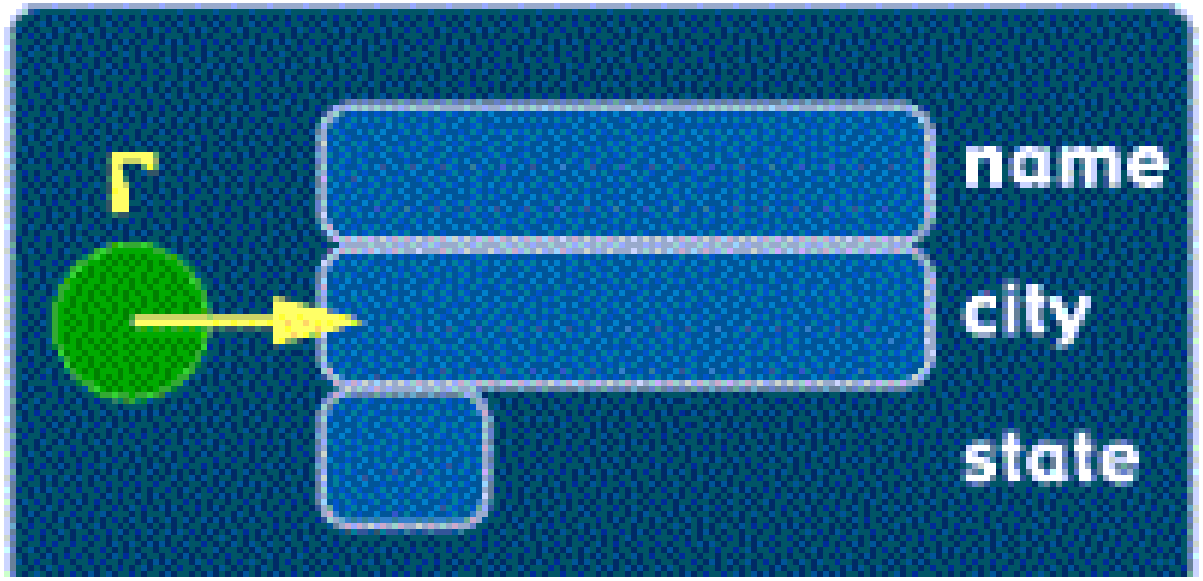


Advanced Pointers

Objectives

To learn and appreciate the following concepts

- Operations on pointers
- Pointers and Arrays
- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers

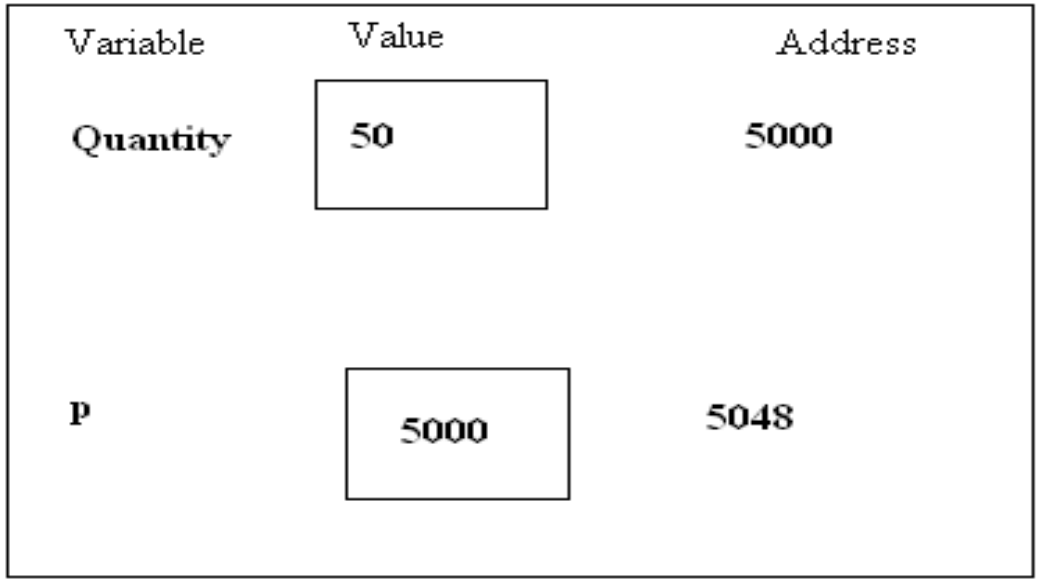
Session outcome

At the end of session one will be able to understand

- Operations on pointers
- Pointers and Arrays
- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers

Pointers- A recap…

```
int Quantity;     //defines variable Quantity of type int
int* p;           //defines p as a pointer to int
p = &Quantity;    //assigns address of variable Quantity to pointer p
```



Now…

```
Quantity = 50;//assigns 50 to Quantity
*p = 50; //assigns 50 to Quantity
```

Pointer expressions

- Pointers can be used in most valid C expressions. However, some special rules apply.
- You may need to surround some parts of a pointer expression with parentheses in order to ensure that the outcome is what you desire.
- As any other variable, a pointer may be used on the right side of an assignment operator to assign its value to another pointer.

Pointer Expressions - Example

- Eg: int a=10, b=20,c,d=10;

  int *p1 = &a, *p2 = &b;

Expression	A	b	c
c = *p1p2; OR *p1 * *p2 OR (*p1) * (*p2)**	10	20	**200**
c = c + *p1;	10	20	**210**
c = 5 * - *p2 / *p1; **OR** (5 * (- (*p2)))/(*p1) **//space between / and * is required**	10	20	**-10**
***p2 =*p2 +10;**	10	30	

Operations on Pointer Variables

- **Assignment** – the value of one pointer variable can be assigned to another pointer variable of the same type
- **Relational operations** - two pointer variables of the same type can be compared for equality, and so on
- **Some limited arithmetic operations**
  - integer values can be added to and subtracted from a pointer variable
  - value of one pointer variable can be subtracted from another pointer variable
  - Shorthand Increment and Decrement Operators

Valid Pointer Operations - Example

- int a = 10, b = 20, *p1, *p2, *p3, *p4;
- **p1 = &a;** //assume address of a = 2004
- **p2 = &b;** //assume address of b = 1008

Assume an integer occupies 4 bytes

Pointer Operations	Example expression	Result
Addition of integers from pointers	**p3 = p1 + 2**	value of p3 = 2004 + 4*2 = 2012
Subtraction of integers from pointers	**p4 = p2 – 2**	value of p4 = 1008-4*2 = 1000
Subtraction of one pointer from another	**c = p3– p1**	Value of c = 2012– 2004= 2
Pointer Increment	**p1++**	Value of p1 = 2008
Pointer Decrement	**--p1**	Value of p1 = 2004

Invalid Operations:

- Pointers are not used in division and multiplication.

  p1 / *p2;

  p1*p2;

  p1/3; are not allowed.

- Two pointers can not be added.

  p1 + p2 is illegal.

Program to exchange two values using pointers

```
int main() {

int x, y, t, *a, *b;
a=&x; b=&y;
printf("Enter the values of a and b: \n");
scanf("%d %d", a, b); // equivalent to scanf("%d %d", &x, &y);
 t=*a;
*a=*b;
*b=t;

printf("x = %d \n", x);
printf("y = %d", y);

return 0;
}
```

```
Enter the values of a and b:
10  5
x= 5
y = 10
```

Pointers and arrays

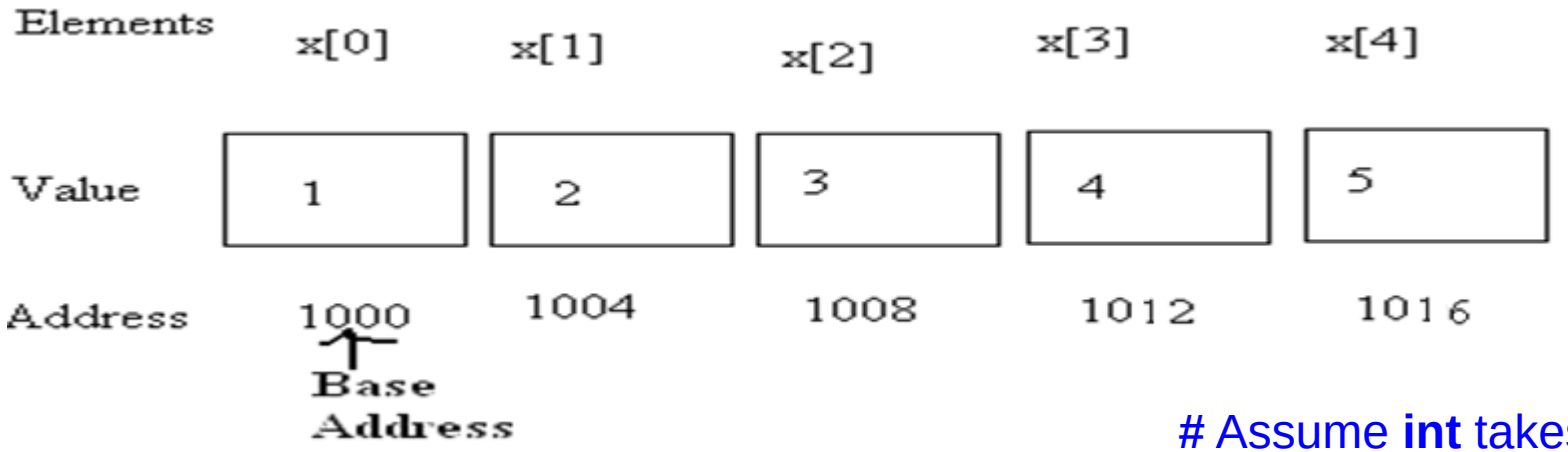
- When an array is declared, the compiler allocates a **Base Address (BA)** and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The **base address** is the location of the first element (index 0) of the array.
- The compiler also defines the array name as a **constant pointer** to the first element.

Pointers and arrays

- An array x is declared as follows and assume the base address of **x** is **1000.**

int x[5] ={ 1,2,3,4,5};

- Array name **x**, is a **constant pointer**, pointing to the first element **x[0]** .
- For example, if value of **x** is **1000 (Base Address)**, the location of **x[0]**.
  i.e. **x** is same as **&x[0]** equals **1000** (in the illustration below)



Assume **int** takes 4 bytes

Array accessing using Pointers

- An **integer pointer variable p**, can be made to point to an array as follows:

```
int  x[5] ={ 1,2,3,4,5};
int *p;
p = x;     OR      p = &x[0];
```

- **Following statement is Invalid:**

```
p = &x ; //Invalid
```

- **Successive array elements can be accessed** by writing:

```
printf("%d", *p);
        p++;
or
printf("%d", *(p+i));
        i++;
```

Pointers and arrays

- The relationship between **p** and **x** is shown below:

```
p= &x[0];        (=1000) BASE ADDRESS
p+1=>&x[1]       (=1004)
p+2=>&x[2]       (=1008)
p+3=>&x[3]       (=1012)
p+4=>&x[4]       (=1016)
```

- **Address of an element of x is given by:**

Address of **x[i]** = **base address** + **i** * **scale factor of (int)**

Address of x[3]= 1000 +(3*4) = 1012

Assume **int** takes 4 bytes

Array accessing using *array name* as a pointer - Example

```
// array accessed with constant pointer array name arr
int main()
{

 int arr[5] = { 31, 54, 77, 52, 93 };

 for(int j=0; j<5; j++)    //for each element,
 printf("%d ", *(arr+j));        //print value

 return 0;
}
```

Array accessing using Pointers - Example

```
// array accessed with explicit pointer ptr

int main() {

  int arr[5] = { 31, 54, 77, 52, 93 };
  int* ptr;   //pointer to arr
  ptr = arr;   //points to arr

 for(int j=0; j<5; j++)    //for each element
 printf("%d ", *ptr++);

return 0;
}
```

"ptr" is a pointer which can be used to access the elements.

Sum of all elements stored in an array

```
int main() {
   int  *p, sum=0, i=0;
   int  x[5] ={5, 9, 6,3,7};
   p=x;
   while(i<5) {
      sum+=*p;
      i++;
      p++;
   }
   printf("sum of elements = %d“, sum);
   return 0;
}
```

```
sum of elements =30
```

Pointers & Character strings

```
//length of the string
int main() {
   char name[15];
   char *cptr=name;
   printf("Enter some word to find its length: \n“);
   scanf(“%s”, name);
   while(*cptr!= '\0')
         cptr++;
   printf("length= %d“, cptr-name);
   return 0;
}
```

```
Enter some word to find its length
Computer
length=8
```

Pointers & Character strings

- The statements

```
char name[10];
char  *cptr =name;
```

declares cptr as a pointer to a character array and assigns address of the first character of name as the initial value.

- The statement `while(*cptr!='\0')`

is true until the end of the string is reached.

- When the while loop is terminated, the pointer cptr holds the address of the null character ['\0'].
- The statement `length = cptr – name;` gives the length of the string name.

Pointers & Character strings

- A constant character string always represents a pointer to that string.
- The following statements are valid.

```
char *name;

name =“Delhi”;
```

These statements will declare **name** as a **pointer** to character array and assign to name the constant character string “Delhi”.

Pointers and 2D arrays

```
int a[ ][2]={   {12, 22},
                {33, 44} };
int  (*p)[2];
p=a; // initialization
```

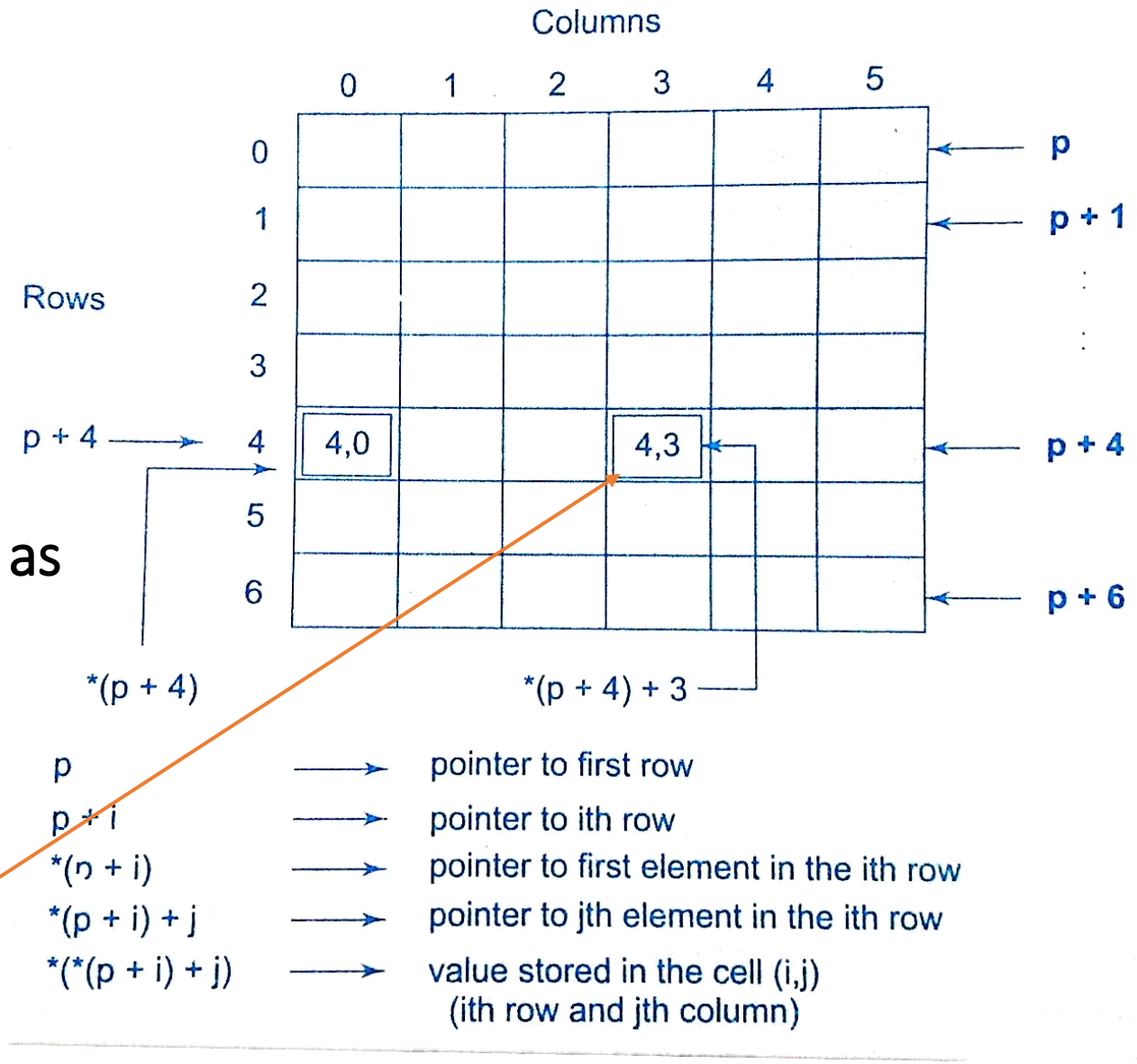
So, an element in 2d represented as

((a+i)+j)

or

((p+i)+j)

((p+4)+3)



Pointers and 2D arrays

```
// 2D array accessed with pointer

int main() {

   int i, j, (*p)[2], a[][2] = {{12, 22}, {33, 44} };

   p=a;

   for(i=0;i<2;i++) {
      for(j=0;j<2;j++)
         printf("%d \t", *(*(p+i)+j));
      printf("\n");
   }

   return 0;
}
```

Array of pointers - concept

- We can use pointers to handle a table of strings.

  char sports[5][15];

  sports is a table containing 5 sport names, each with a maximum length of 15 characters (including '\0')

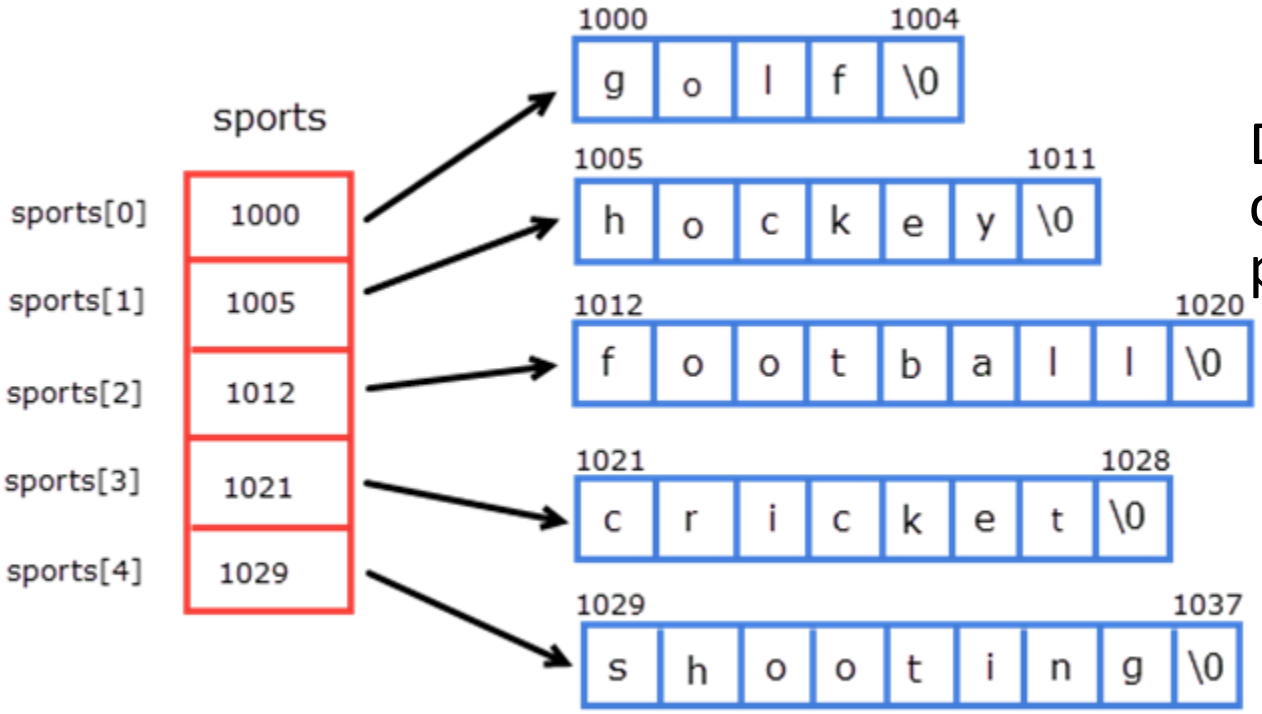
- Total **storage** requirement for **sports** is **75 bytes**.

  But rarely all the individual strings will be equal in lengths.

- We can use a pointer to a string of varying length as

char *sports[5] = { "golf", "hockey", "football", "cricket", "shooting" };

Array of pointers



Memory representation of array of pointers

So, **char *sports[5] = { "golf",**
"hockey",
"football",
"cricket",
"shooting"};

Declares **sports** to be an **array of 5 pointers** to characters, each pointer pointing to a particular sport.

sports[0] → golf
sports [1]→ hockey
sports [2]→ football
sports [3]→ cricket
sports [4]→ shooting

This declaration allocates 33 bytes.

Array of pointers

The following statement would print out all the 5 names.

```
for(i=0; i<=5; i++)
  printf("%s",sports[i]);
```

or

```
printf("%s", *(sports + i));
```

To access the j^{th} character in the i^{th} name, we may write as

```
*(sports[i] + j)
```

The character array with rows of varying lengths are called **ragged arrays** and are better handled by pointers.

Benefits (use) of pointers

- Pointers provide **direct access** to memory.
- Pointers provide a way to **return more than one value** to the functions.
- Reduces the **storage space and complexity** of the program.
- Reduces the **execution time** of the program.
- Provides an **alternate way** to access array elements
- Pointers can be used to **pass information back and forth** between the calling function and called function.

Drawbacks of pointers

- Uninitialized pointers might cause **Segmentation fault**.
- Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to **Memory leak**.
- Pointers are **slower** than normal variables.
- If pointers are updated with incorrect values, it might lead to **Memory corruption**.

Summary

- Pointers -recap
- Basic operations on pointers
- Pointers and Arrays
- Pointers and Character Strings
- Pointers and 2D
- Array of Pointers