Accessing a Memory Address

Each variable in a program stores its contents in the computer's memory, and each chunk of the memory has an address number. For a given variable, the memory address can be accessed using an ampersand in front of the variable. To see an example of this, execute the following code which displays the hexadecimal (https://en.wikipedia.org/wiki/Hexadecimal) memory addresses of the variables `i` and `j`:

In [ ]:
```
#include <iostream>
using std::cout;

int main() {
    int i = 5;
    int j = 6;

    // Print the memory addresses of i and j
    cout << "The address of i is: " << &i << "\n";
    cout << "The address of j is: " << &j << "\n";
}
```

Compile & Execute | Explain

Loading terminal (id_1lhn359), please wait...

At this point, you might be wondering why the same symbol `&` can be used to both access memory addresses and, as you've seen before, pass references into a function. This is a great thing to wonder about. The overloading of the ampersand symbol `&` and the `*` symbol probably contribute to much of the confusion around pointers.

> **The symbols `&` and `*` have a different meaning, depending on which side of an equation they appear.**

This is extremely important to remember. For the `&` symbol, if it appears on the left side of an equation (e.g. when declaring a variable), it means that the variable is declared as a reference. If the `&` appears on the right side of an equation, or before a previously defined variable, it is used to return a memory address, as in the example above.

Try using the cell above to create new variables and print out their addresses!

Storing a Memory Address (int type)

Once a memory address is accessed, you can store it using a pointer. A pointer can be declared by using the `*` operator in the declaration. See the following code for an example:

In [ ]:
```
#include <iostream>
using std::cout;

int main()
{
    int i = 5;
    // A pointer pointer_to_i is declared and initialized to the address of i.
    int* pointer_to_i = &i;

    // Print the memory addresses of i and j
    cout << "The address of i is:          " << &i << "\n";
    cout << "The variable pointer_to_i is: " << pointer_to_i << "\n";
}
```

Compile & Execute | Explain

Loading terminal (id_okmu891), please wait...

As you can see from the code, the variable `pointer_to_i` is declared as a pointer to an `int` using the `*` symbol, and `pointer_to_i` is set to the address of `i`. From the printout, it can be seen that `pointer_to_i` holds the same value as the address of `i`.

Getting an Object Back from a Pointer Address

Once you have a pointer, you may want to retrieve the object it is pointing to. In this case, the `*` symbol can be used again. This time, however, it will appear on the right hand side of an equation or in front of an already-defined variable, so the meaning is different. In this case, it is called the "dereferencing operator", and it returns the object being pointed to. You can see how this works with the code below:

In [ ]:
```
#include <iostream>
using std::cout;

int main()
{
    int i = 5;
    // A pointer pointer_to_i is declared and initialized to the address of i.
    int* pointer_to_i = &i;

    // Print the memory addresses of i and j
    cout << "The address of i is:          " << &i << "\n";
    cout << "The variable pointer_to_i is: " << pointer_to_i << "\n";
    cout << "The value of the variable pointed to by pointer_to_i is: " << *pointer_to_i << "\n";
}
```

Compile & Execute | Explain

Loading terminal (id_azwdndx), please wait...

In the following example, the code is similar to above, except that the object that is being pointed to is changed before the pointer is dereferenced. Before executing the following code, guess what you think will happen to the value of the dereferenced pointer.

In [ ]:
```
#include <iostream>
using std::cout;

int main() {
    int i = 5;
    // A pointer pointer_to_i is declared and initialized to the address of i.
    int* pointer_to_i = &i;

    // Print the memory addresses of i and j
    cout << "The address of i is:          " << &i << "\n";
    cout << "The variable pointer_to_i is: " << pointer_to_i << "\n";

    // The value of i is changed.
    i = 7;
    cout << "The new value of the variable i is                  : " << i << "\n";
    cout << "The value of the variable pointed to by pointer_to_i is: " << *pointer_to_i << "\n";
}
```

Compile & Run | Explain

Loading terminal (id_4ooyod1), please wait...

As you can see, an object or variable can be changed while a pointer is pointing to it.

Try it Yourself!

Don't forget to experiment with the code in the cells above! Coding a few pointer variables is a great way to get used to the syntax.