Chapter 5: References and Pointers

Notes

- Reference variables can be created along-side normal variables of the same type, just like pointer variables. They can even be initialized to variables further up in the list.

Introduction

This chapter is all about accessing a variable's value without using the variable's name. On one hand you can do this through references and on another you can use the variable's memory address with pointers. If you can get past this chapter, you'll have done what many would-be C++ programmers have gotten stumped at. But I believe I can get you through this with minimal scratches or brain damage. Make sure your health insurance covers "mental disability by C++ pointers" though. ☺ Hopefully this chapter reads easier than it was to write!

Reference Variables

A reference variable, or simply *reference*, is a nickname to a variable. It's used like the original variable name and it has the same effect on the same value. To create a reference variable you write out a variable declaration as normal, but precede the name with the *reference operator* or ampersand. Next you must initialize the new reference variable by assigning it to a variable:

```
int &reference = variable;
```

A reference does not have a value of its own. It is simply a reference to another variable of the same type, another identity for the same thing, a "nickname". A reference cannot exist without referring to something. For this reason a reference variable must always be initialized.

When you initialize a reference variable, it is a special assignment operation. Usually an assignment expression copies the value of the right operand into the storage unit of the left operand. In the case of a reference declaration, the assignment does no copying at all. Instead it simply creates an alias to the specified variable that you can use in the same way. We can use this moniker in any place we would normally use the variable and it will effect the same value.

The following creates an 'int' called 'x' and a reference to it called 'ref_x':

```
int x = 5;
int &ref_x = x;
```

This reference, ‘ref_x’, is simply a nickname for ‘x’. The type of ‘ref_x’ must be the same as ‘x’. And ‘ref_x’ must be initialized like any reference variable. If I had simply written ‘`int &ref_x;`’ the compiler would reject it. Always initialize your reference variables! This whole statement simply creates a nickname for ‘x’ called ‘ref_x’. We can use ‘ref_x’ in anyplace that ‘x’ is used and it will act the same way.

Since it is a nickname, any changes to ‘ref_x’ are reflected in both ‘ref_x’ and ‘x’ (because they are two names for one storage unit). So, if our storage unit was a crate for storing oranges, it would first be given a label ‘x’. Then by creating the reference, we’d also give it another label: ‘ref_x’. Anytime we want to refer to that crate of damn delicious oranges, we could use the name ‘x’ *or* ‘ref_x’:

<crate of oranges analogy>

A variable reference acts like nicknames we give each other in real life. You may know your friend both by his name and a nickname or alias. Both names, the real one and the alias, refer to the same person. This is how variable references are. Both the reference variable and the original variable refer to the same value.

<friend with real name and nickname picture>

The following program demonstrates the relationship of a reference and the value it refers to:

```
#include <iostream.h>

int main()
{
    int x = 5;
    int &ref_x = x;
    cout << “The value of ‘ref_x’ is ” << ref_x << endl;
    ref_x += 5;
    cout << “The value of ‘x’ is ” << x << endl;
    return 0;
}
```

The output of the above program is:

```
The value of ‘ref_x’ is 5
The value of ‘x’ is 10
```

First the ‘int’ variable ‘x’ is created, and then a reference called ‘ref_x’ is made to it. The next line prints the value of ‘ref_x’. Because this is a reference to ‘x’ the value printed is that of ‘x’ itself. Remember, ‘ref_x’ is simply a nickname to the same variable which is ‘x’ in this case. Armed with this knowledge, we know that the next line adds five (5) to the value of ‘x’ because ‘ref_x’ is referencing the same value. Lastly the value of ‘x’ is printed, which is now ten (10).

A reference variable can be used like any normal variable. It also follows the same rules of scope. If you were to create a reference inside a nested block, you would not be able to use that same reference *after* the block ended. This is because the reference variable would have been local, gone out of scope, and been destroyed. The following program would not compile:

```
#include <iostream.h>

int main()
{
    int x = 5;
    {
        int &ref_x = x;
        cout << “’ref_x’ is “ << ref_x << endl;
    }
    ref_x += 5;
    return 0;
}
```

When the statement ‘ref_x += 5;’ is reached, that reference variable has already gone out of scope and been destroyed. Remember that even though it is a special variable, it still follows the same rules of scope.

Blarg.

Memory Addresses

All variables in C++ have a *memory address*. This is not to say that each variable has a second value which represents its address. Rather, memory in C++ is laid out in a single line and each variable occupies a portion of that. The address of a variable is its *offset* from the beginning of available memory. Even though this relates very closely to how many machines *use* memory; it doesn’t have to and it is a C++ concept that can be dealt with appropriately by the compiler.

The memory area in C++ is a single line of binary data, divided up by bytes. Thus every variable must use a portion that equates to one or more bytes. And the address of the variable is the *byte offset* from zero. In other words, each new memory address is a new byte. When referring to data that occupies multiple bytes, you would use the first memory address. This is like going to the front of a new house rather than coming in through the basement or something.

Let’s think of crates, lots of them. Each crate occupies one or more bytes in memory. Obviously, the more bytes occupied, the bigger the crate and the more that crate can hold. Let us imagine that they are divided by square meters (going metric baby) and that all of them are organized on a single line; just like bytes in memory. Every meter with no crate is an unused byte of memory. The number of meters from the start of the line is the *address* of the crate: its location:

<analogy of crates>

A variables address (location) is its distance from zero, measured in bytes. Thus a variable with the address 1675 is at the 1,675th byte in available memory[1]. A variable’s type determines the storage unit it uses and that in turn is the number of bytes that it occupies. A ‘`long`’ uses a quad-word storage unit which is four (4) bytes. Therefore a ‘long’ occupies four bytes of memory, and the next variable must occur at least four (4) bytes from the ‘long’ variable’s address:

<long variable at 1675 in memory>

To get the memory address of a variable you use the *address operator* which is the ampersand (`&`). The result of this *unary* operation is the memory address of the right operand. It is commonly read as “get the address of”. So, when put in front of a variable, such as ‘`x`’, it would read “get the address of variable ‘`x`’”. The following program creates a variable and prints its address:

```
#include <iostream.h>

int main()
{
    int x = 5;
    cout << “Address of ‘x’ is ” << &x << endl;
    return 0;
}
```

Note, if you will, that ‘`cout`’ will automatically show a memory address as a hexadecimal number, rather than decimal. This address can be expressed as an integer number as you have seen, but in mathematical operations it will be treated as a unique format. For example, trying to assign the address of a variable to a normal integer-based variable will generate a compiler error:

```
int x = 5;
int y = &x;
```

The storage unit required to hold a memory address varies from system to system, much like the storage unit of an ‘int’ varies. In fact, the storage unit is typically the same as for an ‘int’, but you should not rely on this. You can create a *pointer variable* with the correct storage unit for a memory address, and use it to refer to another variable.

[1] Memory addresses are typically expressed in hexadecimal. But since we all natively speak decimal, I’ll begin with that until you understand more about this topic.

Pointers

A pointer variable, commonly known simply as *pointer*, is similar to a reference variable in that it is a way of indirectly referring to another value, except it is *not* a nickname. A pointer variable *is* a real variable and has a numeric value like any other[2]. However, the value of a pointer is the *address* of another variable. Pointer variables are the only ones allowed by the compiler to store memory addresses safely. Usually the pointer stores the address to another variable; this is known as *pointing* to a variable:

<simple picture of pointer variable containing address of other variable>

To create a pointer variable, you declare it as you normally would with a type and a name and possibly an initial value. However, you must precede the name with an asterisk to signify that it is a pointer variable. The type you give the pointer variable is the type of variable that the pointer variable will point to. For example:

```
int *p;
```

This would create a pointer variable called ‘p’ which would point to variables of type ‘int’. You can create multiple pointer variables in a single declaration, but each identifier must be preceded by an asterisk to signify that it is a pointer. The following would create a normal ‘int’ named ‘x’ and a pointer named ‘p’:

```
int x, *p;
```

Whereas the following would create two pointers, ‘x’ and ‘p’:

```
int *x, *p;
```

Address Assignment

The first thing you need to do in order to use a pointer variable is assign it an address. We can get the address of a variable using the ampersand. Once you have a variable’s address through such an expression, you can use a normal assignment operator to put it into a pointer:

```
int x = 5;
int *px = &x;
```

The above creates a variable called ‘`x`’. The address of ‘`x`’ is then used to initialize the pointer ‘`px`’ which points to ‘`int`’-type variables. So what it does is get the address of ‘`x`’ and assign it to the pointer variable ‘`px`’. If you attempted to put the address of a

[2] Although a pointer is a numeric variable, it is illegal (therefore impossible) to create a reference to a pointer variable.

different type into ‘`px`’ you would get a compiler error. Therefore the following would not work:

```
float x = 5.5;
int *px = &x;
```

Just as with normal variables, you can use assignment operators outside of initialization. This means that unlike reference variables you can change the address stored by the pointer at any time within your program. The following works just fine:

```
int x = 5;
int y = 10;
int *p = &x;
p = &y;
```

Storing the memory address of a variable in a pointer means nothing really until you use it. The pointer’s value is simply another variable’s address. If we think about our crates, a pointer variable would be analogous to an empty crate with a note in it reading the address of another crate. The address is stored *as* the pointer’s value, like the crate containing the pointer note. Actual data is stored elsewhere, in the memory location referred to by the pointer. It is in the crate whose address is specified.

Unlike references, pointers do not have to be initialized. They can be assigned the address of a variable anytime within their lifetime:

```
int x;
int *px = 0;
px = &x;
```

If a pointer is not going to be initialized to the address of another variable, it is good practice to initialize it to zero (0) as I have done above.

Dereference

Its not much good storing the memory address of another variable unless you’re going to use it to affect the other variable. But any mathematical operations we do to the pointer variable will simply affect the address that it stores, not the value that is *at* the address. To modify the value at the specified address, you must first *dereference* the pointer.

Dereferencing can be thought of as actually going to the value. When you find a crate that has a note mentioning the address of another crate, you would be dereferencing the current crate by going to the one whose address is mentioned. Once you’re at that point you can modify the value; not the memory address stored by the pointer.

To dereference a pointer, you precede it with the unary *pointer-indirection operator* which is an asterisk (`*`). This operator is commonly seen as meaning “the variable pointed to by”. The indirection operator is the inverse of the address operator. Where the address operator gets the address of the variable, the indirection operator gets the variable from the address.

Once a pointer has been dereferenced it can be used like the variable it is pointing to. The following demonstrates this:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int x = 5;
06      int *px = &x;
07      cout << “The value of ‘x’ is ” << (*px) << endl;
08      *px += 5;
09      cout << “The value of ‘x’ is ” << x << endl;
10      return 0;
11  }
```

The output of this program is:

```
The value of ‘x’ is 5
The value of ‘x’ is 10
```

Let’s look at some of these lines individually.

```
06      int *px = &x;
```

This is read as “assign the address of variable ‘x’ in the pointer variable ‘px’”. We use the address operator on ‘x’ to get the variable’s address. That is then used to initialize the pointer variable ‘px’ which points to variables of type ‘int’ (which ‘x’ happens to be).

```
07      cout << “The value of ‘x’ is ” << (*px) << endl;
```

The indirection operator is used with ‘px’ here. Thus, here it means “the variable pointed to by ‘px’”. The variable ‘px’ points to is ‘x’, meaning the address of ‘x’ is the value of ‘px’. By dereferencing the pointer ‘px’ we get the variable ‘x’. I enclosed this in parenthesis to make sure it was done first. When using the indirection operator within a complex statement, I try to do this to avoid ambiguities. This statement would break down as follows:

```
cout << “The value of ‘x’ is ” << (*px) << endl;
cout << “The value of ‘x’ is ” << x << endl;
cout << x << endl;
cout << endl;
```

The first expression done is the pointer-indirection operation or dereferencing. In terms of our crates, we wouldn’t count the oranges in ‘px’ (because it contains a pointer-note to another crate and therefore really *has* no oranges), we’d count the oranges in the crate that ‘px’ refers to.

```
08      *px += 5;
```

This line reads as “Add five to the variable pointed to by ‘px’”. Since the variable ‘px’ points to is ‘x’, this line adds ‘5’ to ‘x’. Or more specifically, ‘5’ is added to the value stored in memory at the address pointed to by ‘px’. First ‘px’ is dereferenced so it becomes the value it points to. Then ‘5’ is added to that value.

Here we get into the reason the indirection operator is named what it is. A variable has direct access to the value it represents. A pointer variable, on the other hand, indirectly accesses that value *through* that value’s address. Referencing a value through a pointer is called *indirection*. To perform this indirect access you dereference the pointer.

Let’s think of our crates. We got these five oranges we have to put somewhere. The ‘px’ crate, however, has a note with the address of another crate in it. You can’t directly put the oranges into a crate because first you must walk over to the crate at the specified address. Then you put the oranges there. You have indirectly accessed that crate because you first had to go through the pointer.

Although I show ‘(*px)’ as being evaluated to simply ‘x’ this is a little bit of a gray lie. Dereferencing a pointer will yield a value that is the storage unit and type of the pointer variable’s type. In this case ‘x’ happens to be the same type and storage unit as what the pointer expects to be pointing to. The *type* of the pointer variable determines what value type the pointer will be dereferenced to, not the variable whose address was stored.

Pointer Types

A pointer’s value is simply a *memory address*. This is not necessarily the address of another variable, though very well could be. If you assign an address of a variable, the pointer never “sees” or “knows” about the variable you used. It simply has a memory address. And memory addresses are simply a byte-based offset from the beginning of available memory. So what does the pointer really know about the variable? Nothing, it just has an address.

My analogy with the crates for pointers is a crate that doesn’t contain oranges, but contains a note with the address of another crate. What do you know about the crate at that address? Nothing! You simply have this address.

When you dereference a pointer, it will represent the storage unit at the address it contains. But since it knows nothing of the variable whose address it contains, how does it know which storage unit to use. Even more so, how does it know the number format (integer / floating point / memory address) of that value? This is where a pointer’s type comes into play.

It is assumed that what a pointer points to is the address of a variable whose type is the same as the pointer. Thus far in my examples, this has been true so there are no problems. But it *is* possible for a pointer to contain the address of a variable whose type is different than its own. Example: a ‘float’ pointer could conceivably contain the address of a ‘char’ variable. Sure the storage units of a ‘float’ and ‘char’ are completely different, but the address of everything in C++ requires the same storage unit. So, if you were to dereference this ‘float’ pointer, it would represent much more than was originally intended:

<picture of memory, showing ‘char’ storage unit enclosed in ‘float’ storage unit>

The type of a pointer variable is a *type safe* mechanism in C++. If you try to assign the address of a ‘char’ variable to a ‘float’ pointer you will get a compiler error. This is not to say that it can’t be done, but you must *force* it to happen. You can do this through some precarious *casting* which was covered previously.

But the type of a pointer does *not* determine its storage unit. The storage unit is the same for all pointer variables, because all memory address require the same storage unit on a particular platform. To see the size of the storage unit required for memory addresses on your system, run the following program:

```
#include <iostream.h>

int main()
{
    cout << “Memory Address requires ‟ << sizeof(int*)
         << ‟ bytes.″ << endl;
    return 0;
}
```

The above program prints the number of bytes necessary to hold an address on your platform. It does this using the ‘`sizeof`’ operator which I previously covered.

Void Pointers

It is possible to create a pointer variable that has no type and therefore cannot be dereferenced. The type used to create this pointer is the keyword ‘void’. A pointer with this type is pointing to an abstract place in memory. It can hold a memory address like any other pointer, but what it points to is not expected to be anything in particular. For example, an ‘int’ pointer is thought to be storing an address to an ‘int’ variable. Nothing like this is assumed, however, with ‘void’ pointers.

All type safe mechanisms in C++ are *off* when using void pointers. This means the compiler will not complain if you give it the address of a ‘float’, ‘int’, or anything else. The following creates a void pointer and assigns it first the address of an ‘int’ and then a ‘float’:

```
int x;
float y;
void *p = &x;
p = &y;
```

Null Pointers

It is commonly acknowledged that no data shall exist at address zero (0). Therefore the value zero is also known as “null”. A *null pointer* is a pointer whose value is address zero or null. When you create a pointer variable it is best to initialize it to this value if you are not going to use the address of another variable. Another way to represent a null address is through the ‘NULL’ constant which means the same thing. The following pointer variables are both initialized to address zero or null:

```
int *p1 = 0;
int *p2 = NULL;
```

It may be deceiving, but ‘NULL’ is not a keyword. The line ‘#include <iostream.h>’ will import the necessary stuff for using this constant. It was once considered very bad programming to use the literal zero (0) instead of the constant ‘NULL’. This over-zealous prudence has much faded, but you may still find it among some programmers. My advice is to go with the flow. If your instructor or employer is adamant about using ‘NULL’ then do it, otherwise make your own decision.

Pointer Assignments

When you create two ‘int’ variables, you can easily assign, or copy, the value of one to the other. Since pointers are really just variables, they have the same capability. But when you do this, remember that you are assigning the *value* of the pointer which is a memory address. Unless you use the indirection operator, the assignment will not be of the value whose address is stored in the pointer. Consider the following:

```
int x = 5;
int *px = &x;
int *p = px;
```

The value of ‘p’ and ‘px’ will both be the same thing: the address of ‘x’. The value of ‘x’, which is ‘5’, is not touched or exchanged because the pointers are only dealing with that variable’s address. The following, however is different *and* will not work:

```
int x = 5;
int *px = &x;
int *p = *px;
```

This will generate a compiler error. The last statement is initializing the pointer ‘p’ to the value ‘*px’. Here the indirection operator is used so this statement is “get the value pointed at by ‘px’”. So, it is trying to assign the *value* of ‘x’ to ‘p’ rather than its address. In this case the value of ‘x’ is five (5). The statement is equivalent to:

```
int *p = 5;
```

We know this is illegal because you cannot implicitly assign an integer value to a pointer. The compiler must know it is an address or you must force it using explicit casting. It is important to recognize how a pointer’s value is used. Either the pointer’s actual value, which is a memory address, is used or the value referred to by the memory address is used. You need to be able to distinguish between both.

References are Pointers

I have been slightly misleading when I said that references are not variables because they have no storage space of their own. In fact, references *are* pointers deep down. So deep, in fact, that it is impossible to tell the truth without understanding how C++ is represented in assembler or machine language. References are an easier/safer style of pointer interaction. Where pointers are immensely complex and can do all sorts of crazy things, a reference is always just a nickname to a variable.

It is actually impossible to tell that a reference has storage space using any C++ code. Take the following example:

```
#include <iostream.h>

int main()
{
    int x = 0;
    int &ref_x = x;
    cout << ‟&x = ‟ << &x << endl
         << ‟&ref_x = ‟ << &ref_x << endl;
    return 0;
}
```

The output of this program will be two identical memory addresses. Why? Because ‘`ref_x`’ is a reference to ‘`x`’, the address operation will be done using the actual variable (‘`x`’) rather than the reference (‘`ref_x`’). Logically, this makes perfect sense. It also makes it impossible for me to convince you that reference variables are pointers underneath. In a later chapter we will revisit this concept. For now, use references like you always would and be content with their safe and simple semantics.