C++ Lecture 11

* Pointers, Dynamic Variables, Type Definitions, and Dynamic Arrays
* CIS 251 • Shelby-Hoover Campus

Pointers

* A **pointer** is the memory address of a variable
* Such an address can be stored in a pointer variable
* The pointer variable can then be **dereferenced** to access the value stored in that variable

Declaring Pointer Variables

* The difference between declaring a variable of a particular type and a pointer variable of that type is the addition of an asterisk before the variable name  
  + int myInt; // Declares a variable that can store a value of type int
  + int \*myIntPtr; // Declares a variable that can store a pointer to a variable of type int
* Pointer variables and ordinary variables of the same type can be declared in the same statement:  
    
  int \*ptr1, \*ptr2, val1, val2;

Assigning Addresses

* To assign the address of an existing variable to a pointer variable, use the **address of operator**, & (ampersand)
* Example:  
    
  int myGrade = 88, \*gradePtr;  
  gradePtr = &myGrade;
* After these two lines, the pointer variable gradePtr points to the variable myGrade (it stores the address of myGrade)

Dereferencing a Pointer

* To access the value of the variable pointed to by a pointer variable, use the **dereferencing operator**, \* (asterisk)  
  + Example: \*gradePtr = 92;  
    // assigns the value 92 to the variable pointed to by gradePtr, which is myGrade
  + Example: cout << \*gradePtr << endl;  
    // displays the value of the variable pointed to by gradePtr, which is myGrade
* If you forget the dereferencing operator, you’re actually dealing with the address itself:  
    
  cout << gradePtr << endl;  
  // displays the address stored in gradePtr, which is the address of myGrade

Pointer Assignment

* Using an assignment statement between two *plain* pointer variables copies the *address* stored in the second pointer variable to the first pointer variable:  
    
  int \*ptr1, \*ptr2, value1 = 82;  
  ptr1 = &value1;  
  ptr2 = ptr1; // both ptr1 and ptr2 store the address of value1
* Using an assignment statement between two *dereferenced* pointer variables copies the *value* of the variable pointed to by the second pointer variable to the variable pointed to by first pointer variable:  
    
  int \*ptr1, \*ptr2, value1 = 82, value2 = 91;  
  ptr1 = &value1;  
  ptr2 = &value2;  
  \*ptr2 = \*ptr1; // both value1 and value2 hold the value 82

Dynamic Variables

* A pointer variable can be used to allocate memory dynamically, pointing to a variable that hasn’t been declared separately
* When you allocate memory for a single variable using a pointer variable, this new variable is known as a **dynamic variable**
* The key word new represents the dynamic allocation of memory for a variable:  
    
  int \*myIntPtr;  
  myIntPtr = new int;
* The only way to access this value is to dereference the pointer:  
    
  \*myIntPtr = 9;

Variables in Memory

* Dynamic variables are stored in the heap / freestore
  + It is possible to run out of memory in the heap
  + To prevent this, use a delete statement to deallocate (release) memory that has been allocated to a variable that is no longer needed:  
      
    delete myIntPtr;
  + The pointer variable itself still exists, but it no longer points to a variable
* Be careful not to dereference a pointer variable not currently pointing to a variable; this is known as a **dangling pointer**
* Variables that are created using a normal declaration are called **automatic variables** or **ordinary variables** (a program does not have to allocate or deallocate memory for these variables)

Type Definitions

* Sometimes it can be difficult to remember to include the asterisk for each pointer variable being declared
* C++ allows a program to define a new type based on another type using the key word typedef
  + Example:  
      
    typedef int\* IntPtr;
  + Every variable declared to be of type IntPtr can store a pointer to a variable of type int; the asterisk is “built in”
* To declare a variable of this type, use the new type name:  
    
  IntPtr myIntPtr;
* The type definition may be local (within a function) or global (at the top of the program, to be available to all functions)
* This solves a problem with passing a pointer by reference (determining the order of the asterisk and the ampersand)

Pointers and Arrays

* Chapter 7 mentions that an array variable stores the address of the first element in the array; the subscript accesses a memory location relative to this starting address
* A pointer variable may be assigned an array directly (no need to use an ampersand)
* A programmer may apply a subscript to a pointer variable pointing to an array as with the array itself:  
    
  typedef int\* IntPtr;  
    
  const int SIZE = 20;  
  int myArray[SIZE];  
  IntPtr myPtr;  
    
  myPtr = myArray; // no ampersand needed  
    
  for (int sub = 0; sub < SIZE; sub++)  
   cin >> myPtr[sub];
* This does not work in the other direction: a program cannot assign a pointer to an array variable  
  (e.g., myArray = myPtr; )

Dynamic Arrays

* The array declarations in chapter 7 (automatic arrays, or ordinary arrays) require a constant or literal for the size
* A pointer variable may be used to allocate memory for a **dynamic array**
  + The size of the array may be a named constant, a literal value, or a variable (a program cannot use a variable to set the size of an ordinary array)
  + This size is still fixed for the remainder of the program
* The syntax for a dynamic array declaration is the same as that for a dynamic variable except that the size is specified in square brackets after the type:  
    
  typedef double\* DoubleArrayPtr;  
    
  int size;  
  DoubleArrayPtr myArrayPtr;  
    
  cout << "How many values do you want to enter? ";  
  cin >> size;  
    
  myArrayPtr = new double[size];

Using a Dynamic Array

* The array elements in a dynamic array can be accessed using normal subscript notation after the pointer variable name:  
    
  for (int sub = 0; sub < size; sub++)  
  {  
   cout << "Enter value #" << sub + 1 << ": ";  
   cin >> myArrayPtr[sub];  
  }
* Deallocating the memory for a dynamic array requires a slightly modified delete statement, with empty square brackets before the pointer variable name to indicate that the memory being released is for an entire array, not for a single variable:  
    
  delete [] myArrayPtr;

Pointer Arithmetic

* When a pointer variable points to the first element of the array, the dereferencing operator can be used to access individual elements by adding the subscript to the pointer variable itself (in parentheses):  
    
  typedef int\* IntArrayPtr;  
    
  const int SIZE = 10;  
  int values[SIZE];  
  IntArrayPtr ptr;  
    
  ptr = values;  
    
  for (int i = 0; i < SIZE; i++)  
   cin >> \*(ptr + i);
* A pointer variable may also be “advanced” to the next or previous array element using the increment and decrement operators (not recommended for a dynamic array’s primary pointer)

Multidimensional Dynamic Arrays

* Declaring a multidimensional dynamic array requires declaring an array of pointer variables, where each pointer variable is allocated a set of elements:  
    
  typedef int\* IntArrayPtr;  
  IntArrayPtr \*m = new IntArrayPtr[3];  
  for (int i = 0; i < 3; i++)  
   m[i] = new int[4];
* Accessing the individual elements of a multidimensional dynamic array uses the same syntax as accessing such elements in a multidimensional ordinary array (provide a separate subscript in square brackets for each dimension)
* Deleting a multidimensional array requires deallocating the memory for each row before deallocating the array as a whole:  
    
  for (int i = 0; i < 3; i++)  
   delete [] m[i];  
  delete [] m;

The Point of Pointers

* Pointer variables that point to individual ordinary variables have limited use
* Pointer variables that point to arrays allow a programmer to create arrays with sizes based on input
* A frequent use of pointer variables is in building data structures with classes (chapter 10)
  + In a linked list, each item contains a pointer variable that points to the next item in the list
  + In a tree, each item contains two pointer variables: one that points to the item to the “left” of the current item, and another that points to the item to the “right” of the current item
  + The concept of using pointers to connect items in a data structure is in chapter 13 (beyond CIS 251)