Contents

[Introduction 1](#_Toc57061705)

[References 1](#_Toc57061706)

[Pass-By-Reference 2](#_Toc57061707)

[Pass-By-Reference with Const 3](#_Toc57061708)

[Memory Address 4](#_Toc57061709)

[Pointers 5](#_Toc57061710)

[Dereference 6](#_Toc57061711)

[Null Pointer 6](#_Toc57061712)

[Review 6](#_Toc57061713)

## Introduction

A computer’s *memory* is a sequence of [bytes](https://en.wikipedia.org/wiki/Byte). We can number the bytes from 0 to the last one. Each number, known as an *address*, represents a location in the memory.

Everything we put into memory has an address. For example, when we declare and initialize an int variable named power:

int power = 9000;

This will set aside an int-size piece of memory for the variable power somewhere and put the value 9000 into that memory. But where is “somewhere”? How is it useful?

In this lesson, we will answers these questions by learning about:

* References
* Pointers

These are some of the most powerful features in C++; they allow programmers to directly manipulate memory – the most critical and scarce resource in computer – in order to optimize performance.

However, references and pointers are also sometimes considered two of the most complex and difficult features in C++.

So let’s get started.

## References

In C++, a *reference* variable is an alias for something else, that is, another name for an already existing variable.

So suppose we make Sonny a reference to someone named Songqiao. You can refer to the person as either Sonny or Songqiao.

Let’s take a look at how we can do this with code. Suppose we have an int variable already called songqiao, we can create an alias to it by using the & sign in the declaration:

int &sonny = songqiao;

So here, we made sonny a reference to songqiao.

Now when we make changes to sonny (add 1, subtract 2, etc), songqiao also changes.

Two things to note about references:

1. Anything we do to the reference also happens to the original.
2. Aliases cannot be changed to alias something else.

## Pass-By-Reference

So what’s a good use case for references? Let’s take a look.

Previously, when we passed parameters to a function, we used normal variables and that’s known as *pass-by-value*. But because the variables passed into the function are out of scope, we can’t actually modify the value of the arguments.

*Pass-by-reference* refers to passing parameters to a function by using references. When called, the function can modify the value of the arguments by using the reference passed in.

This allows us to:

* Modify the value of the function arguments.
* Avoid making copies of a variable/object for performance reasons.

The following code shows an example of pass-by-reference. The reference parameters are initialized with the actual arguments when the function is called:

void swap\_num(int &i, int &j) {  
   
  int temp = i;  
  i = j;  
  j = temp;  
   
}  
   
int main() {  
   
  int a = 100;  
  int b = 200;  
   
  swap\_num(a, b);  
   
  std::cout << "A is " << a << "\n";  
  std::cout << "B is " << b << "\n";  
   
}

Notice that the int &i and int &j are the parameters of the function swap\_num().

When swap\_num() is called, the values of the variables a and b will be modified because they are passed by reference. The output will be:

A is 200  
B is 100

Suppose we didn’t pass-by-reference here and have the parameters as simply int i and int j in the swap\_num() function, then i and j would swap, but a and b wouldn’t be modified.

And the output will then be:

A is 100  
B is 200

To reiterate, using references as parameters allows us to modify the arguments’ values. This can be very useful in a lot cases.

## Pass-By-Reference with Const

Remember const? The const keywords tells the compiler that we won’t change something.

For example, in the following code, we are telling the compiler that the double variable pi will stay at 3.14 through out the program:

double const pi = 3.14;

If we try to change pi, the compiler will throw an error.

Sometimes, we use const in a function parameter; this is when we know for a fact that we want to write a function where the parameter won’t change inside the function. Here’s an example:

int triple(int const i) {  
   
  return i \* 3;  
   
}

In this example, we are not modifiying the i. If inside the function triple(), the value of i is changed, there will be a compiler error.

So to save the computational cost for a function that doesn’t modify the parameter value(s), we can actually go a step further and use a const reference:

int triple(int const &i) {  
   
  return i \* 3;  
   
}

This will ensure the same thing: the parameter won’t be changed. However, by making i a reference to the argument, this saves the computational cost of making a copy of the argument.

## Memory Address

So we haved learned about references (aliases), which are created by using the & symbol in a variable declaration. But the & sign can have another meaning.

The “address of” operator, &, is used to get the *memory address*, the location in the memory, of an object.

Suppose we declare a variable called:

int porcupine\_count = 3;

Have you wondered where the variable porcupine\_count is stored on the computer? We can find out by printing out &porcupine\_count:

std::cout << &porcupine\_count << "\n";

It will return something like:

0x7ffd7caa5b54

This is a memory address represented in [hexadecimal](https://en.wikipedia.org/wiki/Hexadecimal). A memory address is usually denoted in hexadecimal instead of binary for readability and conciseness.

The double meaning of the & symbol can be tricky at first, so make sure to note:

* When & is used in a declaration, it is a reference operator.
* When & is not used in a declaration, it is an address operator.

## Pointers

In C++, a *pointer* variable is mostly the same as other variables, which can store a piece of data. Unlike normal variables, which store a value (such as an int, double, char), a pointer stores a memory address.

While references are a newer mechanism that originated in C++, pointers are an older mechanism that was inherited from C. We recommend avoiding pointers as much as possible; usually, a reference will do the trick. However, you will see pointers a lot in the wild, particularly in older projects, where they are used in a very similar way to references.

Pointers must be declared before they can be used, just like a normal variable. They are syntactically distinguished by the \*, so that int\* means “pointer to int“ and double\* means “pointer to double“.

int\* number;  
double\* decimal;  
char\* character;

So suppose we have a variable called gum:

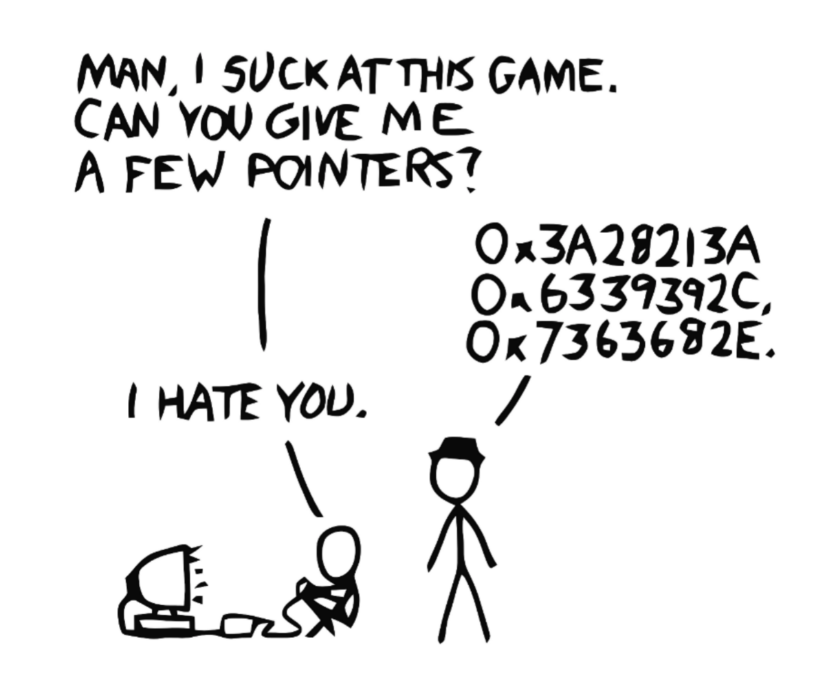
int gum = 8;

We can create a pointer to it by:

int\* ptr = &gum;

* int\* makes it a pointer rather than a normal variable.
* ptr is the pointer name.
* &gum is the memory address of the other variable gum.

So now ptr has a value of gum‘s memory address.



**Note:** Syntactically, spaces around \* do not matter, but the best practice is to have it after the data type.

int\* number;  
int \*number;  
int \* number;

## Dereference

So now we learned what a pointer is and how to create one, but is there a way to obtain the value pointed to by the pointer?

The asterisk sign \* a.k.a. the dereference operator is used to obtain the value pointed to by a variable. This can be done by preceding the name of a pointer variable with \*.

int blah = \*ptr;

The double meaning of the \* symbol can be tricky at first, so make sure to note:

* When \* is used in a declaration, it is creating a pointer.
* When \* is not used in a declaration, it is a dereference operator.

## Null Pointer

When we declare a pointer variable like so, its content is not intialized:

int\* ptr;

In other words, it contains an address of “somewhere”, which is of course not a valid location. This is [dangerous](https://en.wikipedia.org/wiki/Uninitialized_variable)! We need to initialize a pointer by assigning it a valid address.

But suppose we don’t know where we are pointing to, we can use a null pointer.

nullptr is a new keyword introduced in C++11. It provides a typesafe pointer value representing an empty pointer.

We can use nullptr like so:

int\* ptr = nullptr;

**Note:** In older C/C++ code, NULL was used for this purpose. nullptr is meant as a modern replacement to NULL.

## Review

In this lesson, you have learned:

* References
  + Pass-by-reference
  + Pass-by-reference with const
* Memory addresses and how to access them
* Pointers
  + Dereferencing a pointer
  + nullptr

// Reference  
int &reference = original;  
   
// Pointer  
int\* pointer = &original;