Lesson 4 - Pointers
Referencing and dereferencing

Summary

This lesson aims to provide an understanding of pointers and memory allocation.

New Concepts

Referencing, dereferencing, values and indirect access.

Pointers

We begin by declaring an integer variable `a` on line 7. As we have seen we can find out its value by using the `a` identifier. Now we introduce the reference operator &. See how it is applied to `a` in the `cout` statement on line 11. The reference operator is used to ask where a variable is in memory (its location).

Let us now create something called a pointer, see line 13. A pointer is a memory location that points to another memory location. Here we see the creation of `b` as a pointer and the assigning of the memory location `a` to it. Notice how pointers need types as well, so the compiler knows what can and cannot be done using the pointer (this is useful to the programmer as the compiler can highlight when there is a problem - for example, adding a character to an integer).

Now let us introduce the dereference operator `*` on line 15. We are not concerned with the contents of memory location `b` as a final result, but what the memory location in `b` actually points to (i.e., `b` holds a memory location value and it is this memory location we are interested in). If we remove the `*` then we can find out what is in memory location `b`. This should be the memory location where the variable `a` is located.

```cpp
 1 #include <iostream>
 2 
 3 using namespace std;
 4 
 5 int main () {
 6 
 7    int a = 100;
 8 
 9    cout << "Value in memory location a is " << a << "\n";
10 
11    cout << "Address of (the reference to) memory location a is "<< &a << "\n";
12 
13    int *b = &a;
14 
15    cout << "Value in memory location pointed to by b is " << *b << "\n";
16 
17 
18    cout << "Value in memory location b is " << b << "\n";
19    cout << b << " should be the same as the memory location for a: " << &a << "\n";
20    cout << "b is actually at memory location " << &b << endl;
```

pointers.cpp

We can create pointers that point to pointers! Well, it may seem strange but we sometimes need this if we are building a chain of references in memory (lists and the like). A pointer to a pointer is described using `**`. On line 21 we create a pointer `c` that points to the pointer `b`. We assign the value of `b`'s memory location value and then print it out so we can see what is going on.

The `cout` on line 24 can display interesting things like “what is the pointer value pointed at by `c`?” i.e., what is the value in `b`?

```cpp
21     int** c;
22 
23     c = &b;
24     cout << "c is now pointing to what b points to: " << **c << endl;
25 
26     cout << "c points to: " << *c << endl;
27 
28     cout << "c actually holds the value: " << c << endl;
29     cout << c << " is the same value as b's memory location: " << &b << endl;
30     cout << "c is actually at memory location " << &c << endl;
31 
32     cout << "c is eventually retrieving what is in memory location " << **&c << endl;
33     cout << "the value of a is in memory location " << &a << endl;
34 
35     int x;
36     cout << "Type in a number to finish" << "\n";
37     cin >> x;
38 
39     return 0;
40 }
```

pointers.cpp

One final note, we can’t write `*a` because we have not declared the type of the memory at location `100` (`a` contains 100). This is a very important aspect of pointers in that a pointer simply holds a memory location, it is this memory location whose type must be declared.

C++ Style

Pointers tend to be a source of many software bugs, even for programmers with significant experience with C++. If your program causes a ‘segmentation fault’ when executed, then the likelihood is that the error is to do with use of pointers. Languages like Java and C# do not allow the use of pointers at all, preferring to trade power for safety. C++ provides ‘reference’ types in addition to pointers which remove some of the dangers of pointers (more on references later). C++ Programmers are also encouraged to use several constructs provided by C++ libraries called ‘smart’ pointers (more on those later, too). Keep in mind, however, while smart pointers are an option for writing new software you must understand ‘naked’ pointers because of their prevalence in ‘legacy code’.

Exercises

1. Given that a pointer holds the value of a memory address, why is it permitted to add an integer data type value to a pointer variable but not a double data type?

2. Suppose we have a pointer to `float` data type which contains the memory address value 100. If we add the integer value 3 to this pointer, what will be the value of the pointer if `float` data types are 4 bytes in length?

3. Write a function called `swap` that takes two pointers to integer arguments and returns void. Inside the function, swap the values of the integers. Write a main function that calls this function and verify that the value of the integers has indeed been swapped after this function call.

4. Pointer arithmetic can be used to access specific values when a sequence of such values exist. Create an array of 10 integers numbered 1 to 10 with the syntax:
   `int arr[10] = {1,2,3,4,5,6,7,8,9,10};`
   Now create a function which accepts a integer pointer as its argument and returns void. Implement the function to print out the even numbers of the array using pointer arithmetic.