C++ Study Notes

COLLABORATORS			
	TITLE : C++ Study Notes		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Frederick Ollinger	May 28, 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 Arrays and Pointers

1.4.1 What Are Arrays?

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.

1.4.2 How To Use A Pointer To An Array

Here we demonstrate that pointers are merely indexes into arrays. It also shows that the increment operator will automatically increment the size of the type. That is, an int is different from a double, but if you increment a given type of pointer, it will automatically increment properly. Because of this, in order for this to work, you must know the size of the type at compile time.

```
#include <stdio.h>

int main()
{
    int arr[4] = { 1, 2, 3, 4 };
    int *parr = arr;
    printf("parr: [%i] \n", *parr);
    parr++;
    printf("parr: [%i] \n", *parr);

    return 0;
}
```

Output:

```
parr: [1]
parr: [2]
```

1.5 Dynamically Allocating Memory I (Malloc and Free)

1.5.1 malloc()

In C (and in any programming language) each variable uses memory. C is unique in that it's one of the few langauges which exposes some of the details of this proceess.

In this example, we're going to create an array, but we don't know at compile time how large to make it.

```
#include <stdio.h>   // printf()
#include <stdlib.h>  // malloc()
#include <string.h> // bzero()

#define ARRAY_SIZE 5

int main()
{
    int *pint;
    pint = (int *) malloc(sizeof(int) * ARRAY_SIZE);
    bzero(pint, sizeof(int) * ARRAY_SIZE);

    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("[%i]: [%i] \n", i, *(pint+i));
    }

    return 0;
}
```

Output:

```
[0]: [0]
[1]: [0]
[2]: [0]
[3]: [0]
[4]: [0]
```

Note, one can modify ARRAY_SIZE and allocate and zero different amounts of memory.

Note that you should get in the habit of reading the manpages for things like bzero.

```
 $ man bzero
```

1.5.2 free()

Note that each time we call malloc() we need to call free(). If we don't then we'll create a memory leak. Thus, in the last program, there's a memory leak.

Below is an example of using free().

```
#include <stdio.h>   // printf()
#include <stdlib.h>  // malloc()
#include <string.h> // bzero()

char* getstr() {
    char *pchar;
    pchar = (char *) malloc(sizeof(char) * 10);
    strcpy(pchar, "my string");
    return pchar;
}

int main()
{
    char *str = getstr();
    printf("[%s] \n", str);
    free(str);
    return 0;
}
```

1.5.3 Stack vs. Heap

I got this from: *Memory : Stack vs Heap*.

Declaring a variable on the stack is:

```
    int i;
```

When you use malloc(), you are declaring a variable on the heap. Which way you decide to do things depends on a few factors.

"Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted)."

Thus, with the stack you don't have to do your own memory management.

"Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope."

1.6 Dynamically Allocating Memory II: New, Delete

Now that we understand malloc/free, we're going to look at the same concepts in C++. While you can use malloc/free in C++, generally, we use new/delete.

```
#include <stdio.h>   // printf()
#include <stdlib.h>  // malloc()
#include <string.h> // bzero()

#define ARRAY_SIZE 5

int main()
{
    int *p = new int();
    *p = 10;

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

    delete p;
    p = NULL;

    return 0;
}
```

In this example, we have basically made the pointer.c example, but we have written it so it uses new/delete. Just like malloc, new will allocate on the heap and thus needs to be freed.

1.7 Segmentation Faults (Program Crashes)

A segmentation fault is where we try to reference memory which is not available.

1.7.1 Uninitialized Pointer

In this example, we don't set our pointer to anything.

```
int main()
{
    int *p;
    *p = 10;
    return 0;
}
```

Output:

```
Segmentation fault
```

1.7.2 Using The Debugger To Find Why A Program Crashed

Though this segfault is simple, we might need to find a segfault in a more complex program. We can use the debugger to do this. First, though, we need to turn on core dumps. Core files are dumped when we segfault. But they are off by default on most Linux distros.

```
ulimit -c unlimited
```

Output with core dumps on:

```
Segmentation fault (core dumped)
```

Now let's try to find out where crash is:

```
 $ gdb segfault1 core
```

```
Core was generated by `./segfault1'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00000000004004ae in main () at ../segfault1.cpp:4

warning: Source file is more recent than executable.
4           *p = 10;
(gdb) bt
#0  0x00000000004004ae in main () at ../segfault1.cpp:4
```

In this example, we were able to find the line that the segfault was on.

1.7.3 Using A Pointer After It's Deleted

Though these examples are contrived, the actual bugs are very common. In this case, we delete a pointer then try to access it after it's been deleted.

```
#include <stddef.h> // for NULL
#include <stdio.h> // for printf

int main()
{
    int *p = new int();
    *p = 10;

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

    delete p;
    p = NULL;

    *p = 20;

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

    return 0;
}
```

It's an excercise for the reader to find the line where we segfault as well as running gdb on this example.

1.8 NULL vs. nullptr

I got this from: *A name for the null pointer: nullptr*.

C++ introduced nullptr to replace NULL in C++11. Thus, in new C++ code, you should NEVER use NULL, but rather use nullptr.

```
#include <stdio.h> // for printf

int main()
{
    int *p = new int();
    *p = 10;

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

    delete p;
    p = nullptr;

    if (nullptr == p) {
        printf ( "\n ERROR: p is uninitialized. \n");
  return -1;
    }

    *p = 20;

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

    return 0;
}
```

The main advantage of nullptr is that it's less ambiguous as NUL can be confused by the compiler as 0.