# ARRAYS, STRING AND STRUCTURES

## C++ Arrays

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

### Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement –

```
double balance[10];
```

### Initializing Arrays

You can initialize C++ array elements either one by one or using a single statement as follows –

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array –

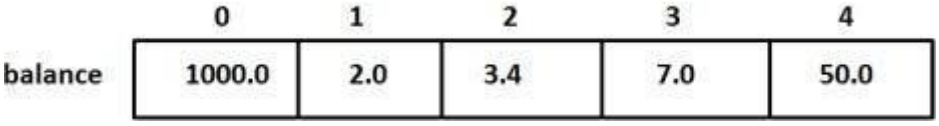
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above –




### Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays –

```
#include <iostream>

using namespace std;


#include <iomanip>

using std::setw;


int main () {


   int n[ 10 ]; // n is an array of 10 integers


   // initialize elements of array n to 0

   for ( int i = 0; i < 10; i++ ) {

      n[ i ] = i + 100; // set element at location i to i + 100
```

```
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;


   // output each array element's value
   for ( int j = 0; j < 10; j++ ) {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }


   return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result –

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

### Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer –

| Sr.No | Concept & Description |
|---|---|
| 1 | **Multi-dimensional arrays**<br><br>C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | **Pointer to an array**<br><br>You can generate a pointer to the first element of an array by simply |

| | |
|---|---|
| | specifying the array name, without any index. |
| 3 | **Passing arrays to functions**<br><br>You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 4 | **Return array from functions**<br><br>C++ allows a function to return an array. |

## C++ Strings

C++ provides following two types of string representations –

- The C-style character string.
- The string class type introduced with Standard C++.

### The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

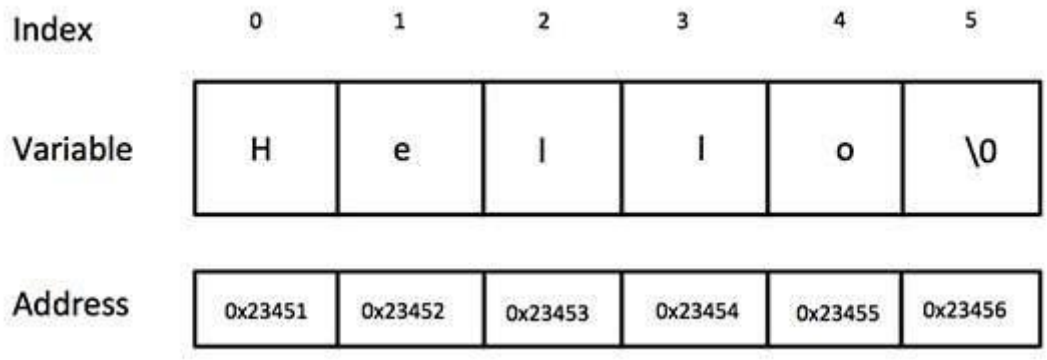
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++ –




Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string –

```
#include <iostream>

using namespace std;

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings –

| Sr.No | Function & Purpose |
|---|---|

| | |
|---|---|
| 1 | **strcpy(s1, s2);**<br>Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**<br>Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**<br>Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**<br>Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**<br>Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**<br>Returns a pointer to the first occurrence of string s2 in string s1. |

Following example makes use of few of the above-mentioned functions –

```
#include <iostream>
#include <cstring>

using namespace std;

int main () {

   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;
```

```
   // copy str1 into str3

   strcpy( str3, str1);

   cout << "strcpy( str3, str1) : " << str3 << endl;


   // concatenates str1 and str2

   strcat( str1, str2);

   cout << "strcat( str1, str2): " << str1 << endl;


   // total lenghth of str1 after concatenation

   len = strlen(str1);

   cout << "strlen(str1) : " << len << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces result something as follows –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

### The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example –

```
#include <iostream>

#include <string>


using namespace std;


int main () {


   string str1 = "Hello";
```

```
   string str2 = "World";

   string str3;

   int  len ;


   // copy str1 into str3

   str3 = str1;

   cout << "str3 : " << str3 << endl;


   // concatenates str1 and str2

   str3 = str1 + str2;

   cout << "str1 + str2 : " << str3 << endl;


   // total length of str3 after concatenation

   len = str3.size();

   cout << "str3.size() :  " << len << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces result something as follows –

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

## C++ Data Structures

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title

- Author
- Subject
- Book ID

### Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this –

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

### Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure –

```
#include <iostream>

#include <cstring>

using namespace std;

struct Books {

   char  title[50];

   char  author[50];

   char  subject[100];

   int   book_id;
```

```
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;
   return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result –

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

### Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example –

```
#include <iostream>

#include <cstring>


using namespace std;

void printBook( struct Books book );


struct Books {

   char  title[50];

   char  author[50];

   char  subject[100];

   int   book_id;

};


int main() {

   struct Books Book1;        // Declare Book1 of type Book

   struct Books Book2;        // Declare Book2 of type Book


   // book 1 specification
```

```
   strcpy( Book1.title, "Learn C++ Programming");

   strcpy( Book1.author, "Chand Miyan");

   strcpy( Book1.subject, "C++ Programming");

   Book1.book_id = 6495407;


   // book 2 specification

   strcpy( Book2.title, "Telecom Billing");

   strcpy( Book2.author, "Yakit Singha");

   strcpy( Book2.subject, "Telecom");

   Book2.book_id = 6495700;


   // Print Book1 info

   printBook( Book1 );


   // Print Book2 info

   printBook( Book2 );


   return 0;

}

void print Book( struct Books book ) {

   cout << "Book title : " << book.title <<endl;

   cout << "Book author : " << book.author <<endl;

   cout << "Book subject : " << book.subject <<endl;

   cout << "Book id : " << book.book_id <<endl;

}
```

When the above code is compiled and executed, it produces the following result –

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
```

```
Book subject : Telecom
Book id : 6495700
```

### Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows –

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows –

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows –

```
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept –

```
#include <iostream>

#include <cstring>


using namespace std;

void printBook( struct Books *book );


struct Books {

   char  title[50];

   char  author[50];

   char  subject[100];

   int   book_id;

};

int main() {

   struct Books Book1;        // Declare Book1 of type Book

   struct Books Book2;        // Declare Book2 of type Book


   // Book 1 specification
```

```
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

When the above code is compiled and executed, it produces the following result –

```
Book title : Learn C++ Programming
```

```
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

### The typedef Keyword

There is an easier way to define structs or you could "alias" types you create. For example –

```
typedef struct {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Books;
```

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example –

```
Books Book1, Book2;
```

You can use **typedef** keyword for non-structs as well as follows –

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y and z are all pointers to long ints.