MIT OpenCourseWare
http://ocw.mit.edu

6.096 Introduction to C++
January (IAP) 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

6.096 **Lecture 5 Notes**

Pointers

Pointers can be one of the most confusing and dangerous parts of C++, but if used well, they are incredibly powerful, and form the basis of much of Object-Oriented Programming (a technique we'll be seeing very soon).

Variables and Memory:

When you declare a variable, the computer associates the name you choose with a particular location in memory and stores a value there. For instance, when you write int x;, the compiler might assign x memory location 2597 (or 000A25 in hexadecimal, the base-16 representation often used for memory locations).

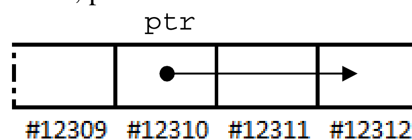
When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

C++ allows us to perform either one of these steps independently on a variable with the & and * operators:

- &x evaluates to the address of x in memory.
- *(&x) takes the address of x and *dereferences* it – it retrieves the value at that location in memory.

As with arrays, it is often helpful to visualize memory addresses by using a row of adjacent cells to represent memory locations, as below. Each cell represents 1 byte. The dot-arrow notation indicates that the variable ptr, stored in location 12310, points to the value in memory location 12312.



Pointers and their behavior:

Pointers are just variables storing numbers – but those numbers happen to be memory addresses, usually addresses of other variables. A pointer that stores the address of some variable x is said to *point to* x. To access the value of x, we'd need to dereference the pointer.

Motivating pointers:

Pointers allow us to manipulate data much more flexibly. Particularly when we start discussing more complex data types – types that consist of large composites of the types we've seen so far – we'll see that being able to just pass around pointers to the data can be much more useful and efficient than passing around copies of the data itself.

Just a taste of what we'll be able to do with pointers:

- More flexible pass-by-reference
- Manipulate complex data structures efficiently
    - This is especially useful for manipulating data that, unlike arrays, isn't stored in memory all at once, and so is scattered in various places in memory
- Use polymorphism – calling functions on data without even knowing exactly what kind of data it is (more on this later in the course)

Pointer usage:

To declare a pointer variable named x that points to an integer variable named y:

```
int *x = &y;
```

&y returns the address of the integer variable y, and int *x declares a pointer to an integer value which we set to the address of y.

The general scheme for declaring pointers is:

```
data_type *pointer_name; // or data_type *pointer_name = initial_value;
```

Just like any other data type, we can pass pointers as arguments to functions. The data type of a pointer to an integer is int *, so the same way we'd say void func(int x) {…}, we can say void func(int *x){…}.

Once a pointer is declared, we can *dereference* it with the `*` operator to access its value:

```
cout << *x; // Prints the value pointed to by x, which in the above example
would be y's value
```

Without the `*` operator, the identifier x refers to the pointer itself, not the value it points to:

```
cout << x; //Outputs a hexadecimal representation of the memory address of y
```

Using the * operator:

The usage of the * operator with pointers can be confusing. The operator is used in two different ways:

1. When declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer, and not, say, an int or a char.
2. When using a pointer that has been set to point to some value, * is placed before the pointer name to dereference it – to access or set the value it points to.

References:

We discussed passing arguments by reference in the lecture on functions. But that is not the only context in which we can declare references. Just like when we write func(y) to call a function defined as void

func(int &x) {…}, x becomes another name – an alias – for the value of y in memory, we can declare a reference variable locally, as well:

```
int y;

int &x = y;        // Makes x an alias – a reference – to y
```

After these declarations, changing x will change y and vice versa, because they are two names for the same thing.

References basically function like pointers that are dereferenced every time they are used. Having a reference named x and using x in expressions is functionally identical to having a pointer named x and using *x in the expressions. However, because you cannot directly access the memory location stored in a reference (as it is dereferenced every time you use the reference), you cannot change the location to which a reference points, whereas you can change the location to which a pointer points.

Pointers and arrays:

The name of an array is in fact just a pointer to the first element in the array. Saying array[3] tells the compiler to start with the memory address of the first element in array, jump three elements down in memory, and dereference. In other words, it says to return the element that is 3 away from the starting element.

This explains why arrays are always passed by reference – passing an array is really passing a pointer.

This also explains why array indices start at 0: when we want the first element of the array, we want the element that is 0 away from the start of the array.

Pointer arithmetic:

Pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory.

Say we have a pointer ptr to some element in an array of long integers. When we increase ptr by 1, we don't just want to move it to the next byte in memory, since it's pointing to a value that takes up multiple bytes. We want to move it to the next element in the array. The C++ compiler automatically takes care of this, using the appropriate step size for adding to and subtracting from pointers.

Thus, ptr++ moves ptr to point to the next element of the array.

Similarly, ptr2 - ptr (assuming ptr2 points to some other element of the array) gives the number of array elements between ptr2 and ptr.

All addition and subtraction operations on pointers use the appropriate increment size.

Pointer offset notations:

Because of the interchangeability of pointers and arrays, subscript notation (the square brackets with a number inside) can be used with pointers as well as arrays. These two possibilities are referred to as *array-subscript* and *pointer-subscript notations*.

An alternative way to express the same thing is *pointer-offset notation*, in which you explicitly add your offset to the pointer and dereference the resulting address. For instance, an alternate way to express a[3] (where a is some pointer) is *(a+3). Both of these expressions are interpreted by the compiler in the same way.

Passing by reference with pointers:

Example:

```
void squareByRef( int *numPtr ) {
        *numPtr = *numPtr * *numPtr;
}
```

This function, when called on a number, uses pointer syntax to square the value of the variable whose address is passed in.

Null, uninitialized, and deallocated pointers:

Some pointers do not point to valid data and cannot be dereferenced. Any pointer set to 0 is called a *null pointer*, and since there is no memory location 0, it is an invalid pointer. Any attempt to dereference it will result in a runtime error.

Dereferencing pointers to data that has been erased from memory also usually causes runtime errors.

Example:

```
int *myFunc() {
        int test = 4;
        return &test;
}
```

Any pointer set to the return value of this function will not be valid once the function exits.

`char *` and `char[]`:

Arrays of chars and pointers to chars are interchangeable. A string is really just an array of characters. (If it's a hard-coded string that actually appears in your code, it is loaded into program memory and assigned a memory address at program startup.) When you set a char * to a string, you are really setting it to point to the first character in the array that holds the string.

You cannot modify a string declared as a constant (e.g. char *str = "Hi!";). To do so is either a syntax error or a runtime error, depending on how you try to do it. You can, however, modify an array of characters. The way to declare such an array follows the usual array declaration syntax:

```
char timbuktu[] = {'t', 'i', 'm', 'b', 'u', 'k', 't', 'u', '\0' };
```

The last character in this array is the *null character*, whose ASCII code is 0. It is required at the end of every string to mark the end of the string in memory (though it is inserted automatically for any string in quotes). This is called a *null-terminated string*.

Function pointers:

So far we've had a strong distinction between program logic and data – data is variables, program instructions are functions. But functions to the computer are just sequences of instructions stored in a location in memory. If we store that location in a variable, we get a pointer to a function.

See Lab 4 for the syntax of declaring and using function pointers.

const pointers:

There are 2 places the const keyword can be placed within a pointer variable declaration. The reason for this is that there are 2 different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

const int *ptr;

declares a changeable pointer to a constant integer. The value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.

int * const ptr;

declares a constant pointer to changeable integer data. The value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.

const int * const ptr;

forbids changing either the address ptr contains or the value it points to.

String objects:

There are numerous functions in the C++ standard library for dealing with char *'s. Many of these involve pre-allocating memory of a certain quantity, and so on. This gets to be unwieldy for longer function calls.

Don't worry about syntax, but accept for now that you can declare a variable of type string (once you've included the cstring standard header.

A string can be assigned to any string of any length.

To call one of the string-related functions on a string str, we type, str.funcName(…).

The mathematical operators (+, ==, etc.) are also available for use on strings.

▪