Chapter 7

Pointers

Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline…

- Pointers: What & Why?
- Understanding Pointers
- Working with Pointers
- Chain of Pointers
- Pointer Arithmetic
- Pointers and Arrays
- Array of Pointers
- Assignments

Pointers: What & Why?

- **What is a Pointer?:**
  - **[Definition]: A pointer is a variable that can store the address of another variable or function.**

    In this chapter, we will discuss how can we make a variable (a pointer) to store the address of another *variable*. However, the 2nd part of the definition i.e.,, how a pointer can be made to store the address of a *function* will be discussed in the chapter "Functions" in the section "Pointers to Functions".

  - **It is a derived data type in C.**

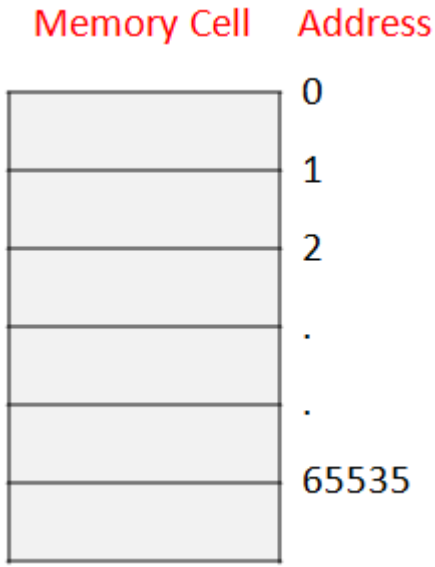
Advantages of Using Pointers:

- Pointers are more efficient in handling arrays.
- Pointers can be used to return multiple values from a function.
- Pointers permit reference to functions.
- Pointers allow dynamic memory management.
- Pointers allow an efficient tool for manipulating data structures like linked lists, stacks, queues, trees, graphs etc.
- Pointers increase the execution speed and thus reduce the execution time.

Understanding Pointers

- **Let us 1st get an overall understanding of the computer's memory (RAM):**
  - **The computer's memory is typically organized into a sequence of "cells" (storage cells). A cell is normally of one byte (8 bits).**
  - **Each cell (byte) is uniquely identified by a number, called its address.**
  - **The addresses always starts from 0 and the last address depends on the memory size.**

The memory layout of a computer system having 64KB RAM is shown here. (We have represented the addresses in decimal format. However, normally they are represented in hexadecimal format).



[NOTE]: A detailed discussion on the memory layout is made in the Chapter "Storage Classes of Variables".

- **Now, what happens when we declare a variable?**

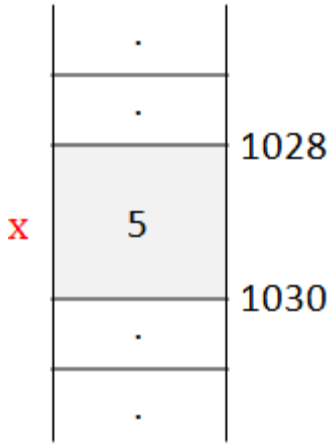
Let us consider the following declaration:

```
int x = 5;
```

The above declaration tells the C compiler to take the following actions:

1. **Reserves** 2 bytes **of unallocated space somewhere in the memory.**
2. **Associate the name** x **with this memory location.**
3. **Store the value** 5 **at this location.**

Here, we have *assumed* that the compiler has selected the memory location 1028 **(from** 1028 **to** 1030**) for the variable** x**.**

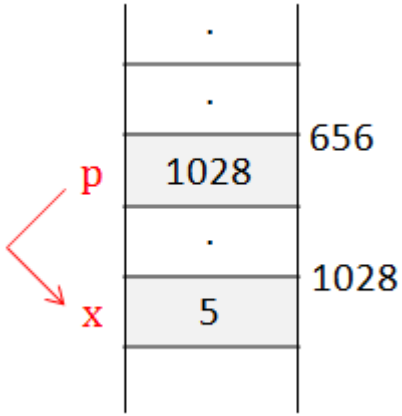


- **The point here is that the address** (address always means starting address) **of the variable** x **is a number (an unsigned positive integer) and since it is a number, it can also be collected in another variable.**

  Say, we chose a variable 'p' **for this purpose. If this is done, then the variable** 'p' **is called a pointer, because here** 'p' **is a variable that holds the address of (i.e., points to) another variable** 'x'.

  - **In our case, since** x **is an integer variable,** p **is called a pointer to an integer or an integer pointer.**

Note that, the pointer 'p' **is also a variable, and it also has an address (in our case it is** 656**).**



- **In the next section** "Working with Pointers" **we will discuss how to declare a variable to be of type pointer and how to store an address within it. But before this there are two questions that comes in mind:**

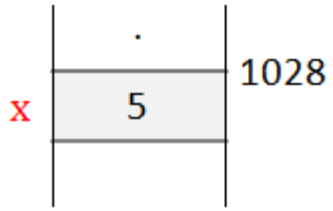
  1. **How can we determine the address of a variable so that we can store it within a pointer variable?**
     - **This question comes in mind because, the address of a variable is actually system dependent. In the previous discussion we have *assumed* the address of** x **to be** 1028.
  2. **Can we access the *value* of a variable through it address? If yes, how?**
     - **It is because, if we can't, then there is no meaning of storing the address of a variable in a pointer.**

- **C provides two operators for the above two issues:**
  - **The operator &:**
    - **It is called the 'address of' operator.**
    - **When prefixed with a variable, it returns the address of the variable i.e., `&x` will return `1028`.**
  - **The operator *:**
    - **It is called 'value at address' or 'indirection' operator.**
    - **When prefixed with an address, it returns the value stored at that address. i.e., `*(&x)` will return `5`.**

- **Let us clarify the concept through the same previous example:**

  Assume that, we have declared a variable like `int x = 5`, **and it is stored in the location** 1028. **Then,**

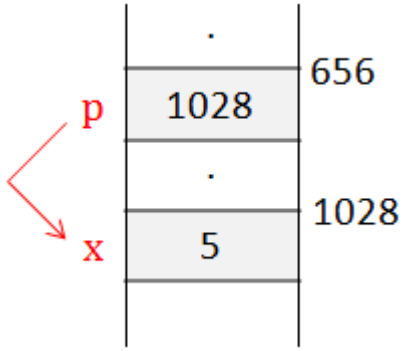
Expression	Result
x	5
&x	1028
*(&x)	5



Now, if we declare a pointer 'p' **that stores the address of** 'x' (we will see in the next section, how to declare a pointer and how to store an address within it).

Then,

Expression	Result
p	1028
*p	5
&p	656
*(&p)	1028



Working with Pointers

Declaring Pointers

- **Syntaxes:**

(1)
```
data_type* pointer_name;
```

e.g.,
```
int* p;          /* This declaration tells the compiler that 'p' is a pointer
                    variable (because a * precedes it) that can store the address
                    of an integer variable. In other words, 'p' is an integer
                    pointer. */
int* p, q;       /* 'p' and 'q' both are integer pointers. */
float* m, n;     /* 'm' and 'n' both are floating point pointers. */
```

(2)
```
data_type *pointer_name;
```
(Preferred)

e.g.,
```
int *p;          /* 'p' is an integer pointer. */
int *p, *q;      /* 'p' and 'q' both are integer pointers. */
float *m, n;     /* 'm' is a floating-point pointer, but 'n' is a floating-point
                    variable. */
```

- **A declaration causes the compiler to allocate memory for the pointer variable.**
  Initially, the pointer variable contains some garbage value (since it is not assigned to any address).
- **[NOTE]: How can we justify the usage of * in a declaration like "`int *p;`"?:**
  Let us go by the meaning of *. It stands for 'value at address'. Thus `int *p` would mean, the value at the address contained in 'p' is an integer.

Initializing Pointers

- **Importance: Every pointer variable need to be initialized** (to the address of a variable)**, because as pointed out earlier, an uninitialized pointer would contain some garbage value that will be interpreted as a memory address. Hence the programs with uninitialized pointers will produce erroneous results.**
- **Syntax: Demonstrated through the following example**

```
int x;
int *p;  /* Declaration */
p = &x;  /* Initialization */
```

```
int x;
int *p = &x;  /* Declaration and Initialization */
```

- **Notes:**

1. **We must ensure that a pointer variable always points to the corresponding type of data.**

```
int x, *p;
float m;
p = &m;  /*Error, because 'p' is an integer pointer. It can't hold the address
           of a floating-point variable*/
p = &x;  /* OK */
```

2. **A declaration like the following is perfectly legal.**

```
int x, *p = &x;  /* Three in one . Note that here 'x' is declared first*/
```

However, the following declaration is illegal.

```
int *p = &x, x;  /* Error */
```

3. **In case it is not decided to which variable a pointer should point, it is always a good practice to initialized the pointer to a NULL or 0 (zero).**

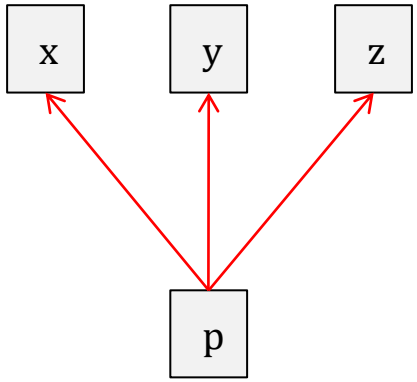
```
int *p = NULL; or int *p = 0;
```

4. **With the exception to a NULL or 0, no other constant value could be assigned to a pointer variable.**

```
int *p = 1028; /* Absolute address, Error */
```

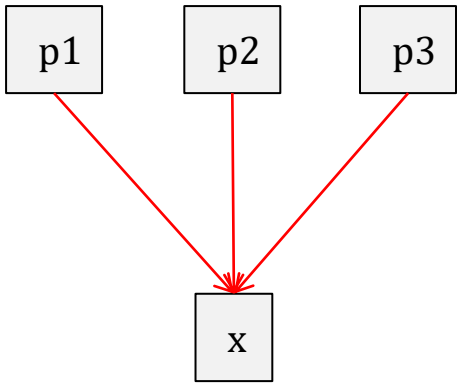
5. **We can make the same pointer to point to different variables at different times as per our need.**

```
int x, y, z, *p;
...
p = &x;
...
p = &y;
...
p = &z;
```



6. **We can also use different pointers to point to the same variable if needed.**

```
int x, *p1, *p2, *p3;
...
p1 = &x;
...
p2 = &x;
...
p3 = &x;
```



Accessing a Variable Through its Pointer

- **As we have already discussed, if 'p' is a pointer to a variable 'x' i.e.,**

```
int x = 5;
int *p = &x;
```

Then, the value of 'x' could be accessed by the expression *p. i.e.,

```
printf("%d", *p);
```

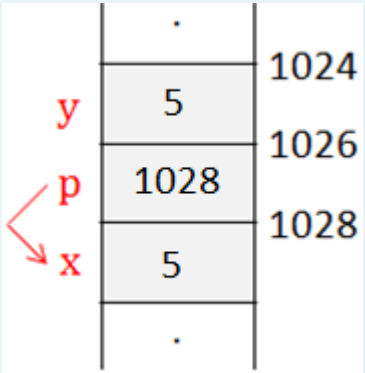
would print 5.

A Programming Example

- **The following program summarizes all the concepts that we have learnt so far.**

Try to figure out the outputs with the help of the *memory map*.

```
/* PR7_1.c: Pointer notations */
# include <stdio.h>

void main()
{
     int x, *p, y;
     x = 5;
     p = &x;
     y = *p;

     printf("\nx = %d", x);
     printf("\nx = %d ; Address of x = %u", x, &x);
     printf("\nx = %d ; Address of x = %u", *(&x), &x);
     printf("\nx = %d ; Address of x = %u", *p, p);
     printf("\n\np = %u ; Address of p = %u", p, &p);
     printf("\np = %u ; Address of p = %u", *(&p), &p);
     printf("\ny = %d ; Address of y = %u", y, &y);
     *p = 25;
     printf("\nNow x = %d", x);
}
```



Output

```
x = 5
x = 5 ; Address of x = 1028
x = 5 ; Address of x = 1028
x = 5 ; Address of x = 1028
p = 1028 ; Address of p = 1026
p = 1028 ; Address of p = 1026
y = 5 ; Address of y = 1024
Now x = 25
```

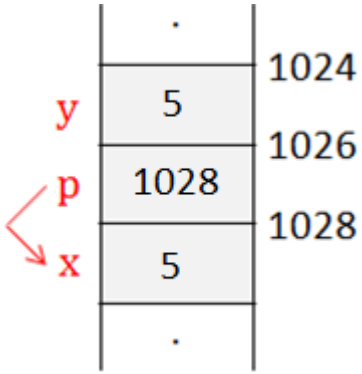
Value of x
x = *(&x) = *p

Address of x
&x = p

- **A NOTE:**

In the "memory map" given in the last program the variables are shown (to be stored) in the reverse order of their declaration.

```
int x, *p, y;
```



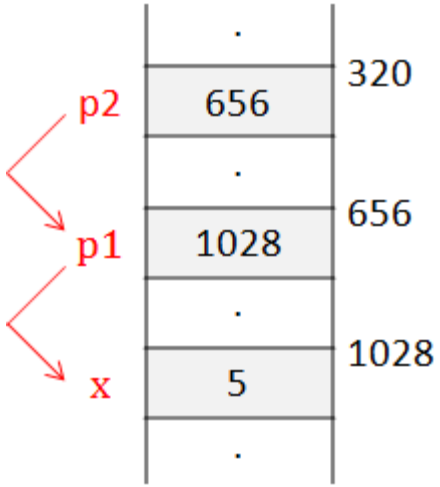
There is a reason behind this.

These variables are actually stored in an area of RAM known as the "stack area or stack segment". In this part of RAM, the variables are organized in bottom to top manner (like in a stack) in the order of declaration.

We shall have a detailed discussion on this in the Chapter "Storage Classes of Variables".

Chain of Pointers

- The concept of pointers can be further extended. It is possible to make a pointer to point to another pointer, thus creating a chain of pointers, as shown.
  - This is also known as "multiple indirection".
  - Here, p2 is a pointer to the integer pointer p1.



- **Declaration:** A pointer to a pointer is declared using an additional indirection operator, as shown

```
data_type **pointer_name;
```

For example, here p2 will be declared as:

```
int **p2;
```

- **Initialization:** **A pointer to a pointer must be initialized to the address of a pointer. For Example:**

```
int x, *p1, **p2; /* Declaration*/
x = 5;
p1 = &x;
p2 = &p1; /* Initialization */
```

```
/* Declaration and Initialization at the same place*/
int x = 5, p1 = &x, p2 = &p1;
```

- **Accessing The Variable:** **We can access the target variable (in our case 'x') by applying the indirection operator twice to the pointer to pointer. i.e.,**

```
printf("%d", **p2);
```

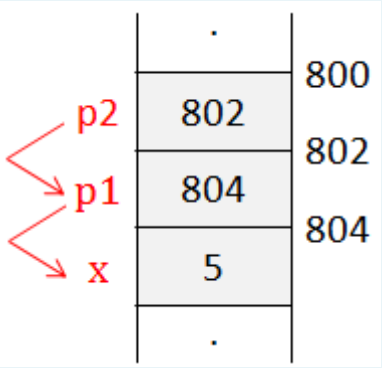
will print 5.

- [NOTE]:

Like we have defined *pointer to pointer*, we can have *pointer to pointer to pointer* (int ***p). In principle, there is no limit on how far we can go one extending this concept. However, beyond to *"pointers to pointers"* is rarely used, because they are very difficult to comprehend.

Programming Example:

```
/* PR7_2.c: Pointer to pointer demonstration */

# include <stdio.h>

void main()
{
     int x = 5, *p1, **p2;
     p1 = &x;
     p2 = &p1;

     printf("\nAddress of x = %u", &x);
     printf("\nAddress of x = %u", p1);
     printf("\nAddress of x = %u", *p2);
     printf("\nAddress of p1 = %u", &p1);
     printf("\nAddress of p1 = %u", p2);
     printf("\nAddress of p2 = %u", &p2);
     printf("\nValue of p1 = %u", p1);
     printf("\nValue of p2 = %u", p2);
     printf("\nValue of x = %d", x);
     printf("\nValue of x = %d", *(&x));
     printf("\nValue of x = %d", *p1);
     printf("\nValue of x = %d", **p2);

     getch();
}
```



Output

```
Address of x = 804
Address of x = 804
Address of x = 804
Address of p1 = 802
Address of p1 = 802
Address of p2 = 800
Value of p1 = 804
Value of p2 = 802
Value of x = 5
Value of x = 5
Value of x = 5
Value of x = 5
```

Pointer Arithmetic

- The following operations are allowed on pointers:

  1. Addition of an integer to a pointer.

     When ever a pointer is incremented (by 1), it is incremented to the number of bytes taken by its data type.

```
int x;
float y;
...
int *p = &x;    /* Let 'p' is initialized to 6000 (i.e., the address of x = 6000) */
float *q = &y;  /* Let 'q' is initialized to 2000 (i.e., the address of y = 2000) */
...
printf("\np = %u", p);       /* Output: p = 6000 */
printf("\n++p = %u", ++p);   /* Output: ++p = 6002 (i.e., 6000+1*2) */
printf("\np+7 = %u", p+7);    /* Output: p+7 = 6016 (i.e., 6002+7*2) */
printf("\n7+p = %u", 7+p);    /* Output: 7+p = 6016 (i.e., 6002+7*2) */
...
printf("\nq = %u", q);       /* Output: q = 2000 */
printf("\nq+1 = %u", q+1);   /* Output: q +1= 2004 (i.e., 2000+1*4) */
printf("\nq+7 = %u", q+7);   /* Output: q +7= 2028 (i.e., 2000+7*4) */
```

2. Subtraction of an integer from a pointer.

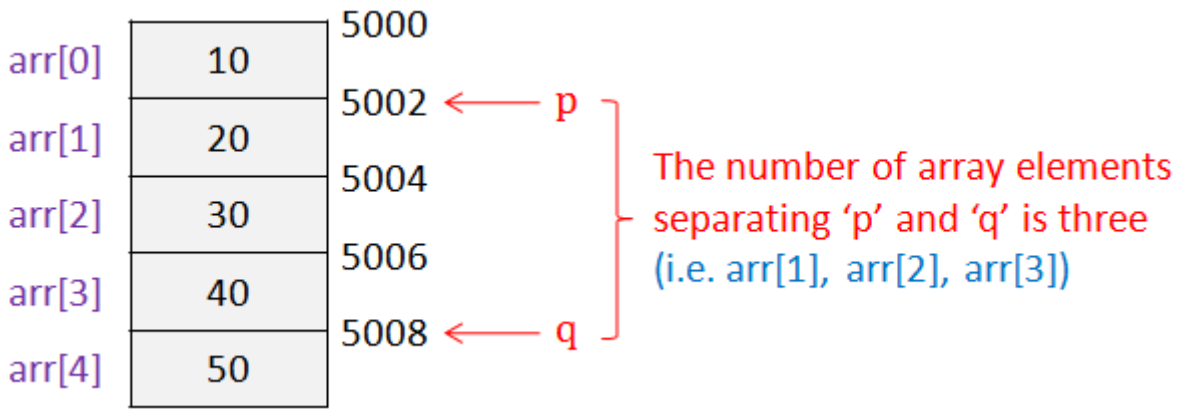
When ever a pointer is decremented (by 1), it is decremented to the number of bytes taken by its data type.

```
int x;
float y;
...
int *p = &x;    /* Let 'p' is initialized to 6000 (i.e., the address of x = 6000) */
float *q = &y;  /* Let 'q' is initialized to 2000 (i.e., the address of y = 2000) */
...
printf("\np = %u", p);       /* Output: p = 6000 */
printf("\np-1 = %u", p-1);   /* Output: p-1 = 5998 (i.e., 6000-1*2) */
printf("\np-5 = %u", p-5);   /* Output: p-5 = 5990 (i.e., 6000-5*2) */
printf("\n5-p = %u", 5-p);   /* Error*/
...
printf("\nq = %u", q);       /* Output: q = 2000 */
printf("\n--q = %u", --q);   /* Output: --q  = 1996 (i.e., 2000-1*4) */
printf("\nq-5 = %u", q-5);   /* Output: q -5 = 1976 (i.e., 1996-5*4) */
```

3. Subtraction of one pointer from another.

One pointer variable can be subtracted from another pointer variable provided both point to elements of the same array. The result is the number of array elements separating the pointers.

```
int arr[] = {10, 20, 30, 40, 50};/* Let the base address of the
                                    array is 5000 */
...
int *p, *q;
p = &arr[1];
q = &arr[4];
...
printf("q-p = %d", q-p); /* Output: q -p = 3*/
```



4. Comparison of two pointer variables using the relational operators (<, <=, >, >=, ==, !=).

Comparison of two pointer variables is effective (makes sense) provided both points to the variables of same data type (usually the same array). However, any comparison of pointers that points to separate or unrelated variables makes no sense.

```
int arr[] = {10, 20, 30, 40, 50};/* Let the base address of the
...                                  array is 5000 */
int *p, *q;
p = &arr[1];
q = &arr[4];
...
printf("\nStatus(p<q) = %d", p<q);    /* Output: Status(p<q) = 1*/
printf("\nStatus(p<=q) = %d", p<=q); /* Output: Status(p<=q) = 1*/
printf("\nStatus(p>q) = %d", p>q);    /* Output: Status(p>q) = 0*/
printf("\nStatus(p>=q) = %d", p>=q); /* Output: Status(p>=q) = 0*/
printf("\nStatus(p==q) = %d", p==q); /* Output: Status(p==q) = 0*/
printf("\nStatus(p!=q) = %d", p!=q); /* Output: Status(p!=q) = 1*/
```

- **The following operations are NOT allowed on pointers. They will produce compilation error.**
    1. **Addition of two pointers** (e.g., p1+p2)
    2. **Multiplication of two pointers** (e.g. p1*p2)
    3. **Multiplication of a pointer with a constant** (e.g. p1*3)
    4. **Division of two pointers** (e.g. p1/p2)
    5. **Division of a pointer with a constant** (e.g. p1/3)

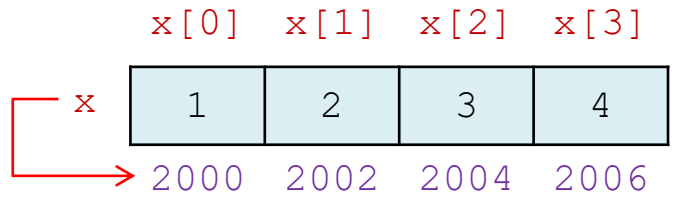
Pointers & Arrays

Pointers & One-Dimensional Array

- **Pointers and 1D arrays are related in a vary interesting way:**
  - **When an 1D array is declared, the compiler defines the array name as a constant pointer to the 1st element of the array i.e., the name of the array corresponds to the address of the 1st element (the base address).**

    For Example:

    When we declare an 1D array 'x' as: $\text{int x}[4] = \{1, 2, 3, 4\};$



Then, x = The base address = Address of the 1st element (i.e., &x[0])

- **Now, let us consider the expression:** x+1

  It will result in 2002. **It is because, the value of** x = 2000 **(the base address of the array) and when ever a pointer is incremented, it is incremented to the number of bytes taken by its data type** (here it is `int`).

  It can be observed that 2002 **is the address of** x[1].

  i.e., x+1 = &x[1].

  Similarly, x+2 = &x[2].

  And so on…

  In general, x+i = i+x = Address of the element at index 'i' (i.e., &x[i])

- **Now, since** x+i = &x[i]

  ⇒ *(x+i) = *(&x[i]) = x[i]

  So, *(x+i) = *(i+x)= The element at index 'i' (i.e., x[i])

  In fact, this is what the C compiler does internally. When we say x[i], the C compiler internally converts it to *(x+i).

- **Let us summaries what we have learnt.**

For a 1D array 'x' declared as: int x[4];

- x
  = The base address
  = Address of the 1st element (i.e., &x[0])
- (x+i) = (i+x)
  = Address of the element at index 'i' (i.e., &x[i])
- *(x+i) = *(i+x)
  = The element at index 'i' (i.e., x[i] or, [i]x)

Programming Example:

```
/* PR7_3.c: A program using pointers to compute the sum of all elements stored in a 1D array */

# include <stdio.h>

void main()
{
    int x[] = {12,3, 9, 89, 11};
    int i, sum = 0;

    printf("\nElement\t Address\t Value\n");
    printf("-------\t -------\t -----");
    for(i=0;i<5;i++)
    {
        printf("\nx[%d]\t %u\t %d", i, (x+i), *(x+i));
        sum = sum + *(x+i);
    }
    printf("\n\nSum = %d", sum);
    getch();
}
```

Output

```
Element  Address  Value
-------  -------  -----
x[0]     2000     12
x[1]     2002     3
x[2]     2004     9
x[3]     2006     89
x[4]     2008     11

Sum = 124
```

- **[A Note]: Which method should one use, accessing the array elements by using pointers (like *(x+i)), or accessing them by using subscripts (like x[i])?**

  The answer is, accessing the array elements by using pointers is always faster than accessing them by using subscripts. However, from the programmers' point of view accessing the array elements by using subscripts seems to be more convenient (easy to understand). So, it depends…

Pointers & Two-Dimensional Array

- **The Relationship: Like the 1D arrays, pointers can also be used to manipulate 2D arrays. Following is the relationship:**

  For a 2D array 'x' declared as: int x[3][2];

  - x = *x = *(x +0)= x[0]
    = Base address of the array
    = Starting address of the '0th' 1D array
    = Address of the element at row '0' and column '0' (&x[0][0])
  - (x+i)= *(x+i)= x[i]
    = Starting address of the 'ith' 1D array
    = Address of the element at row 'i' and column '0' (&x[i][0])
  - *(x+i)+j = x[i]+j
    = Address of the element at row 'i' and column 'j' (&x[i][j])
  - *(*(x+i)+j) = x[i][j]
    = The element at row 'i' and column 'j' (x[i][j])

Programming Example:

```
/* PR7_4.c: Manipulation of 2D array using pointers */

# include <stdio.h>

void main()
{
    int x[3][2] = {1, 2, 3, 4, 5, 6};
    int i, j;

    printf("\nBase address: %u, %u, %u, %u\n", x, *x, x[0], &x[0][0]);
    for(i=0;i<3;i++)
    {
        printf("\nStarting address of %dth 1D array:
        %u, %u, %u, %u", i, (x+i), *(x+i), x[i], &x[i][0]);
    }
    printf("\n");
    for(i=0;i<3;i++)
    {
       for(j=0;j<2;j++)
       {
           printf("\nAddress of element at row/column (%d, %d):
           %u, %u, %u", i, j, *(x+i)+j, x[i]+j, &x[i][j]);
       }
    }
```

[Cont.]

```
    printf("\n\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<2;j++)
        {
            printf("\nThe element at row/column (%d, %d):
            %u, %u", i, j, *(*(x+i)+j), x[i][j]);
        }
    }
    getch();
}
```

Output is shown in the next slide.

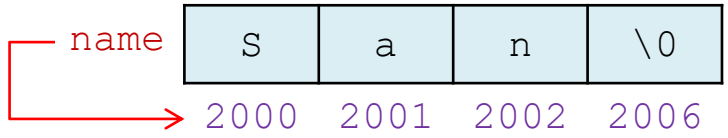
Output

```
Base address: 1016, 1016, 1016, 1016

Starting address of 0th 1D array: 1016, 1016, 1016, 1016
Starting address of 1th 1D array: 1020, 1020, 1020, 1020
Starting address of 2th 1D array: 1024, 1024, 1024, 1024

Address of element at row/column (0, 0): 1016
Address of element at row/column (0, 1): 1018
Address of element at row/column (1, 0): 1020
Address of element at row/column (1, 1): 1022
Address of element at row/column (2, 0): 1024
Address of element at row/column (2, 1): 1028

The element at row/column (0, 0): 1
The element at row/column (0, 1): 2
The element at row/column (1, 0): 3
The element at row/column (1, 1): 4
The element at row/column (2, 0): 5
The element at row/column (2, 1): 6
```

- **The Relationship:** As we have discussed in the chapter "Arrays & Strings", a string is nothing but a *1D array* of characters terminated by a NULL character. Hence, the name of the string corresponds to its base address.

  For Example: If we declare a string as: char name[] = "San";

  Then, name = the base address



  This is the reason why we are not preceding a '&' before the name of the string while scanning through `scanf()`.

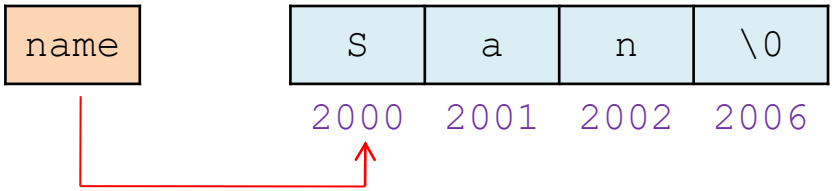
```
scanf("%s", name);
```

- **One More Thing: C supports an alternative method to create string using pointers to character.**

For Example:

```
char *name = "San";
```

It creates a string "San" **and stores its base address in the pointer** "name".



The effect is same as declaring the above string like char name[] = "San";

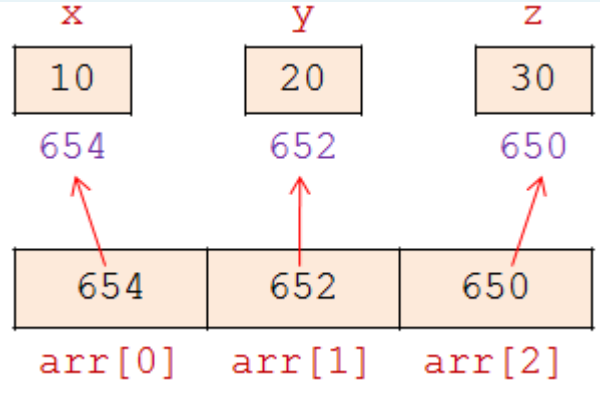
Array of Pointers

- **The Concept: The way there can be array of integers or array of floats, similarly there can be array of pointers.**
  - **Since a pointer variable always contains an address, an array of pointers would nothing but a collection of addresses. These addresses can be addresses of isolated variables or addresses of elements of another array, or any other addresses.**
  - **All the rules that apply to an ordinary array apply to an array of pointers.**

The program given in the next slide will clarify the concept

Programming Example:

```
/* PR7_5.c: Demonstration of array of pointers */

# include <stdio.h>

void main()
{
    int *arr[3]; /* Array of integer pointers */
    int i, x=10, y=20, z=30;

    arr[0] = &x;
    arr[1] = &y;
    arr[2] = &z;

    for(i=0;i<3;i++)
        printf("\narr[%d] = %d", i, *(arr[i]));

    getch();
}
```

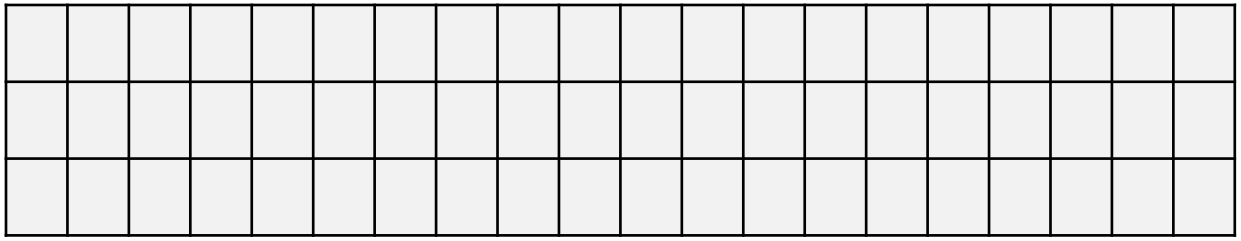


Output

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
```

- **Practical Use:** Array of pointers are practically useful in handling an array of strings.
  - Consider the following declaration: `char name [3][20];`

    This says that the `name` is a table containing three names, each with the max. length of 20 characters (including the \0). Hence, the total storage requirements are 60 bytes.

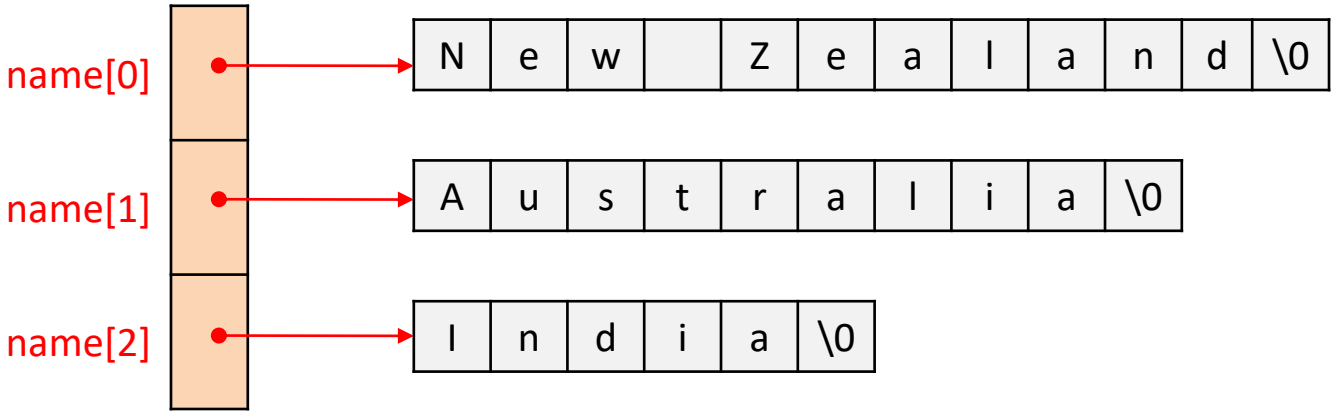


  - However, we know that rarely the names could be the equal length. So, there will be a wastage of some storage space.

- **This situation could be handled by using an array of pointers (to characters), as shown below:**

```
char *name[3] = {
                    "New Zealand",
                    "Australia",
                    "India"
                }
```

Here, `name` is an *array of three pointers* to characters, each pointer pointing to a particular name.



The following statements would print all the three names:

```
for(i=0;i<3;i++)
   printf("\n%s", name[i]);
```

To access the 'j^{th}' character in the 'i^{th}' name, we may use the expression:

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

- **NOTES:**
  - An array of strings with different length are called 'ragged arrays'.
  - It is always advised to deal with a table of strings (whether of fixed or variable length) by using the above approach (i.e., array of pointers to characters), because it is easy and saves space.

Assignments

Complete the experiments given in "Lab Manual - Section 9".

End of Chapter 7