POINTERS& USER-DEFINEDTYPES IN CPP

A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to *point to* the second.

Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. Apointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

*type *name;*

where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name.*

The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly. (Pointer arithmetic is discussed later in this chapter.)

The Pointer Operators

The pointer operators were discussed in Chapter 2. We will take a closer look at them here, beginning with a review of their basic operation. There are two special pointer operators: * and **&**. The **&** is a unary operator that returns the memory address of its operand. (Remember, a unary operator only requires one operand.)

For example,

m = &count;

places into **m** the memory address of the variable **count**. This address is the computer's internal location of the variable. It has nothing to do with the value of **count**. You can think of **&** as returning "the address of." Therefore, the preceding assignment statement means "**m** receives the address of **count**." To understand the above assignment better, assume that the variable **count** uses memory location 2000 to store its value. Also assume that **count** has a value of 100. Then, after the preceding assignment, **m** will have the value 2000. The second pointer operator, *, is the complement of **&**. It is a unary operator that returns the value located at the address that follows.

For example, if **m** contains the memory address of the variable **count**,

```
q = *m;
```

places the value of **count** into **q**. Thus, **q** will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in **m**. You can think of * as "at address." In this case, the preceding statement means "**q** receives the value at address **m**." Both **&** and * have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal.

You must make sure that your pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type **int**, the compiler assumes that any address that it holds points to an integer variable—whether it actually does or not. Because you can assign any address you want to a pointer variable, the following program compiles without error, but does not produce the desired result:

```
#include <stdio.h>
int main(void)
{
double x = 100.1, y;
int *p;
/* The next statement causes p (which is an integer pointer) to point to a double. */
p = (int *)&x;
/* The next statement does not operate as expected. */
y = *p;
printf("%f", y); /* won't output 100.1 */
return 0;
}
```

This will not assign the value of **x** to **y**. Because **p** is declared as an integer pointer, only 4 bytes of information (assuming 4-byte integers) will be transferred to **y**, not the 8 bytes that normally make up a **double**. *In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast. In C, casts should be used for most pointer conversions.*

Pointer Expressions

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions. Pointer **Assignments** As with

any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer.

For example,

```
#include <stdio.h>
int main(void)
{
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
printf(" %p", p2); /* print the address of x, not x's value! */
return 0;
}
```

Both **p1** and **p2** now point to **x**. The address of **x** is displayed by using the **%p printf( )** format specifier, which causes **printf( )** to display an address in the format used by the host computer.

Pointer Arithmetic

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let **p1** be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long. After the expression p1++; **p1** contains 2002, not 2001. The reason for this is that each time **p1** is incremented; it will point to the next integer. The same is true of decrements. For example, assuming that **p1** has the value 2000, the expression p1--; causes **p1** to have the value 1998. Generalizing from the preceding example, the following rules govern pointer arithmetic. Each time a pointer is incremented, it points to the memory location

Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers **p** and **q**, the following statement is perfectly valid: if(p<q) printf("p points to lower memory than q\n");

Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a

pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf