to (or initialized with) a **modifiable lvalue**.
- Lvalue references cannot be bound to **non-modifiable lvalues** or **rvalues** (because it would allow us to modify a const variable through the non-const reference).
- The **type** of the reference must match the type of the referent (modifiable lvalue).
- Once initialized, a reference cannot be reassigned as aliases to other variables.
- Reference variables follow the same scoping and duration rules that normal variables do.

```
int main() {  
  
    int x{ 5 };  
    int& ref{ x }; // valid: lvalue reference bound to a modifiable lvalue  
  
    const int y{ 5 };  
    int& ref1{ y }; // invalid: can't bind to a non-modifiable lvalue  
    int& ref2{ 0 }; // invalid: can't bind to an r-value  
  
    double z{ 6.0 };  
    int& ref3{ z }; // invalid: reference to int cannot bind to double variable  
}
```



Lvalue Reference to const

- By using the `const` keyword when declaring an lvalue reference, we can create a reference to **non-modifiable lvalues** to access but not modify them.
- Lvalue reference to `const` can also bind to a **modifiable lvalue**. In such a case, we can use the reference to access the lvalue, but because reference is `const`, we can not modify the value of the lvalue through the reference. However, we still can modify the value of the lvalue directly.

```
#include <iostream>
int main() {
    const int x { 5 }; // x is a non-modifiable lvalue
    const int& ref1 { x }; // okay: ref1 is a an lvalue reference to a const value

    std::cout << ref1 << "\n"; // okay: we can access the const object
    ref1 = 6; // error: we can not modify a const object

    int y { 5 }; // y is a modifiable lvalue
    const int& ref2 { y }; // okay: we can bind a const reference to a modifiable lvalue

    std::cout << ref2 << "\n"; // okay: we can access the object through our const reference
    ref2 = 7; // error: we can not modify an object through a const reference
    y = 6; // okay: y is a modifiable lvalue, we can still modify it through the original identifier
}
```

Best Practice: Favor lvalue references to `const` unless you need to modify the object being referenced through the reference.

Lvalue Reference to const

We can initialize an lvalue reference to const with an rvalue. When this happens, a temporary object is created and initialized with the rvalue, and the reference to const is bound to that temporary object. The lifetime of the temporary object is extended to match the lifetime of the reference.

```
#include <iostream>
int main() {
    const int& ref{ 5 }; // A temporary object holding value 5 is created
                        // and ref is bound to it (5 is an rvalue)

    std::cout << ref << "\n"; // Prints 5
} // Both ref and the temporary object die here
```

Summary:

- Lvalue references can only bind to modifiable lvalues.
- Lvalue references to const can bind to modifiable lvalues, non-modifiable lvalues, and rvalues. This makes them a much more flexible type of reference.

Passing Arguments by Reference

Passing Arguments to Functions

Two ways to pass arguments to functions are **pass-by-value** and **pass-by-reference**.

- With **pass-by-value**:
 - A **copy** of the argument's value is made and passed to the called function.
 - Changes to the copy in the called function do not affect the original variable's value in the caller.
 - **Disadvantage** is that if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.
- With **pass-by-reference**:
 - The caller gives the called function the ability to **access the caller's data directly**.
 - It is **good for performance** reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Pass-by-Value

```
#include <iostream>
#include <string>
void printValue(int y) {
    std::cout << y << '\n';
} // y is destroyed here
void printValue(std::string y) {
    std::cout << y << '\n';
} // y is destroyed here
int main() {
    int x{2};
    printValue(x); // x is passed by value (copied) into parameter y (inexpensive)
    std::string s{"Hello, world!"}; // s is a std::string
    printValue(s); // s is passed by value (copied) into parameter y (expensive)
}
```

- **Fundamental types** are **cheap** to copy, however, most of the types provided by the standard library (such as `std::string`) are **class types** that are usually **expensive** to copy.
- One way to avoid making an expensive copy of an argument when calling a function and make the program more efficient is to use **pass-by-reference** instead of pass-by-value.

Pass-by-Reference

(reference to non-const)

When using pass by reference, we declare a function parameter as a reference type. When the function is called, each reference parameter is bound to the appropriate argument in the caller. Because the reference acts as an alias for the argument, no copy of the argument is made.

When using pass by reference to non-const,

- any changes made to the reference parameter in the function will affect the original value of the argument.
- only modifiable lvalue arguments are acceptable.

```
#include <iostream>
#include <string>
void printValue(std::string& y) { // y is bound to a modifiable lvalue (s)
    std::cout << y << '\n';
} // y is destroyed here
void printValue(int& y) { // y is bound to a modifiable lvalue
    std::cout << y++ << '\n'; // this modifies the actual object x
} // y is destroyed here
int main() {
    std::string s{"Hello, world!"}; // s is a modifiable lvalue
    printValue(s); // s is passed by reference
    int x{ 5 }; // x is a modifiable lvalue
    printValue(x); // x is passed by reference
    std::cout << x << '\n'; // x has been modified
    const int z{ 5 }; // z is a non-modifiable lvalue
    printValue(z); // error: z is a non-modifiable lvalue
    printValue(5); // error: 5 is an rvalue
}
```



Pass-by-Reference

(reference to const)

A reference to const can (1) bind to modifiable lvalues, non-modifiable lvalues, and rvalues (i.e., any type of argument) (2) guarantee that the function can not change the value being referenced (in most cases, we don't want our functions modifying the value of arguments).


```
#include <iostream>

void printValue(const int& y) { // y is a const reference
    std::cout << y << '\n';
    // ++y; // not allowed: y is const
}

int main() {
    int x{ 5 };
    printValue(x); // ok: x is a modifiable lvalue

    const int z{ 5 };
    printValue(z); // ok: z is a non-modifiable lvalue

    printValue(5); // ok: 5 is a literal rvalue
}
```



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.

As always, the function prototype and header must agree.

```
#include <iostream>
#include <string>
void printValue(int a, int& b, const std::string& c);
int main() {
    int x{ 5 };
    const std::string s{ "Hello, world!" };
    printValue(15, x, s);
}
void printValue(int a, int& b, const std::string& c) {
    std::cout << a << ", " << b << ", " << c << "\n";
}
```

Best Practices:

- 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).
- Prefer pass-by-value for objects that are cheap to copy (e.g., fundamental types, enumerated types), and pass-by-const-reference for objects that are expensive to copy (e.g., class types including `std::string`, `std::array`, `std::vector`). If you're not sure whether an object is cheap or expensive to copy, favor pass-by-const-reference.

Pointers

Address-of Operator (&)

When the code is executed, a piece of memory from RAM will be assigned to variable `x`. The address-of operator (&) returns the memory address of its operand (Memory addresses are typically printed as hexadecimal values). For objects that use more than one byte of memory, address-of will return the memory address of the first byte used by the object.

```
#include <iostream>
int main() {
    int x{ 5 };
    int& ref { x }; // ref is an lvalue reference to x
    std::cout << x << '\n';
    std::cout << ref << '\n';
    std::cout << &x << '\n'; // print the memory address of variable x
}
```

The & symbol has different meanings depending on context:

- When following a type name, & denotes an lvalue reference: `int& ref`.
- When used in a unary context in an expression, & is the address-of operator: `std::cout << &x`.
- When used in a binary context in an expression, & is the Bitwise AND operator: `std::cout << x & y`.

Dereference Operator (*)

The dereference operator (*) returns the value at a given memory address as an lvalue:

```
#include <iostream>
int main() {
    int x{ 5 };
    int& ref { x }; // ref is an lvalue reference to x (when used with a type, & means lvalue reference)
    std::cout << x << '\n';
    std::cout << ref << '\n';
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *(&x) << '\n'; // print the value at the memory address of variable x
                                // (parentheses not required, but make it easier to read)
}
```

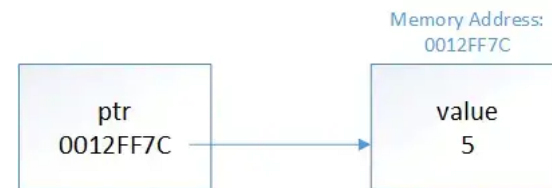
The address-of operator (&) and dereference operator (*) work as opposites: address-of gets the address of an object and dereference gets the object at an address.

Pointers & Initialization

A pointer is an object that holds a memory address (typically of another variable) as its value.

```
#include <iostream>
int main() {
    int x { 5 }; // normal variable
    int& ref { x }; // an lvalue reference to an int value (bound to x)
    int* ptr; // an uninitialized pointer to an int value (holds a garbage address)
    int* ptr2{}; // a null pointer
    int* ptr3{ &x }; // a pointer initialized with the address of variable x
    std::cout << *ptr3 << '\n'; // use dereference operator to access the value at the address that ptr3 is holding}
}
```

- Note that this asterisk is part of the declaration syntax for pointers, not a use of the dereference operator. When declaring a pointer type, place the asterisk next to the type name.



- A pointer that has not been initialized contains a garbage address and dereferencing it will result in undefined behavior. Because of this, you should always initialize your pointers to a known value.

Pointer Initialization

Much like the type of a reference has to match the type of object being referred to, the type of the pointer has to match the type of the object being pointed to:

```
int main() {  
    int i{ 5 };  
    double d{ 7.0 };  
    int* iPtr{ &i }; // ok: a pointer to an int can point to an int object  
    int* iPtr2 { &d }; // not okay: a pointer to an int can't point to a double object  
    double* dPtr{ &d }; // ok: a pointer to a double can point to a double object  
    double* dPtr2{ &i }; // not okay: a pointer to a double can't point to an int object  
}
```

Initializing a pointer with a literal value is disallowed:

```
int* ptr{ 5 }; // not okay  
int* ptr{ 0x0012FF7C }; // not okay, 0x0012FF7C is treated as an integer literal
```

Pointers & Lvalue References

```
#include <iostream>
int main() {
    int x{ 5 };
    int& ref { x };
    int* ptr { &x }; // ptr initialized with address of variable x
    std::cout << x; // print x's value (5)
    std::cout << ref; // use the reference to print x's value (5)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (5)

    ref = 6; // use the reference to change the value of x
    std::cout << x;
    std::cout << ref; // use the reference to print x's value (6)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (6)

    *ptr = 7; // use the pointer to change the value of x (ptr is dereferenced here)
    std::cout << x;
    std::cout << ref; // use the reference to print x's value (7)
    std::cout << *ptr << '\n'; // use the pointer to print x's value (7)

    int y{ 8 };
    // int& ref { y }; // not allowed!
    ptr = &y; // // change ptr to point at y
    std::cout << "x = " << x << '\n'; // print x's value (7)
    std::cout << "y = " << y << '\n'; // print y's value (8)
    std::cout << "ref = " << ref << '\n'; // (7)
    std::cout << "*ptr = " << *ptr << '\n'; // (8)
}
```

Pointers and References both provide a way to indirectly access another object. However, with pointers, we need to explicitly get the address to point at, and we have to explicitly dereference the pointer to get the value. With references, the address-of and dereference happens implicitly.

We can use assignment with pointers in two different ways:

- To change what the pointer is pointing at (by assigning the pointer a new address),
- To change the value being pointed at (by assigning the dereferenced pointer a new value).

Pointers & Lvalue 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.
- References are “safe” (outside of dangling references), pointers are inherently dangerous.

```
#include <iostream>
int main() {
    int x{ 5 };
    int* ptr{ &x };
    std::cout << *ptr << '\n'; // valid
    {
        int y{ 6 };
        ptr = &y;
        std::cout << *ptr << '\n'; // valid
    } // y goes out of scope, and ptr is now dangling
    // undefined behavior from dereferencing a dangling pointer
    std::cout << *ptr << '\n';
}
```

Much like a dangling reference, a **dangling pointer** is a pointer that is holding the address of an object that is no longer valid (e.g., because it has been destroyed). Dereferencing a dangling pointer will lead to undefined behavior, as you are trying to access an object that is no longer valid.

Null Pointers

Besides a memory address, there is one additional value that a pointer can hold: a null value. A **null value** (often shortened to **null**) is a special value that means something has no value. When a pointer is holding a null value, it means the pointer is not pointing at anything. Such a pointer is called a **null pointer**.

```
#include <iostream>
int main() {
    int* ptr {}; // ptr is a null pointer, and is not holding an address
    int x { 5 };
    ptr = &x; // ptr now pointing at object x (no longer a null pointer)

    int* ptr2 { nullptr }; // using nullptr to initialize a pointer to be a null pointer

    int value { 5 };
    int* ptr3 { &value }; // ptr3 is a valid pointer
    ptr3 = nullptr; // Can assign nullptr to make the pointer a null pointer

    // std::cout << *ptr3 << '\n'; // Dereference the null pointer
}
```

- The **nullptr** keyword represents a null pointer literal. We can use **nullptr** to explicitly initialize or assign a pointer a null value.
- Dereferencing a null pointer results in undefined behavior

Checking for Null Pointers

We can avoid dereferencing a null pointer by using a conditional to ensure a pointer is non-null before trying to dereference it.

```
#include <iostream>
int main() {
    int x { 5 };
    int* ptr { &x };

    // Assume ptr is some pointer that may or may not be a null pointer
    if (ptr == nullptr) // explicit test for equivalence
        std::cout << *ptr << '\n'; // okay to dereference
    else
        ;// do something else that doesn't involve dereferencing ptr (print an error message, etc...)

    // or
    if (ptr) // implicit conversion to Boolean (if ptr is not a null pointer)
        std::cout << *ptr << '\n'; // okay to dereference
    else
        ;// do something else that doesn't involve dereferencing ptr (print an error message, etc...)

    int* ptr2 {};
    std::cout << "ptr2 is " << (ptr2 == nullptr ? "null\n" : "non-null\n"); // explicit test for equivalence
    std::cout << "ptr2 is " << (ptr2 ? "non-null\n" : "null\n"); // implicit conversion to Boolean
}
```

Pointers can implicitly convert to Boolean values: a **null pointer** converts to Boolean value **false**, and a **non-null pointer** converts to Boolean value **true**.

Favor References Over Pointers

Conditionals can only be used to differentiate null pointers from non-null pointers. There is no convenient way to determine whether a non-null pointer is pointing to a valid object, or it is dangling (pointing to an invalid object). When an object is destroyed, any pointers to the destroyed object will be left dangling. It is the programmer's responsibility to detect these cases and ensure those pointers are subsequently set to `nullptr`.

Pointers have the additional abilities of being able to change what they are pointing at, and to be pointed at null. However, these pointer abilities are also inherently dangerous: A null pointer runs the risk of being dereferenced, and the ability to change what a pointer is pointing at can make creating dangling pointers easier.

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

Pointer to const value

A **pointer to a const value** is a (non-const) pointer that points to a constant value. To declare a pointer to a const value, use the const keyword before the pointer's data type.

```
int main() {  
    const int x{ 5 }; // const  
    // int* ptr { &x }; // compile error: cannot convert from const int* to int*  
  
    const int* ptr { &x }; // okay: ptr is pointing to a "const int"  
  
    // *ptr = 6; // not allowed: we can't change a const value  
  
    const int y{ 6 };  
    ptr = &y; // okay: ptr now points at const int y  
  
    int z{ 5 }; // non-const  
    const int* ptr2 { &z }; // ptr points to a "const int"  
    // *ptr2 = 6; // not allowed: ptr points to a "const int" so we can't change the value through ptr  
    z = 6; // allowed: the value is still non-const when accessed through non-const identifier x  
}
```

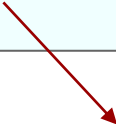
Because a pointer to const is not const itself (it just points to a const value), we can change what the pointer is pointing at by assigning the pointer a new address.

A pointer to const can also point to non-const variables. A pointer to const treats the value being pointed to as constant, regardless of whether the object at that address was initially defined as const or not.

const pointers

We can also make a pointer itself constant. A const pointer is a pointer whose address can not be changed after initialization. To declare a const pointer, use the const keyword after the asterisk in the pointer declaration.

```
int main() {  
    int x{ 5 };  
    int y{ 6 };  
  
    int* const ptr { &x }; // okay: the const pointer is initialized to the address of x  
                          // ptr will always point to x  
    // ptr = &y; // error: once initialized, a const pointer can not be changed.  
  
    *ptr = 6; // okay: the value being pointed to is non-const  
}
```



Because the value being pointed to is non-const, it is possible to change the value being pointed to via dereferencing the const pointer

const pointer to a const value

It is possible to declare a const pointer to a const value by using the const keyword both before the type and after the asterisk. A const pointer to a const value can not have its address changed, nor can the value it is pointing to be changed through the pointer. It can only be dereferenced to get the value it is pointing at.

```
int main() {
    int value { 5 };
    const int* const ptr { &value }; // a const pointer to a const value
}
```

Recap:

```
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
}
```