Pointers to Other Object Types

Although the type of object being pointed to must be included in a pointer declaration, pointers hold the same kind of value for every type of object: just a memory address to where the object is stored. In the following code, a vector is declared. Write your own code to create a pointer to the address of that vector. Then, dereference your pointer and print the value of the first item in the vector.

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

int main() {
    // Vector v is declared and initialized to {1, 2, 3}
    vector<int> v {1, 2, 3};

    // Declare and initialize a pointer to the address of v here:

    // The following loops over each int a in the vector v and prints.
    // Note that this uses a "range-based" for loop:
    // https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-for-range
    for (int a: v) {
        cout << a << "\n";
    }

    // Dereference your pointer to v and print the int at index 0 here (note: you should print 1):

}
```

Compile & Execute | See Solution

Loading terminal (id_fr9qte8), please wait...

Hint: If you've created a pointer to `v`, say `pv`, you can retrieve `v` with `(*pv)`.

Passing Pointers to a Function

Pointers can be used in another form of pass-by-reference when working with functions. When used in this context, they work much like the references that you used for pass-by reference previously. If the pointer is pointing to a large object, it can be much more efficient to pass the pointer to a function than to pass a copy of the object as with pass-by-value.

In the following code, a pointer to an int is created, and that pointer is passed to a function. The object pointed to is then modified in the function.

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

void AddOne(int* j)
{
    // Dereference the pointer and increment the int being pointed to.
    (*j)++;
}

int main()
{
    int i = 1;
    cout << "The value of i is: " << i << "\n";

    // Declare a pointer to i:
    int* pi = &i;
    AddOne(pi);
    cout << "The value of i is now: " << i << "\n";
}
```

Compile & Execute | Explain

Loading terminal (id_699poaf), please wait...

When using pointers with functions, some care should be taken. If a pointer is passed to a function and then assigned to a variable in the function that goes out of scope after the function finishes executing, then the pointer will have undefined behavior at that point - the memory it is pointing to might be overwritten by other parts of the program.

Returning a Pointer from a Function

You can also return a pointer from a function. As mentioned just above, if you do this, you must be careful that the object being pointed to doesn't go out of scope when the function finishes executing. If the object goes out of scope, the memory address being pointed to might then be used for something else.

In the example below, a reference is passed into a function and a pointer is returned. This is safe since the pointer being returned points to a reference - a variable that exists outside of the function and will not go out of scope in the function.

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

int* AddOne(int& j)
{
    // Increment the referenced int and return the
    // address of j.
    j++;
    return &j;
}

int main()
{
    int i = 1;
    cout << "The value of i is: " << i << "\n";

    // Declare a pointer and initialize to the value
    // returned by AddOne:
    int* my_pointer = AddOne(i);
    cout << "The value of i is now: " << i << "\n";
    cout << "The value of the int pointed to by my_pointer is: " << *my_pointer << "\n";
}
```

Compile & Execute | Explain

Loading terminal (id_05lniic), please wait...

Your turn!

Try modifying the code above to get a better sense of how things work. If you are curious, try printing the memory address of `i` in the `main`, and the address of `j` inside the `AddOne` function. Since `j` is a reference to `i`, they should have the address!