Contents

References and Pointers

1 Introduction

A computers memory is a sequency of bytes. 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 adresss. For example, when we declare and initilize 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.

2 References

In c++, 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 Mark. You can refer to the person as either Sonnny or Mark

Heres how we do that with code:

Suppose we have an *int* variable called *Mark*, we can create an alias to it by using the & sign in the declaration.

```
int &sonny = mark;
```

So here, we made *sonny* a reference to *mark*

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

Note:-

Two things to note about references

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

3 Pass-By-Reference

Pass-by-reference referes 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.

Note:-

Arrays do this by default. If you wanted to modify an array passed into the function, using pass-by-refernence is not needed.

You would do this to modify all other data types, like int and such

This allows us to:

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

The following code snippet shows an example of pass-by-reference

```
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";

}
```

4 Pass-By-Reference with Const

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 modifying 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 values(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

5 Memory Address

The `&` symbol 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;
```

We can find where this variable is stored in memory by printing out `&porcupine_count`

```
std::cout << &porcupine_count << "\n";
```

It will return something like:

```
0x7ffd7caa5b54
```

This is a memory address represented in **hexadecimal**. A memory address is usually denoted in hexadecimal instead of binary for readability and concisness.

Note:-

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

6 Pointers

In C++, a **pointer** variable is mostly the same as other variables, which can store a piece of date.

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. It is recommended to avoid pointers as much as possible; usually, a reference will do the trick.

Note:-

pointers must be delcared 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;
```

We can point our pointers to the memory address of variables by writting

```
int* var = &otherVar;
```

7 Dereference

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 foo = *ptr;
```

Note:-

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

8 Null Pointer

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

```
int* ptr;
```

In other words, it contains an address of "somewhere", which is of course not a valid location. This is **dangerous** We need to initilize 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`