Rajarata University of Sri Lanka
Department of Physical Sciences

COM1407
Computer Programming

LECTURE 11 –PART II

WORKING WITH POINTERS

PIYUMI HERATH

Pointers & Arrays

- The main reasons for using pointers to arrays are ones of notational convenience and of program efficiency.
- Pointers to arrays generally result in code that uses less memory and executes faster.

- Pointers to arrays is useful when you want to sequence through the elements of an array.

Pointers & Arrays (Cont…)

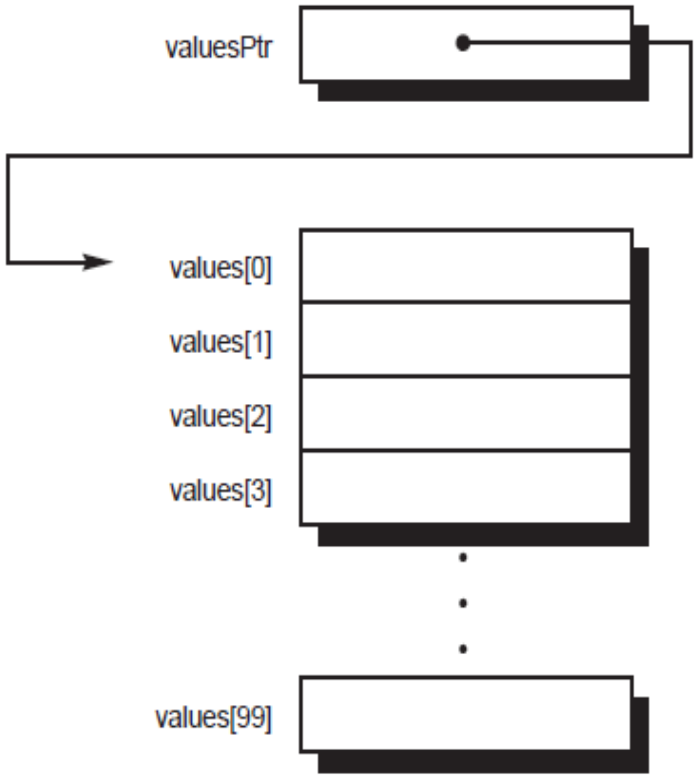
- An array name without brackets is a pointer to the array's first element.
- So, if a program declared an array data[], data (array's name) is the address of the first array element and is equivalent to the expression &data[0] that means references the address of the array's first element.
- The **array's name is, therefore a pointer to the array's first element.**

Pointers & Arrays (Cont…)

int values [100] = {1,2,…, 100};

int *ptr;

ptr = values;



Or

ptr = &values[0];

If you do following

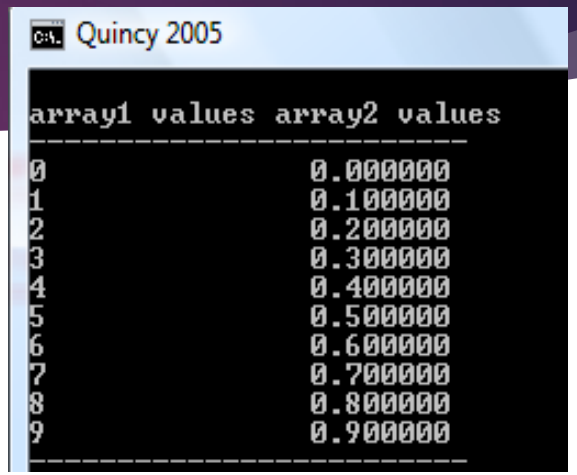
ptr = &values[1];

It points to second element

Once you store the address of the first element in ' ptr ', you can access the array elements using * ptr, *(ptr +1), *(ptr+2) and so on

Pointers & Arrays (Cont…)

```
#include <stdio.h>
#define MAX 10
int main(){
        int array1[MAX] = {0,1,2,3,4,5,6,7,8,9};
        int *ptr1, count;
        float array2[MAX] = {0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};
        float *ptr2;

        ptr1 = array1;
        ptr2 = array2;

        printf("\narray1 values array2 values");
        printf("\n-----------------------");
        for(count = 0; count < MAX; count++)
                printf("\n%i\t\t%f", *ptr1++, *ptr2++);
        printf("\n-----------------------\n");
        return 0;
}
```



The increment and decrement operators are particularly handy when dealing with pointers. Applying the increment operator to a pointer has the same effect as adding one to the pointer, while applying the decrement operator has the same effect as subtracting one from the pointer.

Pointers & Arrays (Cont…)

- Generally, the relationship is as follows:
  - *(array1) == array1 [0] //first element
  - *(array1 + 1) == array1 [1] //second element
  - *(array1+ 2) == array1 [2] //third element
  - …
  - …
  - *(array1 + n) == array1[n] //the nth element

Pointers & Arrays (Cont…)

- In general, the process of indexing an array takes more time to execute than does the process of accessing the contents of a pointer.
- In fact, this is one of the main reasons why pointers are used to access the elements of an array—the code that is generated is generally more efficient.
- Of course, if access to the array is not generally sequential, pointers accomplish nothing, as far as this issue is concerned, because the expression `*(pointer + j)` takes just as long to execute as does the expression `array[j]`.

Array of Pointers

- Like we declare arrays of int, float or char etc, we can also declare an array of pointers.

- Syntax: datatype *array_name[size];

```
int *arrop[5];
```

Here *arrop* is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables. In other words, you can assign 5 pointer variables of type pointer to int to the elements of this array.

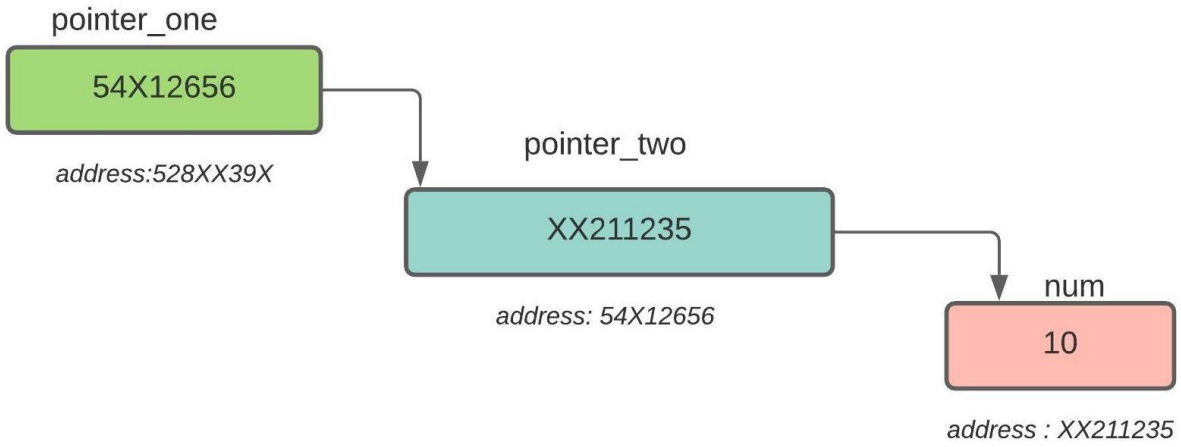
Eg:

```
#include<stdio.h>
int main()
{
    int *arrop[3];
    int a = 10, b = 20, c = 50, i;

    arrop[0] = &a;
    arrop[1] = &b;
    arrop[2] = &c;

    for(i = 0; i < 3; i++)
    {
        printf("Address = %d\t Value = %d\n", arrop[i], *arrop[i]);
    }
    return 0;
}
```

```
Address = 6356736        Value = 10
Address = 6356732        Value = 20
Address = 6356728        Value = 50
```

Pointers to Pointers

- When a pointer holds the address of another pointer then such type of pointer is known as **pointer-to-pointer**
- Graphically, the construct of a pointer to pointer can be depicted as shown below.
- *pointer_one* is the first pointer, pointing to the second pointer, *pointer_two* and finally *pointer_two* is pointing to a normal variable *num* that hold integer 10.



Pointers to Pointers (Cont...)

- In order to indirectly access the target value pointed to by a pointer to a pointer, the asterisk operator must be applied twice.
- For example, the following declaration:

```
int **SecondPtr;
```

  tells the compiler that SecondPtr is a pointer to a pointer of type integer.

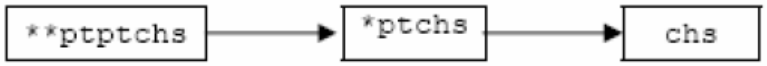
- Pointer to pointer is rarely used but you will find it regularly in programs that accept argument(s) from command line.

Pointers to Pointers (Cont...)

- Consider the following declarations:

```
char chs; /* a normal character variable */
char *ptchs; /* a pointer to a character */
char **ptptchs; /* a pointer to a pointer to a character */
```

- If the variables are related as shown below:



- We can do some assignment like this:

```
chs = 'A';
ptchs = &chs;
ptptchs = &ptchs;
```

Eg:

```
int main() {
    int num=500;
    int *ptr;
    int **ptr_ptr;

    ptr = &num;
    ptr_ptr = &ptr;

    printf("\n Value of num is: %d", num);
    printf("\n Value of num using ptr is: %d", *ptr);
    printf("\n Value of num using ptr_ptr is: %d\n", **ptr_ptr);

   printf("\n Address of num is: %p", &num);
    printf("\n Address of num using ptr is: %p", ptr);
    printf("\n Address of num using ptr_ptr is: %p \n", *ptr_ptr);

    printf("\n Value of Pointer ptr is: %p", ptr);
    printf("\n Address of Pointer ptr is:%p",&ptr);
    printf("\n Value of Pointer ptr_ptr is: %p\n", ptr_ptr);
    printf("\n Address of Pointer ptr_ptr is:%p \n",&ptr_ptr);

    return 0; }
```

```
Value of num is: 500
Value of num using ptr is: 500
Value of num using ptr_ptr is: 500

Address of num is: 0060FF10
Address of num using ptr is: 0060FF10
Address of num using ptr_ptr is: 0060FF10

Value of Pointer ptr is: 0060FF10
Address of Pointer ptr is:0060FF0C
Value of Pointer ptr_ptr is: 0060FF0C

Address of Pointer ptr_ptr is:0060FF08
```

Pointers & Functions

- You can pass a pointer as an argument to a function in the normal fashion, and you can also have a function return a pointer as its result.
- The pointers can be passed as formal parameters to a function in the same way as a normal pointer variable.
- The value of the pointer is copied into the formal parameter when the function is called.
- Although the pointer cannot be changed by the function, the data elements that the pointer references can be changed.

Pointers & Functions (Cont...)

```
#include <stdio.h>
void test (int *int_pointer)
{
        *int_pointer = 100;
}
int main (void)
{
        int i = 50, *p = &i;
        printf ("Before the call to test i = %i\n", i);
        test (p);
        printf ("After the call to test i = %i\n", i);
        return 0;
}
```

Pointers & Functions (Cont...)

```
#include <stdio.h>
void exchange (int * pint1, int *  pint2)
{
      int temp;
      temp = *pint1;
      *pint1 = *pint2;
      *pint2 = temp;
}
```

```
int main (void)
{
     int i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;
     printf ("i1 = %i, i2 = %i\n", i1, i2);
     exchange (p1, p2);
     printf ("i1 = %i, i2 = %i\n", i1, i2);
     exchange (&i1, &i2);
     printf ("i1 = %i, i2 = %i\n", i1, i2);

     return 0;
}
```

Pointers & Functions (Cont...)

- Similarly, C also allows to return a pointer from a function.

```
#include<stdio.h>
int *fun();
int main(){
   int *ptr;
   ptr=fun();
   printf("%i",*ptr);
   return 0;
}
int *fun() {
        int p, *point;
        int p = 12;
        point=&p;
        return point;
}
```

Pointers to Character Strings

- One of the most common applications of using a pointer to an array is as a pointer to a character string.
- The reasons are ones of notational convenience and efficiency.

  (In C programming, a string is a sequence of characters terminated with a null character \0.

  For example:

  char c[] = “This is a string";

  Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.)

Pointers to Character Strings (Cont…)

- So, if textPtr is declared to be a character pointer, as in

```
char *textPtr;
```

then the statement

```
textPtr = "A character string.";
```

assigns to textPtr a *pointer* to the constant character string "A character string."

Pointers to Character Strings (Cont…)

- Be careful to make the distinction here between character pointers and character arrays, as the type of assignment just shown is *not* valid with a character array.
- So, for example, if text is defined instead to be an array of chars, with a statement such as

```
char text[80];
```

  then you *could not* write a statement such as

```
text = "This is not valid.";
```

- The *only* time that C lets you get away with performing this type of assignment to a character array is when initializing it, as in

```
char text[80] = "This is okay.";
```

Pointers to Character Strings (Cont…)

- If ***str_pnt*** is a character pointer, initializing ***str_pnt*** with the statement

```
char *str_pnt = "This is okay.";
```

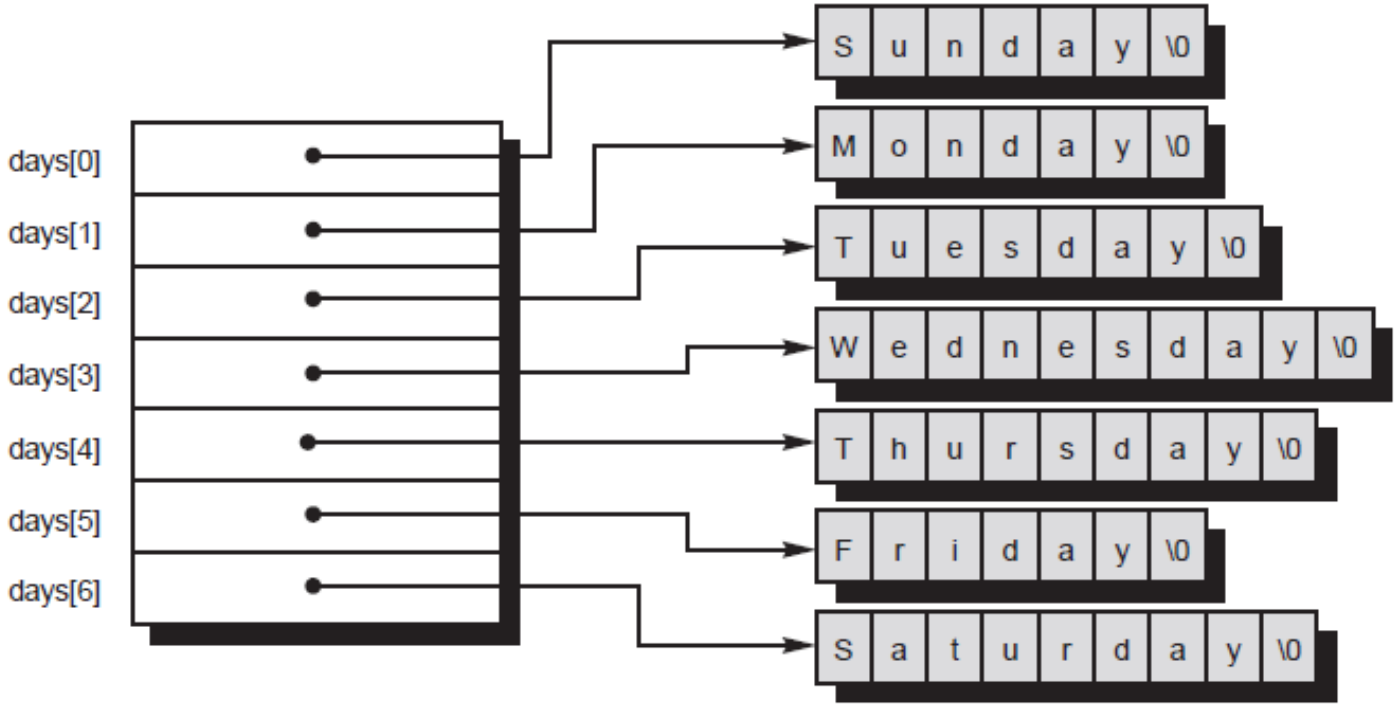
means that *str_pnt* is a pointer to the character string "This is okay.“

- As another example of the distinction between character strings and character string pointers, the following sets up an array called days, which contains *pointers* to the names of the days of the week.

```
char *days[] = { "Sunday", "Monday",
    "Tuesday", "Wednesday", "Thursday",
            "Friday","Saturday" };
```

- So days[0] contains a pointer to the character string "Sunday", days[1] contains a pointer to the string "Monday", and so on.

Pointers to Character Strings (Cont…)



Reference

- Chapter 11 - Programming in C, 3rd Edition, Stephen G. Kochan

Thank You