Introduction to Pointers

Introduction

C++ supports special primitive data types called *pointers* that store, read from, and write to memory addresses. With pointers, you can access other variables indirectly or refer to blocks of memory generated at runtime.

Used correctly, pointers can perform wonders. As you'll see later on in CS106B and CS106X, pointers can define recursive data structures like binary trees, linked lists, and graphs. Pointers are also used extensively in systems programming, since they provide a way to write values to specific memory regions. However, pointer power comes at a high price, and it would not be much of a stretch to claim that almost all program crashes are caused by pointer errors.

This handout acts as a quick introduction to pointers, primarily to give a background for Handout #06, which discusses C strings. Please see the course reader for CS106B and CS106X for more information on pointers.

What is a Pointer?

At the basic level, a pointer is simply a variable that stores a data type and a memory address. For example, a pointer might encode that an `int` is stored at memory address 0x47D38B30, or that there is a `double` at 0x00034280. To declare a pointer, use the syntax `Type * variableName`, where `Type` is the type of variable the pointer will point to and `variableName` is the name of the newly-created variable. For example, to create a pointer to an `int` called `myPointer`, you'd write

```
int * myPointer;
```

The declarations `int* myPointer` and `int *myPointer` are also valid.

Pointers need not point solely to primitive types. Here's code that creates a variable called `myStrPtr` that points to a C++ `string`:

```
string * myStrPtr;
```

When declaring multiple variables in a single statement, be careful when creating pointers. Consider this line of code:

```
int *myPtr1, myPtr2; // Legal, but incorrect.
```

Here, `myPtr1` is declared as an `int *` pointer to integer, but the variable `myPtr2`, despite its name, is just a plain old `int`. The star indicating a pointer only sticks onto the first variable name it finds, so if you declare multiple pointers, you need to preface each with a star, as in

```
int *myPtr1, *myPtr2; // Legal and correct.
```

Initializing a Pointer

When you create a pointer using the above steps, you are simply declaring a variable. The pointer does not point to anything. Like all other primitive types, the pointer will be holding garbage data and could be pointing anywhere in memory. Thus, before you attempt to work with a pointer, you must be sure to initialize it.

Initializing a pointer simply means assigning it the address of a variable. When this happens, the pointer is said to *point* to the variable in the address, called the *pointee*. While there are many ways to set up pointees (as you'll see later in this handout), perhaps the simplest is to have the pointer point to a local variable declared on the stack. For example, consider this code snippet:

```
int myInteger = 137;
int * myIntPointer;
```

We'd like to have `myIntPointer` point to the variable `myInteger`. However, we cannot simply write `myIntPointer = myInteger`. `myIntPointer` is an `int *`, not a regular `int`, so the assignment isn't defined. Since pointers store memory addresses, to make `myIntPointer` point to `myInteger`, we have to assign `myIntPointer` the memory address of `myInteger`. Given any variable, you can obtain a pointer to that variable using the "address-of" operator by prefacing the name of the variable with an `&`. For example, we can write `&myInteger` to return the address of `myInteger` as an `int *`. Therefore, to make `myIntPointer` point to `myInteger`, we'd write

```
myIntPointer = &myInteger; // Pointer now holds the address of myInteger.
```

Note that the assignment only works because `myIntPointer` is an `int *` and `myInteger` is an `int`. In general, you cannot use pointers of mixed types in expressions,* so code like the following will not compile:

```
double myDouble = 2.71828;
int *myIntPtr = &myDouble; // ERROR: Can't convert from double * to int *!
```

You can also initialize a pointer by making it point to the same variable another pointer points to. In other words, once you have a pointer to an address in memory, you can copy that pointer as many times as you'd like. For example, here's code to initialize a pointer to a local variable, then initialize another pointer to point to the same location:

```
int myInteger = 137;
int *myIntPointer = &myInteger;
int *myOtherIntPtr = myIntPointer;
```

Note that when setting up `myIntPointer` to point to `myInteger`, we assigned it the value `&myInteger`, using the address-of operator. However, when setting up `myOtherIntPtr`, we simply assigned it the value of `myIntPointer`. Had we written `myOtherIntPtr = &myIntPointer`, we would have gotten a compilation error, since `&myIntPointer` is the address of the `myIntPointer` variable, not the address of its pointee. In general, use straight assignment when assigning pointers to each other, and use the address-of operator when assigning pointers the addresses of other variables.

---

* When we cover inheritance, you'll see an exception to this rule. If you are truly brave, you can also use the special pointer type `void *` as a "universal pointer" that can be assigned to any pointer type you wish.

Dereferencing a Pointer

We now can initialize pointers to point to memory addresses, but what else can we do with them? The major pointer operation is the *pointer dereference*, a way to access and modify the contents of the pointee. In the above example, once `myIntPointer` is initialized to point to `myInteger`, we can dereference `myIntPointer` to read and write the value of `myInteger`.

To dereference a pointer, you preface the name of the pointer with a star, as in `*myIntPointer`. The dereferenced pointer then acts identically to the variable it's pointing to. For example, consider the following code snippet:

```
int myInteger = 137;
int *myIntPointer = &myInteger;
*myIntPointer = 42;
cout << myInteger << endl;
```

In the first two lines, we create a variable called `myInteger` and a pointer `myIntPointer` that points to it. In the third line, `*myIntPointer = 42`, we dereference the pointer `myIntPointer` to get a reference to the variable it refers to, in this case `myInteger`, and assign it the value 42. In the final line we print the value 42, since we indirectly overwrote the contents of `myInteger` in the previous line.

Admittedly, the star notation with pointers can get a bit confusing since it means either "declare a pointer" or "dereference a pointer." However, with a little practice you'll be able to differentiate between the two.

Before you deference a pointer, you must make sure that you've set it up correctly. Consider the following example:

```
int myInteger = 137;
int *myIntPointer; // Note: myIntPointer wasn't initialized!
*myIntPointer = 42;
cout << myInteger << endl;
```

This code is identical to the above example, except that we've forgotten to initialize `myIntPointer`. As a result, the line `*myIntPointer` will result in *undefined behavior*. When `myIntPointer` is created, like any other primitive type, it initially holds a garbage value. As a result, when we write `*myIntPointer = 42`, the program will almost certainly crash because `myIntPointer` is pointing to an arbitrary memory address.

`new` and `delete`

Up to this point, we've only used pointers to refer to other variables declared on the stack. However, there's another place to store variables called the *heap*, a region of memory where variables can be allocated and deallocated at runtime. While right now it might not be apparent why you would choose to allocate memory in the heap, as you'll see later in CS106B, CS106X, and CS106L, heap storage is critically important for creating powerful programs.

Heap storage gives you the ability to create and destroy variables as needed during program execution. If you need an integer to hold a value, or want to create a temporary `string` object, you can use the C++ `new` operator to obtain a pointer to one. The `new` operator sets aside a small block

of memory to hold the new variable, then returns a pointer to the memory. For example, here's some code that allocates space for a `double` and stores it in a local pointer:

```
double *dynamicDouble = new double;
*dynamicDouble = 2.71828; // Write the value 2.71828
```

Unlike other languages like Java, C++ does not have support for automatic garbage collection. As a result, if you allocate any memory with `new`, you must use the C++ `delete` keyword to reclaim the memory. Here's some code that allocates some memory, uses it a bit, then deletes it:

```
int *myIntPtr = new int;
*myIntPtr = 137;
cout << *myIntPtr << endl;
delete myIntPtr;
```

Note that when you write `delete myIntPtr`, you are *not* destroying the `myIntPtr` variable. Instead, you're instructing C++ to clean up the memory pointed at by `myIntPtr`. After writing `delete myIntPtr`, you're free to reassign the `myIntPtr` to other memory and continue using it.

There are several important points to consider when using `delete`. First, calling `delete` on the same dynamically-allocated memory twice results in undefined behavior and commonly corrupts memory your program needs to function correctly. Thus you must be very careful to balance each call to `new` with one and exactly one call to `delete`. This can be tricky. For example, consider the following code snippet:

```
int *myIntPtr1 = new int;
int *myIntPtr2 = myIntPtr1;
delete myIntPtr1;
delete myIntPtr2;
```

At first you might think that this code is correct, since both `myIntPtr1` and `myIntPtr2` refer to dynamically-allocated memory, but unfortunately this code double-`delete`s the dynamically-allocated memory. Since `myIntPtr1` and `myIntPtr2` refer to the same dynamically-allocated memory, calling `delete` on both variables will try to clean up the same memory twice.

Second, after you call `delete` to clean up memory, accessing the reclaimed memory results in undefined behavior. In other words, calling `delete` indicates that you are done using the memory and do not plan on ever accessing it again. While this might seem simple, it can be quite complicated. For example, consider this code snippet:

```
int *myIntPtr1 = new int;
int *myIntPtr2 = myIntPtr1;
delete myIntPtr2;
*myIntPtr1 = 137;
```

Since `myIntPtr1` and `myIntPtr2` both refer to the same region in memory, after the call to `delete myIntPtr2`, both `myIntPtr1` and `myIntPtr2` point to reclaimed memory. However, in the next line we wrote `*myIntPtr1 = 137`, which tries to write a value to the memory address. This will almost certainly cause a runtime crash.

`new[]` and `delete[]`

Commonly, when writing programs to manipulate data, you'll need to allocate enough space to store an indeterminate number of variables. For example, suppose you want to write a program to play a variant on the game of checkers where the board can have any dimensions the user wishes. Without using the `Grid` ADT provided in the CS106 libraries, you would have a lot of trouble getting this program working, since you wouldn't know how much space the board would take up.

To resolve this problem, C++ has two special operators, `new[]` and `delete[]`, which allocate and deallocate blocks of memory holding multiple elements. For example, you could allocate space for 400 integers by writing `new int[400]`, or a sequence of characters twelve elements long with `new char[12]`.

The memory allocated with `new []` stores all the elements in sequential order, so if you know the starting address of the first variable, you can locate any variable in the sequence you wish. For example, if you have fourteen `int`s starting at address 1000, since `int`s are four bytes each, you can find the second integer in the list at position 1004, the third at 1008, etc. Therefore, although `new[]` allocates space for many variables, it returns only the address of the first variable. Thus you can store the list using a simple pointer, as shown below:

```
int *myManyInts = new int[137];
```

Once you have a pointer to a dynamically-allocated array of elements, you can access individual elements using square brackets `[]`. For example, here's code to allocate 200 integers and set each one equal to its position in the array:

```
const int NUM_ELEMS = 200;
int *myManyInts = new int[NUM_ELEMS];
for(int i = 0; i < NUM_ELEMS; i++)
    myManyInts[i] = i;
```

As with regular `new`, you should clean up any memory you allocate with `new[]` by balancing it with a call to `delete[]`. You should not put any numbers inside the brackets of `delete []`, since the C++ memory manager is clever enough to keep track of how many elements to delete. For example, here's code to clean up a list of `int`s:

```
int *myInts = new int[100];
delete [] myInts;
```

Note that `delete` and `delete []` are *completely different operators*. That is, you must be extremely careful not to clean up an array of elements using `delete`, nor a single element using `delete []`. Doing so will have disastrous consequences for your program and will almost certainly cause a crash at some point.

More to Explore

This introduction to pointers has been rather brief and does not address several important pointer topics. While Handout #06 on C strings will cover some additional points, you should consider reading into some of these additional topics for more information on pointers. Also, please be sure to consult your course readers for CS106B/X.

1. **Static arrays**: C and C++ let you allocate arrays of elements on the stack as well as in the heap. Arrays are relatively unsafe compared to ADT classes like `Vector`, but are a bit faster and arise in legacy code. Arrays are strongly related to pointers, and you should consider looking into them if you plan to use C++ more seriously.
2. **Smart pointers**: Pointers are difficult to work with – you need to make sure to `delete` or `delete []` memory once and exactly once, and must be on guard not to access deallocated memory. As a result, C++ programmers have developed objects called *smart pointers* that mimic standard pointers but handle all memory allocation and deallocation behind the scenes. Smart pointers are easy to use, reduce program complexity, and eliminate all sorts of errors. However, they do add a bit of overhead to your code. If you're interested in seriously using C++, be sure to look into smart pointers.

Practice Problems

1. Give two reasons why you cannot assign a `int *` to a `double *`.
2. Is the variable declaration `int **myPtr` legal? If so, what does it mean? If not, why not?
3. What's wrong with this code snippet?
```
int myInteger = 137;
int *myIntPtr = myInteger;
*myIntPtr = 42;
```
4. Is this code snippet legal? If so, what does it do? If not, why not?
```
int myInteger = 0;
*(&myInteger) = 1;
```
5. When using `new[]` to allocate memory, you can access individual elements in the sequence using the notation `pointer[index]`. Recall that `pointer[index]` instructs C++ to start at the memory pointed at by `pointer`, march forward `index` elements, and read a value. Given a pointer initialized by `int *myIntPtr = new int`, what will `myIntPtr[0] = 42` do? What about `myIntPtr[1]`?