C++ Programming Fundamentals

Elvis C. Foster

Lecture 05: Structures, Arrays, and Pointers

In your introduction to C++ (lectures 1 and 2), it was noted that in addition to primitive data types, there were more advanced data types that the programmer could define. Structures, arrays and pointers are examples of these advanced data types.

There is a close relationship among structures, arrays and pointers. In fact, C++, in many instances, treats arrays and pointers in the same way. Later in the lecture, you will see that:

- You can define a structure which includes arrays and pointers.
- You can define an array of structures as well as an array of pointers.
- You can define pointers to any data type, including structures and arrays.

We shall begin by introducing basic concepts about structures. This will be followed by a relatively more detailed discussion of arrays, then pointers. We will then revisit structures and put the pieces together.

The lecture proceeds under the following captions:

- Introduction to Structures
- Introduction to Arrays
- Strings
- Multi-Dimensional Arrays
- Arrays of Strings
- Introduction to Pointers
- Pointers and Function Arguments
- Pointer Arithmetic
- Association of Pointers With Arrays
- Caution With Pointers
- Pointers to Structures and Arrays of Structures
- Arrays of Pointers
- Passing Structures to & Returning Structures from Functions
- Passing/Returning Arrays and Pointers to/from Functions
- Structures with Arrays, Sub-structures, and/or Pointers
- Self-Referencing Structures
- Unions
- Bit-Fields
- Summary and Concluding Remarks

5.1 Introduction to Structures

A C++ structure is a collection of variables, possibly of different data types, referenced under the same name. In other programming languages, various terms are used to describe the concept of a structure:

- in Pascal and COBOL, the term *record* is used;
- in RPG-400, the term *data structure* is used (this is somewhat confusing);
- in purely object oriented languages such as Java, the term *class* is used (as you will see later in the course a class is a more sophisticated object).

Structures are also called *compound data types*, as well as *aggregate data types*, to mean that they are made up of component variables of different types. The component variables are referred to as *elements* or *members*. The syntax for defining a structure is shown in figure 5-1:

Figure 5-1: Syntax for Structure Definition

```
StructureDefinition ::=
struct [<StructName>]
{
        <memberDeclaration1>;
        <memberDeclaration2>;
         [* <nemberDeclaration>; *]
}       [<instanceName>] [* ,<instanceName> *];
```

Following are a few points of clarification that you should remember:

1. The structure declaration is in fact a type definition — it defines a data type, identified by the structure name.

2. The structure may consist of any number of members; each member is declared in a manner that is similar to variable declaration.

3. Instances of the structure may be declared in the normal manner for variable declaration. Alternately, they may be specified as part of the structure declaration. Each instance has all members of the structure.

Example 5-1: The following code snippets achieve the same end; however, the approach on the right is more flexible.

```
struct studType
{
int studNumber;
char lastName[15];
char firstName[15];
char studMajor[30];
float studGPA;
} stud1, stud2;
```

Is equivalent to:

```
struct studType
{
int studNumber;
char lastName[15];
char firstName[15];
char studMajor[30];
float studGPA;
};
// …
studType stud1, stud2;
```

5.1 Introduction to Structures (continued)

A structure instance may be initialized by assignment or by a function that returns another structure instance of the same type. Elements of a structure instance are referenced by the dot operation, thus:

```
<instanceName.memberName>
```

Example 5-2: Given the structure definition of Example 5-2, we may access structure instance members as follows:

```
strcpy(stud2.firstName, "Bruce"); // stud2.firstName = "Bruce";
strcpy(stud2.lastName, "Jones"); // stud2.lastName = "Jones";
// …
char sYear[4], sSequence[3];  // For storing string version of year and sequence number
// …
stud2.studNumber  = atoi (strcat (sYear, sSequence));
stud1 = stud2;
```

Nested structures (i.e. structures that consist of other structures) are allowed as in the following example.

Example 5-3: In the following code snippet, **dateType** and **studType** are both structures.

```
struct dateType
{
int year , month, day};
…
struct studType
{
int studNumber;
char lastName[15];
char firstName[15];
dateType dateOfBirth;
char studMajor[30];
float studGPA;
} thisStud;
// …
if (thisStud.dateOfBirth.year < 1998) cout << "Welcome to the YMCA!";
else cout << "Sorry, you are too young!";
```

Structures may be used in arrays (arrays of structures) or pointers (pointers to structures); they may also be used as function arguments, or returned from functions. You will encounter examples of these possibilities later in this lecture and as the course progresses.

5.2 Introduction to Arrays

Arrays were introduced earlier in chapter 1. This section briefly reviews what was then established, and then adds additional clarifications on the topic.

An array is a finite list of items of a given base type. The simplest kind of array is the one-dimensional (1-D) array. The syntax for its definition is shown in figure 5-2.

Figure: 5-2: Syntax for Array Definition

1-D_ArrayDefinition ::=
<BaseType> <ArrayName> "["<ArrayLength>"]";

Note:
1. The square bracket here does not mean "optional" but it is required as prat of the syntax, hence the use of quotation marks.
2. The array length specified must be positive integer literal or an integer variable that has a positive value.
3. The base type specified may be any valid primitive or advanced data type.

Array subscripting is also done by using the square brackets. The first element in any C++ array has subscript value of 0; the n^{th} element has subscript value n-1.

Example 5-4:

```
double courseScore [32];       // defines double array of 32 elements
// Subscript values will be 0, 1, 2, 3, ….31.
// Each element could represent the earned score for a particular course and student.
```

Here are two important guidelines to remember when working with arrays:

1. You may not assign one array to another via the assignment statement. Array assignment must be made to array subscripts. There is an exception for strings which will be discussed later.

2. Boundary checking must be done by the programmer; C++ does not check for boundary violation prior to runtime.

5.2.1 Initializing 1-D Arrays

An array is initialized in one of two ways — either via an array initialization assignment or via an initialization loop. The array initialization assignment allows you to specify specific array element values within curly braces; this may appear with the array declaration or shortly afterwards. This option is applicable only for simple arrays where the base type is a primitive data type, and the number of elements is small. The required syntax is shown below (figure 5-3).

Figure 5-3: Array Initialization for Trivial Arrays

ArrayInitialization ::=
<ArrayName>= {<literal0>, <literal1> …};

5.2.1 Initializing 1-D Arrays (continued)

If your array is large (for instance with ten or more items), or the array type is not a primitive type, you need to write an initialization loop. Your initialization loop may be any iterative loop; the **for-statement** is widely used for this.

Example 5-5: The following code snippets illustrate the two approaches to array initialization.

```
double salesList [12] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

// The code above is equivalent to the following:
int x, double salesList [12];
…
for (x = 1; x <= 12; x++) salesList [x - 1] = 0.0;
```

There is only one exception to this matter of array initialization: a string (which is a 1-D character array) may be initialized by using a string constant, specified in double quotes.

Example 5-6: The following code snippet illustrates null-terminated-string initialization (more on this in the next section).

```
char thisString [5] = "    ";
// is equivalent to
char thisString [5] = { ' ', ' ', ' ', ' ', '\0'}
```

5.2.2 Finding the Smallest and Largest Item in a List

Finding the smallest item and/or largest item in a list is a very common problem that people have to solve. For instance, a school teacher or college professor may want to know the best and worst performance from a class on a particular assignment. Figure 5-4 illustrates a generic algorithm for solving the problem. Assuming the declarations of example 5-4, figure 5-5 shows a C++ implementation of the algorithm.

Here are a few points to note about the algorithm and its C++ implementation:

1. The algorithm receives an array of data items of a given base type. This array is called **thisList**.

2. The variable or object **lowes**t is initialized to the highest possible value. This is so because on each iteration through the loop, it will be compared with each item and switched if necessary.

3. The variable or object **highes**t is initialized to the lowest possible value. This is so because on each iteration through the loop, it will be compared with each item and switched if necessary.

4. Depending on the **highLowFlag**, the algorithm returns the highest or lowest item in the list.

Figure 5-4: Finding the Largest and Smallest Item in a List

```
Algorithm: highLow(thisList, thisLength, highLowFlag) Returns data item of the base-type of thisList
// Assume thisList be an array of any given base-type;
// Assume thisLength an integer;
Let x be an integer;
Let highOrLow be a variable of the base-type of thisList;
Let highLowFlag be a character; // 'H' represents high, 'L' represents low
Let lowest be a variable of the base-type of thisList, initialized to the highest possible value;
Let highest be a variable of the base-type of thisList, initialized to the lowest possible value;

START
  If (highLowFlag = 'H')
     For (x varying from 1 to thisLength with increments of 1) do the following:
          If (thisList[x - 1] > highest)
            highest :=  thisList[x - 1];
          End-If
     End-For;
     highOrLow := highest;
  End-If;

  If (highLowFlag = 'L')
     For (x varying from 1 to Length with increments of 1) do the following:
          If (thisList[x - 1] < lowest)
            lowest :=  thisList[x - 1];
          End-If
     End-For;
     highOrLow := lowest;
  End-If;

  Return highOrLow;
STOP
```

Figure 5-5: C++ Implementation of the highLow Algorithm for a list of Real Numbers

```
double highLow(double thisList[ ], char hghLowFlag)
{
   int x;
   double highOrLow, highest = DBL_MIN, lowest = DBL_MAX;
   int size = sizeof(thisList) / sizeof(thisList[0]); // determine the array size

   if (highLowFlag == 'H')
   { for (x = 1, x <= size; x++) if (thisList[x-1] > highest) highest = thisList[x-1];
     highOrLow = highest;
   }

   if (highLowFlag == 'L')
   { for (x = 1, x <= size; x++) if (thisList[x-1] < lowest) lowest = thisList[x-1];
     highOrLow = lowest;
   }

   return highOrLow;
}
```

5.2.2 Finding the Smallest and Largest Item in a List (continued)

There is a more efficient way of finding the smallest or largest item in a list; it involves the use of a more advanced algorithm called a binary search. However, we will forego that discussion at this time; you will likely learn about binary search in a more advanced course such as Data Structures and Algorithms.

5.3 Null-Terminated Strings

C++ supports two types of strings, namely, *null-terminated strings* (NTS) and regular strings that are instances of the **string** class. This section discusses null-terminated strings; we will defer a discussion of regular strings until lecture 6. An NTS is a 1-D character array; for this reason, the alternate reference of *C-string* is often used. C-strings represent the most common use of the 1-D array. Many programming languages provide a special data type for strings, but C++ is not one of them. However, there is a **string** class, which will be discussed in lecture 6.

C++ terminates an NTS with the null character (‘\0’). Because of this, an NTS variable must be declared to be one character longer than the longest string that it will hold.

5.3.1 Reading a String from the Keyboard

Use the **cin** function to read strings that do not contain white space. If the string contains white space, you must write a loop that uses the **getchar()** function. Alternately, you may use the **gets (…)** function, which requires an argument that is a string (review lecture 1).

Note: Neither **cin** nor **gets** performs any boundary checking on the array. Consequently, if the input string is longer than the size of the destination array, the array will be over-written.

5.3.2 C-String Manipulation Functions

The previous lecture covered the commonly used string manipulation functions. They belong to one of two header files: **<cstring.h**> and **<cctype.h**> (**<string.h>** and **<ctype.h>** for older compilers).

The most common functions relate to string copy (**strcpy** and **strncpy**), string concatenation (**strcat** and **strncat**), string comparison (**strcmp** and **strncmp**) and string length determination (**strlen**). Their uses are illustrated in example 7 below.

Contemporary versions of C++ provide a **string** *class* in the header file **<string>** (the next lecture discusses classes in more detail). This class provides a constructor that allows you to define a string instance in the normal way that you would define a variable. This will be revisited in the upcoming lecture.

5.3.2 String Manipulation Functions (continued)

Example 5-7: Figure 5-6 provides a program listing that illustrates manipulation of c-strings; in this case, the user is prompted to key in his/her name, which is then used for some basic manipulations.

Figure 5-6: Illustrating C-String Manipulation

```
// Program: EFString
// Purpose to Illustrate the Use of c-strings
// Author: Elvis Foster

#include <cstring.h>
#include <iostream.h>
using namespace std;
typedef char* cString; // Defines String as a type

void main(int argc, char *argv[])
{
  cString lastName = new char[16]; lastName = " ";
  cString firstName = new char[16]; firstName = " ";
  cString fullName = new char[31]; fullName = " ";
  signed int x;
  char thisChar;
  cout << "Illustration of Strings\n";
  cout << "\n Please enter your first name: ";
  gets(firstName);
  cout << "Please enter your Surname: ";
  gets(lastName);
  strcpy(fullName, firstName);
  strcpy(fullName, lastName);   // firstName will be overwritten by lastName
  cout << fullName << endl;

  strcpy(fullName, lastName);  // Copies Surname to fullName
  strcat(fullName, " "); // Appends space to fullName
  strcat(fullName, firstName);   // firstName is appended to fullName
  cout << fullName << "  " << strlen(fullName) << endl;

  x = strcmp(firstName, lastName);
  if (x > 0)
  cout << firstName << " > " << lastName << endl;
  else if (x < 0)
      cout << firstName << " < " << lastName << endl;
      else cout << firstName << " = " << lastName << endl;

  cout << "Press any key to continue: ";
  cin >> thisChar;

}
```

5.3.2 String Manipulation Functions (continued)

As you observe usage of these c-string functions, please note the following points of clarification:

1. Concatenation starts at the first non-null character in the destination string (left to right).

2. Copy starts at the first position (i.e. position 0 in the destination string.

4. If you are using **strcmp** to check for one condition only, you can specify it as part of a larger expression. Here is an example: if (strcmp (FirstName, SurName) > 0) …
   The reason this format was not used in example 5-7 is that it would result in a call of **strcmp** twice instead of once. In the interest of efficiency, this alternative was avoided.

5. String constants are specified in double quotes and are typically used to initialize strings. Note however that character values are specified in single quotes (as in sample program **MyGrade** of lecture 3).

```
e.g. char yourName[ ]  = “          ”;
```

Consider this assignment statement: `string1 = string2;`
A statement like this often causes a compilation error when used with null-terminated strings. There are two possible ways to resolve this:

- Use the **strcpy** function
- Do a character-by-character assignment within a loop

Example 5-8: The code snippet below illustrates the two approaches to assigning one string to another.

```
typedef char* cString; // Defines cString as a type
// . . .
cString string1, string2;
// . . .
strcpy(string1, string2); // This statement is equivalent to
for (x = 0; x < strlen(string1); x++) string1[x] = string2[x];
```

Exercise:

1. Write a C++ function to accept a string as input argument, convert it to lower case and return it to the calling statement.
2. Write a C++ function to accept a string as input argument, convert it to upper case and return it to the calling statement.

5.4 Multi-Dimensional Arrays

Quite often in programming, we need to be able to use arrays of dimension greater than one. C++ allows the programmer to define arrays of any number of dimensions.

5.4.1 Two Dimensional Arrays

For a two-dimensional (2-D) array, two lengths are required; these are traditionally construed as rows and columns. The required is represented in figure 5-7:

Figure 5-7: Syntax for 2-D Arrays

2-D_ArrayDefinition ::=
<Type> <ArrayName> "["<Length1>"]" "["<Length2>"]";

Note:
1. The square bracket here does not mean "optional" but it is required as prat of the syntax, hence the use of quotation marks.
2. The array length specified must be positive integer literal or an integer variable that has a positive value.
3. The data type specified may be any valid primitive or advanced data type.

Here are a few points about 2-D arrays to remember:

1. Unlike many other languages that use commas to separate dimensions, each dimension is specified in its own set of brackets.

2. In accessing elements of the 2-D array, double subscripting with two brackets is also required.

3. Conceptually, a 2-D array is stored in a row-column (row major) manner. The upcoming example should help to clarify these points.

Example 5-9: Consider tracking the monthly performance of thirty sales representatives. We could do this via three arrays: a 1-D array to store the names of the sales persons, another 1-D array to store the months of the year, and a 2-D array to store the performance of the sales persons. Figure 5-8 illustrates the basic idea.

Figure 5-8: Representing a 2-D Array for Storing Monthly Performance of Sales Persons

Assume that there are 30 salesmen and we wish to store monthly sales for each salesman for 12 months. Thus,

	Jan	*Feb*	*Dec*
S1	1026	1045	1522
...			
S30	964	920	1255

The first element is salePerf [0][0]; the last element is salesPerf [29][11]. The C++ declaration required would be:
double salesPerf [30] [12];

An alternate declaration might be:

```
typedef double salesMatrix [30] [12];
// . . .
salesMatrix salesPerf;
```

```
// Array declarations
const  int ROW_MAX = 30;
const  int COL_MAX = 12;
typedef double salesMatrix [ROW_MAX] [COL_MAX];
typedef char* cString;

// . . .
cString salesPerson [ROW_MAX];
cString month [COL_MAX];
salesMatrix salesPerf;
// . . .
// Initialize the arrays
int x, y;
for (x = 1; x <= ROW_MAX; x++) {salesPerson [x – 1] = new char[31]; salesPerson [x – 1] = " ";}
for (y = 1; y <= COL_MAX; y++) month [y – 1] = new char[4];
month = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

for (x=1; x <=ROW_MAX; x++)
     for (y = 1; y <= COL_MAX; y++) salesPerf [x - 1] [y - 1] = 0.0;
// . . .
```

5.4.1 Two Dimensional Arrays (continued)

Example 5-10: Assuming the definition of the **salesPerf** array of the previous example, figure 5-9 shows a C++ function to average sale for a particular salesperson.

Figure 5-9: Function to Determine the Average Sale of a Particular Salesman

```
// This function will accept the sales performance matrix and subscript for the related salesperson as input parameters
double salesPerfAverage (salesMatrix thisPerf, int sPerson)
{
    double totalSales = 0.0; double averageSale = -1.0;
    int y;

    if (sPerson <= ROW_MAX)
    {
        for (y = 1; y <= COL_MAX; y++) totalSales += thisPerf [sPerson - 1] [y - 1];
        averageSale = totalSales / COL_MAX;
    }
    return averageSale;
}
// Note: A negative average indicates that the subscript for the related salesperson was invalid
```

5.4.2 Multi-Dimensional Arrays

An n-dimensional (n-D) array can be specified by simply specifying **n** dimensions in the declarations. The required syntax is summarized in figure 5-10:

Figure 5-10: Syntax for n-D Arrays

n-D_ArrayDefinition ::=
<Type> <ArrayName> "["<Length1>"]" "["<Length2>"]" . . . "["<LengthN>"]";

Note:
1. The square bracket here does not mean "optional" but it is required as part of the syntax, hence the quotation marks.
2. The array length specified must be positive integer literal or an integer variable that has a positive value.
3. The data type specified may be any valid primitive or advanced data type.

Dimensions greater than 2 are not very prevalent in usage, due to the fact that these arrays have the potential to consume a lot of space.

5.5 Array of Strings

A 2-D character array is basically an array of strings. Recall that in dealing with strings (section 5.3) you can access (input and output) a string by simply referring to its name. For an array of strings, you can access a string by simply specifying its first (leftmost) index.

Example 5-11: The following code snippets illustrate how you may manipulate an array of strings.

```
// If we have
    char studentName [20] [30];  // 20 student names each of length 30 bytes
//  then
    cout << studentName [0]      // displays the first student name
    // …
    cout << studentName [19]     // displays the 20th student name

// We can therefore have the following:
   gets (studentName [x]);
    // …
   cout << "Well done " << studentName [x]; // where x lies in the range 0 . . 19
```

```
// Alternate Declarations
typedef char* cString; // Defines cString as a type
const  int sMAX = 20;
const  int lMAX = 31;
// . . .
cString studentName [20];          // 20 student names each of length to be determined
// . . .
// To allocate 30 bytes for each student name, we could do the following:
for (int x = 1; x <= sMAX; x++) {studentName [x – 1] = new char[lMAX]; studentName [x – 1] = " ";}
// . . .

//  then
    cout << studentName [0]      // displays the first student name
    // …
    cout << studentName [19]     // displays the 20th student name

// We can therefore have the following:
   gets (studentName [x]);
    // …
   cout << "Well done " << studentName [x]; // where x lies in the range 0 . . 19
```

5.6 Introduction to Pointers

A pointer is a variable that points to an address (typically of another variable). The syntax for defining a pointer is as follows (figure 5-11):

Figure 5-11: Syntax for Defining a Pointer

```
PointerDefinition ::=
<BaseType> *<VariableName>;
```

Example 5-12: Following are two pointer declarations

```
int *thisOne, *thatOne;  // Defines two pointers, thisOne and thatOne, which point to integers
float *thisFloat; // Defines a pointer, thisFloat, which points to a real number
```

The unary operator & means "address of" — it gives the address of an object. For this reason, it is often used to retrieve the address of a non-pointer variable. The address-of operator (&) applies to variables and array elements only; it does not apply to expressions, constants, or register variables.

Example 5-13: The statements below illustrate how the address-of operator works.

```
int thisNum; int* thisPointer;
thisPointer = &thisNum; // thisPointer now points to the address of thisNum
```

The asterisk (*) is used as the indirection (or de-referencing) operator in the context of pointer. When applied to a pointer, the indirection operator must be placed in front of the pointer variable; in this context, it literally means, "value of" the pointer variable that is being referenced.

Example 5-14: The following code illustrates the use of the address-of and indirection operators.

```
int x =100, y = 200, z[10];
int *ip, *ip2;              // pointers to integers
ip =&x;                     //  ip points to x
y = *ip;                    // y = 100
*ip = 0;                    // x =0
ip = &z[2];                 // ip points to z[2]
ip2 = &z[9];                // ip2 points to z[9]

// The following two statements are incorrect
ip = *ip2 ;                 // trying to assign an integer to a pointer
*ip = ip2;                  // trying to assign a pointer to an integer

//But the following is acceptable
ip = ip2;                   // ip now points to the same address as ip2
```

Note: Observe the multiple roles of the asterisk (*): when sandwiched between two numeric operands, it represents multiplication; when appearing in front of a pointer variable, it's the indirection operator; when appearing after a data type, it represents a pointer declaration. This feature is a case of *operator overloading*, a concept that will be revisited in lecture 7.

5.7 Pointers and Function Arguments

A pointer may be sent as an argument to a function for one of two reasons.
- to facilitate a call by reference;
- the pointer defines a character string.

The latter application will be discussed later; the former concept was introduced in the previous lecture (please review section 4.2). To add more clarity, let us look at another example. Suppose we have a function **F1** that manipulates two variables, **value1** and **value2**. It calls a second function **F2** to perform calculations on the variables and return control to **F1** for further processing. There are three ways to approach this problem (outlined in figure 5-12):
- Declare **value1** and **value2** as global variables;
- Declare **F2** to accept arguments as pointers to **value1** and **value2**;
- Declare **F2** to use reference parameters for **values1** and **value2**.

Figure 5-12: Changing Variables in one Function, from Another Function

```
// Inadequate solution: F2 has no impact on value1 and value2
void F1 ( )
{
        int value1, value2;
        // ...
        F2 (value1, value2); // call F2
        // ...
}

void F2 (float val1, val2)
{
        // ...
        val1 =…;
        val2 =…;
        // ...
}
/* If value1 and value2 were to be declared globally, then F2 would not need parameters but would have access to both
variables */
```

```
// Adequate solution: F2 has an impact on value1 and value2
void F1 ( )
{
        int value1, value2;
        // ...
        F2 (&value1, &value2);        // Arguments passed as address references
        // ...
}

void F2 (float *val1, val2)           // Parameters declared as pointers
{
        // ...
        *val1 = …;                    // F2 changes the objects in the referenced addresses /
        *val2 = ….;
}
```

Figure 5-12: Changing Variables in one Function, from Another Function (continued)

```
// An alternate solution with reference parameters (the preferred solution):
void F1 ( )
{
        int value1, value2;
        // ...
        F2 (value1, value2);
        // ...
}

void F2 (float &val1, &val2)            // Parameters declared as reference parameters.
{
        // An implicit link is established with supplied arguments at function call.
        val1 = …;               // No indirection or address-of operator required
        val2 = …;
}
```

5.8 Pointer Arithmetic

There are four arithmetic operators that can be used with pointers: +, -, ++ and --.
Let p be a pointer of on a given base type. Then
p++ points to the next address location from the original value of p.
p- - points to the previous location from the original value of p.
p+i points to the i^{th} (address) location after p.
p-i points to the i^{th} location prior to p.

Example 5-15: The following statements illustrate how pointer arithmetic works.

```
// Let p be a pointer to integers and assume the initial address value of 1000
// Also assume 32-bit integers (i.e. each integer occupies 4 bytes)
int* p;
p++;  // Assigns value 1004 to p
p- - // Assigns value 996
p = p+4; // Assigns value 1016
p = p-4; // Assigns value 984
```

Please note the following:

1. Only integer values can be added to or subtracted from a pointer variable. Each increment/decrement advances/retreats by the number of bytes represented by the base type.

2. If two pointers **p1** and **p2** are defined on the same base type, then **p1-p2** yields the number of elements that separate the two locations (pointed to by p1 and p2 respectively). However **p1 + p2** is not allowed.

5.9 Association of Pointers with Arrays

In C++, there is a close relationship between pointers and arrays. In fact, most array implementations can be substituted by pointers implementations (typically more efficiently). The converse, though possible is not always advisable.

Example 5-16: You may create an NTS either via a character array or a pointer.

```
char thisString [20];        //  Creates a C-string of 20 bytes
char* thisString;            // Creates a C-string of variable length; space must be subsequently allocated
```

Note: When an un-indexed array name is used in an expression, it yields an implicit pointer to the first element in the array (however, that pointer cannot be changed). Other elements of the array may be referenced either by array subscript or by pointer arithmetic. This principle applies to all arrays in general, irrespective of the base type.

Example 5-17: Here is another illustration that an NTS may be implemented via a character array or pointer to characters.

```
// Consider the following declarations
char thisString1 [20];
char *thisString2;
int x;

// We can have the assignment
strcpy(thisString2, thisString1); // thisString2 = thisString1;
// which is equivalent to
thisString2 = &thisString1[0];

/* Given the assignment above, observe that
thisString2 + x  corresponds to &thisString1 [x] so that
*(thisString2+ x ) corresponds to thisString1 [x]
* (thisString2 + 4) corresponds to thisString1 [4] and
thisString2 +4 corresponds to &thisString1 [4]   */
```

String Constants: There is an exception in the way pointers are handled in respect of C-strings: if the subscript is not specified, the entire string is accessed; subscripting when used, refers to specific characters within the string.

Example 5-18: Not specifying a subscript means accessing the entire string

```
char thisString1 [20];
char *thisString2;
// …
// The following are valid statements
thisString2 = new char [20]; thisString2 = “pointers are fun”;
thisString1 = “I love pointers”;
cout << thisString2 << “  ” << thisString1;
```

5.9 Association of Pointers with Arrays (continued)

From the preceding two examples in this section, it should be clear to you that a C-string may be declared either as a character array, or a pointer to characters. For instance, if you want to define a function to accept a string as a parameter, you could use either of the two approaches. It is important that you appreciate this feature about C-strings; in many cases, pointer implementations (of strings) are more flexible than array implementations.

Dynamic Lists: You can capitalize on the fact that C++ treats arrays like pointers by manipulating dynamic arrays (lists) in the following way:

- Create a pointer of the base-type of interest.
- Determine the number of items to be added to the list at a specific point in time.
- Use the **new** operator to allocate memory for items to be added to the list (possibly in addition to items already in the list).
- Construct an iterative loop for adding the new items to the list, treating the list as you would an array.

You have already seen examples of how the **new** operator is used. More formally, here is the syntax (figure 5-13):

Figure 5-13: Syntax for Defining a Dynamic List, and Allocating Memory for it

```
DynamicListDefinition ::=
<BaseType>* <TargetList>; // You must specify the base-type and the name of the target-list
// …
<TargetList> = new <BaseType>"["<Length>"]"; // You must specify the amount of items
// …
```

C++ will allocate memory (in bytes) equivalent to the product of the length of the base-type, and the specified length of the list.

Example 5-19: Figure 5-14 provides section of code illustrating how you can load student records in a list.

Figure 5-14: Illustrating Construction and Manipulation of a Dynamic List

```
// …
typedef char* cString;
struct studentRec
{
 cString sID_Number;
 cString sName, sSex, sMajor;
};
typedef studentRec* studentList;

studentList inputStudents(); // Function Prototype
```

Figure 5-14: Illustrating Construction and Manipulation of a Dynamic List (continued)

```
// ...
int main(int argc, char *argv[])
{
   studentList sList; // sList will contain multiple student records
   // ...
   sList = inputStudents(); // sList is loaded by simply calling function inputStudents()
  // ...
}

// Input Student function
studentList inputStudents()
{
    int x, numStuds;
    bool exitTime = false;
    char exitKey, dummy;
    studentList thisList; // A list implemented via pointers

    // Find out how many students are to be processed and create an array for them
    cout<< "\nEnter desired number of students: ";
    cin >> numStuds; char dummy = getchar();

    // Allocate Memory for Array
     thisList = new studentRec [numStuds];

     // Accept information for each student in the list
     for (x = 1; x <= numStuds; x++) // For each student
     {
          studentRec currentS;
          currentS.sID_Number = new char [10];
          currentS.sName = new char [31];
          currentS.sMajor = new char [31];
          currentS.sSex = new char [7];
          currentS.sID_Number = current.SName = currentS.sSex = currentS.sMajor = " ";

          cout << "\nEnter details for student " << (x) <<endl;
          cout <<  "\nID Number: ";
          gets(currentS.sID_Number);
          cout <<  "\nName: ";
          gets(currentS.sName);
          cout << "\nSex: ";
          gets(currentS.sSex);
          cout << "\nMajor: ";
          gets(currentS.sMajor);

          thisList [x-1] = currentS; // Store in the list (as if in an array)
      };   // End For each student

     return thisList; // Return the list
}// End Input Student
```

5.10 Caution with Pointers

In using pointers, you should always take caution to avoid exception errors resulting in program crash. Following are some guidelines:

1. Never attempt to use a pointer before giving it an initial value. By convention, if a pointer contains the null value, it is assumed to point to nothing. Give all unused pointers the null value and avoid using null pointers, except to change their values.

2. If p is a pointer, then to check for p being null, use the construct.

```
if (p)        // if p is not null
if (!p)       // if p is null
```

3. You can introduce several levels of multiple indirections — where you can create pointers to other pointers. However, it is advised to avoid going beyond two levels.

   For example:

```
int **thisNumber; // defines a pointer to a pointer of integers
```

4. Logical errors associated with pointers are not always easy to track, so be careful.

5. Comparing pointers of different base types is invalid. For valid comparisons, not only must the base type be the same, the pointers must be accessing the same data structure.

5.11 Pointers to Structures and Arrays of Structures

As conveyed and illustrated in in example 5-14, you can define pointers of structures. This strategy is very useful when maintaining dynamic lists.

Example 5-20: Figure 5-15 illustrates how a pointer of student records (structures) could be constructed and manipulated.

Figure 5-15: Illustrating Pointer to Student Structures

```
struct dateType
{ int year , month, day};
  // ...
struct studType
{
     int studNumber;
     char studLastName[15];
     char studFirstName[15];
     dateType studDateofBirth;
     char studMajor[30];
     float studGPA;
};
studType* studPtr; studType aStudent;
// ...
/* To reference elements, use the brackets to ensure that the dot-operator does not take precedence over the
indirection operator, as illustrated below: */
studPtr = &aStudent;
cout<< (* studPtr).studNumber << "     " << (*studPtr).studLastName; // is equivalent to
cout << aStudent.studNumber << "  "<< aStudent.studLastName;

// Alternately, use the arrow (->) operator, as illustrated below:
studPtr = &aStudent;
cout << studPtr->studNumber << "  " << studPtr->studLastName; // is equivalent to
cout << aStudent.studNumber << "  " << aStudent.studLastName;
```

Notice the use of the arrow (→) operator in this example. This is a special indirection operator for more complex situations with pointers. Essentially, if **p** is a pointer to a structure (or some other complex object), then **(*p).element** is equivalent to **p→element** in any expression. So in the example, **(*studPtr).studNumber** is equivalent to **studPtr →studNumber**.

5.11 Pointers to Structures and Arrays of Structures (continued)

You may define an array of structures. This strategy is very useful when maintaining finite lists.

Example 5-21: Figure 5-16 illustrates how a pointer of student records (structures) could be constructed and manipulated.

Figure 5-16: Illustrating Array of Student Structures

```
struct dateType
{ int year , month, day};
  // ...
struct studType
{
     int studNumber;
     char studLastName[15];
     char studFirstName[15];
     dateType studDateofBirth;
     char studMajor[30];
     float studGPA;
};

// ...
studType sList[30];
// ...
// To reference elements, use array subscript and the dot-operator as in the following example
cout << sList[x].studNumber << "  " << sList[x].studLastName;
// ...
```

5.12 Arrays of Pointers

You can define an array of pointers. This strategy is applicable to a scenario where you desire to maintain a finite list of dynamic lists.

Example 22: Suppose that we wanted to maintain separate pointers to students where each pointer represents a list of student objects. The required declaration is shown in figure 5-17.

Figure 5-17: Illustrating Array of Pointers to Student Structures

```
struct dateType
{ int year , month, day};
  // ...
struct studType
{
     int studNumber;
     char studLastName[15];
     char studFirstName[15];
     dateType studDateofBirth;
     char studMajor[30];
     float studGPA;
};
// ...
studType* studPtr[20];                    // Twenty pointers of StudType
// ...
/* To reference each pointer, we use the indirection operator along with the array subscript and the dot-operator, or
the -> operator as illustrated below:              */
cout << (*studPtr[x]).studNumber << "   " << (*studPtr[x]).studLastName;  // or
cout << studPtr[x]->studNumber << "  " << studPtr[x]->studLastName;
```

Again, notice the use of the arrow (→) operator in this example, as well as in figure 5-15. To reiterate for emphasis, if **p** is a pointer to a structure (or some other complex object), then **(*p).element** is equivalent to **p→element** in any expression. So in the example, **(*studPtr[x]).studNumber** is equivalent to **studPtr[x] →StudNumber**, and **(*StudPtr[x]).studLastName** is equivalent to **studPtr[x]→studLastName**.

5.13 Passing Structures to and Returning Structures from Functions

As mentioned earlier (section 5.1), structures may be used as function parameters; they may also be returned from functions. Obviously, in order for a structure to be used as a function parameter, it must first be declared (globally) so that it is known to the related function.

In the upcoming example a function accepts and returns an argument of the same type (a structure). Note however, that generally speaking, the return type and the parameter type need not be the same. The sole purpose of the example is to illustrate that a function may accept as argument, a structure variable, and/or return a structure variable.

Example 5-23: Assuming the student record of earlier discussions, suppose that it was desirous to have a function that receives a student record as input argument, makes changes to it, and then returns it to the calling statement. Figure 5-18 illustrates how you would set this up.

Figure 5-18: Illustrating how a Structure is passed to and from a Function

```
struct dateType
{ int year , month, day};
  // ...
struct studType
{
     int studNumber;
     char studLastName[15];
     char studFirstName[15];
     dateType studDateofBirth;
     char studMajor[30];
     float studGPA;
};
// ...
studType modifyStud (studType aStudent); // Function Prototype
// ...
void main ( )
{
  studType thisStudent;

  // ...
  thisStudent = modifyStud (thisStudent);
  // ...
}

/* modifyStud changes the value of a studType instance and returns the new value to the calling statement. An
alternative to this would be to let modifyStud have a reference parameter, but without a return type. Review
section 5.7 */

studType modifyStud (studType aStudent)
{
  // ... // Note: All elements of the structure are known to this function; modifications would be made as required
  return aStudent;
}
```

5.14 Passing/Returning Arrays/Pointers to/from Functions

You can pass pointers and/or arrays to functions in the standard way that arguments are sent to corresponding parameters of the function in question. You can also return a pointer or an array from a function; this was illustrated in figure 5-14 of section 5.9. To return an array from a function is a bit tricky: you must first define the array as an advanced (programmer defined) type. You do this by using the **typedef** statement. The same approach may be used for pointer. However, for pointers, you have more flexibility — in the function definition, you can indicate that the function returns a pointer by simply specifying the pointer type in the normal way. Moreover, remember that you can define a list as a pointer, and then manipulate it as an array; this is the preferred approach.

Example 5-24: Figure 5-19 illustrates how you could construct a function that accepts a dynamic list of student records, manipulates the list, and returns it to the calling statement.

Figure 5-19: Passing and Returning a Dynamic List to and from a Function

```
struct dateType
{ int year , month, day};
  // ...
struct studType
{
     int studNumber;
     char studLastName[15]; char studFirstName[15];
     dateType studDateofBirth; char studMajor[30]; float studGPA;
};
// ...
const int LIM = 50;
typedef studType* studList // Dynamic list of student records
//typedef studType studList[LIM]; // Alternate static list of 50 student records

studList constructList (studList thisList); // prototype
// ...
void main ( )
{
studList sList;
// ...
sList = constructList (sList);
// ...
} // End of main function

studList constructList (studList thisList)
{ // Inserts students in a list of student records
     int x, numStud; studType thisStud;
     // Prompt for numStud; then use to construct thisList
     cout << "Number of Students: "; cin >> numStud; char dummy = getchar();
     for (x = 1; x <= numStud; x++)
     {
       // . . . Prompt for fields of thisStud . . .
       thisList[x – 1] = thisStud; // Store in the list
     }
     return thisList;
}
```

5.15 Structures with Arrays, sub-structures, and/or Pointers

As you probably have gathered by now, a structure could contain members that are arrays, sub-structures, or pointers. Observe examples 5-1, 5-3, and 5-14 through 5-19; they all involve structures that include character arrays (C-strings), and in some cases pointers and/or nested structures. Moreover, there is nothing forbidding other kinds of arrays [apart from C-strings] from being included in a structure.

A structure containing other sub-structure(s) is said to have nested structure(s). Additionally, a structure may also be used for other advanced data types (ADTs) not fully covered in this course — ADTs such as linked lists, stacks, queues, trees, graphs, etc. as typically covered in a course on Data Structures and Algorithms; they are clarified further in lecture 12. In constructing structures that include other advanced components, it is highly recommended that you observe the KISS strategy (keep it simple but not simplistic).

5.16 Self-Referencing Structures

A self-referencing structure is a structure that has a pointer that points to itself. These kinds of structures find relevance in the management of linked lists, queues, trees, graphs, and other ADTs.

Example 5-25: Figure 5-20 illustrates two approaches to representing a linked list of student records.

Figure 5-20: Representing a Linked List of Student Records

```
// A linked list of student records may be defined as follows:
struct dateType
{ int year , month, day};
// ...
struct studList
{
    int studNumber; char studLastName[15];  char studFirstName[15];
    dateType studDateofBirth;
    char studMajor[30];  float studGPA;
    studList* next;
} frontS, currentS, rearS;
```

```
// Alternate definition of a linked list of student records:
typedef char* cString;
struct dateType
{ int year , month, day};
// ...
struct studType
{
    int StudNumber; cString studLastName, studFirstName;
    dateType studDateofBirth;
    cString studMajor;  float studGPA;
}
```

```
struct studList
{
    studentType info;
    studList* next;
}
// …
studList frontS, currentS, rearS;
```

5.16 Self-Referencing Structures (continued)

For self-referencing structures and other ADTs, two operators are typically used: **new** and **delete** for allocating and returning memory space to and from the program. These data structures will be revisited later in the course (lecture 12); however, a full treatment is more appropriately covered in a course in Data Structures and Algorithms.

5.17 Unions

A C++ union is a memory location that is shared by two or more variables (of different types). The programmer keeps track of which type is currently stored in the union and uses it appropriately. Only one type is applicable at a time.

A union is declared in a manner that is similar to a structure declaration. The main difference is that the **union** keyword takes the place of the **struct** keyword.

Example 5-26: Following are two illustrations for definition of a union called **itemType**.

```
union itemType
{
int itemNumber;
char itemSerial[9];
} item;
```

Is equivalent to:

```
union itemType
{
int itemNumber;
char itemSerial[9];
};
// …
itemType item;
```

Considering this illustration, following are a few related guidelines:

1. In this illustration of example 5-26, **item** is referenced either by **itemNumber** or **itemSerial**, but not both.

2. When the union is declared, the compiler will allocate space to facilitate the longest variable type in the union.

3. Elements of the union are accessed via the dot-operation in a manner that is similar to accessing of a structure. For example, if **year** is a numeric variable, we may have the following assignment:

   ```
   item.itemNumber = year*rand( );
   ```

4. Any valid data type (including arrays, structures, pointers, etc.) may be included in a union.

5.17 Unions (continued)

An anonymous union is a union that does not have a type name. Its members are treated as normal variables and are accessed without the dot-operator.

Example 5-27: Below is an anonymous union consisting of three fields.

```
union
{   int itemNumber, year;
    char itemSerial[9];
};
// …
itemNumber = year * rand();
```

Anonymous unions can be useful in gathering miscellaneous program variables together, thus making your code more organized.

5.18 Bit-Fields

Another unique feature about C++ is the ability of the programmer to specify bit-fields within a structure. Bit-fields are particularly useful when space conservation is critical. For instance, you can store several Boolean values in one byte. Secondly, certain device interfaces transmit information encoded in bits within one byte. A third advantage of bit-fields is data encryption — certain encryption algorithms need to access bits within a byte.

A bit-field is indicated by simply specifying the length (in bits) after a colon. Bit-fields are commonly used for analyzing the input from hardware devices.

Example 5-28: The following example shows a structure consisting of bit-fields that may be applicable to a hardware device.

```
// The status port for a serial communications adapter may be defined as follows:
struct  deviceStatusType
{
    unsigned deltaCleartoSend : 1; // change in Clear-to-send signal
    unsigned deltaDataSetReady : 1; // change in Data-set-ready signal
    unsigned trailingEdge : 1;
    unsigned deltaRecieveLine : 1; // change in receive line
    unsigned cearToSend : 1;
    unsigned dataSetReady : 1;
    unsigned ring : 1;
    unsigned recieveLine : 1;
} Status;
```

Note: Bit-fields cannot be included in an array; they cannot be declared as **static**; and they may be mixed with non-bit elements in a structure.

5.19 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- A structure is a collection of variables, possibly of different data types, referenced under the same name. It defines a new data type upon which variables (more precisely, object instances) can be declared.
- An array is a finite list of items of a given base type. C++ supports 1-D arrays and multi-dimensional arrays. It is generally a good idea to initialize an array via an initialization loop. However, if the array is small, it can be initialized (at declaration) without using such a loop.
- A string is a special 1-D array of characters. A string can also be implemented as a pointer to characters. C++ provides special string functions for the manipulation of strings.
- A pointer is a variable that points to an address of a specific base type. Through pointers, you can construct and manage dynamic lists of data. In fact, C++ manages pointers in a manner that is similar to its management of arrays. You can declare a data item as a pointer and manipulate it as an array, thus adding flexibility to your code.
- You can define pointers of structures or an array of structures. You can also have pointers and/or arrays within structures. You can also define and manipulate an array of pointers.
- You can obtain a dynamic list by defining it as a pointer to a specific base-type, and manipulating it as an array.
- You can pass structures, pointers, or arrays to functions. You can also return a pointer, array, or structure from a function.
- C++ allows you to define self-referencing functions in a compact and efficient manner.
- A union is a memory location that is shared by two or more variables (of different types). The programmer keeps track of which type is currently stored in the union and uses it appropriately. Only one type is applicable at a time. However, an anonymous union is a union that does not have a type name. Conveniently, it can be used to define miscellaneous variables in a program.
- C++ facilitates the definition of bit-fields in a structure. A bit-field stores data in bits instead of bytes.

Structures, arrays, and pointers: these form part of the core of C++ programming. But wait, there is much more: The next lecture introduces the object-oriented features of C++.

5.20 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapter 9.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapters 8 & 10.

[Kernighan & Richie 1988] Kernighan, Brian W. & Dennis M. Richie. 1988. *The C Programming Language*. Boston: Prentice Hall. See chapters 5 & 6.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapters 5, 6, & 10.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapters 7 & 9.