|  |  |  |  |
| --- | --- | --- | --- |
| Date: | **20-06-2020** | Name: | **Varun G Shetty** |
| Course: | **C programming** | USN: | **4AL17EC093** |
| Topic: | * Arrays * Pointers,strings | Semester & Section: | **6th & ‘B’** |
| GitHub Repository: | **Varunshetty4** |  |  |

**Report:**

**Two dimensional array:** A two-dimensional array is an array of arrays and can be thought of as a table. A two-dimensional array declaration indicates the number of number rows and the number of columns. For example;

int a[2][3]; /\* A 2 x 3 array \*/

Nested curly braces are used to initialize elements row by row, as in the following statement:

|  |  |
| --- | --- |
| int a[2][3] = {  {3, 2, 6},  {4, 5, 20}  }; | The same statement can also take the form:  int a[2][3] = { {3, 2, 6}, {4, 5, 20} }; |

To access an element of a two-dimensional array, both the row index and column index are required. Just as a **for**loop is used to iterate through a one-dimensional array, nested **for**loops are used to traverse a two-dimensional array.

**Using Memory:** C is designed to be a low-level language that can easily access memory locations and perform memory-related operations. For instance, the scanf() function places the value entered by the user at the location, or address, of the variable. This is accomplished by using the & symbol. **&num** is the address of variable **num**.

A memory address is given as a **hexadecimal** number. The following program displays the memory addresses for variables **i** and **k**:

|  |  |
| --- | --- |
| void test(int k);  int main() {  int i = 0;  printf("The address of i is %x\n", &i);  test(i);  printf("The address of i is %x\n", &i);  test(i);  return 0;  }  void test(int k) {  printf("The address of k is %x\n", &k);  } | >>  The address of i is 846dd754  The address of k is 846dd758  The address of i is 846dd754  The address of k is 846dd758  The address of a variable stays the same from the time it is declared until the end of its scope. |

**Pointers:** A pointer is a variable that contains the address of another variable. In other words, it "points" to the location assigned to a variable and can indirectly access the variable. Pointers are declared using the \* symbol and take the form:

**pointer\_type \*identifier**

The actual pointer data type is a hexadecimal number, but when declaring a pointer, you must indicate what type of data it will be pointing to. Asterisk \* declares a pointer and should appear next to the identifier used for the pointer variable.

|  |  |
| --- | --- |
| int j = 63;  int \*p = NULL;  p = &j;  printf("The address of j is %x\n", &j);  printf("p contains address %x\n", p);  printf("The value of j is %d\n", j);  printf("p is pointing to the value %d\n", \*p); | >>  The address of j is ff3652cc  p contains address ff3652cc  The value of j is 63  p is pointing to the value 63 |

• Pointers should be initialized to **NULL** until they are assigned a valid location.  
• Pointers can be assigned the address of a variable using the**ampersand &**sign.  
• To see what a pointer is pointing to, use the \* again, as in **\*p**. In this case the \* is called the indirection or **dereference** operator. The process is called **dereferencing**.

Some algorithms use a pointer to a pointer. This type of variable declaration uses \*\*, and can be assigned the address of another pointer, as in:

|  |
| --- |
| int x = 12;  int \*p = NULL  int \*\*ptr = NULL;  p = &x;  ptr = &p; |

Pointers can be used in **expressions** just as any variable. Arithmetic operators can be applied to whatever the pointer is pointing to. For example: y = \*p + 2;

Note that parentheses are required for the ++ operator to increment the value being pointed to. The same is true when using the -- operator. For ex: (\*p)++;

**Pointers and Array:** Pointers are especially useful with arrays. An array declaration reserves a block of contiguous memory addresses for its elements. With pointers, we can point to the first element and then use address arithmetic to traverse the array:

+ is used to move forward to a memory location

- is used to move backward to a memory location

|  |  |
| --- | --- |
| int a[5] = {22, 33, 44, 55, 66};  int \*ptr = NULL;  int i;  ptr = a;  for (i = 0; i < 5; i++) {  printf("%d ", \*(ptr + i));  } | >>  22 33 44 55 66 |

An important concept with arrays is that an **array name** acts as a **pointer** to the first element of the array. Therefore, the statement **ptr = a** can be thought of as **ptr = &a[0]**. Consider the following statement, which prints the first element of the array: **printf("%d ", \*a);**

Address arithmetic can also be thought of as pointer arithmetic because the operations involve pointers. Besides using + and – to refer to the next and previous memory locations, you can use the assignment operators to change the address the pointer contains. When a pointer is incremented, the memory address increases by the number of bytes being pointed to. In the program above, the pointer increases by 4 when the increment operator is used (ptr++) because the pointer is pointing to an int. You can also use the ==, <, and > operators to compare pointer addresses.

**Pointers and Functions:** Pointers greatly expand the possibilities for functions. No longer are we limited to returning one value. With pointer parameters, your functions can alter actual data rather than a copy of data. To change the actual values of variables, the calling statement passes addresses to pointer parameters in a function.

|  |  |
| --- | --- |
| void swap (int \*num1, int \*num2);  int main() {  int x = 25;  int y = 100;  printf("x is %d, y is %d\n", x, y);  swap(&x, &y);  printf("x is %d, y is %d\n", x, y);  return 0;  }    void swap (int \*num1, int \*num2) {  int temp;  temp = \*num1;  \*num1 = \*num2;  \*num2 = temp;  } | >>  x is 25, y is 100  x is 100, y is 25 |

**Function with Array parameters:** An array cannot be passed by value to a function. However, an array name is a pointer, so just passing an array name to a function is passing a pointer to the array.

|  |  |
| --- | --- |
| int add\_up (int \*a, int num\_elements);  int main() {  int orders[5] = {100, 220, 37, 16, 98};  printf("Total orders is %d\n", add\_up(orders, 5));  return 0;  }  int add\_up (int \*a, int num\_elements) {  int total = 0;  int k;  for (k = 0; k < num\_elements; k++) {  total += a[k];  }  return (total);  } | >>  Total orders is 471 |

Just as a pointer to an array can be passed into a function, a pointer to an array can be returned.

|  |
| --- |
| int \* get\_evens();  int main() {  int \*a;  int k;  a = get\_evens(); /\* get first 5 even numbers \*/  for (k = 0; k < 5; k++)  printf("%d\n", a[k]);  return 0;  }  int \* get\_evens() {  static int nums[5];  int k;  int even = 0;  for (k = 0; k < 5; k++) {  nums[k] = even += 2;  }  return (nums);  } |

Note that **a** pointer, not an array, is declared to store the value returned by the function. Also note that when a local variable is being passed out of a function, you need to declare it as **static** in the function. Keep in mind that **a[k]**is the same as **\*(a + k)**.

**Strings:** A string in C is an **array of characters** that ends with a **NULL** character '\0'.  
A string declaration can be made in several ways, each with its own considerations. For example: **char str\_name[str\_len] = "string";ex: char str1[6] = "hello";**

This creates a string named *str\_name* of *str\_len* characters and initializes it to the value "string". When you provide a string literal to initialize the string, the compiler automatically adds a NULL character '\0' to the char array. For this reason, you must declare the array size to be at least one character longer than the expected string length. A string can also be declared as a set of characters:

**char str3[6] = {'h', 'e', 'l', 'l', 'o', '\0'};**

As with any array, the name of a string acts as a pointer. A **string literal** is a text **enclosed in double quotation** marks. A character, **such as 'b'**, is indicated by **single quotation** marks and **cannot be treated as a string**. A string pointer declaration such as **char \*str = "stuff";** is considered a constant and cannot be changed from its initial value.

To safely and conveniently operate with strings, you can use the **Standard****Library** string functions shown below. Don't forget to include <string.h>.

**strlen**() - get length of a string

**strcat**() - merge two strings

**strcpy**() - copy one string to another

**strlwr**() - convert string to lower case

**strupr**() - conver string to upper case

**strrev**() - reverse string

**strcmp**() - compare two strings