Rajarata University of Sri Lanka

Department of Physical Sciences

COM1407
Computer Programming

LECTURE 11 – PART I

WORKING WITH POINTERS

PIYUMI HERATH

Objectives

- At the end of this lecture students should be able to;
  - Define the C pointers and its usage in computer programming.
  - Describe pointer declaration and initialization.
  - Apply C pointers for expressions.
  - Experiment on pointer operations.
  - Identify NULL pointer concept.
  - Experiment on pointer to pointer, pointer arrays, arrays with pointers and functions with pointers.
  - Apply taught concepts for writing programs.

Introduction

- To understand how to use pointers, you must have a basic knowledge of how a computer stores information in memory.
- Pointers closely relate to **memory manipulation**.
- Basically, a personal computer RAM consists thousands of sequential storage locations, with each location being identified by a unique address.
- Computer's processor also has their own memory, normally called registers and cache.
- They differ in term of access speed, price and their usage.

Introduction (Cont…)

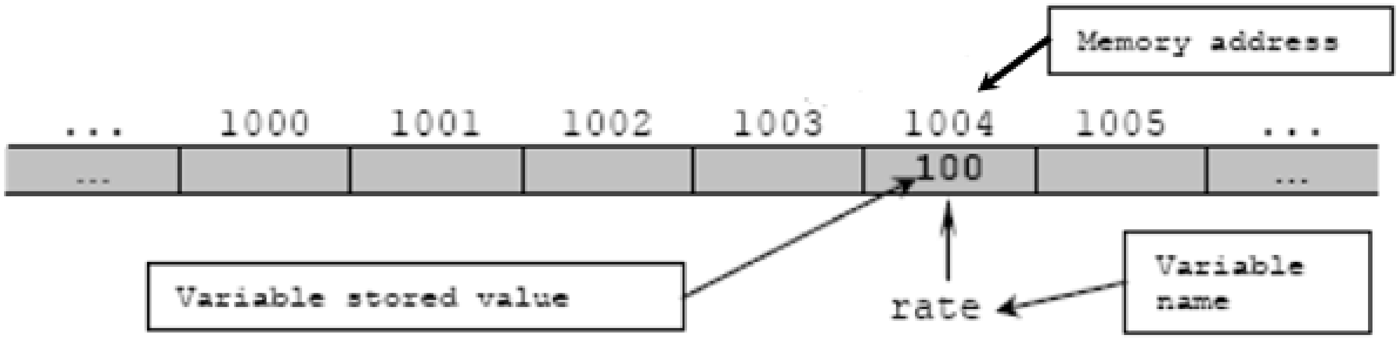
- When the programmer declares a variable in a C/C++ program, the compiler sets aside a memory location with a unique address to store that variable.
- The compiler associates that address with the variable's name.
- When the program uses the variable name, it automatically accesses the proper memory location.
- The locations address remains hidden from the programmer, and the programmer need not be concerned with it.
- What we are going to do is to manipulate the memory addresses by using pointers.

Introduction (Cont…)

- Lets say we declare one variable named *rate* of type integer and assign an initial value as follows:

```
int rate = 100;
```

- Memory allocation for the above variable can be depicted as follows

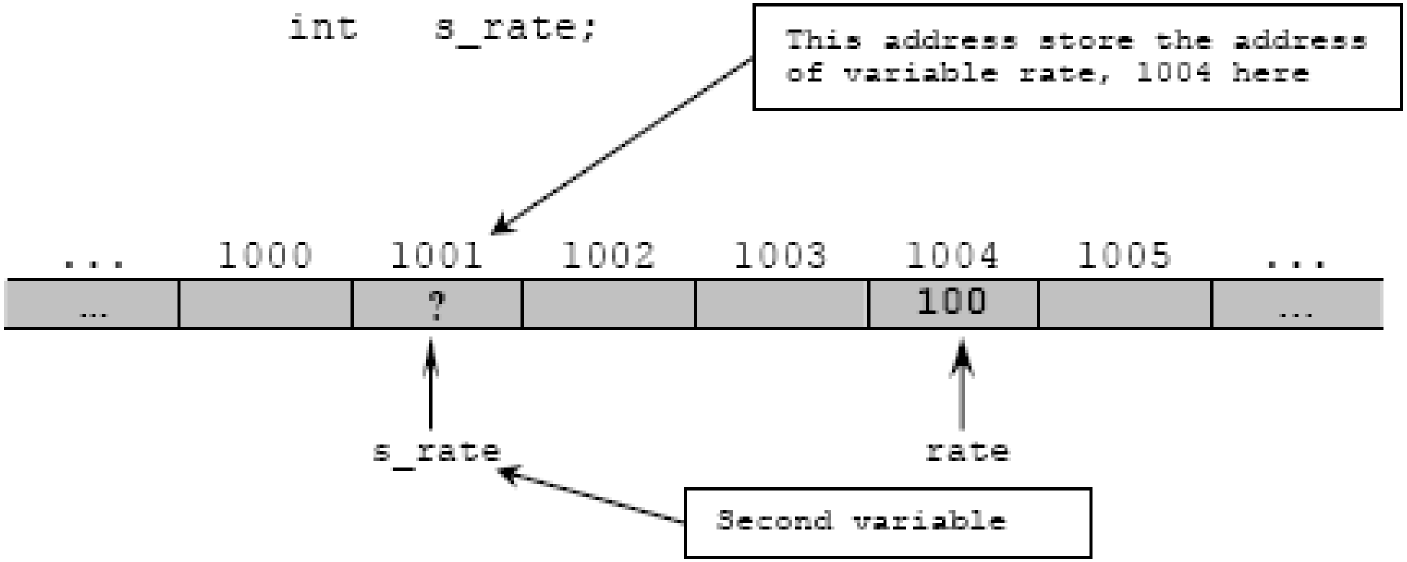


Introduction (Cont…)

- You can see, the memory address of the variable rate (or any other variable) is a number, so can be treated like any other number in C/C++.
- Normally the number is in hexadecimal format.
- Then, if a variable's memory address is known, the programmer can create a second variable for storing a memory address of the first variable.
- We can declare another variable to hold the memory address of variable rate; let say, s_rate

Introduction (Cont...)

- At the beginning, s_rate is uninitialized.
- So, storage has been allocated for s_rate, but its value is undetermined, as shown below.



```
int    s_rate;
```

This address store the address of variable rate, 1004 here

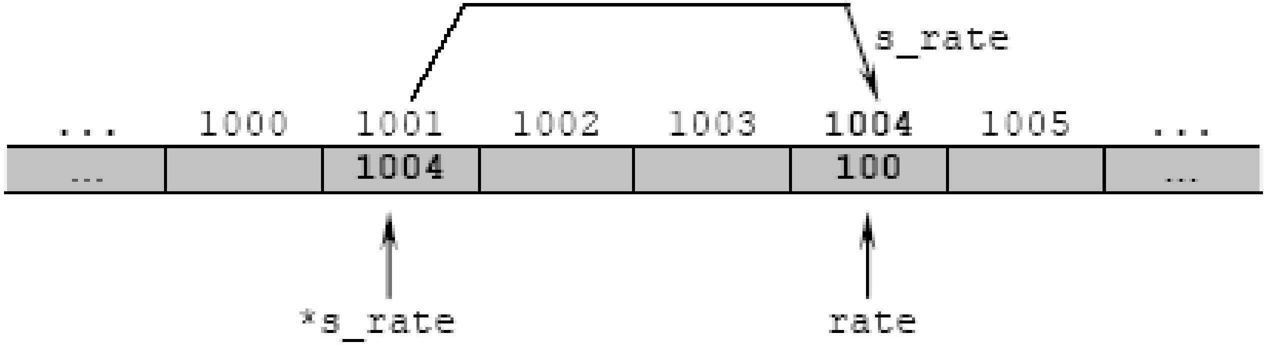
...	1000	1001	1002	1003	1004	1005	...
...		?			100		...

Second variable

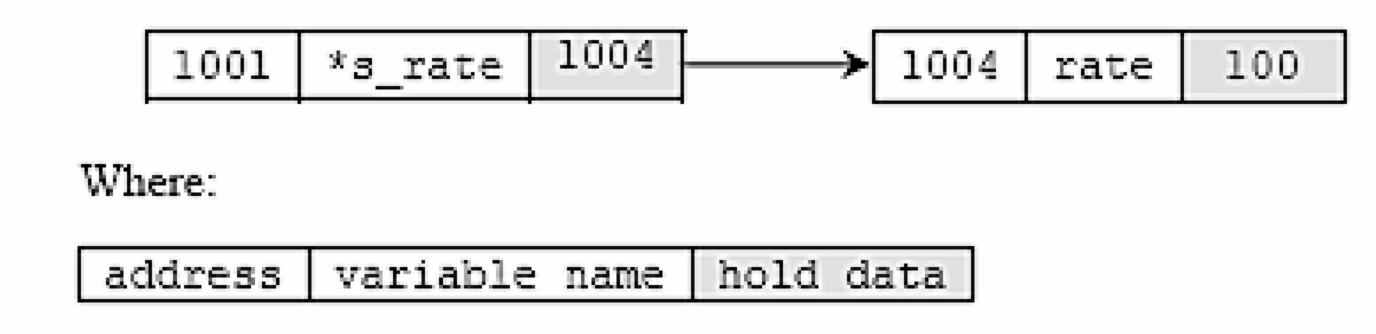
Introduction (Cont…)

- Let store the memory address of variable rate, in variable s_rate, so, s_rate now contains the memory address of rate, which indicates its storage location in memory where the actual data (100) is stored.
- Finally, in C/C++ vocabulary, s_rate is pointing to *rate* or is said a **pointer to *rate*.**

Introduction (Cont…)



- In simplified form ..



1001	*s_rate	1004

→

1004	rate	100

Where:

address	variable name	hold data

Declaring a Pointer Variable

- So the declaration of the pointer variable becomes something like this:

```
int *s_rate;
```

- The asterisk (*) is used to show that is it the pointer variable instead of normal variable.
- **A pointer is a variable that contains the memory address of another variable, where, the actual data is stored.**

Declaring a Pointer Variable (Cont…)

- A pointer is a numeric variable and like other variables, must be declared and initialized before it can be used.
- The following is a general form for declaring a pointer variable:

```
data_type  *pointer_variable_name;
```

For e.g.

```
char* x;
int * type_of_car;
float *value;
```

x is a pointer to a variable of type char.

***, is valid for all the three positions**

The asterisk (*) is called **indirection operator,**

Declaring a Pointer Variable (Cont...)

- Pointers can be declared along with non pointer variables as shown below:

```
char *ch1, *ch2;

// ch1 and ch2 both are pointers to type
  char.

float *value, percent;

// value is a pointer to type float, and
  percent is an ordinary float variable.
```

Initializing Pointers

- Once a pointer is declared, the programmer must initialize the pointer, that is, make the pointer point to something.
- **Don’t make it point to nothing; it is dangerous.**
- Like regular variables, uninitialized pointers will not cause a compiler error, **but using an uninitialized pointer could result in unpredictable and potentially disastrous outcomes.**
- **Until pointer holds an address of a variable, it isn’t useful.**

Initializing Pointers (Cont…)

- C/C++ uses two pointer operators:
  1. Indirection operator (*) – has been explained.
  2. Address-of-operator (&) – means return the address of.
- When & placed before the name of a variable, the address-of-operator returns the memory address of the variable/operand.

Initializing Pointers (Cont…)

- Hence, a pointer variable can be initialized as follows;

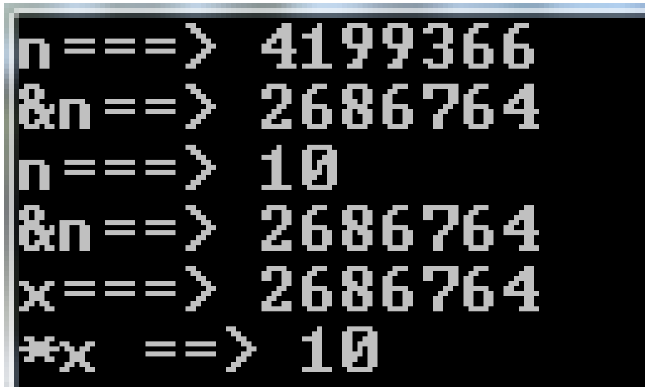
```
pointer_variable_name = &variable;
```

e.g.

```
s_rate = &rate;
```

Initializing Pointers (Cont…)

```
#include <stdio.h>
int main (){
    int n; int *x;

    printf("n===> %i\n",n);
    printf("&n==> %i\n",&n);

    n =10;
    printf("n===> %i\n",n);
    printf("&n==> %i\n",&n);

    x = &n;
    printf("x===> %i\n",x);
    printf("*x ==> %i\n",*x);
    return 0; }
```



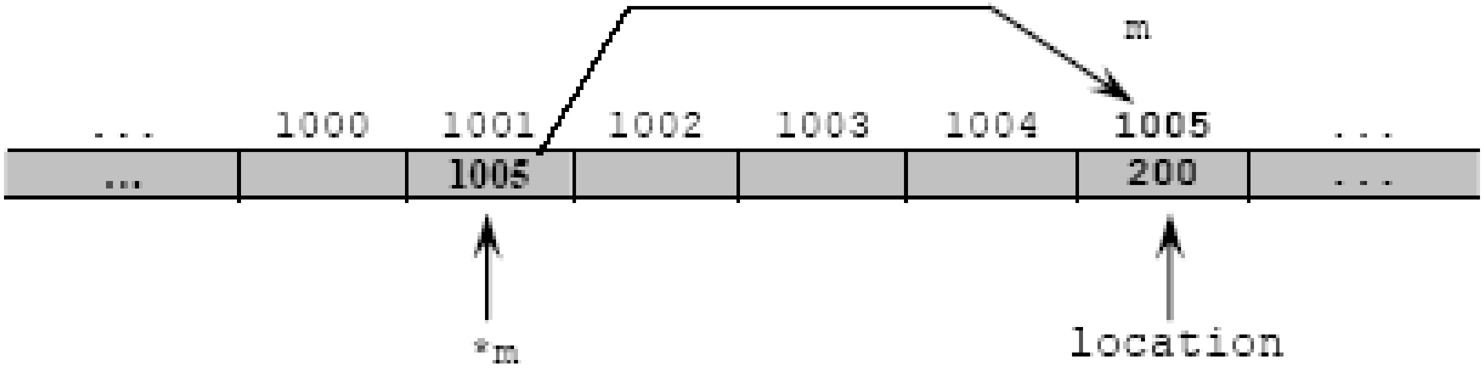
```
n===> 4199366
&n==> 2686764
n===> 10
&n==> 2686764
x===> 2686764
*x ==> 10
```

Initializing Pointers (Cont…)

```
#include <stdio.h>
int main()
{
    int *m;
    int location = 200;
    m = &location;
    printf("The data, *m = %i\n",*m);
    printf("The address of  -location- variable ,the data stored in m = %i\n", m);

return 0;
}
```

Initializing Pointers (Cont…)

- Pointer variable ***m*** receives the address of variable ***location*** or the memory address of the variable **location** is assigned to pointer variable ***m***.



Initializing Pointers (Cont…)

```
#include <stdio.h>
int main ()
{
    /*Following declaration is also legal*/
    int count = 10, x,*int_pointer;
    int_pointer = &count;
    x = *int_pointer;
    printf ("count = %i, x = %i\n", count, x);
    return 0;
}
```

Initializing Pointers (Cont…)

```
#include <stdio.h>
int main (){
    char c = 'Q';
    char *char_pointer = &c;
    printf ("%c %c\n", c, *char_pointer);

    c = '/';
    printf ("%c %c\n", c, *char_pointer);

    *char_pointer = '(';
    printf ("%c %c\n", c, *char_pointer);
    return 0;
}
```

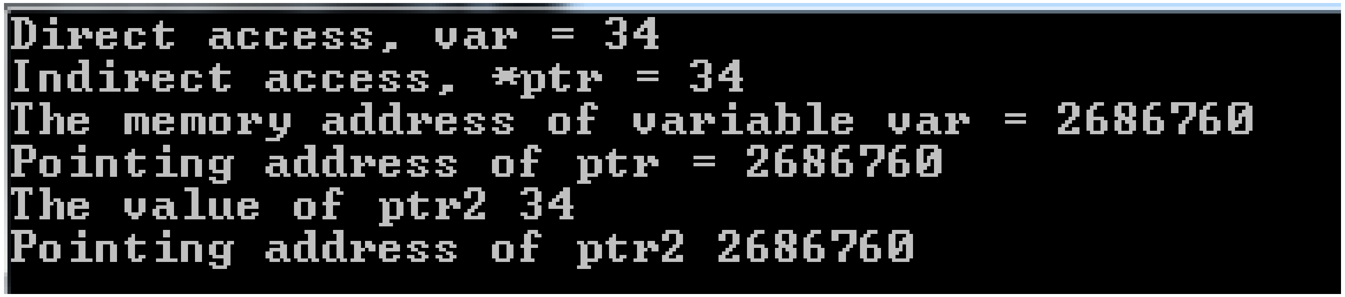
Initializing Pointers (Cont…)

- Where Pointers?
  - The * operator appears before a pointer variable in only two places:
    - When declaring a pointer variable.
    - When dereferencing a pointer variable (to find the data it points to).

Using Pointers in Expressions

```
#include <stdio.h>
int main ()
{
    int var = 34;
    int *ptr,*ptr2;
    ptr = &var;
    printf("Direct access, var = %i\n", var);
    printf("Indirect access, *ptr = %i\n", *ptr);
    printf("The memory address of variable var = %i\n", &var);
    printf("Pointing address of ptr = %i\n", ptr);
    ptr2=ptr;
    printf("The value of ptr2 %i\n",*ptr2);
    printf("Pointing address of ptr2 %i\n",ptr2);
    return 0;
```

Using Pointers in Expressions (Cont…)



```
Direct access, var = 34
Indirect access, *ptr = 34
The memory address of variable var = 2686760
Pointing address of ptr = 2686760
The value of ptr2 34
Pointing address of ptr2 2686760
```

- From the above example, we can:
  1. Access the contents of a variable by using the variable name (var) and is called direct access.
  2. Access the contents of a variable by using a pointer to the variable (*ptr or *ptr2) and is called indirect access or indirection.

Using Pointers in Expressions (Cont...)

```
#include <stdio.h>
int main (void){
    int i1, i2;
    int *p1, *p2;
    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;
    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n"  ,  i1, i2, *p1, *p2);
    return 0;
}
```

Constant Pointers

- A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

- A constant pointer is declared as follows :

  <type of pointer> * const <name of pointer>

- Example :

  int * const ptr;

```
#include<stdio.h>

int main(void)
{
    int var1 = 10, var2 = 20;
    int *const ptr = &var1;
    ptr = &var2; /* This line will throw an error
       as: assignment to read only variable ptr*/

    printf("%d\n", *ptr);
    return 0;
}
```

So once assigned the **const** pointer’s value cannot be changed

```
char c = ‘N‘, d;
char *charPtr = &c;

charPtr = &d; // not valid
*charPtr = 'Y‘; // valid
```

Pointer to constant

- As evident from the name, a pointer through which one cannot change the value of variable it points, is known as a pointer to constant.
- These type of pointers can change the address they point to but cannot change the value kept at those address.
- A pointer to constant is defined as :

```
const <type of pointer>* <name of pointer>
```

- Example :

```
const int* ptr;
```

```
#include<stdio.h>
int main()
{
    int var1 =10;
    const int* ptr = &var1;
    *ptr = 15; ; /* This line will throw an error
       as: assignment to read only location */
    printf("%d\n", *ptr);

    return 0;
}
```

So in **pointer to constants**, once assigned the pointer cannot be change the value of variable it points to.

```
*charPtr = 'Y'; // not valid
 charPtr = &d // valid
```

The Keyword const and Pointers (eg:)

```
#include <stdio.h>
int main ()
{
    char c = 'X', d = 'T';
    char * const charPtr = &c;
    const char *charPtr2 = &c;

    charPtr = &d; //invalid statement
    *charPtr = 'Y'; // Valid statement

    charPtr2 = &d; // Valid statement
    *charPtr2 = 'Y'; // Invalid statement

    return 0; }
```

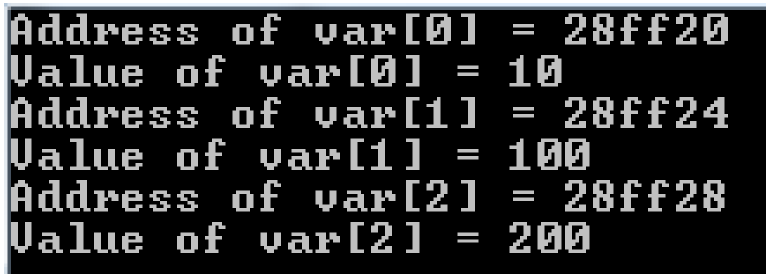
Pointer Arithmetic

- Following arithmetic operations are possible on pointers
  - Increment
  - Decrement
  - Addition
  - Subtraction

  Each time the pointer is incremented, it points to the next integer and similarly, when a pointer is decremented, it points to the previous integer
- ***Differencing***
  - For example, two pointers that point to different elements of the same array can be subtracted to find out how far apart they are.

Pointer Arithmetic (Cont…)

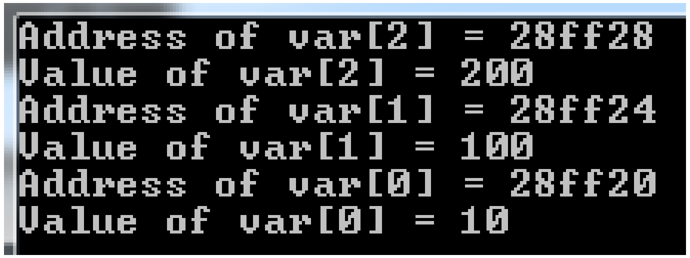
```
#include <stdio.h>
const int MAX = 3;

int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have array address in pointer */
   ptr = var;
   for ( i = 0; i < MAX; i++)
   {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %i\n", i, *ptr );
      /* move to the next location */
      ptr++;
   }
return 0; }
```



```
Address of var[0] = 28ff20
Value of var[0] = 10
Address of var[1] = 28ff24
Value of var[1] = 100
Address of var[2] = 28ff28
Value of var[2] = 200
```

Pointer Arithmetic (Cont...)

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  /* let us have array address in pointer */
  ptr = &var[MAX-1];
  for ( i = MAX; i > 0; i--)
  {
    printf("Address of var[%d] = %x\n", i-1, ptr );
    printf("Value of var[%d] = %d\n", i-1, *ptr );
    /* move to the previous location */
    ptr--;
  }
  return 0; }
```



```
Address of var[2] = 28ff28
Value of var[2] = 200
Address of var[1] = 28ff24
Value of var[1] = 100
Address of var[0] = 28ff20
Value of var[0] = 10
```

Pointer Arithmetic (Cont...)

- **Pointer Comparison**
  - The comparison is valid only between pointers that point to the same array.
  - Under this circumstances, the following relational operators work for pointers operation.

    ==, !=, >, <, >= ,<=

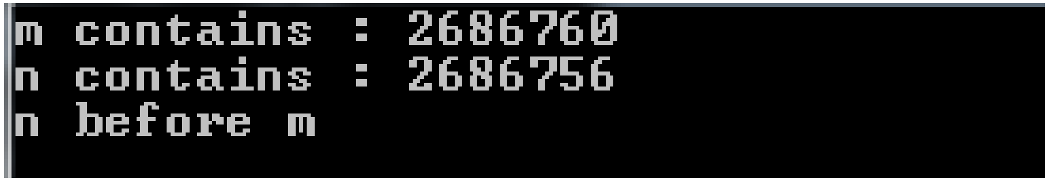
  - A lower array element that is those having a smaller subscript/index , always have a lower address than the higher array elements.
  - Thus if ptr1 and ptr2 point to elements of the same array, the following comparison:

    ptr1 < ptr2

is TRUE If ptr1 points to an earlier member of the array than ptr2 does.

Pointer Operation (Cont…)

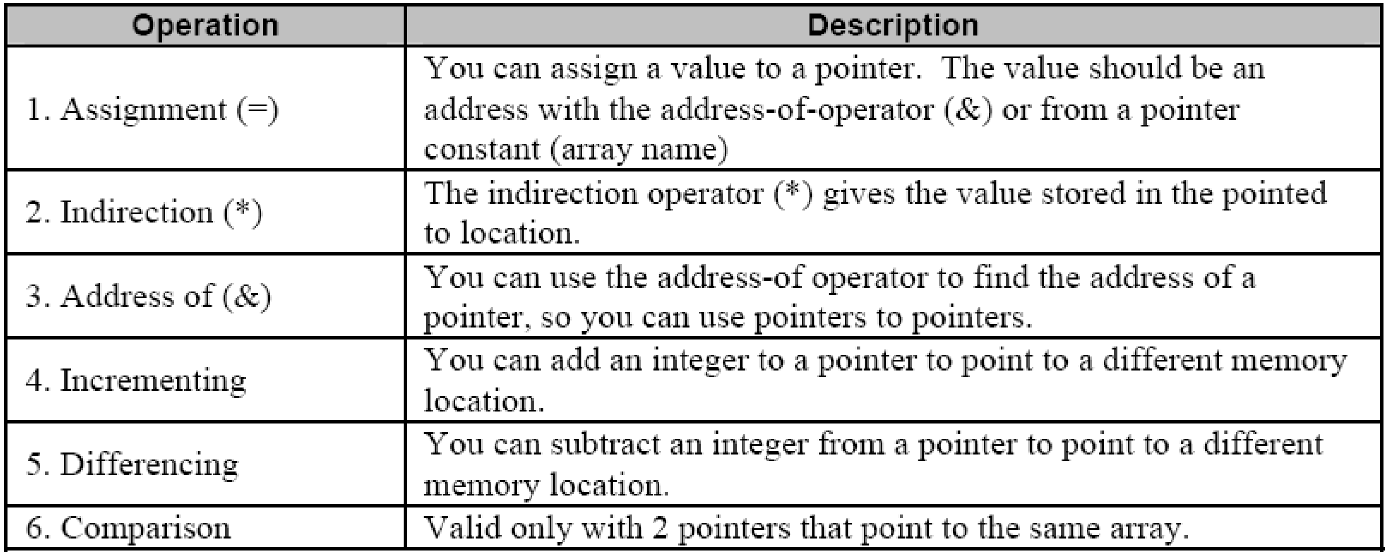
```
#include <stdio.h>
int main (){
    int *m;  int *n;
    int q,r = 35;
    m = &q;   n = &r;
    printf ("m contains : %i\n",m);
    printf ("n contains : %i\n",n);
    if ( m <n ){
        printf ("m before n\n");
    }else{
        printf ("n before m\n");
    }
    return 0; }
```



```
m contains : 2686760
n contains : 2686756
n before m
```

Pointer Operation (Cont…)

- Many arithmetic operations that can be performed with regular variables, such as multiplication and division, do not work with pointers and will generate errors in C/C++.
- The following table is a summary of pointer operations.



Operation	Description
1. Assignment (=)	You can assign a value to a pointer. The value should be an address with the address-of-operator (&) or from a pointer constant (array name)
2. Indirection (*)	The indirection operator (*) gives the value stored in the pointed to location.
3. Address of (&)	You can use the address-of operator to find the address of a pointer, so you can use pointers to pointers.
4. Incrementing	You can add an integer to a pointer to point to a different memory location.
5. Differencing	You can subtract an integer from a pointer to point to a different memory location.
6. Comparison	Valid only with 2 pointers that point to the same array.

NULL Pointers

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned.
- This is done at the time of variable declaration.
- A pointer that is assigned NULL is called a **null**pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries.

NULL Pointers (Cont...)

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );
   printf("The value of ptr is : %i\n", ptr  );

   return 0;
}
```

NULL Pointers (Cont…)

- In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system.
- However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location.
- But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.
- To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

- To play safe, you can also set a pointer to NULL to indicate that it’s no longer in use.

Thank You

NEXT : POINTERS (PART 2)