L E S S O N S E T

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 | Basic understanding of | 15 min. | 167 |  |
| Variables | pointer variables |  |  |  |
| Lab 9.2 |  |  |  |  |
| Dynamic Memory | Basic understanding of | 35 min. | 168 |  |
|  | dynamic memory, new |  |  |  |
|  | and delete operators |  |  |  |
| LESSON 9B |  |  |  |  |
| Lab 9.3 |  |  |  |  |
| Dynamic Arrays | Basic understanding of | 25 min. | 170 |  |
|  | the relationship of |  |  |  |
|  | pointer variables and |  |  |  |
|  | arrays |  |  |  |
| Lab 9.4 |  |  |  |  |
| Student Generated Code | Basic understanding of | 30 min. | 171 |  |
| Assignments | pointers, the (\*) and (&) |  |  |  |
|  | symbols, sort and search |  |  |  |
|  | routines |  |  |  |

157

# P R E - LA B R E A D I N G A S S I G N M E N T

## 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 vari- able, rather than with its address (where in main memory the variable is locat- ed). 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++ pro- gramming? 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 **derefer- ences** 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.

}

1. 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;

1. 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 **dereferenc- ing operator**.

*Example:*

## Using **\*** and **&** Together

cout << \*ptr; // Since ptr is a pointer variable, \*

// dereferences ptr. The value stored at the

// location ptr points to will be printed.

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;

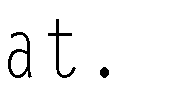
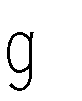
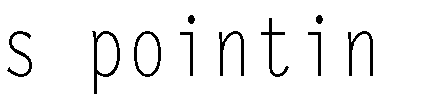
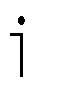
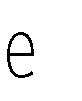
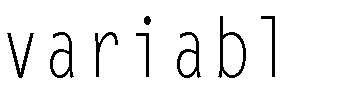
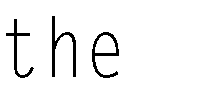
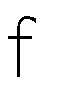
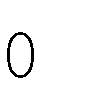
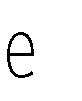
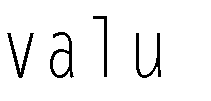
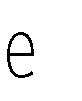
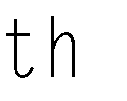
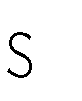
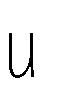
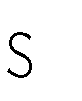
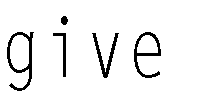
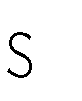
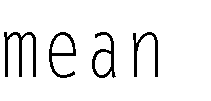
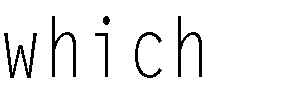
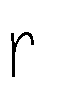
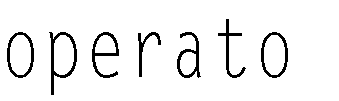
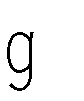
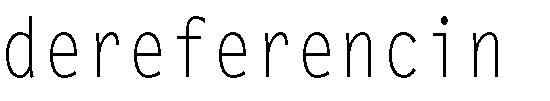
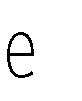
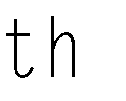
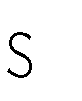
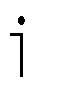
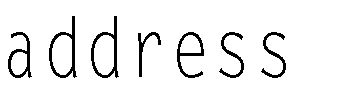
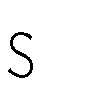
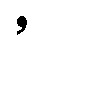
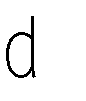
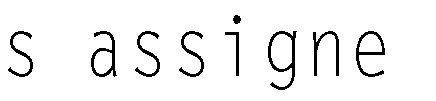
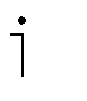
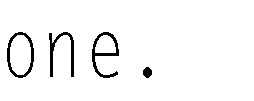
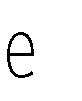
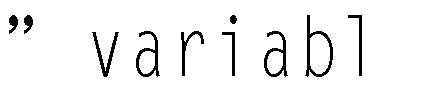
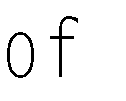
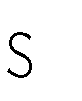
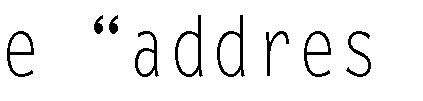
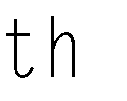
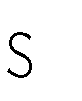
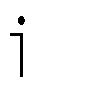
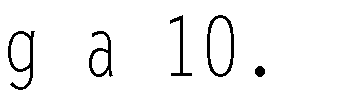
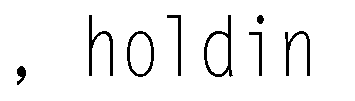
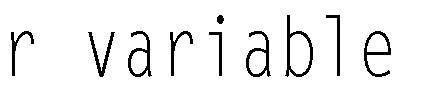
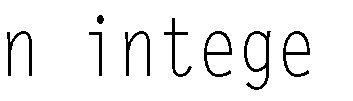
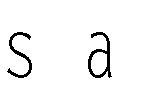
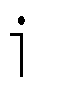
return 0;

}

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

**Output Comments**

**The value of one is 10**



one &one ptr1

\*

one

\*ptr1

ptr1

**The value of &one is 006AF0F4**

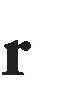
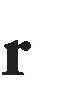
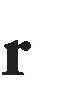
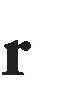
**The value of ptr1 is 006AF0F4 The value of \*ptr1 is 10**

## 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 start- ing 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 start- ing address of the array. Although the first may be the easiest, computer scien- tists 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

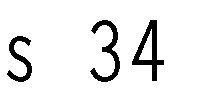
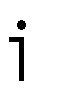
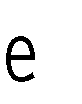
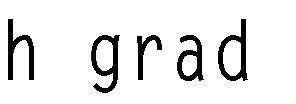
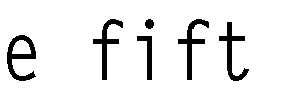
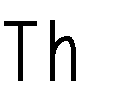
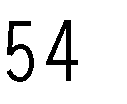
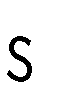
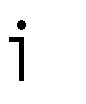
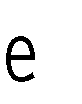
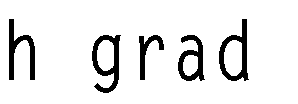
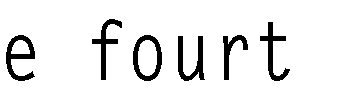
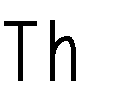
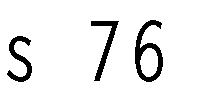
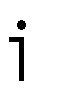
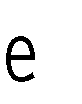
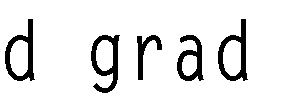
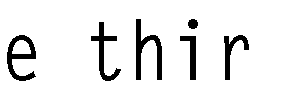
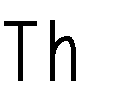
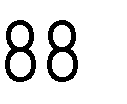
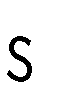
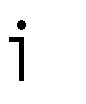
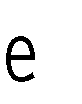
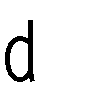
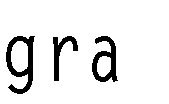
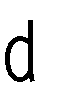
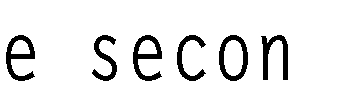
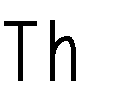
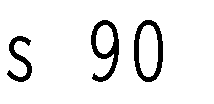
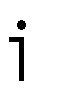
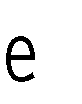
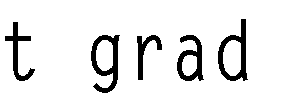
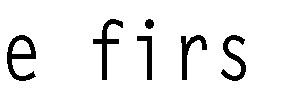
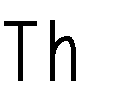
// that holds the starting address of the array.

|  |  |  |  |
| --- | --- | --- | --- |
| cout | <<  << | "The first grade is "  \*grades << endl; | // The \* before grades  // dereferences it so that the  // contents of array location 0  // is printed instead of its  // address. |
| cout cout cout  cout | <<  <<  <<  <<  <<  <<  <<  << | "The second grade is "  \*(grades + 1) << endl; "The third grade is "  \*(grades + 2) << endl; "The fourth grade is "  \*(grades + 3) << endl; "The fifth grade is "  \*(grades + 4) << endl; | // The same is done for the other  // elements of the array. In  // each case, pointer arithmetic  // gives us the address of the  // next array element. Then the  // indirection operator \* gives  // us the value of what is stored  // at that address. |

return 0;

}

What is printed by the program?



## 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 allo- cate 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 stud- ied scope rules, which define where a variable is active. Related to this is the con- cept 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 allo- cate and the operator **delete** is used to deallocate dynamic variables. The com- piler 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 vari- able, the system returns its address and the programmer stores it in a pointer vari- able. 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)

## Review of **\*** and **&**

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.

The \* symbol is used to define pointer variables. In this case it appears in the vari- able 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.

# P R E - LA B W R I T I N G A S S I G N M E N T

167

Pre-Lab Writing Assignment

Fill-in-the-Blank Questions

# L E S S O N 9 A

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;

1. cout << pointer; will print .
2. cout << \*pointer; will print .
3. cout << &pay; will print .
4. cout << pay; will print .

## 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*

cout << "Please input the length of the rectangle" << endl; cin >> length;

cout << "Please input the width of the rectangle" << endl; cin >> width;

**// Fill in code to make lengthPtr point to length (hold its address)**

**// Fill in code to make widthPtr point to width (hold its address)**

area = **// Fill in code to find the area by using only the pointer variables**

cout << "The area is " << area << endl;

if **(// Fill in the condition length > width by using only the pointer variables)**

cout << "The length is greater than the width" << endl;

else if **(// Fill in the condition of width > length by using only the pointer**

**// variables)**

cout << "The width is greater than the length" << endl;

else

cout << "The width and length are the same" << endl;

return 0;

}

}

*Exercise 1:* Complete this program by filling in the code (places in bold). Note: use only pointer variables when instructed to by the comments in bold.

This program is to test your knowledge of pointer variables and the & and

\* symbols.

*Exercise 2:* Run the program with the following data: 10 15. Record the output here .

## LAB 9.2 Dynamic Memory

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

// This program demonstrates the use of dynamic variables

**// PLACE YOUR NAME HERE**

#include <iostream> using namespace std;

const int MAXNAME = 10;

Lesson 9A

169

int main()

{

int pos;

char \*name = nullptr; int \*one = nullptr;

int \*two = nullptr;

int \*three = nullptr; int result;

**// Fill in code to allocate the integer variable one here**

**// Fill in code to allocate the integer variable two here**

**// Fill in code to allocate the integer variable three here**

**// Fill in code to allocate the character array pointed to by name**

cout << "Enter your last name with exactly 10 characters." << endl; cout << "If your name has < 10 characters, repeat last letter. " << endl

<< "Blanks at the end do not count." << endl; for (pos = 0; pos < MAXNAME; pos++)

cin >> **// Fill in code to read a character into the name array**

**// WITHOUT USING a bracketed subscript**

cout << "Hi ";

for (pos = 0; pos < MAXNAME; pos++)

cout << **// Fill in code to a print a character from the name array**

**// WITHOUT USING a bracketed subscript**

cout << endl << "Enter three integer numbers separated by blanks" << endl;

**// Fill in code to input three numbers and store them in the**

**// dynamic variables pointed to by pointers one, two, and three.**

**// You are working only with pointer variables**

//echo print

cout << "The three numbers are " << endl;

**// Fill in code to output those numbers**

result = **// Fill in code to calculate the sum of the three numbers**

cout << "The sum of the three values is " << result << endl;

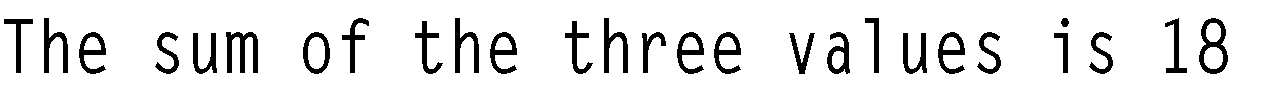
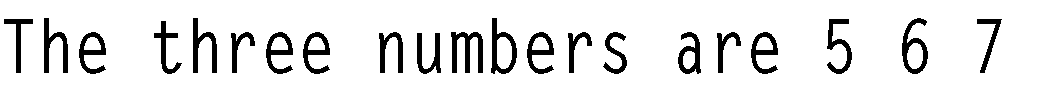
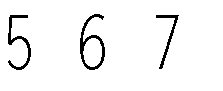
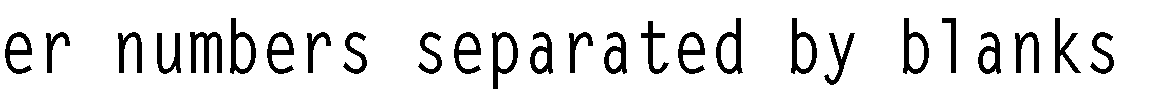
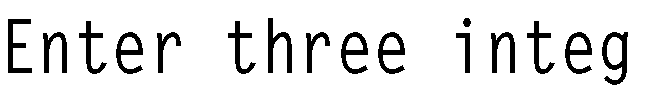
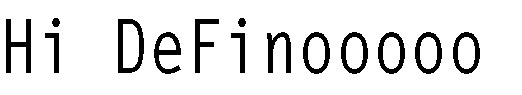
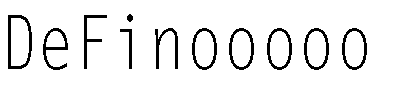
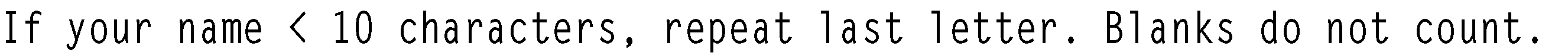
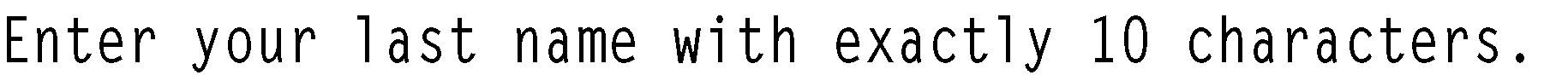
**// Fill in code to deallocate one, two, three and name**

return 0;

}

*Exercise 1:* Complete the program by filling in the code. (Areas in bold) This problem requires that you study very carefully the code already written to prepare you to complete the program.

*Sample Run:*



# L E S S O N 9 B

*Exercise 2:* In inputting and outputting the name, you were asked NOT to use a bracketed subscript. Why is a bracketed subscript unnecessary?

Would using name[pos] work for inputting the name? Why or why not? Would using name[pos] work for outputting the name? Why or why not?

Try them both and see.

## LAB 9.3 Dynamic Arrays

Retrieve program darray.cpp from the Lab 9 folder.

The code is as follows:

// This program demonstrates the use of dynamic arrays

**// PLACE YOUR NAME HERE**

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

int main()

{

float \*monthSales = nullptr; // a pointer used to point to an array

// holding monthly sales

float total = 0; // total of all sales

float average; // average of monthly sales

int numOfSales; // number of sales to be processed

int count; // loop counter cout << fixed << showpoint << setprecision(2);

cout << "How many monthly sales will be processed? "; cin >> numOfSales;

**// Fill in the code to allocate memory for the array pointed to by**

**// monthSales.**

if ( **// Fill in the condition to determine if memory has been**

**// allocated (or eliminate this if construct if your instructor**

**// tells you it is not needed for your compiler)**

)

{

cout << "Error allocating memory!\n"; return 1;

}

cout << "Enter the sales below\n";

for (count = 0; count < numOfSales; count++)

{

cout << "Sales for Month number "

<< **// Fill in code to show the number of the month**

<< ":";

**// Fill in code to bring sales into an element of the array**

}

for (count = 0; count < numOfSales; count++)

{

total = total + monthSales[count];

}

average = **// Fill in code to find the average**

cout << "Average Monthly sale is $" << average << endl;

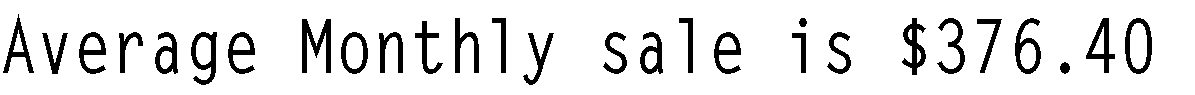
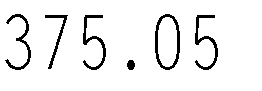
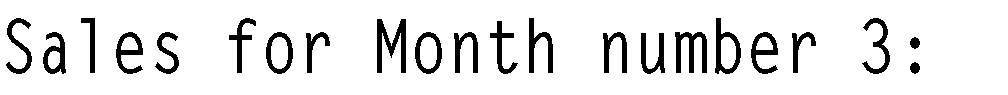
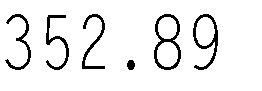
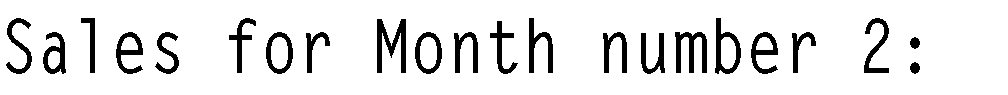
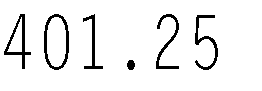
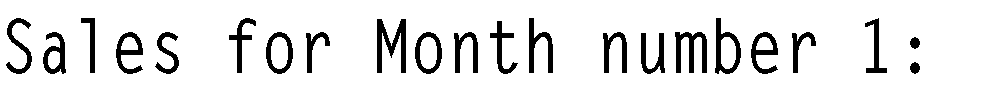
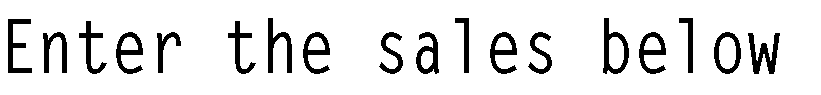
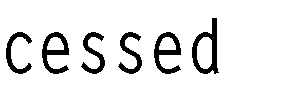
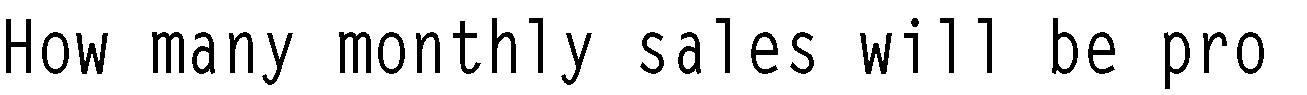
**// Fill in the code to deallocate memory assigned to the array.**

return 0;

}

*Exercise 1:* Fill in the code as indicated by the comments in bold.

*Sample Run:*



## LAB 9.4 Student Generated Code Assignments

In these assignments you are asked to develop functions that have dynamic arrays as parameters. Remember that dynamic arrays are accessed by a point- er variable and thus the parameters that serve as dynamic arrays are, in fact, pointer variables.

*Example:*

void sort(float\* score, int num\_scores); // a prototype whose function has a

// dynamic array as its first

// parameter. It is a pointer variable

.

.

int main()

{

float \*score = nullptr; // a pointer variable

.

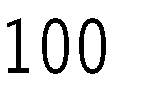
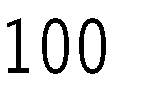
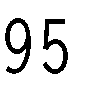
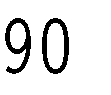
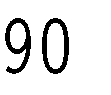
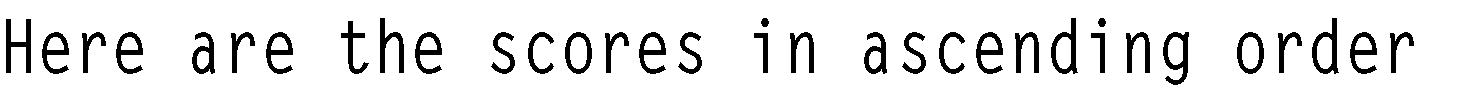
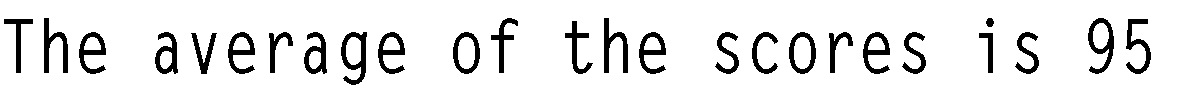
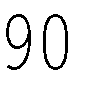
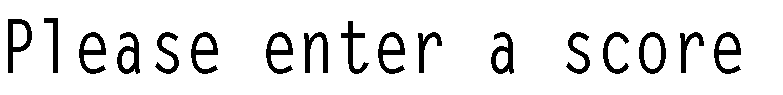
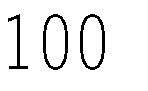
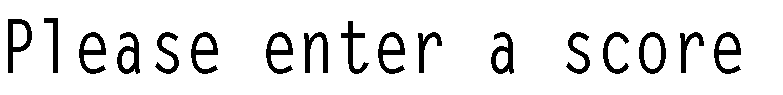
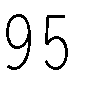
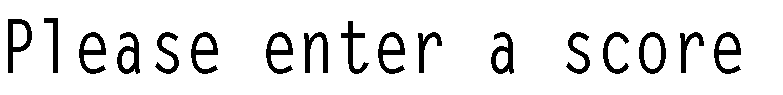
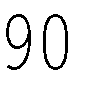
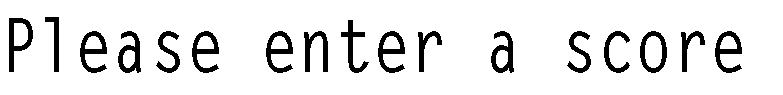
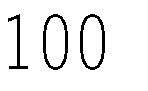
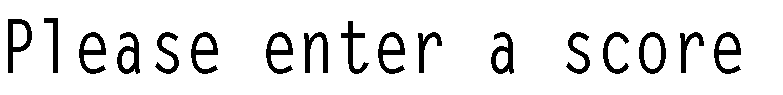
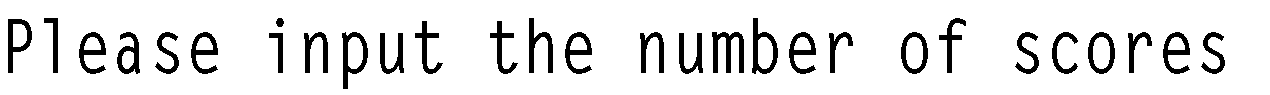
.

score = new float(num\_scores); // allocation of the array sort(score,scoreSize); // call to the function

*Option 1:* Write a program that will read scores into an array. The size of the

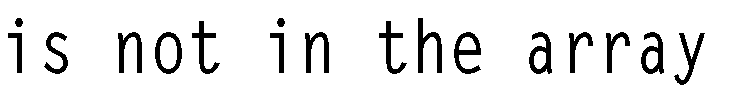
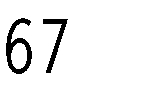
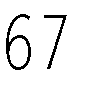
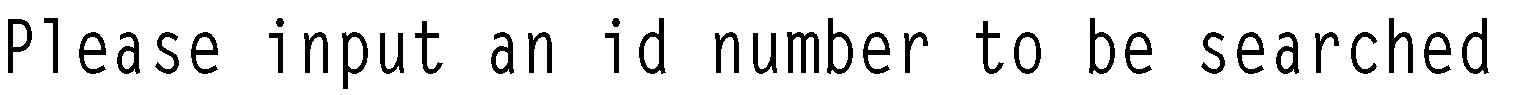
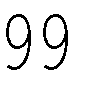
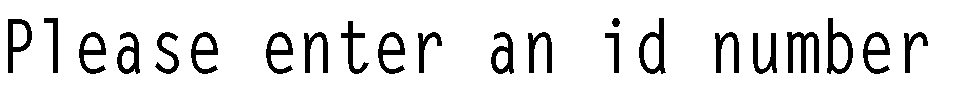
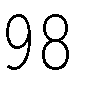
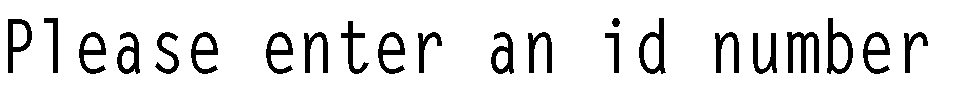
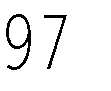
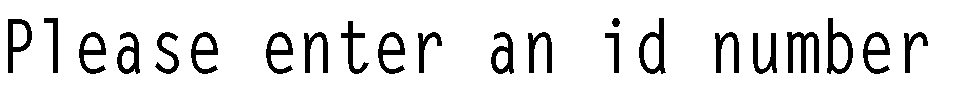
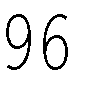
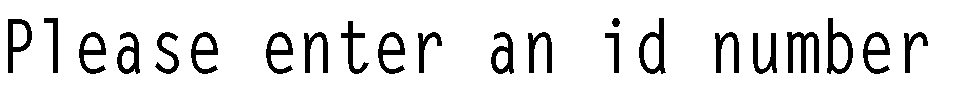
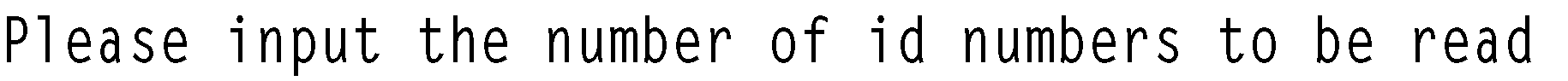
array should be input by the user (dynamic array). The program will find and print out the average of the scores. It will also call a function that will sort (using a bubble sort) the scores in ascending order. The values are then printed in this sorted order.

*Sample Run:*



*Option 2:* This program will read in id numbers and place them in an array. The array is dynamically allocated large enough to hold the number of id numbers given by the user. The program will then input an id and call a function to search for that id in the array. It will print whether the id is in the array or not.

*Sample Run:*



*Option 3:* Write a program that will read monthly sales into a dynamically allocated array. The program will input the size of the array from the user. It will call a function that will find the yearly sum (the sum of all the sales). It will also call another function that will find the average.

*Sample Run:*

