Module 3: Arrays, Records and File Processing

Lesson 1: Arrays

Learning Outcomes

- LO 3.1.1 Declare, access, and modify the contents of one dimensional and two-dimensional array with size n.
- LO 3.1.2 Pass a one and two dimensional array to a function.

All the data types that are covered so far were simple data types. That is, only one value would be stored at a time. However, a structured datatype is composed of several data items in which each data item a simple data type. There are more structured data types that would be discussed on the succeeding chapters however only homogenous structures (i.e. structures with the same datatype, or arrays) would be covered for this chapter.

One Dimensional

As mentioned earlier, an array is a group of similar datatypes with a fixed number of components. So, a one-dimensional array can be illustrated as collection of similar datatypes in list form. Each element has a unique number starting from zero (0) and ending at n – 1, where n is the fixed size of the array. The declaration and initialization of the array are as follows.

Syntax	**Example**
`datatype name [size] = {var[0], var[1], var[2], …var[size-1]};`	`int x[5] = {4,2,6,1,8};`

Array `x` can be illustrated as a column of integers shown on Figure 1.

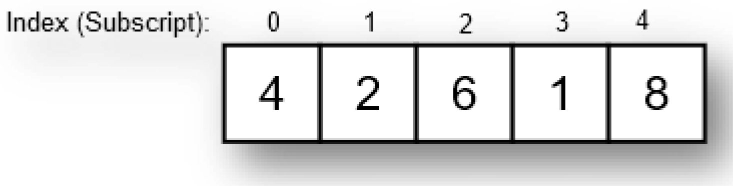

Figure 1: Visualization of a one dimensional array

Accessing values on the array can be done by having the array name followed by its index. Refer to the format below.

Syntax	**Example**

`name [index]`	`/*display first element of array x*/`
	`cout<<x[0];`

However, when the size of the array would increase, it would be difficult to manually access each element, so a counter controlled loop (i.e. typically a `for` loop) would be used to access each element in an array.

Format	**Example**

`for (int i=0;i<size;i++)`
`{`
`      name [i];`
`}`
	`/*display all the elements of array x*/`
	`for (int i=0;i<5;i++)`
	`{`
	`      cout<<x[i]<<",";`

	}

Two Dimensional Array

The previous subsection deals with one-dimensional arrays to store and manipulate in list form. However, there are numerous cases that the stored data is manipulated using table from. Take for instance if you would intend to perform matrix manipulation, the elements are stored on row and column coordinates. The syntax and example for declaring and initializing a two dimensional array is shown below:

Syntax	**Example**

`dataTypename [row][col] = {{rowElem0},`
`{rowElem1},`
`{rowElemN}};`
	`int matrix[2][2] ={{1,2},`
	`{3,4}};`

The example shows a 2 row, 2 column two-dimensional integer array. This can be visualized on Figure 2.

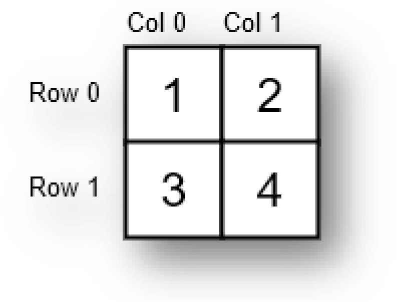

Figure 2: Two dimensional array visualization

Accessing the values on a two-dimensional array would require you to specify the value of the row and the value of the column. But when the size of the array is huge, it is recommended to use a nested loop operation to sequentially access each element.

Format	**Example**

`for (int i=0;i<rowSize;i++)`
`{`
`    for (int j=0;j<colSize;j++)`
`    {`
`        arrayName[i][j];`
`    }`
`}`
	`/* Display each item on the matrix`
	`*/`
	`for (int i=0;i<2;i++)`
	`{`
	`    for (int j=0;j<2;j++)`
	`    {`
	`    cout<<matrix[i][j];`
	`    }`
	`}`

Passing arrays to functions

Arrays in C++ may it be one dimensional or two-dimensional can only be passed to a function by reference. However, it is unnecessary to include an & operator before the variable name on the formal parameter, instead a pair of empty brackets will satisfy the operation. The syntax and example of passing a one-dimensional array shown below.

Syntax	Example

`dataType fName (dataType arrName []);`
`…`
`dataType arrName[size]`
`fName (arrName);`
	`int initArr(int x[]);`
	`…`
	`int x[5];`
	`initArray (x);`

Two dimensional arrays on the other hand requires the size of the column so that the compiler would know where a specific row would end. Since, as mentioned earlier the data on a two-dimensional array is actually stored on the memory in row major ordering. Thus a row boundary is necessary in order to indicate where the current row ends and the next row would begin.

Syntax	Example

`dataType fName (dataType arrName [][colSize]);`
`…`
`dataType arrName[rowSize][colSize]`
`fName (arrName);`
	`int initArr(int xy[][5]);`
	`…`
	`int x[3][5];`
	`initArray (x);`

Array Manipulations (Searching and Sorting)

Locating an item on an array is one of the most common operation on an array. The sequential search is the simplest searching algorithm that exist. It would perform the searching process by comparing the search item on each individual element on the array. The search would either determine the location of the searched item or indicate if the item is found or not (Refer to Figure 3).

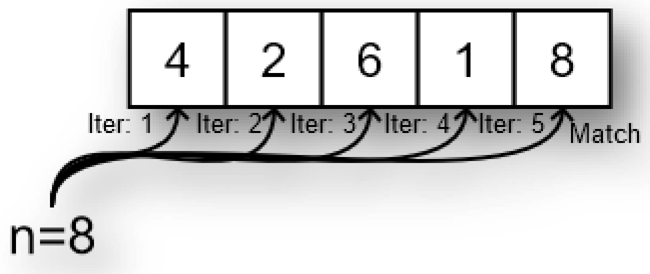

Figure 3: Visualizing linear search

The C++ code implements a sequential search and would determine if the search item exist or not exist on the list.

Example

```
bool isPresent (int lst[], int s, int n)
{
    for (int i=0;i<s;i++)
    {
        if (lst[i]==n)
            return 1;
    }
    return 0;
}
```

Handling data is done more efficiently if the data were sorted if some form of ordering criteria. For example, dictionaries, payrolls, and student list are arranged alphabetically to facilitate the searching. Although there are a lot of sorting algorithm, only the simplest is covered for this chapter.

Given that the arrangement is in ascending order, the bubble sort algorithm would compare the successive elements of an array (i.e. `arr[i]` and `arr[i+1]`) then swap the elements the elements if `arr[i]` is greater than `arr[i+1]`. This would effectively make the smaller element float to the top while the bigger element sink to the bottom. The first iteration would operate on `arr[0]` to `arr[n-1]` the next would be on `arr[0]` to `arr [n-2]` and so on. The iteration would terminate once there would be no swapping left. Refer to Figure 4.

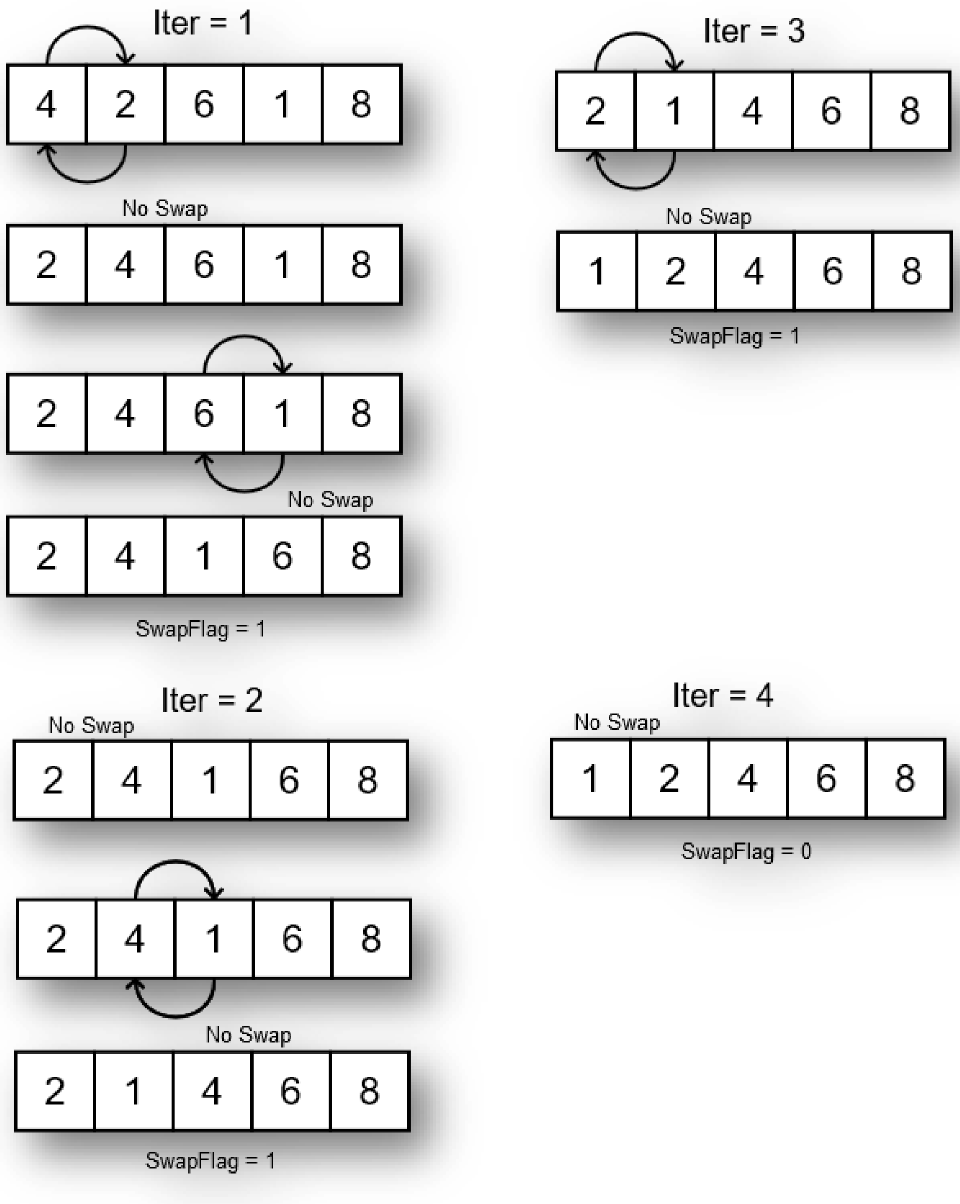

Figure 4: Visualizing bubble sort

The source code for the bubble sorting algorithm is shown below.

Example

```
void bubbleSort (int arr[], int s)
{
    bool swp = 1;
    int temp = 0;
    while (swp)
    {
        swp = 0;
        s--;
        for (int i=0;i<s;i++)
        {
            if (arr[i]>arr[i+1])
            {
                swp = 1;
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1]=temp;
            }
        }
    }
}
```

Precautions when using arrays

Here are some principles that needs to be practiced in order that unexpected errors would be encountered.

- C++ has no out-of-bound index checking. So, when the indicated index is out of range, the program would try to access the component of the specified out-of-bound index. This would result to several strange matters to occur. That is why it is solely the programmer’s responsibility to check the limit of a particular array.
- An equality operator cannot be used to allow aggregate operation on the array. An aggregate operation is the term use manipulate the entire array as a single unit. Example,

```
int x [3] = {1,2,3};
int y[3];
y = x;//this is an erroneous operation
```

Instead the transfer operation can be accomplished through,

```
for(inti=0;i<3;i++)
    y[i] = x[i];
```

Lesson 2: Strings and Vectors

Learning Outcomes

- LO 3.2.1 Allocate memory to manipulate strings
- LO 3.2.2 Use vectors to simplify block memory usage.

Processing characters arrays (i.e. strings) are of special interest in C++, since they are processed differently compared to other datatypes. The widely used character set are ASCII and EBCDIC.

C-style String

Based on chapter 1, strings are an array of characters enclosed in double quotation marks. However in C++, C-strings are null (i.e. \0) terminated, so the null character must only appear on the last position of the string. From the `<cstring>` header, C++ provides predefined functions to handle character arrays. Basic function include, copying, comparing and determining the length (refer to Table)

Function	**Explanation**
`strcpy (s1,s2)`	Copies string s2 to string s1. Note: string s1 must be as large or larger than string s2
`strcmp (s1,s2)`	Returns 0 if s1 and s2 are the same. Returns a value greater than 0 s1 is greater than s2 Returns a value lesser than 0 is s1 is lesser than s2.
`strlen (s1)`	Returns the length of the string excluding the null character.

Comparing string would work by comparing character by character using the system’s ordering sequence. For example:

1. The C-string `"Land"` is less than the C-string `"Water"` because the first character of `"Land"` is less than the first character of `"Water"`.
2. The C-string `"Land"` is greater than the C-string `"Lan"` because the last character of `"Land"` is non-existent on `"Lan"`.
3. The C-string `"Land"` is lesser than the C-string `"land"` because the first character of `"Land"` is lesser than the first character of `"land"`.

Example

```
    char str[10], str1[10];
    strcpy (str,"HelloWorld");
    cout<<str<<endl;
    cout<<strcmp(str,str1)<<endl;
    cout<<strlen(str)<<endl;
```

Inputting data to the string can be done by using the `cin>>` operation however, since C++ has no checking of the bounds, an input that would exceed the size of the character array would cause serious problems on adjacent memories. The `cin.get (str, size)` would be a more favorable way of inputting data to a character array. To produce an output the cout<< operation can readily be used.

Example

```
    cin.get(str,10);
    cout<<str;
```

`string` class type

The string class type in C++ provides a convenient way of processing character array. It provides the same functionality of the C-string header with more nifty features such as automatically increasing the capacity of the string. The string class is defined on the `<string>` header (i.e. `#include <string>` to make use of the class). A string object has a member function named length that returns the size of the string. This means that the string can be treated as an array with the length as its size. Moreover, size the length is the actual occupied elements in a string, accessing illegal indexes is prevented. To declare a string object, use the statements on the table below.

Statement	**Explanation**
`string str`	This statement would construct a new string object.
`string str ("default")`	Constructs a string object initialized with “default”.
`string str (str1)`	Constructs a string object initialized with the contents str1.

When the string object (i.e. str and `str1`) would be created, different functions are available to access elements, modify contents, and compare strings.

Expression	**Explanation**
`str.at(index) or str[index]`	Returns the read and write reference of str at index.
`str = str1`	Copies the contents of str1 to str. The contents of str are erased then allocates memory for str1.
`str +=str1`	Concatenates str with str1 then stores the result on str
`str.empty ()`	Returns true if str is empty false if otherwise
`str.insert(pos, str1)`	Inserts the substring str1 on str starting at position pos.
`str.remove (pos, length)`	Removes a substring with size length on str starting at position pos.
`== != < > =< >=`	Comparison operator for strings. All comparison are lexicographical (dictionary method).
`str.find(str1)`	Locates str1 from str

	then return the index of the first occurrence.

Note that the assignment and comparison operator for string are not applicable for C-strings. The example below, make use of these functions.

Example

```
string str, str1;
str = "Hello";
cin>>str1;//Input: "There"
str += str1;
str.insert(2,"Hi");
cout<<str;//output: HeHilloThere
```

`vector` class type

The arrays that are discussed so far are the one with fixed sizes. Which means that only a fixed number of elements can be stored on the particular array. Other limitation is on the aspect of insertion and deletion. For instance when an element would be inserted on a specific position, some elements would be shifted to in order to accommodate the inserted item. This case also applies on deletion, there must be no empty position in between elements.

The `vector` class in C++ is another alternative to implement a list. A variable that is declared from a `vector` class (note: you must `include <vector>` library) is called the vector object. The `vector` has the flexibility over arrays since it can grow and shrink in size during runtime. So, there is no need to declare the size of the vector during compile time. When declaring a vector object, the type of element that is to be stored must be specified. The table below describes the different ways the vector object is declared.

Statement	**Explanation**
`vector <elemType> vecList`	This operation would create a vector object named vecList without a specified size.
`vector <elemType>vecList (size)`	A vector object named vecList with the specified size would be created and then initialized with default values.

After declaring the vector, basic operations such insertion, deletion and getting the value of the specified element would follow. Given that `vecList` is still the vector object, the table below shows the vector operations.

Expression	**Explanation**
`vecList.at(index) or vecList[index]`	Returns the read/write reference for of vecList at index.
`vecList.front()`	Returns the value of the first element on the vector
`vecList.back()`	Returns the value of the last element on the vector
`vecList.clear()`	Deletes all the items on the vector
`vecList.push_back(elem)`	Insert elem into after the last

	element on vecList.
`vecList.pop_back()`	Deletes the last element on vecList.
`vecList.empty()`	Returns true if the vecList is empty, false if otherwise.
`vecList.size()`	Returns the current number of elements on vecList. The value returned is an unsigned integer
`vecList.max_size()`	Returns the maximum number of elements that can be accommodated by vecList.

Since vector are like arrays, the starting index is 0. Accessing a vector element is similar to an array, however the at statement in a vector would return an exception if the element that is accessed is out of range. Examples are shown below.

Example

```
    vector <int> vecList(5);
    for (int i=0;i<5;i++)
        vecList.at(i) = 1 + i;

    vecList.pop_back();
    vecList.push_back(10);
    vecList.push_back (12);

    for (int j=0;j<vecList.size();j++)
        cout<<vecList.at(j)<<",";
//output: 1,2,3,4,10,12
```

The example would assign values 1, 2, 3, 4, 5 to `vecList` starting from the first element. Then the last element (i.e. 5) is popped and the values of 10 and 12 are pushed to the `vecList` respectively thus producing the output.

Lesson 3: Structures

Learning Outcomes:

- LO 3.3.1 Implement structure and array of structures in a C++ code
- LO 3.3.1 Pass a structure to a function and implement structures within a structure

Declaring and Manipulating Structures

A structure is a group of data items with different data types. Unlike arrays that only group similar data types, structures would better represent real world data, for these data are heterogeneous in nature. The `struct` statement in C++ is reserved word that would begin the definition of the new structure. This would be followed be the name of that particular structure. An open curly brace would follow in order to form a block of the member data for that structure. The definition would end with a closing curly brace followed by a semi-colon. Refer to the general syntax and the example below.

Syntax	**Example**

`struct structName`
`{`
`dataType1 indentifier1`
`dataType2 indentifier2`
`…`
`dataTypeN indentifierN`
`};`
	`struct employee`
	`{`
	`int IDNum;`
	`string name;`
	`double monthlySalary;`
	`};`

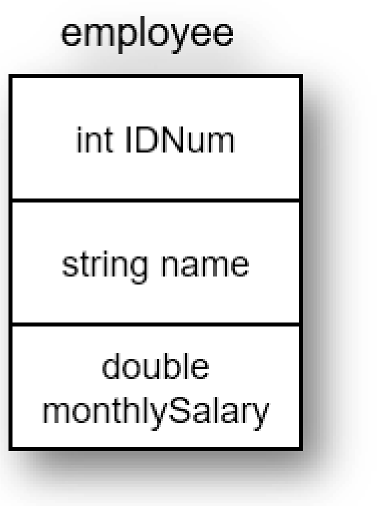

Figure 5: Visualization of the employee structure

After the structure is defined, this would now be similar to a new data type so when declaring you must follow the syntax below.

Syntax	**Example**
`structName structVariableName;`	`employee govtEmp;`

Accessing Structure Components

In arrays the relative position of the element is accessed by supplying the index enclosed in brackets. In a structure, each component (member) is accessed by using the dot (period) operator. The syntax and example for this operation is a follows

Syntax	**Example**
`stuctvariableName.memberName;`	`govtEmp.IDNum = 12345;`
	`govtEmp.name = "Tim Johnson";`
	`govtEmp.monthlySalary = 12550.14;`

Assignment operation for structures

Aggregate operation can be done entirely on a structure. That is, if the structure are of the same type, the contents of onr structure can be transferred entirely to the memory allocation of another structure. The syntax and example is shown below,

Syntax	**Example**
`structVariableName1 =`	`employee newGovtEmp;`
`structVariableName;`	`newGovtEmp = govtEmp;`

Comparison operation for structures

Assignment operation may be done as an aggregate operation, but comparison is done member-wise. This means that a component must be accessed first before it can be compared to another component of the same data type from another structure. The syntax and example is shown below,

Syntax	**Example**
`structvariableName1.memberName`	`if (newGovtEmp.IDNum ==`
`(comparison operator)`	`        govtEmp.IDNum)`
`structvariableName2.memberName;`	`            cout<<"The Same ID";`

Array of Structure

So far the examples that are presented are scalar structure. However real world records would be composed of a list of structured data types. Take for example, there are 20 employees in an institution, rather than declaring an individual structure for each employee and array of structure can be created.

Syntax	**Example**
`structName structVariableName[size];`	`employee companyAEmp[20];`

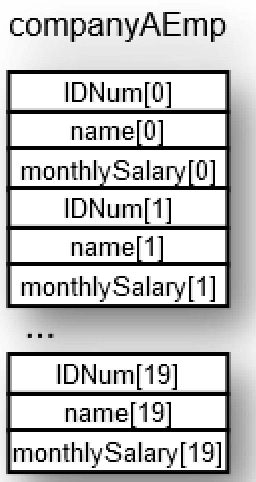

Figure 6: An array of structure example

Now that an array of structure is already declared, the rules when using arrays can be applied. For instance, assigning values on the member for the structure in an element of an array.

Example

```
    companyAEmp[0].IDNum = 1241221;
    companyAEmp[0].name = "Pete Timothy";
    companyAEmp[0].monthlySalary = 5000.10;
```

Structure within Structure

Structures demonstrate that it is capable of organizing data to create meaningful information. For this section it would be exhibited how to improve organization by using structures inside of structures. Take our previous structure definition for employee, if a new complex entry would be added, say for example address. It would be beneficial to create a separate structure to cater this complex entry.

Example

```
struct address
{
    string street;
    string brgy;
    string townCity;
    string province;
    int zipCode;
};

struct employee
{
    int IDNum;
    string name;
    address empAddress;
    double monthlySalary;
};
```

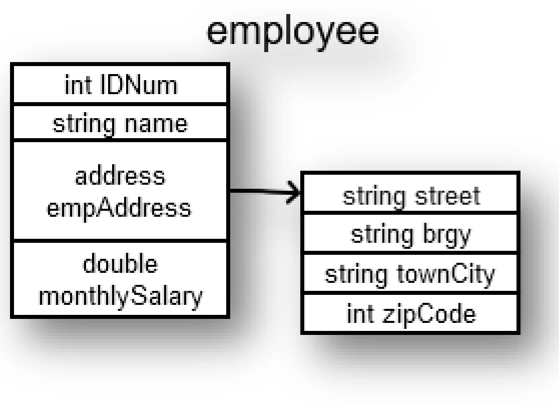

Figure 7: An address structure within an employee structure

When accessing the member of a structure within a structure you must follow the appropriate hierarchy.

Example

```
govtEmp.empAddress.street = "J.P.Rizal";
govtEmp.empAddress.brgy = "Poblacion";
govtEmp.empAddress.townCity = "Baybay";
govtEmp.empAddress.province = "Leyte";
govtEmp.empAddress.zipCode = 6520;
```

Lesson 4: Sequential File Access

Sequential File Access

- LO 3.4.1 Create and open a sequential access file.
- LO 3.4.2 Read and write a record into a sequential access file.
- LO 3.4.3 Test the end of file and close a sequential access file

Creating and Opening a Sequential file

The information that is stored on structure would still exist on the memory. So, if the program would terminate the stored data would be lost. To compensate this, an output file can be created to store the information in a permanent memory medium such as hard disk. Sequential access file or text file is a form of output file that is composed of text lines. Accessing data is similar to the way you access data on a cassette tapes. The first entry must be accessed first before the succeeding entries. Likewise writing data is done in consecutive order.

To create a file object include the `fstream` library which directs the compiler to use the classes that are used for input and output. These classes are the `ofstream` and `ifstream` for output (writing) and input (reading) files respectively. These classes are derived from `ostream` and `istream` which the `cout` and `cin` object exist. They are handled the same way with `cin` and `cout`, but these streams are associated with real files. The syntax and example is shown below.

Syntax	Example
`ifstream fileObject`	`ifstream inEmpRecStor;`
`ofstream fileObject`	`ofstream outEmpRecStor;`

When opening an input or output file you must use the `open()` function to access the files on the computer's disk drive. The syntax and example is shown below.

Syntax	Example

`fileObject.open(filename, mode)`	`outEmpRecStor.open ("EmpRec.txt");`
	`inEmpRecStor.open ("EmpRec.txt");`

The modes that are optional arguments would be used to indicate how the file is to be opened. This would indicate where the position of the file pointer would begin. The table below describes the different operation.

Mode	**Description**
`ios::in`	This is the default mode for the ifstream object. This opens the file and allow for necessary input.
`ios::app`	This mode for the ofstream object would open a file for append. Allowing the program to write new data to the end of the existing file. If the file does not exist, the file is created before being written.
`ios::out`	This is the default mode for the ofstream object. This would create a new and empty file that would be written. If the file exist, its content will be overridden.

There are instances in which the file failed to open. When this happen, the `open()` function would not return any flag indicating the status of the file. So it is useful to check if the file is successfully opened by using the `is_open()` function. This function would return true if the file is successfully opened and false if otherwise. Refer to the syntax and example below.

Syntax	Example

`fileObject.is_open()`	`if (inEmpRecStor.is_open())`
	`{`
	`    //begin reading`
	`}`

Writing and reading a record into a sequential access file

Writing into a sequential file is similar with how you use cout. The << operator is used to write information to the file. The syntax and example is shown below.

Syntax	Example

`fileObject<<data;`	`outEmpRecStor<<companyAEmp[0].IDNum<<endl;`
	`outEmpRecStor<<companyAEmp[0].name<<endl;`
	`outEmpRecStor<<companyAEmp[0].montlySalary<<endl;`

Reading a sequential file is analogous to `cin`. You would use the `>>` operator to read the numerical content of a file place it on variable. On the other hand the `getline()` function would extract the string terminated by the default delimiter `'\n'`. Refer to the syntax and example below.

Syntax	Example

`fileObject>>data;`
`getline(fileObject,stringData, 'delimiter')`
	`inEmpRecStor>>companyAEmp[1].IDNum;`
	`inEmpRecStor.ignore();`
	`getline(inEmpRecStor,companyAEmp[1].name);`
	`inEmpRecStor>>companyAEmp[1].monthlySalary;`

The example used the `ignore()` function to discard the delimiter that is set between the first entry (which is an integer) with the next entry which is a string. However, since the delimiter is read as a character, no `ignore()` function is required for the next entry.

Testing for the end of file (EOF) and closing a sequential file

As indicated earlier, when reading a sequential access file one has to read from the start to the end. That is, a character is read from the file then the next character until the end of line. When all the lines are read this would mean that it is already the end of the file. This task can be accomplished by using the `eof()` function. If the file pointer would be at the end of the file it would return a true value otherwise false. Examine the syntax and example.

Syntax	Example

`fileObject.eof();`	`while(!inEmpRecStor.eof())`
	`{`
	`   //reading file`
	`}`

A sequential access file will now be closed so that the data that is stored in it will not be lost. The close() function would be used for this operation. Examine the syntax and example below.

Syntax	Example

`fileObject.close();`	`inEmpRecStor.close();`
	`outEmpRecStor.close();`