MECH 215

Lecture 5

Pointers

Pointers

- The pointer data type stores memory locations rather than raw data.
- The pointer is declared using the following format:
  - Data_type * pointer_name;
  - Data type can be the usual int, float, double, char, etc…
- To assign an address (memory location) to a pointer we would use the following format:
- Pointer_name=&variable_name;
- This will load the memory address of the variable_name into the pointer.

Pointers

- We can access the memory location that the pointer holds with the derefencing operator *.
- For example we could assign a value to the memory location to another location using: *pointer_name=5;
- This would load the value 5 into the memory location held by pointer_name.
- Note that this is different than changing the value stored in the pointer.

Pointers

- int* some_point; Define Pointer some_point which points to address of integer
- float* over_rainbow; Define pointer over_rainbow which points to address of integer
- int some_value; Defines a memory location that will hold an integer
- float some_where; Defines a memory location that will hold a floating point value

Pointers

- some_point=&some_value; Loads pointer with memory location some_value
- some_where=0.25; Loads the memory location with a value
- cout <<"Mem Location of some_value " <<&some_value <<endl;
- cout <<"Contents of some_point " <<some_point<<endl;
- cout <<"Mem location of some_point " <<&some_point<<endl;
- The screen prints out the memory location of some _value and some point which are different.

Pointers

- some_value=4; loads 4 into memory location
- cout<<"Contents of some_value before deference= "<<some_value<<endl; Will show 4 in this location
- *some_point=5; Loads 5 into memory location (dereference)
- cout <<"Contents of some value after deference= "<<some_value<<endl; Will show a 5 in this location.

Pointers

- int* some_point;
- int some_array[4]={1,2,3,4};
- //
- // Pointer hold memory address
- // in some other memory location. Hold the location
- // of the data not the data itself.
- some_point=&some_array[0];
- for (int index1=0; index1<4; index1++)
- {
- cout<< "Array index = "<<index1 << " Holds "<<some_array[index1]<<endl;
- cout<< "Pointer "<<some_point<<" Holds "<<*some_point<<endl;
- some_point++;
- }

Pointers

- What does the above part of a program do?
- The first two lines declare a pointer called some_point which will hold the address for an integer.
- The next line declares and assigns values to a 4 element array that stores integer values.
- Next the pointer is assigned the memory location of the start of the array (note you could also use some_point=some_array;)
- We then run through a loop and access each element of the array using the array index as well as using dereferencing with the pointer.

Pointers

- We can see that the value in the pointer increases by four.
- But we only increased the index by 1! Some_point++ was written but for a pointer the next location in memory is based on the size that a particular data type requires.
- In this case the integer requires four bytes of memory to hold one value. The next free location that the pointer can access is 4 bytes away from the present value.

Pointers

- Array index = 0 Holds 1
- Pointer 0x7fff55a57d00 Holds 1
- Array index = 1 Holds 2
- Pointer 0x7fff55a57d04 Holds 2
- Array index = 2 Holds 3
- Pointer 0x7fff55a57d08 Holds 3
- Array index = 3 Holds 4
- Pointer 0x7fff55a57d0c Holds 4
- After program runs

Pointers

- int main
- {
- double* some_point;
- double some_array[4]={1,2,3,4};
- some_point=&some_array[0];
- for (int index1=0; index1<4; index1++)
- {
- cout<< "Array index = "<<index1 << " Holds "<<some_array[index1]<<endl;
- cout<< "Pointer "<<some_point<<" Holds "<<*some_point<<endl;
- some_point++;
- }

Pointers

- Array index = 0 Holds 1
- Pointer 0x7fff36eb1f60 Holds 1
- Array index = 1 Holds 2
- Pointer 0x7fff36eb1f68 Holds 2
- Array index = 2 Holds 3
- Pointer 0x7fff36eb1f70 Holds 3
- Array index = 3 Holds 4
- Pointer 0x7fff36eb1f78 Holds 4
- After program is run.

Pointers

- In the program above we see that the pointer and array are to use double precision floating point numbers.
- When the program is run the different size of the data will affect the value stored in the pointer.
- We see that the pointer value will now change by 8 bytes to find the next available space in memory.
- This is a cautionary example when using pointers the math is based on moving through memory not directly manipulating the data held in memory.

- Attempting to deference a non-initialized pointer variable will give undefined and unpredictable results. Merely declaring a pointer variable does not give it an initial value.
- This is a common error which has kept many a C++ programmer up at night trying to debug their program.
- Note that the present build seems to initialize all data memory to zero. This is not always the case.

Void Pointers

- There is one exception to the rule about assigning the address of an “int” to a “int pointer” , the address of a float to a float pointer, etc. C++ allows the programmer to declare a VOID POINTER. A VOID POINTER may be assigned the address of any type. For example;
- void* pointer_to_anything ;
- void* another_pointer_to_anything;
- int an_integer;
- float a_float;
- the following assignments of addresses to void pointers are legal:
- pointer_to_anything = &an_integer;
- another_pointer_to_anything = &a_float;
- Although void pointers may be assigned the address of any data type, VOID POINTERS MAY NOT BE DEREFERENCED DIRECTLY!!!!!!!!!!!!!!!!!

Void Operators

- The reason why it is forbidden to dereference a void pointer directly is that the compiler has no information as to what the void pointer is pointing to.
- Void pointers may be dereferenced PROVIDED THEY ARE CAST TO A POINTER OF THE APPROPRIATE DATA TYPE BY USING THE CAST OPERATOR:
- Programs 33-35 are examples on how to use this type of pointer.

Arrays of Pointers

- Pointers may also be grouped within an array where each of the pointers is given a specific location in memory to hold.
- Each element in the array should have the same characteristics and therefore should be of the same type of pointer.

Dynamic Memory Allocation

- In the case of requesting more memory during a program run, you may use the following syntax.
- Int * some_pointer;
- some_pointer = new int;
- This will request for the system to provide some free memory to use to store an integer.
- You can free up the memory by writing:
- delete some_pointer;

Dynamic Memory Allocation

- The same approach can be used with an array of values
- For example to dynamically allocate memory for an array with user defined size we can write:
- Int array_size;
- Int * some_array_pointer;
- Cin >>array_size;
- Some_array_pointer = new int [array_size];

Dynamic Memory Allocation

- The array could then be accessed using:
- For (int index=0; index<array_size; index++)
- {
- *(some_array_pointer+index)=index; // Puts the value of the index into the array
- }
- When you are finished you can free up the space by writing:
- Delete [] some_array_pointer

Dynamic Memory Allocation

- In the case of multidimensional arrays, the size must be the length of the array stacked end to end so a 2 by 4 array would have a size of 2x4=8.
- This requires some caution when using multiple dimension arrays as the code must take into account the full size of the memory that is requested.

Next Lecture

- In the next lecture we will discuss classes.