C++ Study Notes

COLLABORATORS			
	TITLE : C++ Study Notes		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Frederick Ollinger	July 2, 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

1.6.1 Dynamically Allocating Single Item On Heap With New

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()

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.

1.6.2 Arrays with new/delete

Here we show how to allocate an array using new. In this example, we especially note that we need to use delete[] operator to match new[]. If we don't do this, we won't free all the memory, and we have created a memory leak.

```
#include <stdio.h>   // printf()

void printarr(int *i, int c) {
    printf ( "\n Address of count %i var is: %p \n", c, i + c);
    printf ( "\n Value of count %i var is: %d \n", c, *(i+c));
}

int main()
{
    int size = 5;
```

```
    int count = 0;
    int *arr = new int[size];
    *arr = 10;
    *(arr+1) = 20;

    printarr(arr, count);

    count++;
    printarr(arr, count);

    count++;
    printarr(arr, count);

    delete[] arr;
    arr = nullptr;

    return 0;
}
```

1.7 Broken sizeof()

What do you think this result is?

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

int main()
{
    const char *string0 = "12345678910111213141516171819 20";
    printf("sizeof: [%s] is [%i] \n", string0, sizeof(string0));

    const char *string1 = "1";
    printf("sizeof: [%s] is [%i] \n", string1, sizeof(string1));

    return 0;
}
```

```
sizeof: [12345678910111213141516171819 20] is [8]
sizeof: [1] is [8]
```

Thus, sizeof() does NOT tell you how large a string is, but rather how many bytes are in a pointer. This should be platform dependent.

1.8 Segmentation Faults (Program Crashes)

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

1.8.1 Uninitialized Pointer

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

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

Output:

```
Segmentation fault
```

1.8.2 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.

1.8.3 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.

1.9 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.

1.10 References

References are NOT pointers. They do NOT take a pointer. They take a regular variable's address. From the caller's point of view, they are giving a normal variable. But to the reciever, they are getting an address which can be modified. Thus, references can be used for return values.

NOTE: That references are for C++, ONLY. They are not part of C.

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

void setint(int &i) {
    i = 10;
}

int main()
{
    int j = 5;

    printf("before setint() j is [%i] \n", j);

    setint(j);

    printf("after setint() j is [%i] \n", j);

    return 0;
}
```

Passing a pointer to a function that takes a reference will fail to compile because references are 100% different than pointers.

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

void setint(int &i) {
    i = 10;
}

int main()
{
    int j = 5;
    int *p_j = &j;

    printf("before setint() j is [%i] \n", j);

    setint(p_j);

    printf("after setint() j is [%i] \n", j);

    return 0;
}
```

```
../broken-reference.cpp: In function 'int main()':
../broken-reference.cpp:14:15: error: invalid conversion from 'int*' to 'int' [-fpermissive ↩
    ]
     setint(p_j);
               ^
../broken-reference.cpp:3:6: note:   initializing argument 1 of 'void setint(int&)'
 void setint(int &i) {
      ^
../broken-reference.cpp:14:15: error: cannot bind rvalue '(int)((long int)p_j)' to 'int&'
     setint(p_j);
```

How can you fix the above program so that it will compile and run by changing a single character?

ANSWER: where it says setint(p_j); put an asterisk before the variable: setint(*p_j);

1.11 Pointer To Pointer

Why would you use a pointer to a pointer? There are several reasons. Here's a wrong example.

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

#define STRING1 "Friendship is magic.\0"
#define STRING2 "What the hay?\0"

void make_strings(char *str1, char *str2) {
    str1 = (char *) malloc(sizeof(char) * strlen(STRING1) + 1);
    strncpy(str1,STRING1, strlen(STRING1));

    str2 = (char *) malloc(sizeof(char) * strlen(STRING2) + 1);
    strncpy(str2,STRING2, strlen(STRING2));

    return;
}

int main()
{
```

```
    char *s1, *s2;
    make_strings(s1, s2);
    printf("string 1 [%s] string 2 [%s] \n", s1, s2);
    return 0;
}
```

```
string 1 [(null)] string 2 []
```

Why is this wrong?

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

#define STRING1 "Friendship is magic.\0"
#define STRING2 "What the hay?\0"

void make_strings(char **str1, char **str2) {
    *str1 = (char *) malloc(sizeof(char) * strlen(STRING1) + 1);
    strncpy(*str1,STRING1, strlen(STRING1));

    *str2 = (char *) malloc(sizeof(char) * strlen(STRING2) + 1);
    strncpy(*str2,STRING2, strlen(STRING2));

    return;
}

int main()
{
    char *s1, *s2;
    make_strings(&s1, &s2);
    printf("string 1 [%s] string 2 [%s] \n", s1, s2);
    return 0;
}
```

1.12 Struct

I got this from: *struct (C programming language)*.

"The C struct directly corresponds to the assembly language data type of the same use, and both reference a contiguous block of physical memory, usually delimited (sized) by word-length boundaries. Language implementations which could utilize half-word or byte boundaries (giving denser packing, using less memory) were considered advanced in the mid-eighties. Being a block of contiguous memory, each variable within is located at a fixed offset from the index zero reference, the pointer."

Here's the simplest struct program.

```
struct person {
    int age;       // years
    int height_cm; // cm
};

int main()
{
    struct person Fred;
    Fred.age = 45;
    Fred.height_cm = 70;
    return 0;
}
```

Note here that we can have as many fields as we want of as many different types. When we wish to refernce a field in a struct, we use the dot operator.

Next we'll use malloc to dynamically allocate a pointer to a struct.

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

struct person {
    int age;       // years
    int height_cm; // cm
};

void printPerson(struct person *p) {
    printf("printPerson() age is [%i] and height in cm is [%i] \n", p->age, p->height_cm);
    p->age = 70;
    p->height_cm = 40;
}

int main()
{
    struct person *Fred;
    Fred = (struct person*) malloc(sizeof(struct person));
    Fred->age = 45;
    Fred->height_cm = 70;
    printPerson(Fred);

    printf("main() age is [%i] and height in cm is [%i] \n", Fred->age, Fred->height_cm);

    return 0;
}
```

Output:

```
printPerson() age is [45] and height in cm is [70]
main() age is [70] and height in cm is [40]
```

In this program, we demonstrate that we can easily pass a pointer to a struct to a function. Next, we demonstrate that the arrow operator is used to access a field from the struct. Finally, we recall that since this is passed as a non-const pointer, a function can change the values.

In the next example, we dynamically allocate a char string inside a struct. The point of this example is to gradually add more complex data structures and how they can be constructed using pointers.

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

struct person {
    int age;          // years
    int height_cm;    // cm
    char *first_name; // cm
};

void printPerson(struct person *p) {
    printf("printPerson() name is [%s] \n", p->first_name);
}

int main()
{
    struct person *Fred;
    Fred = (struct person*) malloc(sizeof(struct person));
```

```
    Fred->age = 45;
    Fred->height_cm = 70;

    Fred->first_name = (char*) malloc(sizeof(char) * 5);
    strncpy(Fred->first_name,"Fred\0", 5);

    printf("main() name is [%s] \n", Fred->first_name);

    printPerson(Fred);

    return 0;
}
```

1.13 Dynamic Structures in C

1.13.1 Dynamic Array of C Strings

In this example, we make a dynamic array of C strings.

```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc
#include <string.h> // strncpy

#define MAX_NUMBER_OF_STRINGS 10

struct strstruct {
    char *argv[MAX_NUMBER_OF_STRINGS];
    int argc;
};

void addstring(struct strstruct *st, const char *string) {
    st->argv[st->argc] = (char*)malloc(strlen(string) + 1);
    strncpy(st->argv[st->argc], string, strlen(string) + 1);
    st->argc++;
}

void printstring(struct strstruct *st, int i) {
    printf("print string [%i]: [%s] \n", i, st->argv[i]);
}

int main()
{
    struct strstruct st;
    struct strstruct *p_st = &st;
    st.argc = 0;

    addstring(p_st, "applejack");
    addstring(p_st, "pinkie pie");

    printstring(p_st, 0);
    printstring(p_st, 1);
    printstring(p_st, 2);

    return 0;
}
```

1.13.2 Dynamic Structure to Structure

Filename: structstruct.c

We're going to make a structure that allocates a pointer to another structure.

Our example is going to be based upon the file structure which is found in the Linux kernel. We'll only implement a subset of that.

From `/usr/src/linux-headers-4.7.0-1-common/include/linux/fs.h` :

```
struct file {
    unsigned short f_count; // number of file handles
    struct inode * f_inode; // inode pointing to cooresponding file
};
```

```
struct inode {
    unsigned long i_blocks; // file size in blocks
    char *name;             // file name (not actually part of Linux inode)
};
```

An inode is a structure that tells the kernel everything that it needs to know in order to operate on a file.

Below is the complete program:

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

struct inode {
    unsigned long i_blocks; // file size in blocks
    char *name;             // file name (not actually part of Linux inode)
};

struct file {
    unsigned short f_count; // number of file handles
    struct inode * f_inode; // inode pointing to cooresponding file
};

int main()
{
    static char *filename = "struct.c";
    struct file *myFile;
    myFile = (struct file*) malloc(sizeof(struct file));
    myFile->f_inode = (struct inode*) malloc(sizeof(struct inode));
    myFile->f_count = 1;
    myFile->f_inode->i_blocks = 10;
    myFile->f_inode->name = (char*) malloc(sizeof(char*) * (sizeof(filename)+1));
    strncpy(myFile->f_inode->name, filename, strlen(filename) + 1);

    return 0;
}
```

1.14 Function Pointers

Since we can access memory in C, and functions are in memory, though it's strange, it does make sense that we should be able to access functions in terms of pointers.

Here's a contrived example, which doesn't really do anything, but it does show the syntax of function pointers.

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

int add(int a, int b) {
    return (a+b);
}

int (*addPtr)(int, int);

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

    int c = addPtr(a, b);

    printf("result %i + %i = %i \n", a, b, c);

    return 0;
}
```

In the next example, we show how one would use a function pointer so we can swap out one algorithm for another:

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

int add(int a, int b) {
    return (a+b);
}

int subtract(int a, int b) {
    return (a-b);
}

int (*addPtr)(int, int);
int (*subtractPtr)(int, int);

int compute(int a, int b, int (*f)(int, int)) {
    int i;
    int c = 0;
    for (i = 0; i < a; i++) {
        c = (*f)(i, b);
    }
    return c;
}

int main()
{
    int a = 10;
    int b = 20;
    addPtr = add;
    subtractPtr = subtract;

    int c = compute(a, b, addPtr);
    printf("add result: %i \n", c);

    c = compute(a, b, subtractPtr);
    printf("subtract result: %i \n", c);

    return 0;
}
```