VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

BCSE102L –Structured and Object oriented Programming (Theory)

Dr.R.Priyadarshini

Module III : Pointers

Declaration and Access of pointer Variables, Pointer arithmetic- Dynamic memory allocation-Pointers and arrays –Pointer and functions

Pointers in C

Pointers

What is Pointers?

Pointer is a user defined data type which creates special types of variables which can hold the address of primitive data

Why should I use Pointers?

- Increase the execution speed
- Enable us to access a variable that is defined outside the function
- More efficient in handling the data tables
- Reduce the length and complexity of a program

Pointers

What are the operators used?

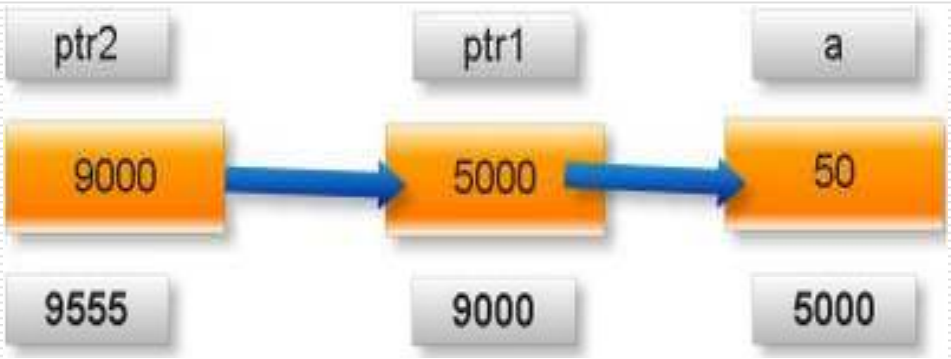
*, &

What are the advantages of using pointers?

- Dynamic memory allocation is possible with pointers.
- Passing arrays and structures to functions
- Passing addresses to functions.
- Creating data structures such as trees, linked lists etc.

Pointers Assignment

```
int a=50;
int *ptr1;
int **ptr2;
ptr1=&a;
ptr2=&pt1;
```




```
void main(){
int x=25;
int *ptr=&x; //statement one
int **temp=&ptr; //statement two
printf("%d %d %d",x,*ptr,**temp);
}
```

Pointers Assignment

```
#include <stdio.h>
int main ()
{
   char ch = ‘a’;
   char* p1, *p2;
   p1 = &ch;
   p2 = p1; // Pointer Assignement Taking Place
   printf (" *p1 = %c And *p2 = %c", *p1,*p2);
   return 0;
}
```

Ans: *p1=a And *p2=a

Pointers Conversion

```
#include <stdio.h>
 int main ()
 {
   int i = 67;
   char* p1
   int *p2;
   p2 = &i;
   p1 = (char *) p2; // Type Casting and Pointer Conversion
   printf (" *p1 = %c And *p2 = %d", *p1,*p2);
   return 0;
 }
```

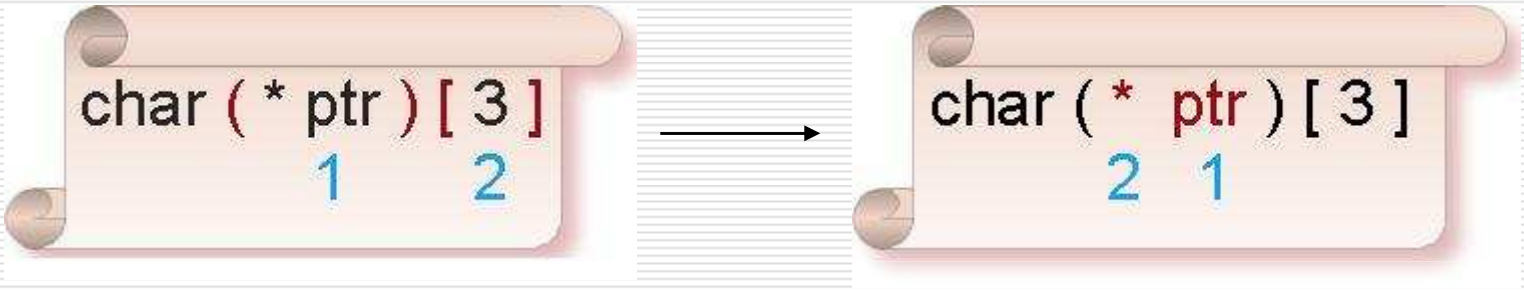
Ans: *p1 = C And *p2 = 67

Pointers

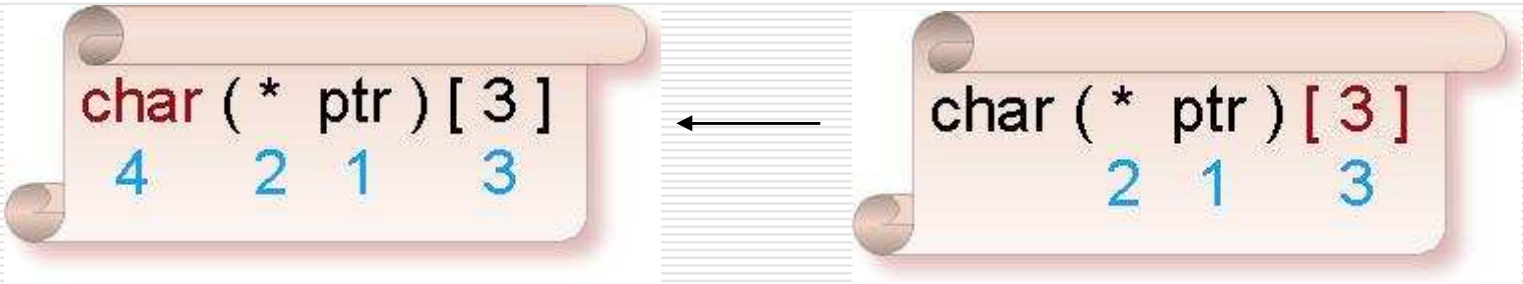
Opertor Precendence and Associativity

Operator	Precedance	Associative
( ) , [ ]	1	Left to right
* , Identifier	2	Right to left
Data type	3	

Reading Pointers



ptr is **pointer** to such one dimensional **array** of size three which content **char** type data



Reading Pointers

1. float (* ptr) (int)
2. void (*ptr) (int (*)[2],int (*) void))
3. int (* (* ptr) [5]) ( )

Arithmetic operation with pointer

Address + Number= Address

Address - Number= Address

Address++ = Address

Address-- = Address

++Address = Address

--Address = Address

Address – Address=Number

```
void main(){
int *ptr=( int *)1000;
ptr=ptr+1;
printf(" %u",ptr);
}
```

Output: 1002

Pointers to function

```
int * function();
void main(){
auto int *x;
int *(*ptr)();
ptr=&function;
x=(*ptr)();
printf ("%d",*x);
}
```

```
int *function(){
static int a=10;
return &a;
}
```

Output: 10

Explanation:

Here function is function whose parameter is void data type and return type is pointer to int data type.

x=(*ptr)()

=> x=(*&function)() //ptr=&function

=> x=function() //From rule *&p=p

=> x=&a

So, ***x** = *&a = a =10

Pointers Supports

- Pointer to array of function
- Pointer to array of string
- Pointer to structure
- pointer to union
- Multi level pointer
- Pointer to array of pointer to string
- Pointer to three dimentional array
- Pointer to two dimensional array
- Sorting of array using pointer
- Pointer to array of array
- Pointer to array of union
- Pointer to array of structure
- Pointer to array of character
- Pointer to array of integer
- Complex pointer

Functions Returning Pointers

```
char display (char (*)[])
void main(){
char c;
char character[]={65,66,67,68};
char (*ptr)[]=&character;
c=display (ptr);
printf ("%c", c);

}

char display (char (*s)[])
{
   **s+=2;
   return **s;
 }
```

Output: C

Explanation: Here function display is passing pointer to array of characters and returning **char** data type.

****s+=2**

=>**s=**s+2

=>**ptr=**ptr+2 //s=ptr

=>**&character= **&character+2 //ptr=&character

=>*character=*character+2 //from rule *&p =p

=>character[0]=character[0]+2 //from rule *(p+i)=p[i]

=>character [0] =67

**s=character [0] =67

Question 1.

```
#include<stdio.h>
 void main()
{
  int a = 320;
  char *ptr;
  ptr =( char *)&a;
  printf("%d ",*ptr);
  getch();
}
```

(A) 2 (B) 320 (C) 64 (D) Compilation error

(E) None of above

Question 1. Explanation

As we know int is two byte data byte while char is one byte data byte. char pointer can keep the address one byte at time.

Binary value of 320 is 00000001 01000000 (In 16 bit)

Memory representation of int a = 320 is:

So ptr is pointing only first 8 bit which color is green and Decimal value is 64.

Question 2.

```
#include<stdio.h>
#include<conio.h>
 void main(){
   void (*p)();
   int (*q)();
   int (*r)();
   p = clrscr;
   q = getch;
   r = puts;
  (*p)();
  (*r)(" Department of Computer Applications");
  (*q)("Error");
}
```

(A) NULL
(B) Department of Computer Applications
(C) Error (D) Compilation error
(E) None of above

Question 2. Explanation

p is pointer to function whose parameter is void and return type is also void.

r and q is pointer to function whose parameter is void and return type is int .

So they can hold the address of such function.

Question 3.?

```
#include<stdio.h>
void main(){
  int i = 3;
  int *j;
  int **k;
  j=&i;
  k=&j;
  printf("%u %u %d ",k,*k,**k);
}
```

(A) Address, Address, 3 (B) Address, 3, 3 (C) 3, 3,3
(D) Compilation error (E) None of above

Question 3. Explanation

Memory representation

Here 6024, 8085, 9091 is any arbitrary address, it may be different.
Value of k is content of k in memory which is 8085
Value of *k means content of memory location which address k keeps.
k keeps address 8085 .
Content of at memory location 8085 is 6024
In the same way **k will equal to 3.
Short cut way to calculate:
Rule: * and & always cancel to each other
i.e. *&a = a
So *k = *(&j) since k = &j
*&j = j = 6024
And
**k = **(&j) = *(*&j) = *j = *(&i) = *&i = i = 3

Question 4?

```
#include<stdio.h>
#include<string.h>
void main(){
char *ptr1 = NULL;
char *ptr2 = 0;
strcpy(ptr1," c");
strcpy(ptr2,"questions");
printf("\n%s %s",ptr1,ptr2);
getch();
}
```

(A) c questions (B) c (null)

(C) (null) (null)

(D) Compilation error(E) None of above

Question 5?

```
#include<stdio.h>
#include<string.h>
void main(){
register a = 25;
int far *p;
p=&a;
printf("%d ",*p);
getch();
}
```

(A) 25 (B) 4 (C) Address

(D) Compilation error

(E) None of above

Question 5

Explanation:

Register data type stores in CPU. So it has not any memory address. Hence we cannot write &a.

Question 6?

```
#include<stdio.h>
#include<string.h>
void main(){
int a = 5,b = 10,c;
int *p = &a,*q = &b;
c = p - q;
printf("%d" , c);
getch();
}
```

Difference of two same type of pointer is always one

(A) 1 (B) 5 (C) -5
(D) Compilation error
(E) None of above

Question 7?

```
#include<stdio.h>
 int main()
{
 int i=3, *j, k;
 j = &i;
 printf("%d\n", i**j*i+*j);
 return 0;
 }
```

A.	30	**B.**	27
C.	9	**D.**	3

Questions 8.?

```
#include<stdio.h>
 int main()
{
 int ***r, **q, *p, i=8;
 p = &i; q = &p; r = &q;
 printf("%d, %d, %d\n", *p, **q, ***r);
 return 0;
 }
```

A. 8,8,8 B. 4000, 4002, 4004

C. 4000, 4004, 4008 D. 4000, 4008, 4016

Questions 9.?

```
#include<stdio.h>
int main()
{
    void *vp;
    char ch=74, *cp="JACK";
    int j=65;
    vp=&ch;
    printf("%c", *(char*)vp);
    vp=&j;
    printf("%c", *(int*)vp);
    vp=cp;
    printf("%s", (char*)vp+2);
    return 0;
}
```

A. JCK B. J65K C. JAK D. JACK

Questions 10.?

```
#include<stdio.h>
 int main()
 {
  int arr[2][2][2] = {10, 2, 3, 4, 5, 6, 7, 8};
  int *p, *q;
  p = &arr[1][1][1];
  q = (int*) arr;
  printf("%d, %d\n", *p, *q);
  return 0;
 }
```

A. 8 10 B. 10 2 C. 8 1 D. Garbage value

Question 11.?

```
#include<stdio.h>
 int main()
{
 char str[] = "peace";
 char *s = str;
 printf("%s\n", s++ +3);
 return 0;
 }
```

A. Peace B. eace C. ace D. ce

Question 12.?

```
#include<stdio.h>
 int main()
 {
   char str1[] = "India";
   char str2[] = "BIX";
   char *s1 = str1, *s2=str2;
   while(*s1++ = *s2++)
    printf ("%s", str1);
    printf("\n");
  return 0;
  }
```

A. IndiaBIX B. BndiaBIdiaBIXia

C. India D. (null)

*ptr++ Vs ++*ptr

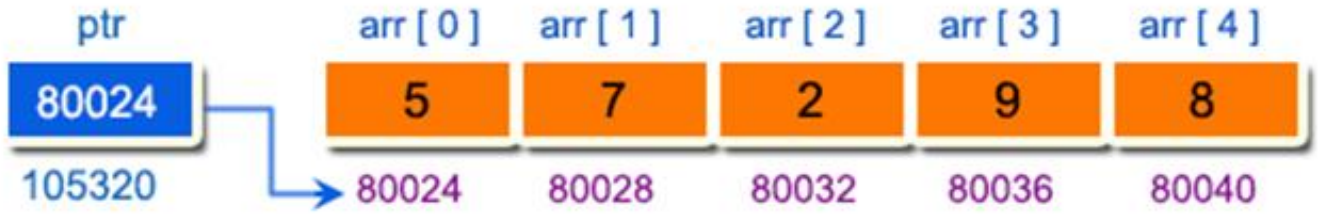
```
#include <stdio.h>
#include <conio.h>
void main()
{
 int *a;
 int b=10; clrscr();
 a=&b;
 printf("%u\n",a);
 *a++;
 printf("%u\n",a);
 ++*a;
 printf("%d",*a);
 printf("%d",b);
 getch();
}
```

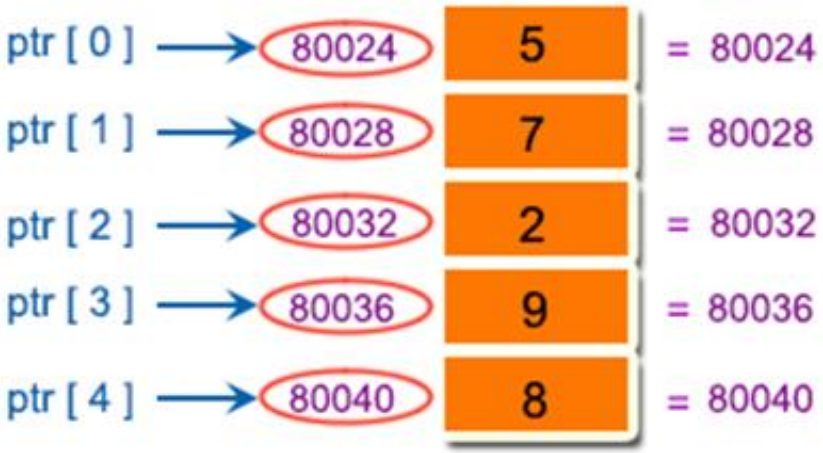
```
65524  65526  1 10
```

20 var ip=&var 1001				
1101	Int b=9; b 1105			1001 int *ip 1118

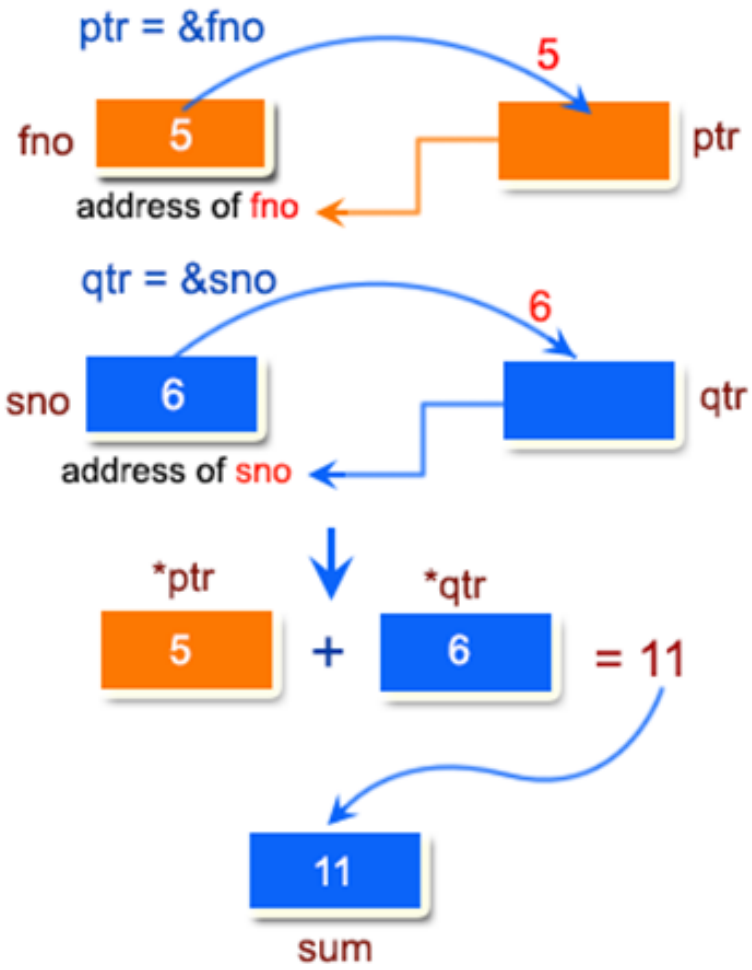
int i ; int *ptr; ptr=i;

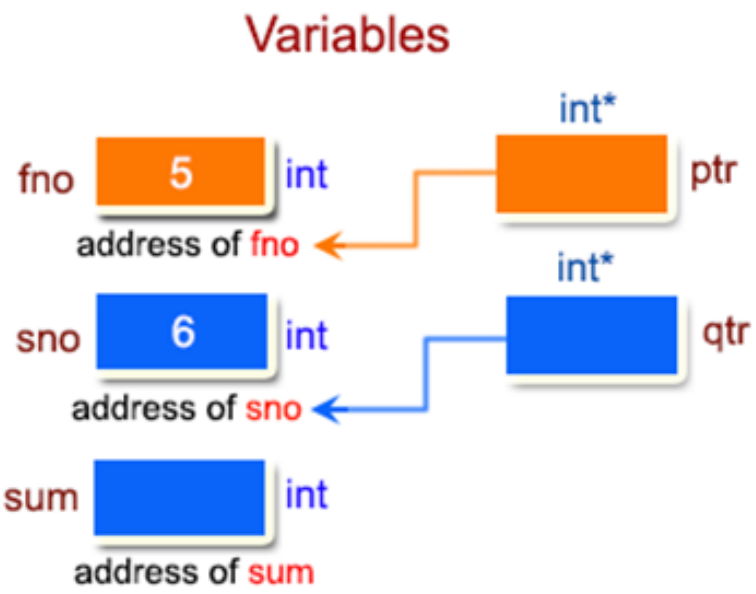
1002-i
1004
1006
1008
1010
1012

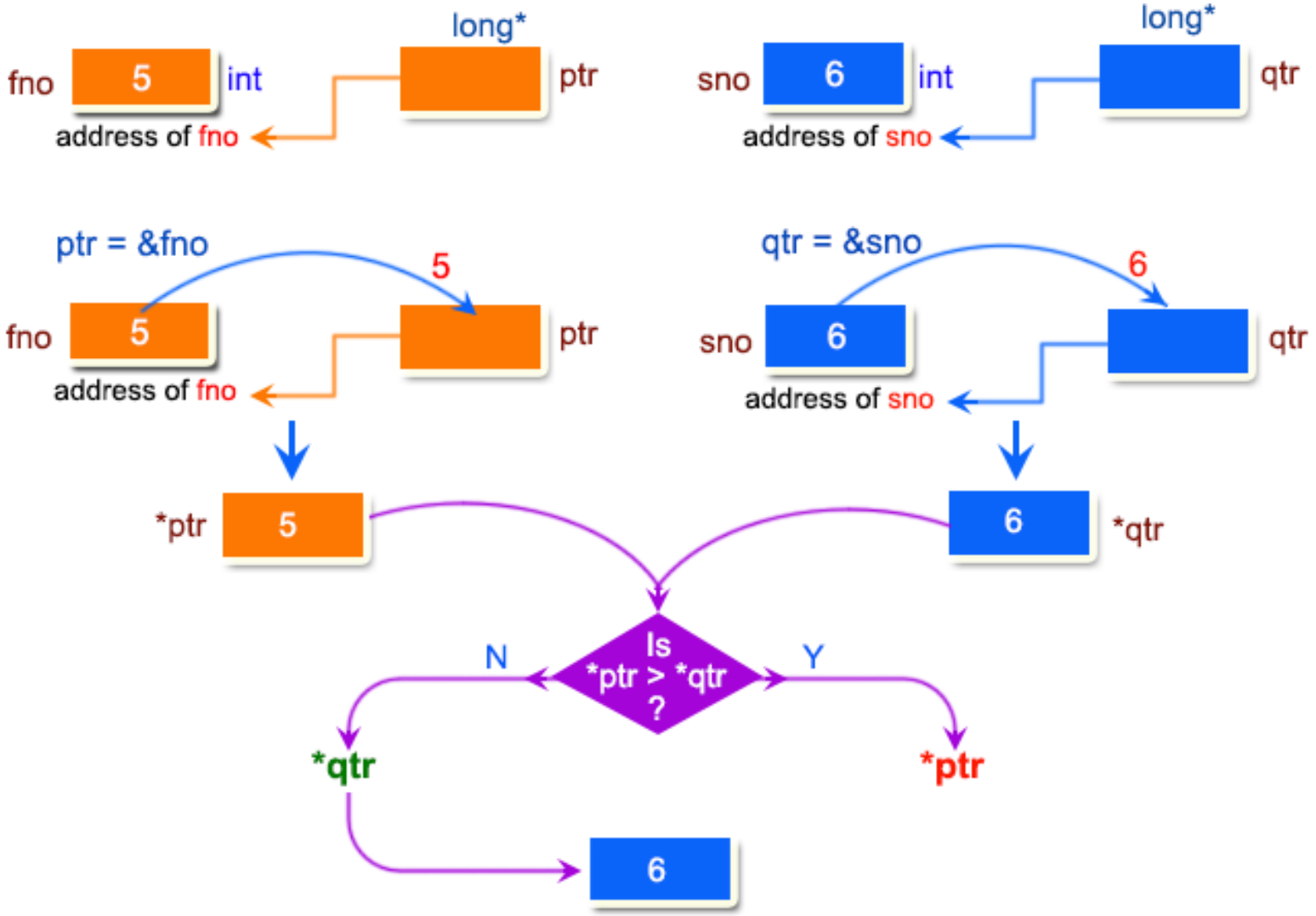

ptr
80024
105320
arr [0]
arr [1]
arr [2]
arr [3]
arr [4]
5
7
2
9
8
80024
80028
80032
80036
80040

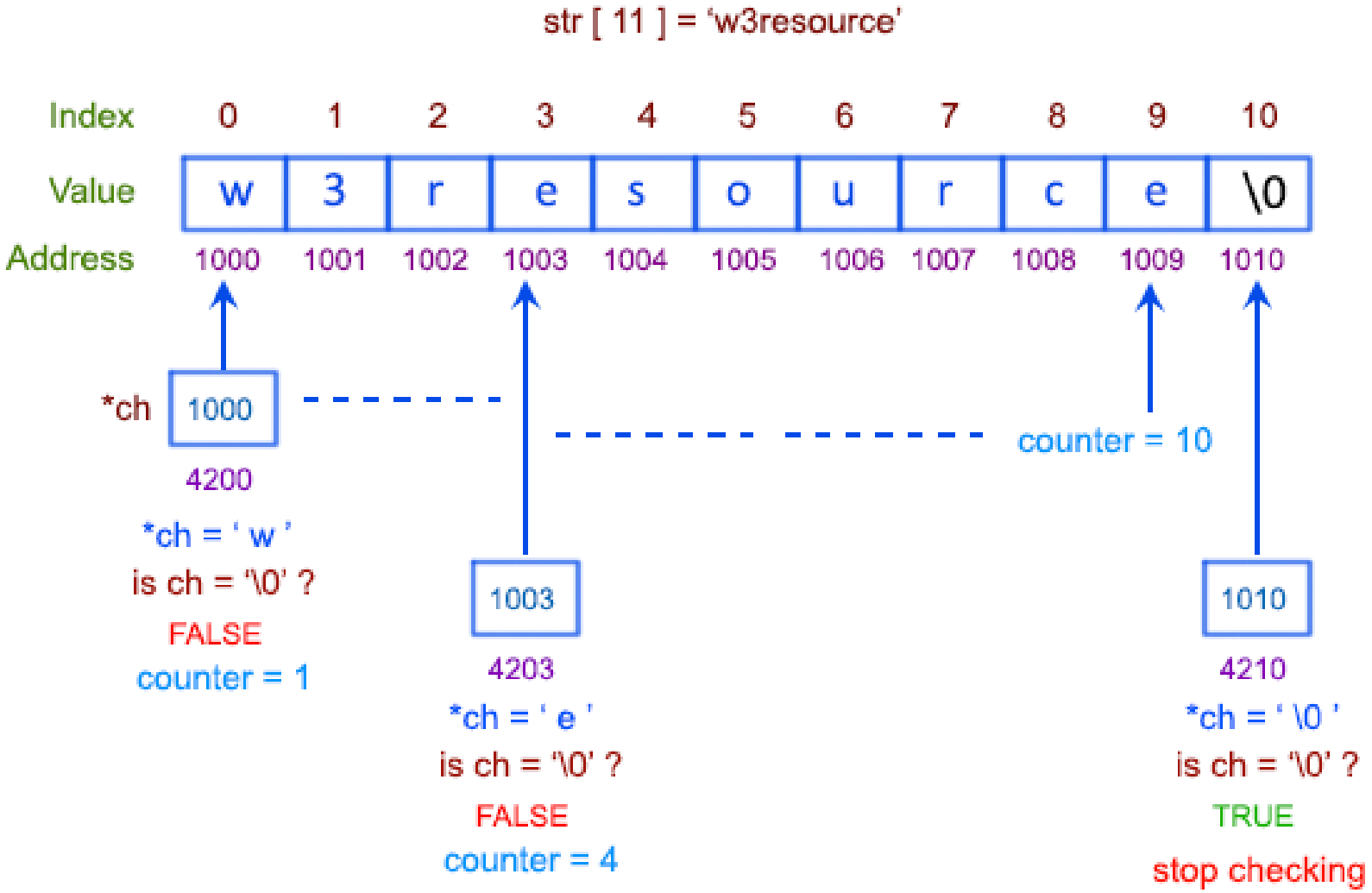

ptr [0]
80024
5
= 80024
ptr [1]
80028
7
= 80028
ptr [2]
80032
2
= 80032
ptr [3]
80036
9
= 80036
ptr [4]
80040
8
= 80040


*ptr [0]
80024
5
= 5
*ptr [1]
80028
7
= 7
*ptr [2]
80032
2
= 2
*ptr [3]
80036
9
= 9
*ptr [4]
80040
8
= 8


ptr = &fno
5
fno
5
ptr
address of fno
qtr = &sno
6
sno
6
qtr
address of sno
*ptr
*qtr
5
+
6
= 11
11
sum


Variables
int*
fno
5
int
ptr
address of fno
int*
sno
6
int
qtr
address of sno
sum
int
address of sum


long*
fno
5
int
ptr
address of fno
long*
sno
6
int
qtr
address of sno
ptr = &fno
5
fno
5
ptr
address of fno
qtr = &sno
6
sno
6
qtr
address of sno
*ptr
5
6
*qtr
Is
*ptr > *qtr
?
N
Y
*qtr
*ptr
6


str [11] = 'w3resource'
Index 0 1 2 3 4 5 6 7 8 9 10
Value w 3 r e s o u r c e \0
Address 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010
*ch
1000
4200
*ch = ' w '
is ch = '\0' ?
FALSE
counter = 1
1003
4203
*ch = ' e '
is ch = '\0' ?
FALSE
counter = 4
counter = 10
1010
4210
*ch = ' \0 '
is ch = '\0' ?
TRUE
stop checking

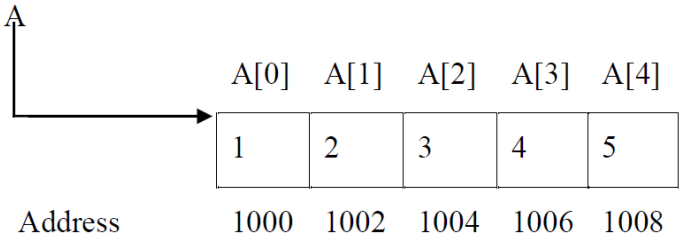
Pointers and Arrays

- We are already familiar with **arrays**, which **store multiple values** of the **same data type**.
- Array elements are stored in **contiguous memory locations.**
- When an array is declared, **compiler allocates required amount of contiguous memory locations** to hold all the elements of the array.
- Memory is allocated by the compiler **during compilation time itself.**
- **Base address** of the array is the **address of the first element** of the array.
- We can create a **pointer** which can be used **to point an array**.
- In an **one dimensional array,** we can use **single subscript** to access the array elements.
- **Name of the 1-D array** or **&array[0]** points to the **address of the first element** of an array.

Pointers and 1D Arrays

- Suppose we declare an **integer** array "**A**" with **5 elements** and assuming **that the base address** of the array a is **1000** and each integer element requires **two bytes** of memory.
- The five elements will be stored as follows:

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



- Here array name "**A**" acts as a **constant pointer** which will give the **base address of the array** and **points to the first element of the array, A[0].**
  - i.e., the value of **A** and **&A[0]** are **same**.

Pointers and 1D Arrays

- We can declare another pointer of type **int** to point to the array **A**.

```
int *p;
p = A;    /* (or) p = &A[0];    both the statements are equivalent. */
```

- Now we can **access every element of the array A using pointer p.**
- **(*p)** is used to access the element **A[0]** which has the **value 1** in the given example.
- To access the **next element in the array A**,
  - increment the pointer p by 1 using **(p++)** or **(++p).**
  - Now we can give ***p** to **get A[1].**
  - It gives the **value 2**.
  - To get the next element in the array A, increment the pointer p by 1 and use *p.

Example 1- Program to demonstrates the use of pointer to access the array elements

```
#include <stdio.h>
int main()
{
        int i;
        int A[5] = {1, 2, 3, 4, 5};
        int *p =A ;     /* Assign starting address to pointer p. Equivalent to  int *p = &a[0];  */
        for (i = 0; i < 5; i++)
        {
           printf("%d  ", *p);
           p++;                          /* Move the pointer to next element in the array  */
        }
        return 0;
}
```

Output:

1 2 3 4 5

Example 2- use array name as a pointer to access the array elements

```
#include <stdio.h>
int main()
{
        int i;
        int A[5] = {1, 2, 3, 4, 5};
        for (i = 0; i < 5; i++)
        {
           printf("%d ", *(A+i));
        }
        return 0;
}
```

- In the above program, **A** is a **constant pointer** which **always points** to **first element A[0]**.
- We **cannot move this pointer to next element using ++ operator** since **base address of the array cannot be changed.**
- To access the elements of the array using a constant pointer A, we can use the following statements:

A gives the address of A[0].	*A or *(A+0) gives the first element 1. i.e., A[0]
A+1 gives the address of A[1].	*(A+1) gives the second element 2 . i.e., A[1]
A+2 gives the address of A[2].	*(A+2) gives the third element 3 . i.e., A[2]
A+3 gives the address of A[3].	*(A+3) gives the fourth element 4 . i.e., A[3]
A+4 gives the address of A[4].	*(A+4) gives the fifth element 5 . i.e., A[4]

In general, to access the i^{th} element of the array, we can use **A[i] or *(A+i)**.

Example 3- Program to display the elements of an array in reverse order using pointer.

```
#include <stdio.h>
int main ()
{
        int  ar[] = {10,20,30,40,50};
        int  i, *ptr;
        /*store the address of last element of the array to pointer*/
        ptr = &ar[4];
        for ( i = 0; i < 5; i++)
        {
                printf("%d ", *ptr );
                /* move ptr to the previous location */
                ptr--;
        }
        return 0;
}
```

Output:

50 40 30 20 10

Example 4 – Program to add the elements of an array using pointer

```
#include <stdio.h>
int main ()
{
        int  ar[] = {10,20,30,40,50};
        int  i, *ptr,sum=0;
        /*store the starting address of the array into pointer*/

         ptr = ar;   /* or ptr=&ar[0]; */
          for ( i = 0; i < 5; i++)
          {
                  sum=sum + *ptr ;
                  ptr++;    /* (or)          ++ptr;  To move ptr to the next location */
          }
          printf("Sum = %d" ,sum);
          return 0;
 }
```

Output:

Sum = 150

Example 5 – Program to demonstrate increment and decrement operators with pointers and arrays.

```
#include <stdio.h>
int  main()
{
        int a[] = {1, 5, 9}, x, y, z;
        int *p = a; /* p points to a[0] */
        ++p;     /* p points to a[1] */
        --p;     /* p points to a[0] again */
        x = ++*p;       /* a[0] is incremented by 1. x=a[0]=2, p still  points to a[0] */
        y = *p++;       /* y=*p which is a[0], Next p  is incremented  which points to a[1] */
        z = *++p;       /* p is points to a[2], z=p[2] i.e. 9 */
        (*p)++;         /* a[2] is incremented by 1 ie., 10, p still points to a[2) */
        ++(*p);         /* a[2] is incremented by 1 ie., 11, p still points to a[2) */
        p--;            /* p points to a[1] */
        printf("a[0] = %d, a[1] = %d, a[2] =%d\n", a[0],a[1],a[2]);
        printf("x = %d,  y = %d, z = %d\n", x, y, z);
        return 0;
}
```

Output:

```
a[0] = 2 ,    a[1] = 5,     a[3] =  11
x = 2,      y = 2,      z = 9
```

Example 6 - Program to copy an array to another array using pointers

```
#include <stdio.h>
void printArray(int *ptr, int size);    /* Function declaration to print array */
int main()
{
        int src[100], dest[100];
        int n, i;
        int *sptr = src;   /* Pointer to src  */
        int *dptr   = dest;     /* Pointer to dest  */
        int *end;
        printf("Enter size of the array: ");
        scanf("%d", &n);
        printf("Enter array: ");
        for (i = 0; i < n; i++)
        {
                scanf("%d", (sptr + i));
        }
        end = &src[n - 1];          /* Pointer to last element of src */
        /* copy source array to destination array */
         while(sptr <= end)
        {
           *dptr = *sptr;
           /* Increment sptr and dptr */
           sptr++;
           dptr++;
        }
        printf("\nDestination array: ");
        printArray(dest, n);
        return 0;
}
```

```
void printArray(int *ptr, int size)
{
        int i;
        for (i = 0; i < size; i++)
        {
                printf("%d  ", *(ptr + i));
        }
}
```

Output:

```
Enter size of the array: 5
Enter array: 1 2 3 4 5
Destination array: 1 2 3 4 5
```

int var – 1002

ip=&var

*ip – Value at that address

DYNAMIC MEMORY ALLOCATION

Disadvantages of ARRAYS

MEMORY ALLOCATION OF ARRAY IS STATIC:
Less resource utilization.

For example:

If the maximum elements declared is 30 for an array, say int test[30], but the user manipulates with only 10 elements, then the space for 20 elements is wasted.

DIFFERENT DATA TYPES COULD NOT BE STORED IN AN ARRAY

- The use of pointer variable instead of arrays, helps in allocation of memory at RUN TIME, thus memory can be allocated or freed anywhere, anytime during the program execution.
    - This is known as DYNAMIC MEMORY ALLOCATION

STATIC vs DYNAMIC

- The allocation of memory for the specific fixed purposes of a program in a predetermined fashion controlled by the compiler is said to be static memory allocation.

- The allocation of memory (and possibly its later de allocation) during the run time of a program and under the control of the program is said to be dynamic memory allocation.

Memory Allocation

- In DYNAMIC memory allocation, the memory is allocated to the variables in the HEAP region.
- The size of HEAP keeps on changing due to creation and death of variables.
- Thus it is possible to encounter “memory overflow” during dynamic memory allocation process.
- In such situation memory allocation library function returns a NULL pointer when they fail to locate enough memory requested.

Memory Allocation Functions

The following are four memory management functions defined in library of C that can be used for allocation and freeing of memory when ever required.

- malloc
- calloc
- free
- realloc

All the above four functions are defined in library file <alloc.h>.

malloc ( )

- malloc : It is a predefined library function that allocates requested size of bytes and returns a pointer to the first byte of the allocated space.

- Pointer returned is of type void that has to be type cast to the type of memory block created (int, char, float, double).

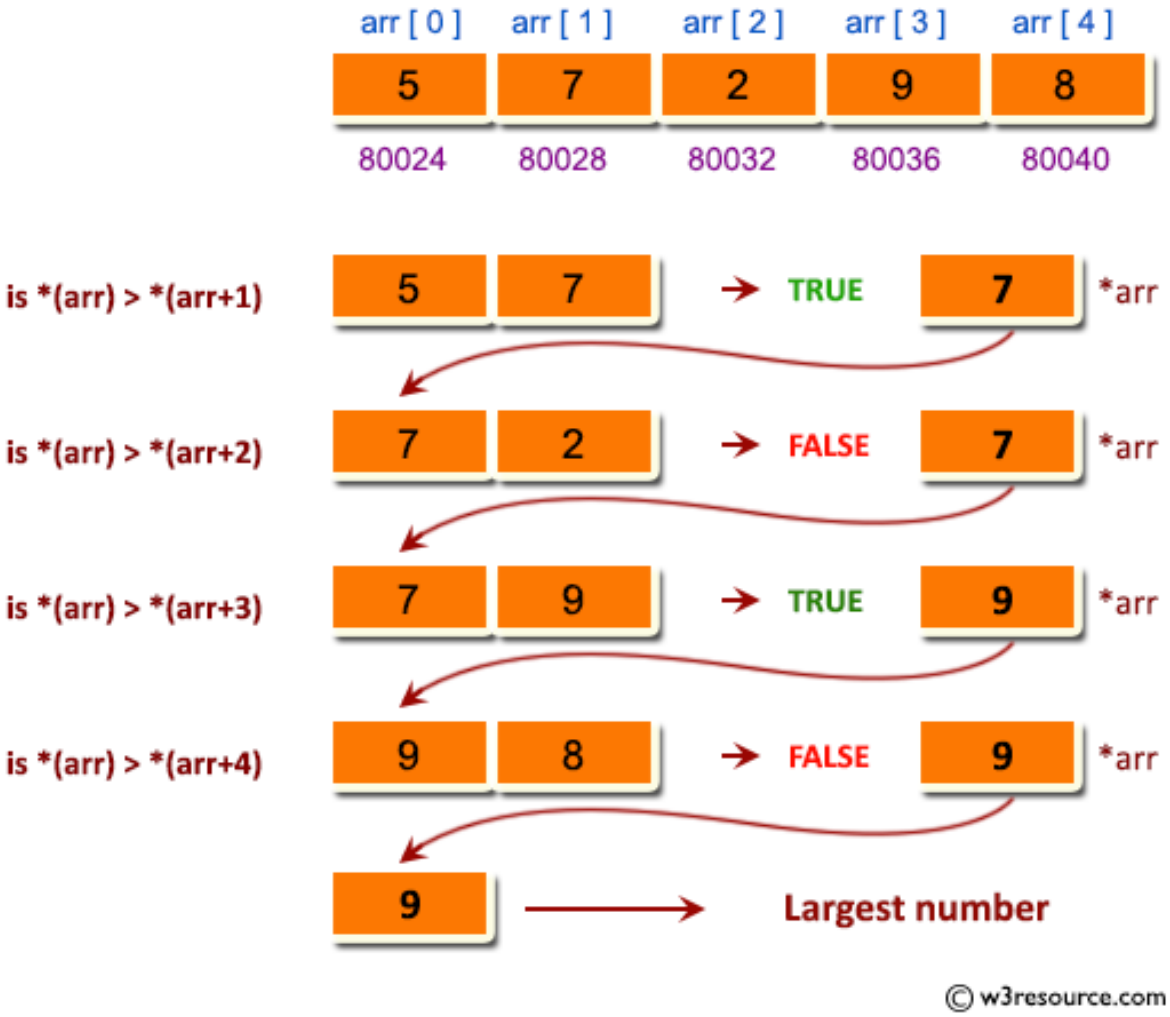
- **General syntax:**

  ptr = (cast –type *) malloc(byte-size) ;

  where ptr is pointer of any data type.

  malloc returns a pointer to an area of memory with size byte-size.

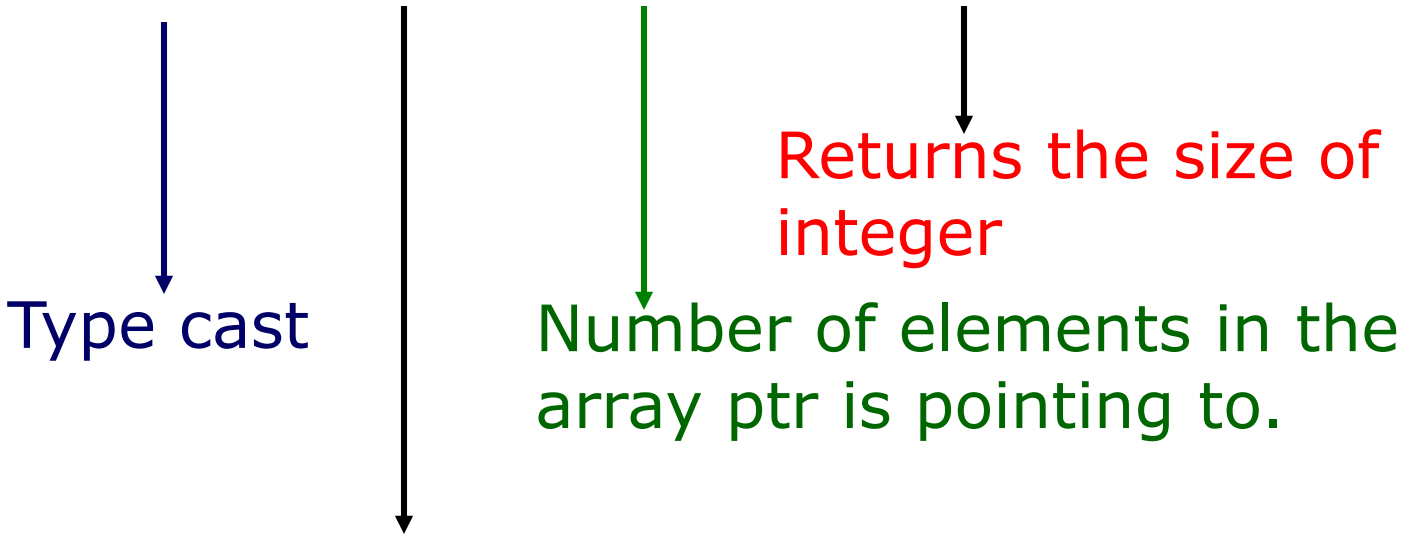
Dynamic Memory Allocation



Example 1

- int *ptr;

```
ptr = (int *) malloc ( 100 * (sizeof (int ) ) ;
```



Allocate 200 bytes i.e declare an array of 100 integers with pointer ptr pointing to the first element of the array

Alternative

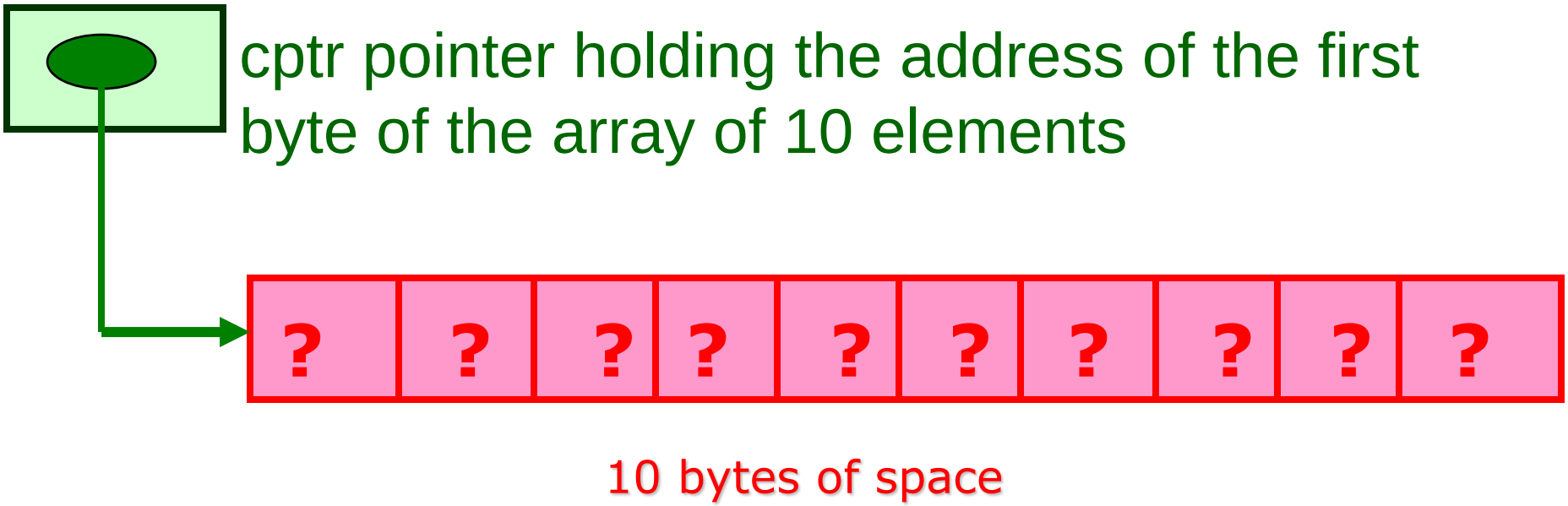
```
int * ptr ;
printf ( “\n Enter the number of elements in the
   array “);
scanf(“%d”, &n);
ptr =  ( int *) malloc ( n x sizeof ( int ) ) ;
```

```
int * ptr ;
ptr =  ( int *) malloc ( 200 ) ;
```

Example 2

```
char * cptr ;
cptr =  ( char * ) malloc ( 10 ) ;
```

This will allocate 10 bytes of memory space for the pointer cptr of type char.



Important

- The storage space allocated dynamically has NO name and therefore its contents are accessed ONLY through a pointer.
- malloc allocates blocks of contiguous memory locations.
- If allocation fails due to in sufficient space on HEAP, it returns NULL pointer. Thus when ever we allocate memory it should be checked whether the memory has assigned successfully or not.

Exercise

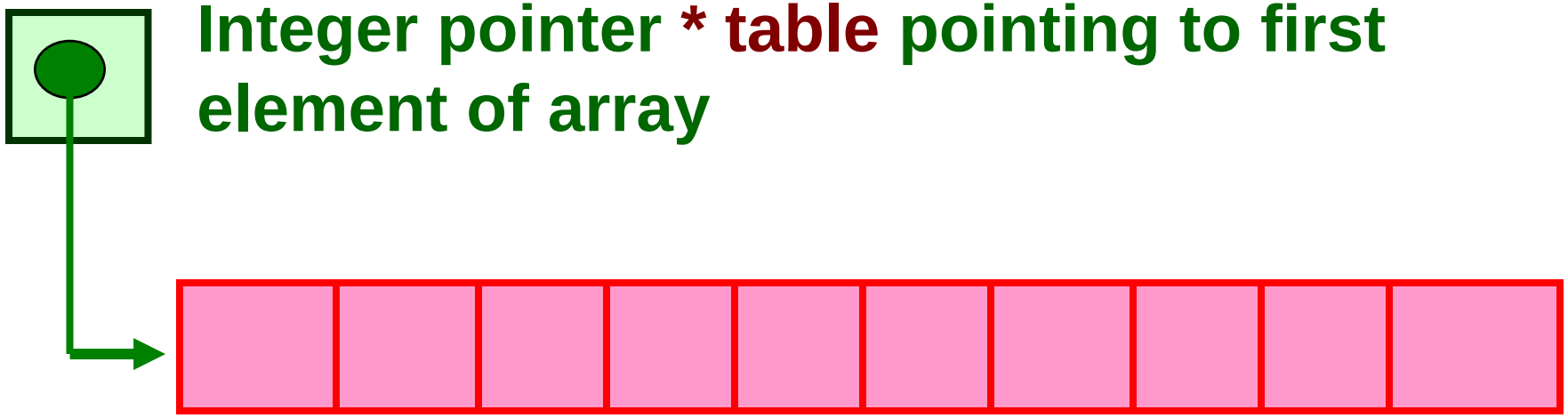
- Write a program to create an array of integers.Size of the array that is number of elements in the array will be specified interactively by the user at RUN TIME.

SOLUTION

```
#include<alloc.h>
int main( ) {
int *p,  *table,  size ;
printf(“\n Enter the size of array “);
scanf(“%d”, &size);
/* memory allocation */
if( table = (int * ) malloc ( size * (sizeof (int) ) ) == NULL )
{     printf(“\n NO space available “);
       exit(1);
}
printf(“\n Address of first byte is %p”,table);
```

Visualization

If size = 10 => 20 bytes will be allocated.

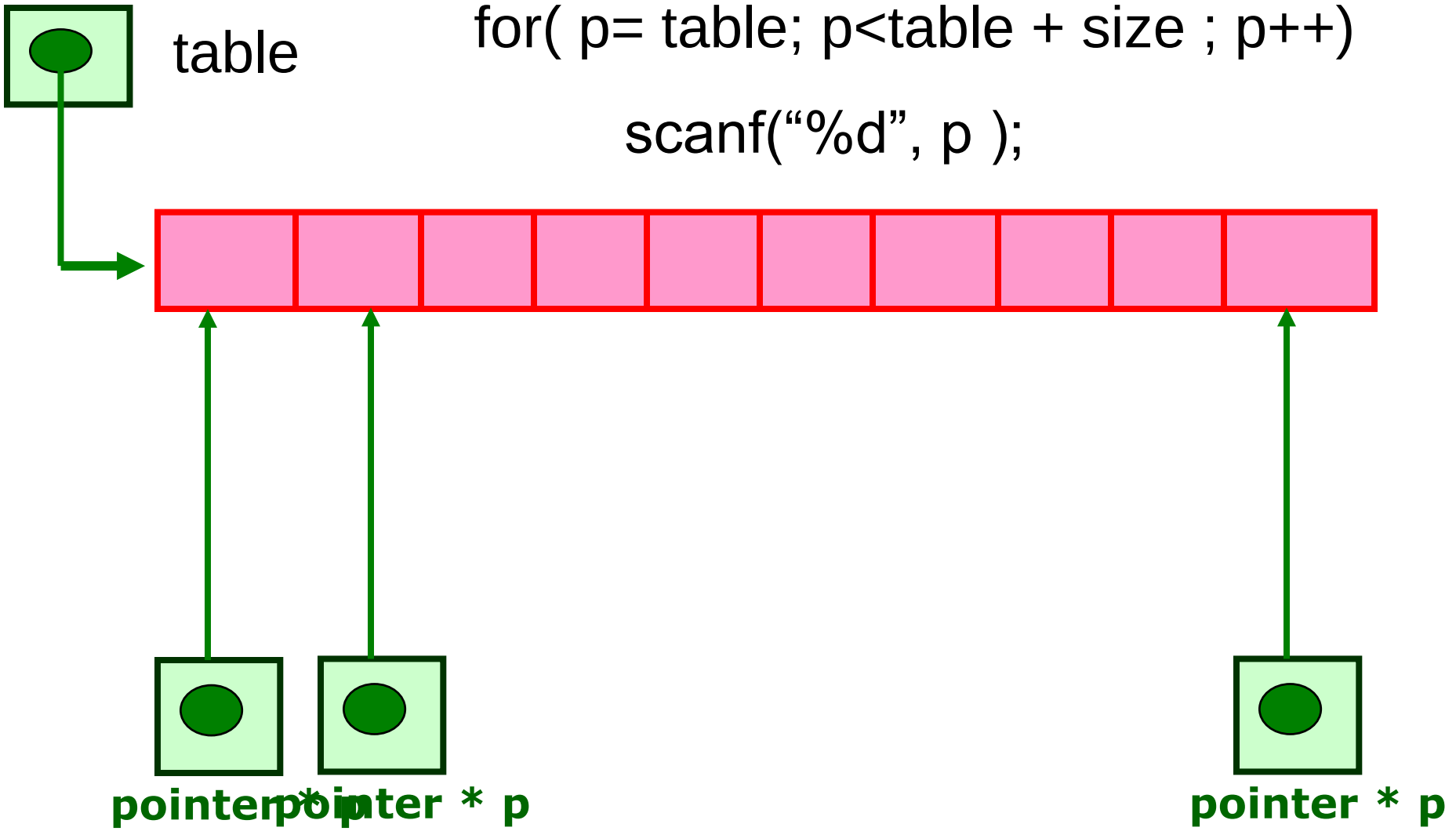


Integer pointer * table pointing to first element of array

SOLUTION cont'd: Reading values in the array

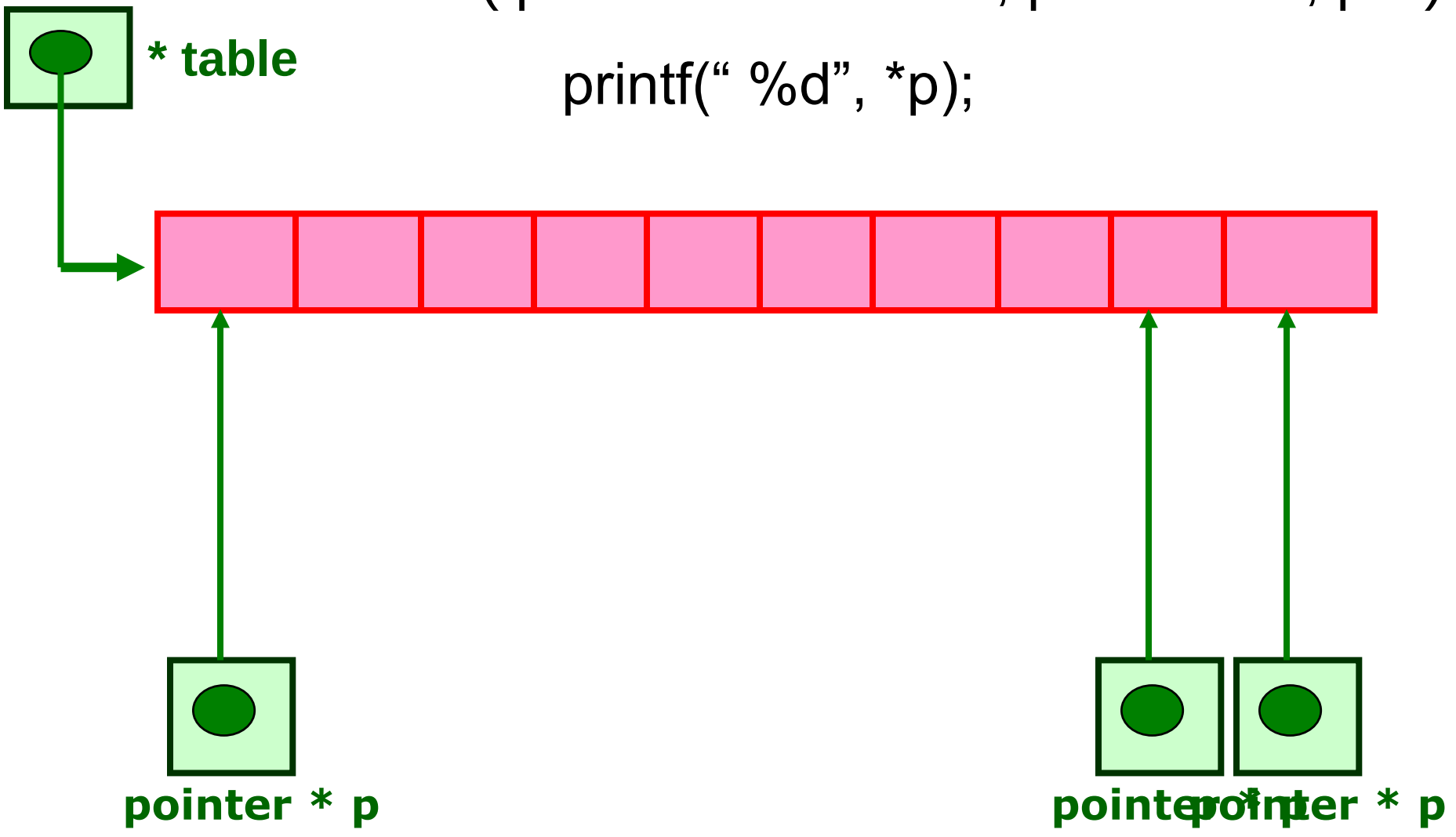
```
// reading values in array
printf("\n Input table value ");
for( p= table; p<table + size ; p++)
  scanf("%d", p );  // no ampersand as p is pointer

// printing values of array in reverse order
for( p = table+size – 1 ; p >= table ; p-- )
   printf("%d", *p);
```

Visualization



Printing values in reverse order

```
for( p = table+size – 1 ; p >= table ; p-- )
    printf(“ %d”, *p);
```



OUTPUT

What is the size of array ? 5
Address of first byte : 077FA

Input values: 11 12 13 14 15

OUTPUT values in reverse order
15
14
13
12
11

calloc ( )

- calloc ( ): It also allocates the requested number of bytes, but the difference is that it takes two parameters:
  - N : number of elements
  - Element_size: size of element in bytes

- Also when the memory is allocated, all the elements are assigned a initial value of zero.
  - This is not provided by function malloc ( )

General syntax:

ptr = (cast _type *) calloc (n, element_size);

- The above statement allocates contiguous space for n elements, each of size element_size.
- All elements are initialized to zero and a pointer to the first byte of the allocated region is returned.
- If there is not enough space, a NULL pointer is returned.

Releasing the Used Space

- In case of Dynamic RUN Time allocation, it is the responsibility of the programmer to release the memory space when it is not required by the program.
- When a block of data in memory is no longer needed by the program, it should be released as memory is a factor in executing complex programs.

free ( )

- To release the memory currently allocated to a pointer using DYNAMIC ALLOCATION use the following function:

```
free ( ptr );
```

where ptr is a pointer to a memory block which has already been created using either malloc or calloc .

Altering the size of memory block

- REALLOCATION function (realloc)

Example

- If the original allocation was done by the statement:

  ptr = (cast_type *) malloc (size) ;

- Then REALLOCATION of space is done by :

  ptr = (cast_type *)realloc (ptr, NEW_SIZE);

How It works ? ? ?

- This function allocates a new memory space of size NEW_SIZE to the pointer variable ptr and returns a pointer to the first byte of the new memory block.
- The NEW_SIZE may be larger or smaller than the original size.
- The new block of memory MAY or MAY NOT begin at the same place as the previously allocated memory block.

- The function guarantees that the old data will remain intact, if it run successfully.

- If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is FREED [lost] .

Memory allocation function

If memory allocation fails, these functions return a NULL pointer.

Since these functions return a pointer to void, when allocating memory use conversion:

```
pi = (int*)malloc(5*sizeof(int)); /* or */

pi = (int*)calloc(5, sizeof(int));

pi = (int*)realloc(pi, 10*sizeof(int));
```

Exercise

- Write a program to store a character string in a block of memory space created by malloc and then modify the same to store a larger string.

SOLUTION

```
int main(void)
{
    char * buffer ;
    /* allocating memory */
    if( (buffer = (char *) malloc (10 ))== NULL )
          { printf(" NOT ENOUGH SPACE " );
      exit (1 );  }
   strcpy(buffer, " NOIDA " );
    /* reallocating space */
    if ( (buffer = (char *) realloc( buffer, 15 ) ) == NULL )
          {          printf(" REALLOCATION FAILS ");
                        exit (1 );
          }
```

```
printf(“ Buffer contains : %s”, buffer );

strcpy( buffer, “ JIIT,Noida”);

Printf(“ Buffer contains : %s “, buffer );

/* freeing the memory */
free ( buffer );
}
```

Two dimensional Array

- In the same manner it is also possible to create a two dimensional array at RUN TIME using malloc ( ) or calloc ( ).

- For creating a two dimensional array, declare a pointer as follows:

```
data_type  ** pointer ;
```

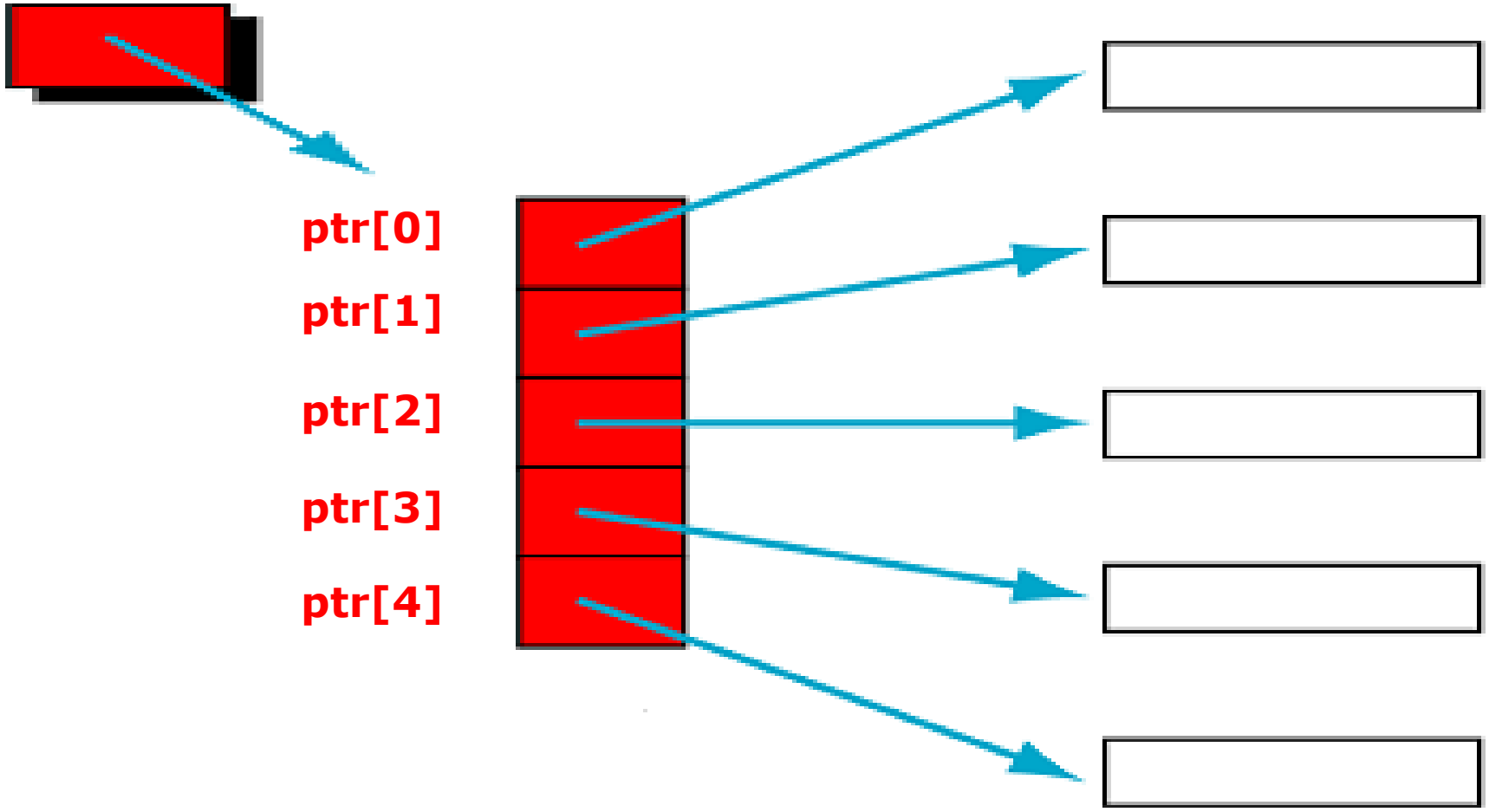
example:

```
int **p;
```

Example: creating a 2D array of m rows and n columns.

```
int main()
{   int **ptr;
    int m, n;
    printf("\n Enter the number of rows:");
    scanf("%d", &m);
    printf("\n Enter the number of columns:");
    scanf("%d", &n);
    /* creating an array of m pointers */
    ptr = (int**) malloc ( m *sizeof(int*));
    /* allocating each pointer a row ( 1 D array)  */
    for (i = 0; i < m; i++)
          ptr[i] = (int *)malloc( n  * sizeof(int));
```

int **ptr

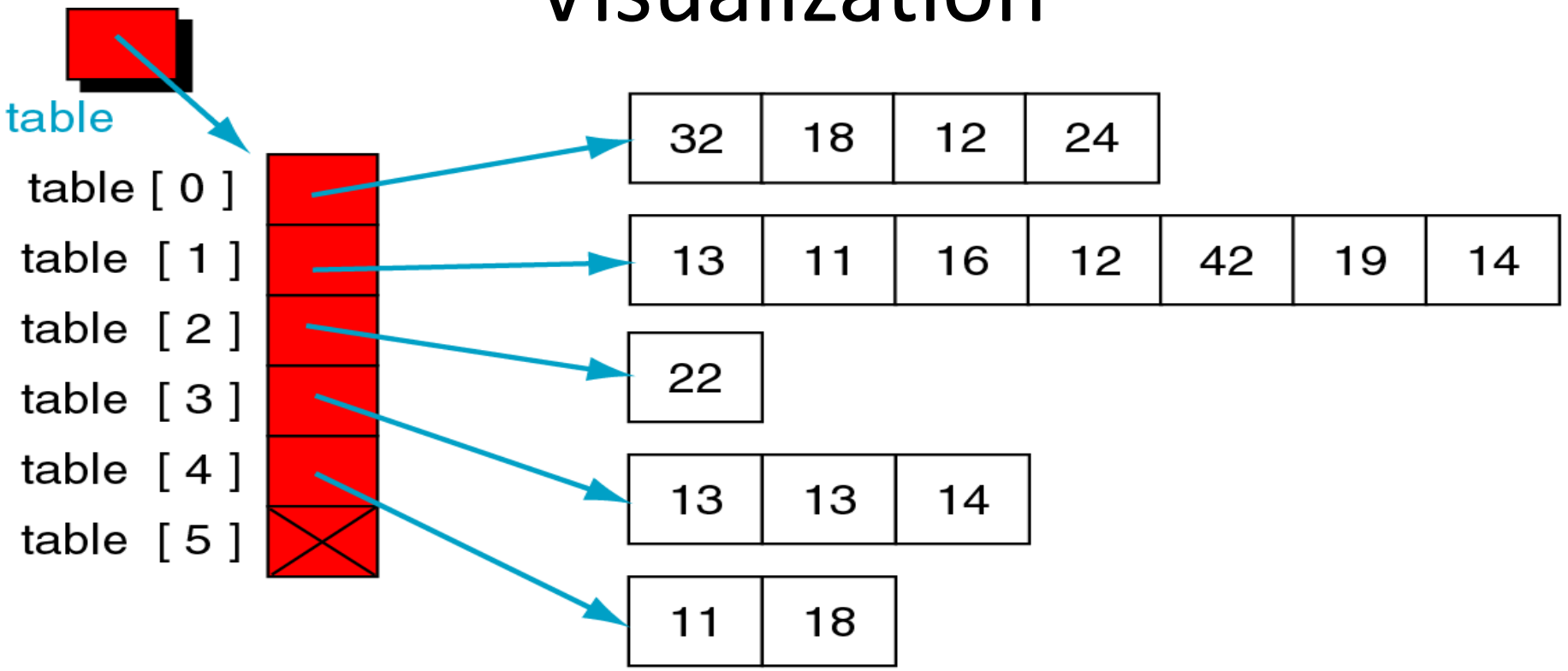


Check This Out !!!

- Is it possible to create a TWO Dimensional array with m rows, where each row will have different number of columns ? ? ?

YES ! ! !

Visualization




```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

Advantages

- There are number of reasons for using pointers:
  - Pointers are more efficient in handling data tables.
  - Pointer reduces the length and complexity of a given problem.
  - Pointer increase the execution speed.
  - The use of a pointer array to character strings results in saving of data storage space in memory.

Thank you