Pointers and References

Introduction

C++ supports special primitive data types called *pointers* and *references* that alias other variables or memory locations. With pointers and references, you can access other variables indirectly or refer to blocks of memory generated at runtime.

Used correctly, pointers and references can perform wonders. As you'll see later in CS106B/X, pointers are crucial in 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 and reference errors.

This handout acts as a quick introduction to pointers and references, primarily to give a background for Handout #06, which discusses C strings. You will probably want to refer to the CS106B/X course reader for more information on pointers and references.

What is a Reference?

As you have learned in CS106B/X, C++ has two parameter-passing mechanisms. The first, *pass-by-value*, passes parameters by initializing the parameter as a copy of the argument. For example, consider the following function:

```
void DoSomething(int x)
{
    x++;  // x is passed by value, so no changes to outside world occur.
}
```

Here, when we call the `DoSomething` function, the parameter `x` is initialized as a copy of the argument to the function, so the line `x++` will not affect the value of any variables outside the function. That is, given the following code:

```
int x = 137;
cout << x << endl;
DoSomething(x);
cout << x << endl;
```

Both the first and second `cout` statements will print the value 137.

The second version of parameter-passing in C++ is *pass-by-reference*, where arguments to the function can be modified by the function. As an example, the following function takes its parameter by reference, so local changes propagate to the caller:

```
void MutateParameter(int& x)
{
    x++;  // x will be updated in the calling function
}
```

Using `MutateParameter` in the following snippet will cause the first `cout` statement to print 137, but the second to print 138:

```
int x = 137;
cout << x << endl;
MutateParameter(x);
cout << x << endl;
```

Notice that the syntax for passing `x` by reference was to declare `x` as a `int& x` rather than simply `int x`. Interestingly, you can use this syntax in other places in your C++ programs to declare variables called *references* which, like parameters that are passed by reference, transparently modify other program variables.

To see how references work, consider the following code snippet:

```
int myVariable = 137;
int& myReference = myVariable; // myReference is a reference to myVariable
```

In this code snippet, we initially declare a variable of type `int` called `myVariable`, then initialize it to 137. In the next line, we create a variable `myReference` of type `int&` and initialize it to the variable `myVariable`. From this point on, `myReference` acts a reference to `myVariable` and like a reference parameter in a function, any changes to `myReference` will update the value of `myVariable`. For example, consider the following code:

```
cout << myVariable << endl;   // Prints 137
cout << myReference << endl;  // Prints 137, the value of myVariable
myReference = 42;             // Indirectly updates myVariable
cout << myVariable << endl;   // Prints 42
cout << myReference << endl;  // Prints 42
```

Here, we print out the values of `myVariable` and `myReference`, which are the same because `myReference` is a reference to `myVariable`. Next, we assign `myReference` the value 42, and because `myReference` is a reference, it updates the value of `myVariable`, as shown with the final two `cout`s.

It is perfectly legal to have several references to the same variable, as shown here:

```
int myInt = 42;
int& ref1 = myInt, &ref2 = myInt; // ref1, ref2 now reference myInt
```

Notice that when declaring several references on the same line, it is necessary to prefix each of them with an ampersand. This is a quirk that helps make the syntax more compatible with that of the C programming language. Forgetting this rule is the source of many a debugging nightmare.

Because references are aliases for other variables, there are several restrictions on references that are not true of other types. First, references must be initialized when they are declared. For example, the following code is illegal:

```
int& myRef; // Error: need to initialize the reference!
```

This restriction at times can be a bit vexing, but in the long run will save you a good deal of trouble. Because references always must be initialized, when working with references you can be more certain that the variable they're referring to actually exists.

Second, because references are aliases, you cannot initialize a reference using a constant value or with the return value of a function. For example, both of the following are illegal:

```
int& myRef1 = 137; // Error: Can't initialize a reference to a constant.
int& myRef2 = GetInteger(); // Error: Can't initialize to a return value.
```

The reason for these restrictions is simple. Suppose that we could legally initialize `myRef` to 137. What would happen if we wrote the following?

```
myRef = 42;
```

Since `myRef` is a reference, when we change the value of `myRef`, we are actually changing the value of some other variable. But if `myRef` is initialized to 137, then this code would be equivalent to writing

```
137 = 42;
```

Which is clearly nonsensical.* Similarly, we cannot initialize `myRef` to the return value of a function, since otherwise we could try to write code to the effect of

```
GetInteger() = 42;
```

Which is similarly illegal.

Technically speaking, a reference can only be initialized with what is known as an *lvalue*, something that can legally be put on the left-hand side of an assignment statement. For example, any variable is an lvalue, since variables can be assigned to. Global constants, on the other hand, are not lvalues because we cannot put a constant on the left-hand side of an assignment statement.

The final point about references is that once a reference has been initialized, it is impossible to change where that reference refers to. That is, once a reference has been bound to a value, we cannot "rebind" the reference to some other object. For example, consider the following code:

```
int myInt = 137, myOtherInt = 137;
int& myRef = myInt, &myOtherRef = myOtherInt;
myRef = myOtherRef;
```

In the last line, we assign `myRef` the value of `myOtherRef`. This does *not* cause `myRef` to begin referring to the same variable as `myOtherRef`; instead, it takes the value of the variable referenced by `myOtherRef` and stores it in the variable referenced by `myRef`.

Pointers

Put simply, a pointer is 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. At a higher level, you can think of a pointer as a more powerful version of a reference. Like references, pointers allow you to indirectly refer to another variable or memory location. Unlike a reference, however, it is possible to change what object a pointer aliases. This versatility is both one of the most useful and one of the most vexing aspects of pointers, since you will have to take care to distinguish between the pointer and its *pointee*, the object at which it points.

---

* Interestingly, however, some older programming languages (notably Fortran) used to allow assignments like this, which would lead to all sorts of debugging headaches.

To declare a pointer, we 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`, you could write

```
int* myPointer;
```

The whitespace around the star in this declaration is unimportant – the declarations `int * myPointer` and `int *myPointer` are also valid. Feel free to use whichever of these you feel is most intuitive, but be sure that you're able to read all three because each arises in professional code.

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;
```

As with references, when declaring multiple pointers in a single declaration, you will need to repeat the star for each variable. For example, consider the following code snippet:

```
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 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 <u>does not point to anything</u>. 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. Notice how pointers differ in this case from references – it is illegal to leave a reference uninitialized, but it is legal (but unsafe) to leave pointers uninitialized.

Initializing a pointer simply means assigning it the *address* (memory location) of the object you want it to point to. When this happens, the pointer is said to *point* to the variable at the address, called the *pointee*. While there are many ways to set up pointees, 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`. Now, if `myIntPointer` were a reference, we could simply write `myIntPointer = myInteger`. However, because `myIntPointer` is a pointer and not a reference, this code would be illegal. When working with pointers, to cause a pointer to point to a location, you must explicitly assign the pointer the address of the object to point to. That is, if we want `myIntPointer` to point to `myInteger`, we need to somehow get the address of the `myInteger` variable. To do this, we can use C++'s *address-of operator*, the `&` operator. For example, here is the code we could use to initialize `myIntPointer` to point to `myInteger`:

```
myIntPointer = &myInteger; // myIntPointer now points to myInteger.
```

It is an unfortunate choice of syntax that C++ uses `&` both to define a variable as a reference type and to get the address of a variable. The difference has to do with context – if `&` is used to declare a variable, it makes that variable a reference. Otherwise, `&` is used to take the address of a variable.

Here we see one major difference between pointers and references. When working with references, to initialize a reference to refer to a variable, we simply used a straight assignment statement. However, when using pointers, we have to explicitly take the address of the object we're pointing at. Why the difference? The main reason is that when working with pointers, there is an explicit difference between the pointer variable and the object being pointed at. References are in some sense not "true" variables because when using a reference, we are always referring to the object the reference aliases rather than the reference itself. That is, if we have the following code:

```
int myInt;
int& myRef = myInt;
myRef++;
```

In the final line, `myRef++`, we are not incrementing `myRef`, but rather the object that it refers to.

Pointers, on the other hand, are true variables that can be reassigned, modified, and transformed without changing the value of the object that they point at. For example, given the following code:

```
int myInt1, myInt2;
int* myPtr;
myPtr = &myInt1;
myPtr = &myInt2;
```

In the last two lines we first make `myPtr` point to `myInt1`, then reassign it to point to `myInt2`. Notice, however, that by saying `myPtr = &myInt1` and `myPtr = &myInt2`, we did **not** change the value of the object that `myPtr` points at. Instead, we simply changed what object the `myPtr` variable was pointing at. If we wanted to change the value of the object pointed at by `myPtr`, we would have to *dereference* it, as explained in a later section.

As with references, pointers are statically-typed and without explicitly subverting the type system can only be made to point to objects of a certain type. For example, the following code is illegal because it tries to make a `int*` point to a `double`:

```
double myDouble = 2.71828;
int* myIntPtr = &myDouble; // Error: Can't store a double* in an int*.
```

Another way to initialize a pointer is to have the pointer begin pointing to the same location as another pointer. For example, if we have a pointer named `myPtr` pointing to some location and we want to create a second pointer `myOtherPtr`, we can legally set `myOtherPtr` to point to the same location as `myPtr`. This is known as *pointer assignment* and, fortunately, has clean syntax. For example, here's code that makes two pointers each point to the same integer by using pointer assignment:

```
int myInteger = 137;
int* myPtr, *myOhterPtr;
myPtr = &myInteger; // Assign myPtr the address of myInteger
myOtherPtr = myPtr; // myOtherPtr now points to the same object as myPtr
```

Note that when setting up `myPtr` to point to `myInteger`, we assigned it the value `&myInteger`, the address of the `myInteger` variable. However, when setting up `myOtherPtr`, we simply assigned it the value of `myPtr`. Had we written `myOtherPtr = &myPtr`, we would have gotten a compilation error,

since `&myPtr` is the location of the `myPtr` variable, not the location 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.

Dereferencing a Pointer

We now can initialize pointers, but how can we access the object being pointed at? This requires a *pointer dereference*, which follows a pointer to its destination and yields the object being pointed at.

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* myPtr = &myInteger;

*myPtr = 42; // Dereference myPtr to get myInteger, then store 42.
cout << myInteger << endl;
```

In the first two lines, we create a variable called `myInteger` and a pointer `myPtr` that points to it. In the third line, `*myPtr = 42`, we dereference the pointer `myPtr` to get a reference to pointee, 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." As with the `&` operator, 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.

Almost all bugs that cause programs to crash stem from dereferencing invalid pointers. You will invariably run into this problem when working with C++ code, so remembering to initialize your pointers will greatly reduce the potential for error.

The `->` Operator

When working with pointers to objects (i.e. `string*`), the pointer dereference operator sometimes can become a source of confusion. For example, consider the following program, which tries to create a pointer to a `string` and then return the length of the `string` being pointed at:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << *myPtr.length() << endl; // Error: See below.
```

This code at first might seem valid – we initialize `myPtr` to point to `myString`, and then try to print out the length of the object being pointed at. Unfortunately, this code is invalid because the `*` operator binds too tightly. The problem is that C++ interprets `*myPtr.length()` as `*(myPtr.length())`; that is, dereferencing the expression `myPtr.length()`. This is a problem because `myPtr.length()` is illegal – `myPtr` is a pointer to a `string`, not a `string` itself, and we cannot apply the dot operator to it.

The proper version of the above code looks like this:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << (*myPtr).length() << endl; // Correct, but there's a better option
```

Here, we've put parentheses around the expression `*myPtr` so that C++ treats `(*myPtr).length()` correctly. But this code is difficult to read and somewhat bulky – all we want to do is to go to the object pointed by `myPtr` and access the `length` member function.

Fortunately, C++ has a special operator called the *arrow operator* that can be useful in these circumstances. Just as you can use the dot operator to access properties of regular objects, you can use the arrow operator to access properties of an object being pointed at. Here is another version of the above code, rewritten using the arrow operator:

```
string myString = "This is a string!";
string* myPtr = &myString; // Initialize myPtr;
cout << myPtr->length() << endl; // Use arrow to select length.
```

Pointer Comparison

Sometimes it is useful to compare two pointers to see if they are equal or unequal. To do so, you can use the == and `!=` operators, as shown here:

```
int myInt1, myInt2;
int* myPtr1 = &myInt1, *myPtr2 = &myInt2;

if(myPtr1 == myPtr2) // Check if the pointers point to the same location
    cout << "Pointers are equal" << endl;
if(myPtr1 != myPtr2) // Check if the pointers point to different locations
    cout << "Pointers are unequal" << endl;
```

Note that the == and `!=` operators check to see if the *pointers* are equal, not the *pointees*. Thus if two pointers are pointing to different objects that have the same value, == will return false. To see if the pointers point to the same value, compare the values of the dereferenced pointers rather than the pointers themselves.

Null Pointers

In some cases, you may want to indicate that a pointer does not actually point to anything. For situations like this, you can assign pointers the special value `NULL`. Any pointer can be assigned the value `NULL`, no matter what the type of that pointer is. For example:

```
int* myIntPtr = NULL;
double* myDoublePtr = NULL;
```

Because `NULL` indicates that a pointer does not actually point at anything, dereferencing a `NULL` pointer leads to undefined behavior. In most cases, it crashes the program. Thus, before you dereference a pointer, make sure that you know it is non-`NULL`. For example, here's code to print out the value a pointer points to if the pointer is non-`NULL`, and an error otherwise:

```
if(myPtr == NULL) // Check if myPtr points to NULL
    cout << "Pointer is NULL!" << endl;
else
    cout << "Value is: " << *myPtr << endl;
```

Unlike Java, in C++ pointers **do not** default to storing `NULL`. This means that if you forget to initialize a pointer, it does not necessarily point to `NULL` and in fact probably points to a random memory location. As a rule of thumb, if you declare a pointer but do not immediately initialize it, for safety reasons you should assign it the value `NULL` so that you can use `NULL`-checks later in your program.

Unlike other languages like Java or Visual Basic, the value `NULL` is not a C++ keyword. To use `NULL`, you need to include the header file `<cstddef>`. However, since virtually every C++ standard library header references `<cstddef>`, in most cases you can ignore the header file.

`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 created 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/X and CS106L, heap storage is critical in almost all nontrivial 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
```

Note that because `new` yields a pointer you do not need to use the address-of operator to assign a pointer to memory created with `new`. In fact, it is illegal to do so, so code to this effect:

```
double* dynamicDouble = &(new double); // Error: Illegal to use & here.
```

Unlike other languages like Java, C++ does not have 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 object. Since `myIntPtr1` and `myIntPtr2` refer to the same object, 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 different operators and cannot be substituted for one another. 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.

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. **Automatic 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.
3. **`nullptr`**: C++ is an constantly-growing language and a new revision of C++, called "C++0x," is currently being developed. In the new version of the language, the value `NULL` will be superseded by a special keyword `nullptr`, which functions identically to `NULL`. Once this new language revision is released, be prepared to see `nullptr` in professional code.

Practice Problems

1. It is illegal in C++ to have a reference to a reference. Why might this be the case? *(Hint: is there a meaningful distinction between the reference and what it refers to?)*
2. However, it *is* legal C++ to have a pointer to a pointer. Why might this be the case? *(Hint: is there a meaningful distinction between the pointer and what it points to?)*
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] = 42`?