# C++ Study Notes

| COLLABORATORS | | | |
|---|---|---|---|
| | *TITLE* :<br><br>C++ Study Notes | | |
| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
| WRITTEN BY | Frederick Ollinger | May 24, 2017 | |

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| 0.0.1 | 2017-05-21 | Initial Version | fko |

# Contents

# Chapter 1

# Pointers

I got this from: *Pointers in C with examples*.

A pointer is a variable that points to a section of memory. Below is a program that points to a block of memory to show how we can access memory in C.

## 1.1 Address of Operator

Before we look at pointers, we'll define what we mean by computer memory in address.c.

```
#include <stdio.h>

int var = 1;
int main()
{
   int num = 10;
   printf("Value of var is: %d \n ", num);
   printf("Address of var is: %p \n", &num);
   return 0;
}
```

```
Value of var is: 10
Address of var is: 0x7ffd5d20b56c
```

Thus, in this context the amperstand is the "Address Of" operator.

## 1.2 Value At Address Operator

Use the asterisk to create a pointer to a variable. The following example shows how pointers work, but it's for demonstration purposes only as we either have an address to a variable or a variable, but usually not both. Here's pointer.c:

```
#include <stdio.h>
int main()
{
    int var =10;
    int *p;
    p= &var;

    printf ( "\n Address of var is: %p \n", &var);
    printf ( "\n Address of var is: %p \n", p);
```

```
    printf ( "\n Address of pointer p is: %p \n", &p);

    /* Note I have used %p for p's value as it should be an address*/
    printf( "\n Value of pointer p is: %p \n", p);

    printf ( "\n Value of var is: %d \n", var);
    printf ( "\n Value of var is: %d \n", *p);
    printf ( "\n Value of var is: %d \n", *( &var));
}
```

Here's the output:

```
 Address of var is: 0x7ffda322535c

 Address of var is: 0x7ffda322535c

 Address of pointer p is: 0x7ffda3225350

 Value of pointer p is: 0x7ffda322535c

 Value of var is: 10

 Value of var is: 10

 Value of var is: 10
```

## 1.3 Sending Pointers To Functions

Here's a classic example which shows how pointers differ from regular variables. Sending a regular variable to a function will yield the same variables when you are done because we pass by value. But if we pass by reference to memory, we can actually allow a function to change a variable.

```
#include <stdio.h>

void swap (int *pa, int *pb) {
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

int main()
{
   int a = 10;
   int b = 20;

   printf("before swap a: [%i] b: [%i] \n", a, b);

   swap(&a, &b);

   printf("after swap a: [%i] b: [%i] \n", a, b);

   return 0;
}
```

Running the program:

```
before swap a: [10] b: [20]
after swap a: [20] b: [10]
```

Often, you don't want to have your variales modified when they are sent to a function. To show this in an api, use the keyword const which allows you to promise that you won't make this change.

```
#include <stdio.h>

void swap (const int *pa, const int *pb) {
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}

int main()
{
   int a = 10;
   int b = 20;

   printf("before swap a: [%i] b: [%i] \n", a, b);

   swap(&a, &b);

   printf("after swap a: [%i] b: [%i] \n", a, b);

   return 0;
}
```

Compiling this gives the following error:

```
cc  '-Iconst@exe' '-I.' '-I..' '-fdiagnostics-color=always' '-pipe' '-D_FILE_OFFSET_BITS ↩
    =64' '-Wall' '-Winvalid-pch' '-O0' '-g' '-fuse-ld=gold' '-MMD' '-MQ' 'const@exe/const.c. ↩
    o' '-MF' 'const@exe/const.c.o.d' -o 'const@exe/const.c.o' -c ../const.c
../const.c: In function 'swap':
../const.c:6:9: error: assignment of read-only location '*pa'
     *pa = *pb;
         ^
../const.c:7:9: error: assignment of read-only location '*pb'
     *pb = tmp;
```

This is a good thing as it can stop us from doing something which we promise not to do.

## 1.4 Treating A Pointer As An Array

I got this from: *Pointers in C with examples*.

First we'll start by covering arrays. An array is a variable that holds multiple values of the same type. Let's make an array of ints.

```
#include <stdio.h>

int main()
{
    int arr[3] ={ 1, 2, 3 } ;
    printf("arr: [%i] \n", arr[1]);

    return 0;
}
```

Output:

```
arr: [2]
```

Let's make a character array, which is a C string. Note that C will automatically allocate the size of the array for us if we leave the number of elements blank.

```
#include <stdio.h>

int main()
{
    char label[] = "Single";
    printf("label: [%s] \n", label);
    printf("label: [%c] \n", label[2]);

    return 0;
}
```

Output:

```
label: [Single]
printf("label: [%c] \n", label[2]);
```

We also learn that we can access a C string just like a normal array.

NEXT SHOW HOW WE CAN USE A POINTER TO AN ARRAY TO ACCESS CHARACTERS.

## 1.5 Stack vs. Heap

TBD

## 1.6 Dynamically Allocating Memory: Malloc, Free, New Delete

TBD

## 1.7 Memory leak

TBD

## 1.8 Segmentation Fault

TBD

## 1.9 Member variables of structs and classes.

TBD

## 1.10 NULL vs. nullptr

TBD