Arrays

Introduction

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.
- Some texts refer to one-dimensional arrays as ***vectors***, two-dimensional arrays as ***matrices***, and use the general term ***arrays*** when the number of dimensions is unspecified or unimportant.

Declaring Arrays

- Array variables are declared identically to variables of their data type, except that the variable name is followed by one pair of square [ ] brackets for each dimension of the array.
- Uninitialized arrays must have the dimensions of their rows, columns, etc. listed within the square brackets.
- Dimensions used when declaring arrays in C must be positive integral constants or constant expressions.

Examples:

```
int i, j, intArray[ 10 ], number;
float floatArray[ 1000 ];
int tableArray[ 3 ][ 5 ];      /* 3 rows by 5 columns */
```

Initializing Arrays

- Arrays may be initialized when they are declared, just as any other variables.
- Place the initialization data in curly {} braces following the equals sign. Note the use of commas in the examples below.
- An array may be partially initialized, by providing fewer data items than the size of the array. The remaining array elements will be automatically initialized to zero.
- If an array is to be completely initialized, the dimension of the array is not required. The compiler will automatically size the array to fit the initialized data.

Examples:

```
int i =  5, intArray[ 6 ] = { 1, 2, 3, 4, 5, 6 }, k;
float sum  = 0.0f, floatArray[ 100 ] = { 1.0f, 5.0f, 20.0f };
double  piFractions[ ] = { 3.141592654, 1.570796327,0.785398163 };
```

Using Arrays

- Elements of an array are accessed by specifying the index (offset) of the desired element within square [ ] brackets after the array name.
- Array subscripts must be of integer type. (int, long int, char, etc.)
- **VERY IMPORTANT:** Array indices start at zero in C, and go to one less than the size of the array. For example, a five element array will have indices zero through four. This is because the index in C is actually an offset from the beginning of the array. (The first element is at the beginning of the array, and hence has zero offset.)
- **Landmine:** The most common mistake when working with arrays in C is forgetting that indices start at zero and stop one less than the array size.
- Arrays are commonly used in conjunction with loops, in order to perform the same calculations on all (or some part) of the data items in the array.

Multidimensional Arrays

- Multi-dimensional arrays are declared by providing more than one set of square [ ] brackets after the variable name in the declaration statement.
- One dimensional arrays do not require the dimension to be given if the array is to be completely initialized. By analogy, multi-dimensional arrays do not require **the first** dimension to be given if the array is to be completely initialized. All dimensions after the first must be given in any case.
- For two dimensional arrays, the first dimension is commonly considered to be the number of rows, and the second dimension the number of columns. We will use this convention when discussing two dimensional arrays.
- Two dimensional arrays are considered by C/C++ to be an array of (single dimensional arrays). For example, "int numbers[5][6]" would refer to a single dimensional array of 5 elements, wherein each element is a single dimensional array of 6 integers. By extension, "int numbers [12] [5] [6]" would refer to an array of twelve elements, each of which is a two dimensional array, and so on.
- Another way of looking at this is that C stores two dimensional arrays by rows, with all elements of a row being stored together as a single unit. Knowing this can sometimes lead to more efficient programs.
- Multidimensional arrays may be completely initialized by listing all data elements within a single pair of curly {} braces, as with single dimensional arrays.
- It is better programming practice to enclose each row within a separate subset of curly {} braces, to make the program more readable. This is required if any row other than the last is to be partially initialized. When subsets of braces are used, the last item within braces is not followed by a comma, but the subsets are themselves separated by commas.
- Multidimensional arrays may be partially initialized by not providing complete initialization data. Individual rows of a multidimensional array may be partially initialized, provided that subset braces are used.
- Individual data items in a multidimensional array are accessed by fully qualifying an array element. Alternatively, a smaller dimensional array may be accessed by partially qualifying the array name. For example, if "data" has been declared as a three dimensional array of floats, then data[1][2][5] would refer to a float, data[1][2] would refer to a one-

dimensional array of floats, and data[1] would refer to a two-dimensional array of floats. The reasons for this and the incentive to do this relate to memory-management issues that are beyond the scope of these notes.

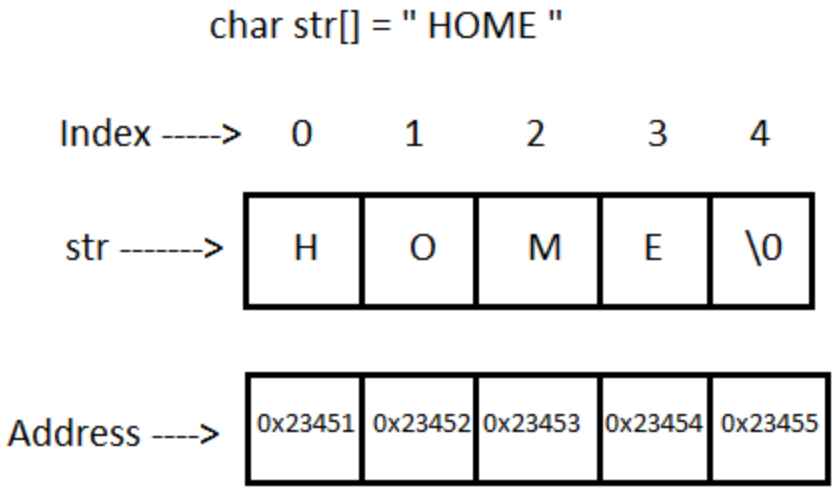
Sample Program Using 2-D Arrays

```
/* Sample program Using 2-D Arrays */

#include <stdlib.h>
#include <stdio.h>

int main( void ) {

    /* Program to add two multidimensional arrays */

    int a[ 2 ][ 3 ] = { { 5, 6, 7 }, { 10, 20, 30 } };
    int b[ 2 ][ 3 ] = { { 1, 2, 3 }, {  3,  2,  1 } };
    int sum[ 2 ][ 3 ], row, column;

    /* First the addition */

    for( row = 0; row < 2; row++ )
        for( column = 0; column < 3; column++ )
            sum[ row ][ column ] =
                a[ row ][ column ] + b[ row ][ column ];

    /* Then print the results */

    printf( "The sum is: \n\n" );

    for( row = 0; row < 2; row++ ) {
        for( column = 0; column < 3; column++ )
            printf( "\t%d",  sum[ row ][ column ] );
        printf( '\n' );   /* at end of each row */
    }

    return 0;
}
```

Character Strings as Arrays of Characters

String is a sequence of characters that are treated as a single data item and terminated by a null character '\0'. Remember that the C language does not support strings as a data type. A **string** is actually a one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "home" contains 5 characters including the '\0' character which is automatically added by the compiler at the end of the string.



// valid

char name[13] = "StudyTonight";

char name[10] = {'c','o','d','e','\0'};

// Illegal

char ch[3] = "hello";

char str[4];

str = "hello";

- The "string" class type is a new addition to C++, which did not exist in traditional C.
- The traditional method for handling character strings is to use an array of characters.
- A null byte (character constant zero, '\0') is used as a terminator signal to mark the end of the string.
- Ordinary quoted strings ("Please enter a number > ") are stored as null-terminated arrays of characters.
- The cstring library contains a number of functions for dealing with traditional arrays of characters.

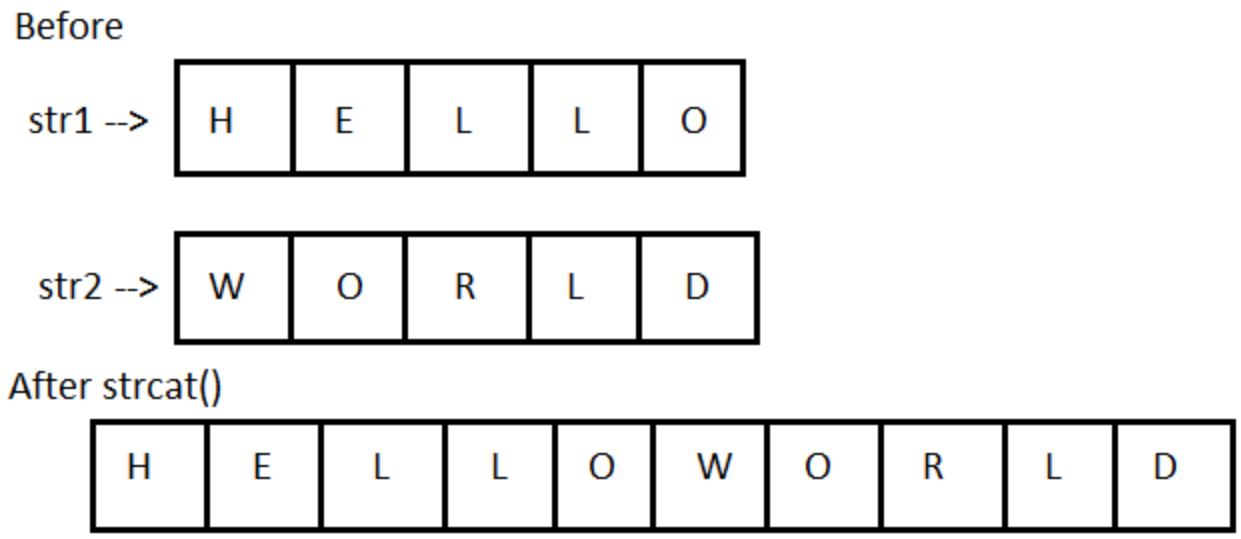
String Handling Functions:

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in the **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	Description
`strcat()`	It is used to concatenate(combine) two strings
`strlen()`	It is used to show the length of a string
`strrev()`	It is used to show the reverse of a string
`strcpy()`	Copies one string into another
`strcmp()`	It is used to compare two string

strcat() function in C:



Syntax:

```
strcat("hello", "world");
```

`strcat()` will add the string **"world"** to **"hello"** i.e ouput = helloworld.

`strlen()` and `strcmp()` function:

`strlen()` will return the length of the string passed to it and `strcmp()` will return the ASCII difference between first unmatching character of two strings.

STRUCTURE

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows 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. The format of the struct statement is as follows −

```
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 the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program −

```
#include <stdio.h>
#include <string.h>

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, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```