LESSON SET

9 Pointers

PURPOSE

1. To introduce pointer variables and their relationship with arrays
2. To introduce the dereferencing operator
3. To introduce the concept of dynamic memory allocation

PROCEDURE

1. Students should read the Pre-lab Reading Assignment before coming to lab.
2. Students should complete the Pre-lab Writing Assignment before coming to lab.
3. In the lab, students should complete labs assigned to them by the instructor.

Contents	Pre-requisites	Approximate completion time	Page number	Check when done
Pre-lab Reading Assignment		20 min.	158	
Pre-lab Writing Assignment	Pre-lab reading	10 min.	167	
LESSON 9A				
Lab 9.1 Introduction to Pointer Variables	Basic understanding of pointer variables	15 min.	167	
Lab 9.2 Dynamic Memory	Basic understanding of dynamic memory, `new` and `delete` operators	35 min.	168	
LESSON 9B				
Lab 9.3 Dynamic Arrays	Basic understanding of the relationship of pointer variables and arrays	25 min.	170	
Lab 9.4 Student Generated Code Assignments	Basic understanding of pointers, the (*) and (&) symbols, sort and search routines	30 min.	171	

PRE-LAB READING ASSIGNMENT

Pointer Variables

A distinction must always be made between a memory location's address and the data stored at that location. A street address like 119 Main St. is a location that is different than a description of what is at that location: the little red house of the Smith family. So far we have been concerned only with the data stored in a variable, rather than with its address (where in main memory the variable is located). In this lesson we will look at addresses of variables and at special variables, called **pointers**, which hold these addresses. The address of a variable is given by preceding the variable name with the C++ address operator (`&`):

```
cout << &sum; // This outputs the address of the variable sum
```

The & operator in front of the variable `sum` indicates that the address itself, and not the data stored in that location, is the value used. On most systems the above address will print as a hexadecimal value representing the physical location of the variable. Before this lesson where have you used the address operator in C++ programming? You may recall that it was used in the prototype and the function heading of a function for parameters being passed by reference. This connection will be explored in the next section.

To define a variable to be a pointer, we precede it with an asterisk (*) and initialize it with the special value `nullptr`:

```
int *ptr = nullptr;
```

The asterisk in front of the variable indicates that `ptr` holds the address of a memory location. Assigning `nullptr` to a pointer variable makes the variable point to the address 0. When a pointer is set to the address 0, it is referred to as a null pointer because it points to "nothing." The `int` indicates that the memory location that `ptr` points to holds integer values. `ptr` is NOT an integer data type, but rather a pointer that holds the address of a location where an integer value is stored. This distinction is most important!

The following example illustrates this difference.

```
int  sum;                  // sum holds an integer value.
int *sumPtr = nullptr;     // sumPtr holds an address where an
                           // integer can be found.
```

By now there may be confusion between the symbols `*` and `&`, so we next discuss their use.

Using the & Symbol

The `&` symbol is basically used on two occasions.

1. The most frequent use we have seen is between the data type and the variable name of a pass by reference parameter in a function heading/prototype. This is called a **reference variable**. The memory address of the parameter is sent to the function instead of the value at that address. When the parameter is used in the function, the compiler automatically **dereferences** the variable. Dereference means that the location of that reference variable (parameter in this case) is accessed to retrieve or store a value.

We have looked at the swap function on several occasions. We revisit this routine to show that the & symbol is used in the parameters that need to be swapped. The reason is that these values need to be changed by the function and, thus, we give the address (location in memory) of those values so that the function can write their new values into them as they are swapped.

Example:

```
void swap(int &first, int &second)
{                   // The & indicates that the parameters
                    // first and second are being passed by
                    // reference.

int temp;

temp = first;     // Since first is a reference variable,
                  // the compiler retrieves the value
                  // stored there and places it in temp.

first = second    // New values are written directly into
second = temp;    // the memory locations of first and second.
}
```

2. The & symbol is also used whenever we are interested in the *address* of a variable rather than its *contents*.

Example:

```
cout  << sum;     // This outputs the value stored in the
                  // variable sum.
cout  << &sum;    // This outputs the address where
                  // sum is stored in memory.
```

Using the & symbol to get the address of a variable comes in handy when we are assigning values to pointer variables.

Using the * Symbol

The * symbol is also basically used on two occasions.

1. It is used to define pointer variables:

```
int *ptr = nullptr;
```

2. It is also used whenever we are interested in the contents of the *memory location* pointed to by a *pointer variable*, rather than the address itself. When used this way * is called the **indirection operator**, or **dereferencing operator**.

Example:

```
cout << *ptr; // Since ptr is a pointer variable, *
              // dereferences ptr. The value stored at the
              // location ptr points to will be printed.
```

Using * and & Together

In many ways * and & are the opposites of each other. The * symbol is used just before a pointer variable so that we may obtain the actual data rather than the address of the variable. The & symbol is used on a variable so that the variable's address, rather than the data stored in it, will be used. The following program demonstrates the use of pointers.

Sample Program 9.1:

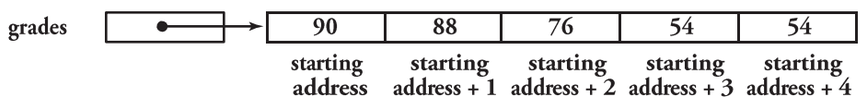
```
#include <iostream>
using namespace std;

int main()
{
   int one = 10;
   int *ptr1 = nullptr;  //  ptr1 is a pointer variable that points to an int

   ptr1 = &one;    //  &one indicates that the address, not the
                   //  contents, of one is being assigned to ptr1.
                   //  Remember that ptr1 can only hold an address.
                   //  Since ptr1 holds the address where the variable
                   //  one is stored, we say that ptr1 "points to" one.

   cout << "The value of one is   " << one   << endl << endl;
   cout << "The value of &one is  " << &one  << endl << endl;
   cout << "The value of ptr1 is  " << ptr1  << endl << endl;
   cout << "The value of *ptr1 is " << *ptr1 << endl << endl;

   return 0;
}
```

What do you expect will be printed if the address of variable one is the hexadecimal value 006AFOF4? The following will be printed by the program.

Output	Comments
The value of one is 10	one is an integer variable, holding a 10.
The value of &one is 006AF0F4	&one is the "address of" variable one.
The value of ptr1 is 006AF0F4	ptr1 is assigned one's address
The value of *ptr1 is 10	* is the dereferencing operator which means *ptr1 gives us the value of the variable ptr1 is pointing at.

Arrays and Pointers

When arrays are passed to functions they are passed by pointer. An array name is a pointer to the beginning of the array. Variables can hold just one value and so we can reference that value by just naming the variable. Arrays, however, hold many values. All of these values cannot be referenced just by naming the array. This is where pointers enter the picture. Pointers allow us to access all the array elements. Recall that the array name is really a pointer that holds the address of the first element in the array. By using an array index, we dereference the pointer which gives us the contents of that array location. If `grades` is an array of 5 integers, as shown below, `grades` is actually a pointer to the first location in the array, and `grades[0]` allows us to access the contents of that first location.



From the last section we know it is also possible to dereference the pointer by using the * operator. What is the output of the following two statements?

```
cout << grades[0];   //  Output the value stored in the 1st array element
cout << *grades;     //  Output the value found at the address stored
                     //  in grades (i.e., at the address of the 1st array
                     //  element).
```

Both statements are actually equivalent. They both print out the contents of the first `grades` array location, a 90.

Access of an individual element of an array through an index is done by **pointer arithmetic**. We can access the second array location with `grades[1]`, the third location with `grades[2]`, and so on, because the indices allow us to move through memory to other addresses relative to the beginning address of the array. The phrase "address + 1" in the previous diagram means to move one array element forward from the staring address of the array. The third element is accessed by moving 2 elements forward and so forth. The amount of movement in bytes depends on how much memory is allocated for each element, and that depends on how the array is defined. Since `grades` is defined as an array of integers, if an integer is allocated 4 bytes, then +1 means to move forward 4 bytes from the starting address of the array, +2 means to move forward 8 bytes, etc. The compiler keeps track of how far forward to move to find a desired element based on the array index. Thus the following two statements are equivalent.

```
cout << grades[2];
cout << *(grades + 2);
```

Both statements refer to the value located two elements forward from the starting address of the array. Although the first may be the easiest, computer scientists need to understand how to access memory through pointers. The following program illustrates how to use pointer arithmetic rather than indexing to access the elements of an array.

Sample Program 9.2:

```
// This program illustrates how to use pointer arithmetic to
// access elements of an array.

#include <iostream>
using namespace std;

int main()
{
    int grades[] = {90, 88, 76, 54, 34};
        // This defines and initializes an int array.
        // Since grades is an array name, it is really a pointer
        // that holds the starting address of the array.

    cout << "The first grade is "          // The * before grades
         << *grades << endl;               // dereferences it so that the
                                           // contents of array location 0
                                           // is printed instead of its
                                           // address.

    cout << "The second grade is "         // The same is done for the other
         << *(grades + 1) << endl;         // elements of the array. In
    cout << "The third grade is "          // each case, pointer arithmetic
         << *(grades + 2) << endl;         // gives us the address of the
    cout << "The fourth grade is "         // next array element.  Then the
         << *(grades + 3) << endl;         // indirection operator * gives
    cout << "The fifth grade is "          // us the value of what is stored
         << *(grades + 4) << endl;         // at that address.

    return 0;
}
```

What is printed by the program?

```
The first grade is 90
The second grade is 88
The third grade is 76
The fourth grade is 54
The fifth grade is 34
```

Dynamic Variables

In Lesson Set 7 on arrays, we saw how the size of an array is given at the time of its definition. The programmer must estimate the maximum number of elements that will be used by the array and this size is static, i.e., it cannot change during the execution of the program. Consequently, if the array is defined to be larger than is needed, memory is wasted. If it is defined to be smaller than is needed, there is not enough memory to hold all of the elements. The use of pointers (and the `new` and `delete` operators described below) allows us to dynamically allocate enough memory for an array so that memory is not wasted.

This leads us to **dynamic variables**. Pointers allow us to use dynamic variables, which can be created and destroyed as needed within a program. We have studied scope rules, which define where a variable is active. Related to this is the concept of **lifetime**, the time during which a variable exists. The lifetime of dynamic variables is controlled by the program through explicit commands to allocate (i.e., create) and deallocate (i.e., destroy) them. The operator **`new`** is used to allocate and the operator **`delete`** is used to deallocate dynamic variables. The compiler keeps track of where in memory non-dynamic variables (variables discussed thus far in this book) are located. Their contents can be accessed by just naming them. However, the compiler does not keep track of the address of a dynamic variable. When the `new` command is used to allocate memory for a dynamic variable, the system returns its address and the programmer stores it in a pointer variable. Through the pointer variable we can access the memory location.

Example:

```
int *one = nullptr;    // one and two are defined to be pointer
int *two = nullptr;    // variables that point to ints

int result;       // defines an int variable that will hold
                  // the sum of two values.

one = new int;    // These statements each dynamically
two = new int;    // allocate enough memory to hold an int
                  // and assign their addresses to pointer
                  // variables one and two, respectively.

*one = 10;        // These statements assign the value 10
*two = 20;        // to the memory location pointed to by one
                  // and 20 to the memory location pointed to
                  // by two.

result = *one + *two;
                  // This adds the contents of the memory
                  // locations pointed to by one and two.
cout << "result = " << result << endl;

delete one;       // These statements deallocate the dynamic
delete two;       // variables.  Their memory is freed and
                  // they cease to exist.
```

Now let us use dynamic variables to allocate an appropriate amount of memory to hold an array. By using the `new` operator to create the array, we can wait until we know how big the array needs to be before creating it. The following program demonstrates this idea. First the user is asked to input the number of grades to be processed. Then that number is used to allocate exactly enough memory to hold an array with the required number of elements for the grades.

Sample Program 9.3:

```
// This program finds the average of a set of grades.
// It dynamically allocates space for the array holding the grades.

#include <iostream>
#include <iomanip>
using namespace std;

// function prototypes
void sortIt (float* grades, int numOfGrades);
void displayGrades(float* grades, int numOfGrades);

int main()
{
    float *grades = nullptr;  // a pointer that will be used to point
                              // to the beginning of a float array
    float total = 0;          // total of all grades
    float average;            // average of all grades
    int numOfGrades;          // the number of grades to be processed
    int count;                // loop counter

    cout << fixed << showpoint << setprecision(2);

    cout << "How many grades will be processed " << endl;
    cin >> numOfGrades;

    while (numOfGrades <= 0)  // checks for a legal value
    {
       cout << "There must be at least one grade. Please reenter.\n";
       cout << "How many grades will be processed " << endl;
       cin >> numOfGrades;
    }

    grades = new float(numOfGrades);
                           // allocation memory for an array
                           // new is the operator that is allocating
                           // an array of floats with the number of
                           // elements specified by the user. grades
                           // is the pointer holding the starting
                           // address of the array.

    if (grades == nullptr)  // nullptr is a special identifier
    {                       // to equal 0.  It indicates a non-valid
                            // address.  If grades is 0 it means the
                            // the operating system was unable to
                            // allocate enough memory for the array.

    cout << "Error allocating memory!\n";
                          // The program should output an appropriate
    return -1;            // error message and return with a value
    }                     // other than 0 to signal a problem.
    cout << "Enter the grades below\n";
```

```
    for (count = 0; count < numOfGrades; count++)
    {
        cout << "Grade " << (count + 1) << ": " << endl;
        cin >> grades[count];
        total = total + grades[count];
    }

    average = total / numOfGrades;
    cout << "Average Grade is " << average << "%" << endl;

    sortIt(grades, numOfGrades);
    displayGrades(grades, numOfGrades);
    delete [] grades;           // deallocates all the array memory

    return 0;
}

//*************************************************
//                sortIt
//
// task:          to sort numbers in an array
// data in:       an array of floats and
//                the number of elements in the array
// data out:      sorted array
//
//*************************************************

void sortIt(float* grades, int numOfGrades)
{
  // Sort routine placed here
}

//*************************************************
//                displayGrades
//
// task:          to display numbers in an array
// data in:       an array of floats and
//                the number of elements in the array
// data out:      none
//
//*************************************************
void displayGrades(float* grades, int numOfGrades)
{
       // Code to display grades of the array
}
```

Notice how the dynamic array is passed as a parameter to the `sortIt` and `displayGrades` functions. In each case, the call to the function simply passes the name of the array, along with its size as an argument. The name of the array holds the array's starting address.

```
sortIt(grades, numOfGrades);
```

In the function header, the formal parameter that receives the array is defined to be a pointer data type.

```
void sortIt(float* grades, int numOfGrades)
```

Since the compiler treats an array name as a pointer, we could also have written the following function header.

```
void sortIt(float grades[], int numOfGrades)
```

In this program, dynamic allocation of memory was used to save memory. This is a minor consideration for the type of programs done in this course, but a major concern in professional programming environments where large fluctuating amounts of data are used.

Review of * and &

The * symbol is used to define pointer variables. In this case it appears in the variable definition statement between the data type and the pointer variable name. It indicates that the variable holds an address, rather than the data stored at that address.

Example 1: `int *ptr1;`

* is also used as a dereferencing operator. When placed in front of an already defined pointer variable, the data stored at the location the pointer points to will be used and not the address.

Example 2: `cout << *ptr1;`

Since `ptr1` is defined as a pointer variable in Example 1, if we assume `ptr1` has now been assigned an address, the output of Example 2 will be the data stored at that address. `*` in this case dereferences the variable `ptr1`.

The `&` symbol is used in a procedure or function heading to indicate that a parameter is being passed by reference. It is placed between the data type and the parameter name of each parameter that is passed by reference.

The `&` symbol is also used before a variable to indicate that the address, not the contents, of the variable is to be used.

Example 3:

```
int *ptr1 = nullptr;
int one = 10;

ptr1 = &one;            // This assigns the address of variable
                        // one to ptr1

cout << "The value of &one is "
     << &one << endl;  // This prints an address

cout << "The value of *ptr1 is "
     << *ptr1 << endl;  // This prints 10, because ptr1 points to
                        // one and * is the dereferencing operator.
```

PRE-LAB WRITING ASSIGNMENT

Fill-in-the-Blank Questions

1. The _____symbol is the dereferencing operator.
2. The _____symbol means "address of."
3. The name of an array, without any brackets, acts as a(n) _____ to the starting address of the array.
4. An operator that allocates a dynamic variable is _____.
5. An operator that deallocates a dynamic variable is _____.
6. Parameters that are passed by _____are similar to a pointer variable in that they can contain the address of another variable. They are used as parameters of a procedure (void function) whenever we want a procedure to change the value of the argument.

Given the following information, fill the blanks with either "an address" or "3.75".

```
float * pointer;
float pay = 3.75;
pointer = &pay;
```

7. `cout << pointer;` will print _____.
8. `cout << *pointer;` will print _____.
9. `cout << &pay;` will print _____.
10. `cout << pay;` will print _____.

LESSON 9A

LAB 9.1 Introduction to Pointer Variables

Retrieve program `pointers.cpp` from the Lab 9 folder.
The code is as follows:

```
// This program demonstrates the use of pointer variables
// It finds the area of a rectangle given length and width
// It prints the length and width in ascending order

// PLACE YOUR NAME HERE

#include <iostream>
using namespace std;

int main()
{

   int length;                // holds length
   int width;                 // holds width
   int area;                  // holds area

   int *lengthPtr = nullptr;  // int pointer which will be set to point to length
   int *widthPtr = nullptr;   // int pointer which will be set to point to width
```

continues