Chp 12.

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task. A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function. The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions. A function can also be referred as a method or a sub-routine or a procedure, etc. **Defining a Function** The general form of a function definition in C programming language is as follows: `return_type function_name( parameter list ) { body of the function }` A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function: • **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**. • **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature. • **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may

contain no parameters.

• **Function Body:** The function body contains a collection of statements that define what the function does. **Example** Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two: `/* function returning the max between two numbers */ int max(int num1, int num2) { /* local variable declaration */ int result; if (num1 > num2) result = num1; else result = num2; return result; }` **Function Declarations** A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts: `return_type function_name( parameter list );` For the above defined function max(),the function declaration is as follows: `int max(int num1, int num2);` Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration: `int max(int, int);` Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program. To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example: `#include <stdio.h> /* function declaration */ int max(int num1, int num2); int main () { /* local variable definition */ int a = 100; int b = 200; int ret; /* calling a function to get max value */ ret = max(a, b); printf("Max value is : %d\n", ret); return 0; } /* function returning the max between two numbers */ int max(int num1, int num2) { /* local variable declaration */ int result;`

`if (num1 > num2) result = num1; else result = num2; return result; }` We have kept max()along with main() and compiled the source code. While running the final executable, it would produce the following result: `Max value is : 200` **Function Arguments** If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways in which arguments can be passed to a function: **Call Type Description** Call by value This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. Call by reference This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by Value The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows. `/* function definition to swap the values */ void swap(int x, int y) { int temp; temp = x; /* save the value of x */ x = y; /* put y into x */ y = temp; /* put temp into y */ return; }` Now, let us call the function **swap()** by passing actual values as in the following example: `#include <stdio.h> /* function declaration */ void swap(int x, int y); int main () { /* local variable definition */ int a =`

100; int b = 200; printf("Before swap, value of a : %d\n", a); printf("Before swap, value of b : %d\n", b);

/* calling a function to swap the values */ swap(a, b); printf("After swap, value of a : %d\n", a); printf("After swap, value of b : %d\n", b); return 0; } Let us put the above code in a single C file, compile and execute it, it will produce the following result: Before swap, value of a :100 Before swap, value of b :200 After swap, value of a :100 After swap, value of b :200 It shows that there are no changes in the values, though they had been changed inside the function. **Call by Reference** The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument. To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly, you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments. /* function definition to swap the values */ void swap(int *x, int *y) { int temp; temp = *x; /* save the value at address x */ *x = *y; /* put y into x */

y = temp; / put temp into y */ return; } Let us now call the function **swap()** by passing values by reference as in the following example: #include <stdio.h> /* function declaration */ void swap(int *x, int *y); int main () { /* local variable definition */ int a = 100; int b = 200; printf("Before swap, value of a : %d\n", a); printf("Before swap, value of b : %d\n", b); /* calling a function to swap the values. * &a indicates pointer to a i.e. address of variable a and * &b indicates pointer to b i.e. address of variable b. */ swap(&a, &b); printf("After swap, value of a : %d\n", a); printf("After swap, value of b : %d\n", b); return 0; } Let us put the above code in a single C file, compile and execute it, to produce the following result:

Before swap, value of a :100 Before swap, value of b :200 After swap, value of a :200 After swap, value of b :100 It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function. By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Chp 13.

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language: • Inside a function or a block which is called **local** variables, • Outside of all functions which is called **global** variables. • In the definition of function parameters which are called **formal** parameters. Let us understand what are **local** and **global** variables, and **formal** parameters. **Local Variables** Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function. `#include <stdio.h> int main () { /* local variable declaration */ int a, b; int c; /* actual initialization */ a = 10; b = 20; c = a + b; printf ("value of a = %d, b = %d`

`and c = %d\n", a, b, c); return 0;`

`}` **Global Variables** Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program. A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program shows how global variables are used in a program. `#include <stdio.h> /* global variable declaration */ int g; int main () { /* local variable declaration */ int a, b; /* actual initialization */ a = 10; b = 20; g = a + b; printf ("value of a = %d, b = %d and g = %d\n", a, b, g); return 0; }` A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example: `#include <stdio.h> /* global variable declaration */`

`int g = 20; int main () { /* local variable declaration */ int g = 10; printf ("value of g = %d\n", g); return 0; }` When the above code is compiled and executed, it produces the following result: `value of g = 10` **Formal Parameters** Formal parameters are treated as local variables with-in a function and they take precedence over global variables. Following is an example: `#include <stdio.h> /* global variable declaration */ int a = 20; int main () { /* local variable declaration in main function */ int a = 10; int b = 20; int c = 0; printf ("value of a in main() = %d\n", a); c = sum( a, b); printf ("value of c in main() = %d\n", c);`

`return 0; } /* function to add two integers */ int sum(int a, int b) { printf ("value of a in sum() = %d\n", a); printf ("value of b in sum() = %d\n", b); return a + b; }` When the above code is compiled and executed, it produces the following result: `value of a in main() = 10 value of a in sum() = 10 value of b in sum() = 20 value of c in main() = 30` **Initializing Local and Global Variables** When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them, as follows: **Data Type Initial Default Value** int 0 char '\0' float 0 double 0

pointer NULL It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Chp14.

Arrays a kind of data structure that can store 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, a 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-dimensional* 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];` Here, *balance* is a variable array which is sufficient to hold up to 10 double numbers. **Initializing** You can initialize an array in C either one by one or using a single statement as follows: `double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};`

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. 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, 7.0, 50.0};` You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array: `balance[4] = 50.0;` The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the 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 the 10th element from the array and assign the value to salary variable. The following example shows how to use all the three above-mentioned concepts viz. declaration, assignment, and accessing arrays: `#include <stdio.h> int main () { int n[ 10 ]; /* n is an array of 10 integers */ int i,j; /* initialize elements of array n to 0 */ for ( i = 0; i < 10; i++ ) {`

`n[ i ] = i + 100; /* set element at location i to i + 100 */ } /* output each array element's value */ for (j = 0; j < 10; j++ ) { printf("Element[%d] = %d\n", j, n[j] ); } return 0; }` When the above code is compiled and executed, it produces the following result: `Element[0] = 100 Element[1] = 101 Element[2] = 102 Element[3] = 103 Element[4] = 104 Element[5] = 105 Element[6] = 106 Element[7] = 107 Element[8] = 108 Element[9] = 109` **Arrays in Detail** Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer: **Concept Description** Multidimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. Passing arrays to functions You can pass to the function a pointer to an

array by specifying the array's name without an index. Return array from a function C allows a function to return an array. Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index. **Multidimensional Arrays** C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration: `type name[size1][size2]...[sizeN];` For example, the following declaration creates a three-dimensional integer array: `int threedim[5][10][4];` **Two-dimensional** The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows: `type arrayName [ x ][ y ];` Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows: Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns. `int a[3][4] = { {0, 1, 2, 3} , /* initializers for row indexed by 0 */ {4, 5, 6, 7} , /* initializers for row indexed by 1 */ {8, 9, 10, 11} /* initializers for row indexed by 2 */ };` The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example: `int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};` **Accessing Array Elements** An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example: `int val = a[2][3];` The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array: `#include <stdio.h> int main () { /* an array with 5 rows and 2 columns*/ int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}}; int i, j; /* output each array element's value */ for ( i = 0; i < 5; i++ ) { for ( j = 0; j < 2; j++ ) { printf("a[%d][%d] = %d\n", i,j, a[i][j] );`

`} } return 0; }` When the above code is compiled and executed, it produces the following result: `a[0][0]: 0 a[0][1]: 0 a[1][0]: 1 a[1][1]: 2 a[2][0]: 2 a[2][1]: 4 a[3][0]: 3 a[3][1]: 6 a[4][0]: 4 a[4][1]: 8` As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions. **Passing Arrays to Functions** If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters. **Way-1** Formal parameters as a pointer: `void myFunction(int *param) { . . . }`

Way-2 Formal parameters as a sized array: `void myFunction(int param[10]) { . . . }` **Way-3** Formal parameters as an unsized array: `void myFunction(int param[]) { . . . }` **Example** Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows: `double getAverage(int arr[], int size) { int i; double avg; double sum; for (i = 0; i < size; ++i) { sum += arr[i]; }`

`avg = sum / size; return avg; }` Now, let us call the above function as follows: `#include <stdio.h> /* function declaration */ double getAverage(int arr[], int size); int main () { /* an int array with 5 elements */ int balance[5] = {1000, 2, 3, 17, 50}; double avg; /* pass pointer to the array as an argument */ avg = getAverage(balance, 5) ; /* output the returned value */ printf("Average value is: %f ", avg); return 0; }` When the above code is compiled together and executed, it produces the following result: `Average value is: 214.400000` As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters. **Return Array from a Function** C programming does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example: `int * myFunction() { . . . }` Second point to remember is that C does not advocate to return the address of a local variable to outside of the function, so you would have to define the local variable as **static** variable. Now, consider the following function which will generate 10 random numbers and return them using an array and call this function as follows: `#include <stdio.h> /* function to generate and return random numbers */ int * getRandom( ) { static int r[10]; int i; /* set the seed */ srand( (unsigned)time( NULL ) ); for ( i = 0; i < 10; ++i) { r[i] =`

rand(); printf("r[%d] = %d\n", i, r[i]); } return r; } /* main function to call above defined function */

int main () { /* a pointer to an int */ int *p; int i; p = getRandom(); for (i = 0; i < 10; i++) { printf("*(p + %d) : %d\n", i, *(p + i)); } return 0; } When the above code is compiled together and executed, it produces the following result: r[0] = 313959809 r[1] = 1759055877 r[2] = 1113101911 r[3] = 2133832223 r[4] = 2073354073 r[5] = 167288147 r[6] = 1827471542 r[7] = 834791014 r[8] = 1901409888 r[9] = 1990469526 *(p + 0) : 313959809 *(p + 1) : 1759055877 *(p + 2) : 1113101911 *(p + 3) : 2133832223 *(p + 4) : 2073354073 *(p + 5) : 167288147 *(p + 6) : 1827471542

*(p + 7) : 834791014 *(p + 8) : 1901409888 *(p + 9) : 1990469526 **Pointer to an Array** It is most likely that you would not understand this section until you are through with the chapter 'Pointers'. Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration: double balance[50]; **balance** is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance**: double *p; double balance[10]; p = balance; It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4]. Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2), and so on. Given below is the example to show all the concepts discussed above: #include <stdio.h> int main () { /* an array with 5 elements */ double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0}; double *p; int i; p = balance;

/* output each array element's value */ printf("Array values using pointer\n"); for (i = 0; i < 5; i++) { printf("*(p + %d) : %f\n", i, *(p + i)); } printf("Array values using balance as address\n"); for (i = 0; i < 5; i++) { printf("*(balance + %d) : %f\n", i, *(balance + i)); } return 0; } When the above code is compiled and executed, it produces the following result: Array values using pointer *(p + 0) : 1000.000000 *(p + 1) : 2.000000 *(p + 2) : 3.400000 *(p + 3) : 17.000000 *(p + 4) : 50.000000 Array values using balance as address *(balance + 0) : 1000.000000 *(balance + 1) : 2.000000 *(balance + 2) : 3.400000 *(balance + 3) : 17.000000 *(balance + 4) : 50.000000 In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, ***p** will give us the value available at the address stored in p, as we have shown in the above example.

Chp 15.

Arrays a kind of data structure that can store 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, a 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-dimensional* 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];` Here, *balance* is a variable array which is sufficient to hold up to 10 double numbers. **Initializing** You can initialize an array in C either one by one or using a single statement as follows: `double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};`

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. 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, 7.0, 50.0};` You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array: `balance[4] = 50.0;` The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the 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 the 10th element from the array and assign the value to salary variable. The following example shows how to use all the three above-mentioned concepts viz. declaration, assignment, and accessing arrays: `#include <stdio.h> int main () { int n[ 10 ]; /* n is an array of 10 integers */ int i,j; /* initialize elements of array n to 0 */ for ( i = 0; i < 10; i++ ) {`

`n[ i ] = i + 100; /* set element at location i to i + 100 */ } /* output each array element's value */ for (j = 0; j < 10; j++ ) { printf("Element[%d] = %d\n", j, n[j] ); } return 0; }` When the above code is compiled and executed, it produces the following result: `Element[0] = 100 Element[1] = 101 Element[2] = 102 Element[3] = 103 Element[4] = 104 Element[5] = 105 Element[6] = 106 Element[7] = 107 Element[8] = 108 Element[9] = 109` **Arrays in Detail** Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer: **Concept Description** Multidimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. Passing arrays to functions You can pass to the function a pointer to an

array by specifying the array's name without an index. Return array from a function C allows a function to return an array. Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index. **Multidimensional Arrays** C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration: `type name[size1][size2]...[sizeN];` For example, the following declaration creates a three-dimensional integer array: `int threedim[5][10][4];` **Two-dimensional** The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows: `type arrayName [ x ][ y ];` Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows: Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns. `int a[3][4] = { {0, 1, 2, 3} , /* initializers for row indexed by 0 */ {4, 5, 6, 7} , /* initializers for row indexed by 1 */ {8, 9, 10, 11} /* initializers for row indexed by 2 */ };` The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example: `int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};` **Accessing Array Elements** An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example: `int val = a[2][3];` The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array: `#include <stdio.h> int main () { /* an array with 5 rows and 2 columns*/ int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}}; int i, j; /* output each array element's value */ for ( i = 0; i < 5; i++ ) { for ( j = 0; j < 2; j++ ) { printf("a[%d][%d] = %d\n", i,j, a[i][j] );`

`} } return 0; }` When the above code is compiled and executed, it produces the following result: `a[0][0]: 0 a[0][1]: 0 a[1][0]: 1 a[1][1]: 2 a[2][0]: 2 a[2][1]: 4 a[3][0]: 3 a[3][1]: 6 a[4][0]: 4 a[4][1]: 8` As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions. **Passing Arrays to Functions** If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters. **Way-1** Formal parameters as a pointer: `void myFunction(int *param) { . . . }`

Way-2 Formal parameters as a sized array: `void myFunction(int param[10]) { . . . }` **Way-3** Formal parameters as an unsized array: `void myFunction(int param[]) { . . . }` **Example** Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows: `double getAverage(int arr[], int size) { int i; double avg; double sum; for (i = 0; i < size; ++i) { sum += arr[i]; }`

`avg = sum / size; return avg; }` Now, let us call the above function as follows: `#include <stdio.h> /* function declaration */ double getAverage(int arr[], int size); int main () { /* an int array with 5 elements */ int balance[5] = {1000, 2, 3, 17, 50}; double avg; /* pass pointer to the array as an argument */ avg = getAverage(balance, 5) ; /* output the returned value */ printf("Average value is: %f ", avg); return 0; }` When the above code is compiled together and executed, it produces the following result: `Average value is: 214.400000` As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters. **Return Array from a Function** C programming does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example: `int * myFunction() { . . . }` Second point to remember is that C does not advocate to return the address of a local variable to outside of the function, so you would have to define the local variable as **static** variable. Now, consider the following function which will generate 10 random numbers and return them using an array and call this function as follows: `#include <stdio.h> /* function to generate and return random numbers */ int * getRandom( ) { static int r[10]; int i; /* set the seed */ srand( (unsigned)time( NULL ) ); for ( i = 0; i < 10; ++i) { r[i] =`

`rand(); printf( "r[%d] = %d\n", i, r[i]); } return r; } /* main function to call above defined function */`

`int main () { /* a pointer to an int */ int *p; int i; p = getRandom(); for ( i = 0; i < 10; i++ ) { printf( "*(p + %d) : %d\n", i, *(p + i)); } return 0; }` When the above code is compiled together and executed, it produces the following result: `r[0] = 313959809 r[1] = 1759055877 r[2] = 1113101911 r[3] = 2133832223 r[4] = 2073354073 r[5] = 167288147 r[6] = 1827471542 r[7] = 834791014 r[8] = 1901409888 r[9] = 1990469526 *(p + 0) : 313959809 *(p + 1) : 1759055877 *(p + 2) : 1113101911 *(p + 3) : 2133832223 *(p + 4) : 2073354073 *(p + 5) : 167288147 *(p + 6) : 1827471542`

`*(p + 7) : 834791014 *(p + 8) : 1901409888 *(p + 9) : 1990469526` **Pointer to an Array** It is most likely that you would not understand this section until you are through with the chapter 'Pointers'. Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration: `double balance[50];` **balance** is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance**: `double *p; double balance[10]; p = balance;` It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4]. Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2), and so on. Given below is the example to show all the concepts discussed above: `#include <stdio.h> int main () { /* an array with 5 elements */ double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0}; double *p; int i; p = balance;`

`/* output each array element's value */ printf( "Array values using pointer\n"); for ( i = 0; i < 5; i++ ) { printf("*(p + %d) : %f\n", i, *(p + i) ); } printf( "Array values using balance as address\n"); for ( i = 0; i < 5; i++ ) { printf("*(balance + %d) : %f\n", i, *(balance + i) ); } return 0; }` When the above code is compiled and executed, it produces the following result: `Array values using pointer *(p + 0) : 1000.000000 *(p + 1) : 2.000000 *(p + 2) : 3.400000 *(p + 3) : 17.000000 *(p + 4) : 50.000000 Array values using balance as address *(balance + 0) : 1000.000000 *(balance + 1) : 2.000000 *(balance + 2) : 3.400000 *(balance + 3) : 17.000000 *(balance + 4) : 50.000000` In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, ***p** will give us the value available at the address stored in p, as we have shown in the above example.