2.3 DYNAMIC MEMORY ALLOCATION

Objectives

- Learn how to allocate and free memory, and to control dynamic arrays of any type of data in general and structures in particular.

- Practice and train with dynamic memory in the world of work oriented applications.
- To know about the pointer arithmetic
- How to create and use array of pointers.

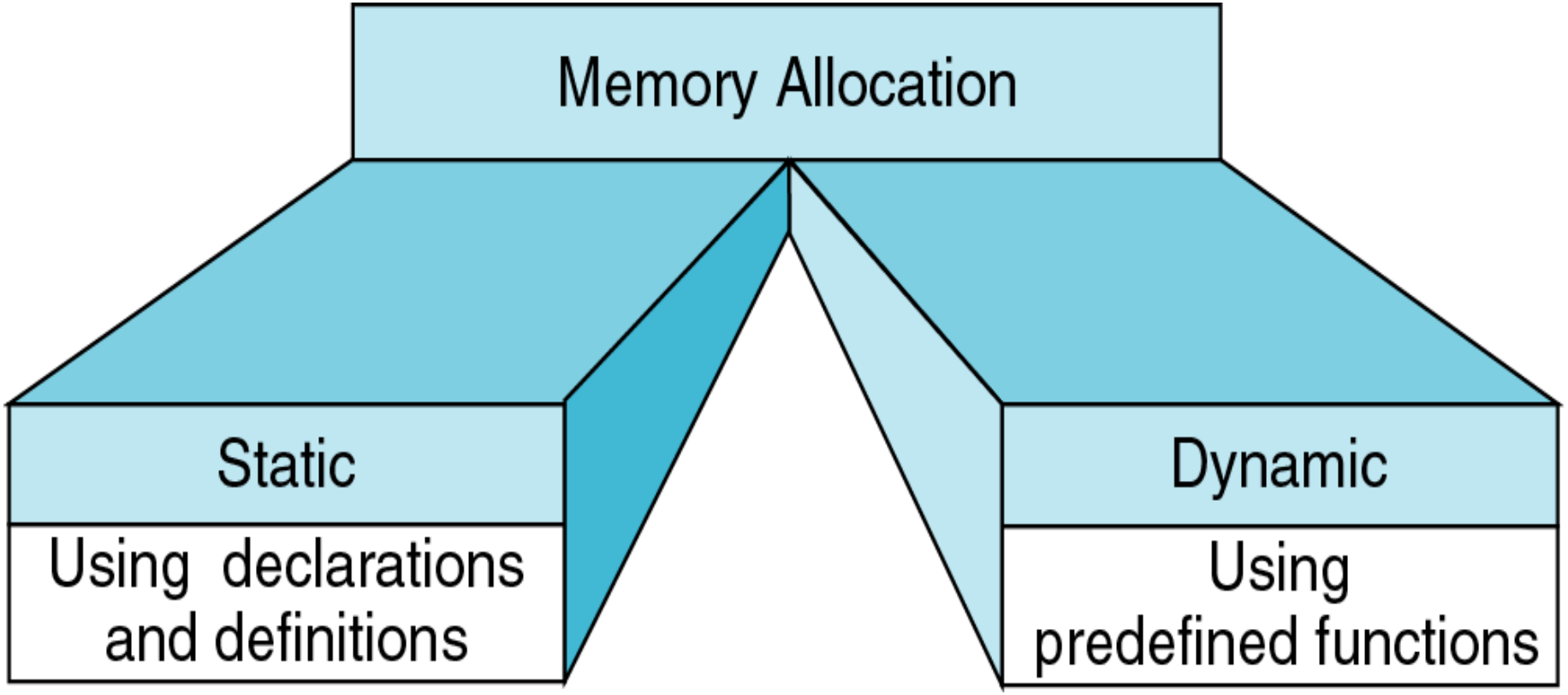
Agenda

- Dynamic memory allocation
  - Malloc
  - Calloc
  - Realloc
  - free
- Pointer Arithmetic
- Array of pointers

Introduction

- While doing programming, if you are aware about the size of an array, then it is easy and you can define it as an array.
- For example to store a name of any person, it can go max 100 characters so you can define something as follows:
  - char name[100]
  - But now let us consider a situation where you have no idea about the length of the text you need to store, for example you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later based on requirement we can allocate memory .

Memory Allocation Function



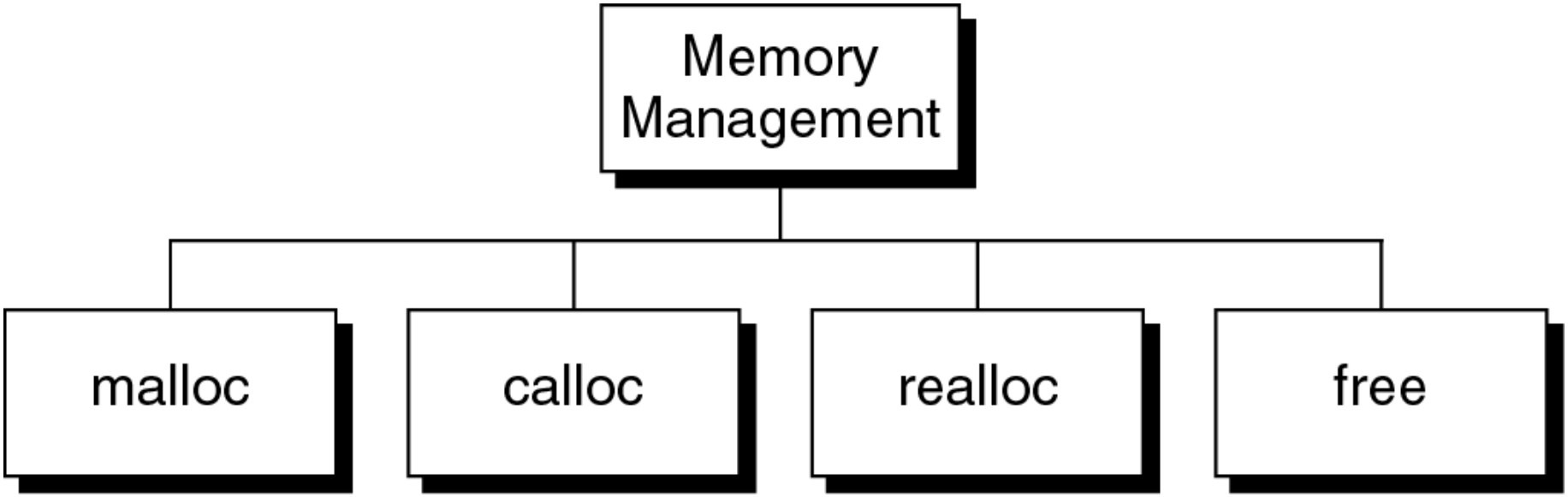
Difference between Static and Dynamic memory allocation

S.no	Static memory allocation	Dynamic memory allocation
1	In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
2	Memory size can't be modified while execution.	Memory size can be modified while execution.
	Example: array	Example: Linked list

Introduction

- Creating and maintaining dynamic structures requires **dynamic memory allocation—** the ability for a program to *obtain more memory space at execution time to hold new values*, and to *release space no longer needed*.

Memory Allocation Functions



Syntax

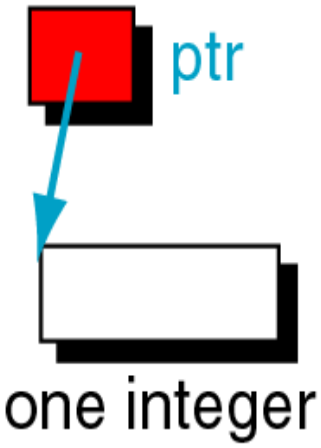
- The following are the function used for dynamic memory allocation
- **void *malloc(int num);**
  - This function allocates an array of **num** bytes and leave them uninitialized.
- **void *calloc(int num, int size);**
  - This function allocates an array of **num** elements each of which size in bytes will be **size**.
- **void *realloc(void *address, int newsize);**
  - This function re-allocates memory extending it upto **newsize**.
- **void free(void *address);**
  - This function releases a block of memory block specified by address.

Block Memory Allocation (malloc)

- Malloc function allocates a block of memory that contains the number of bytes specified in its parameter.
- It returns a void pointer to the first byte of the allocated memory
- The allocated memory is not initialized . We should therefore assume that it will contain unknown values and initialize it as required by our program.
- The function declaration is as follows
  - ***void* malloc (size_t size)***
- If it is not successful malloc return NULL pointer.
- An attempt to allocate memory from heap when memory is insufficient is known as **overflow.**

malloc

- It is up to the program to check the memory overflow
- If it doesn't the program produces invalid results or aborts with an invalid address the first time the pointer is used.




```
if (!(ptr = (int *)malloc(sizeof(int))))
   /* No memory available */
   exit (100) ;
/* Memory available */
...
```

malloc

- Malloc function has one more potential error.
- If we call malloc function with zero size , the results are unpredictable.
- It may return a NULL pointer
- ***Never call malloc with a zero size!!!!!***

Contiguous Memory Allocation (calloc)

- Calloc is primarily used to allocate memory for arrays.
- It differs from malloc only in that it sets memory to null characters.
  - ***void *calloc (size_t element-count, size_t element-size)***

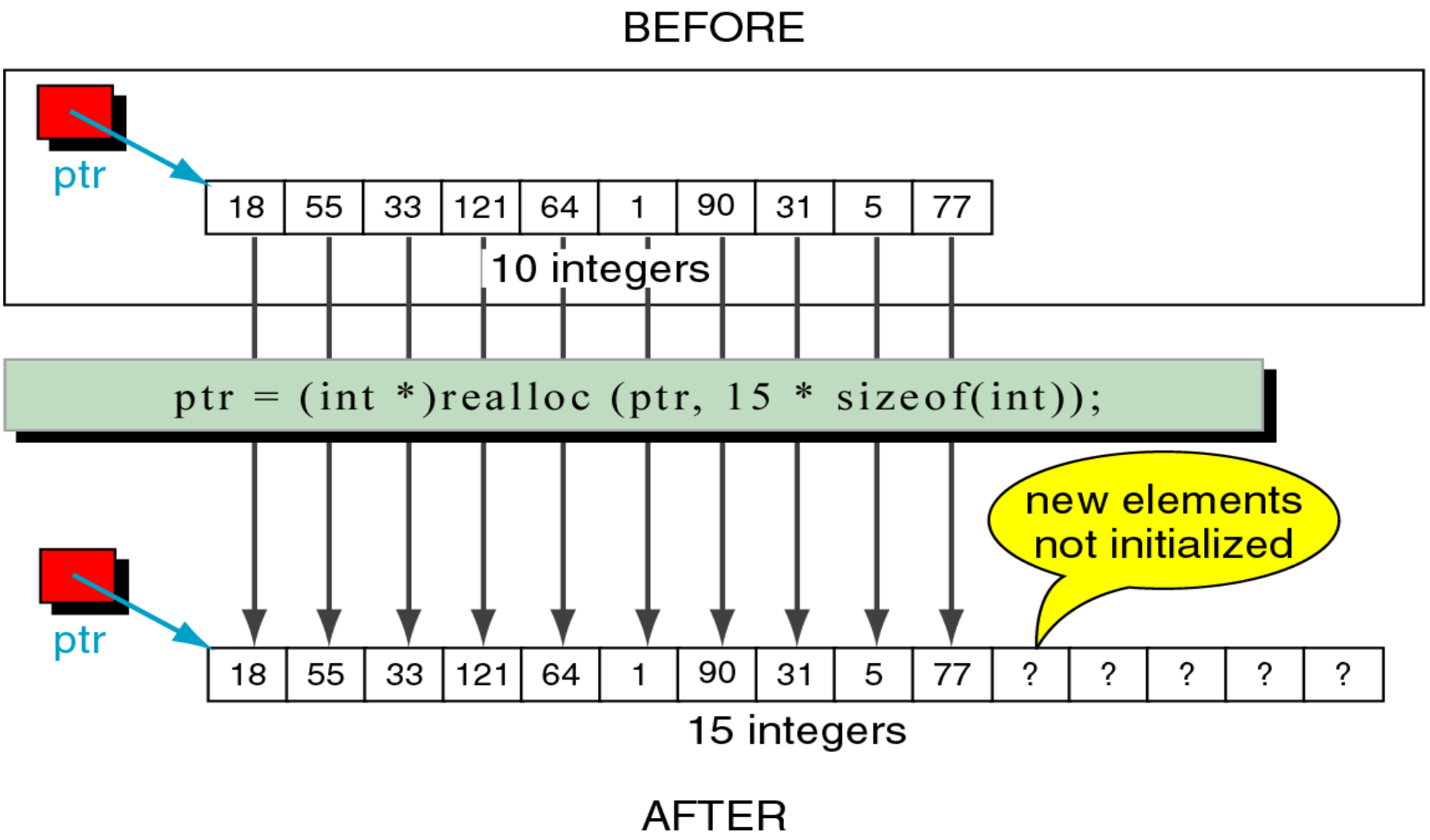

```
if (!(ptr = (int *)calloc (200, sizeof(int))))
    /* No memory available */
    exit (100) ;

/* Memory available */
...
```

Reallocation of memory(realloc)

- The realloc function can be highly inefficient and should be used advisedly.
- When given a pointer to the previously allocated block of memory, realloc changes the size of the block by deleting or extending the memory at the end of the block.
- If memory cannot be extended because of other allocations, realloc allocates a completely new block and copies the existing memory allocation to new allocation, and deletes the old allocation.
  - ***void *realloc (void* ptr, size_t newSize)***

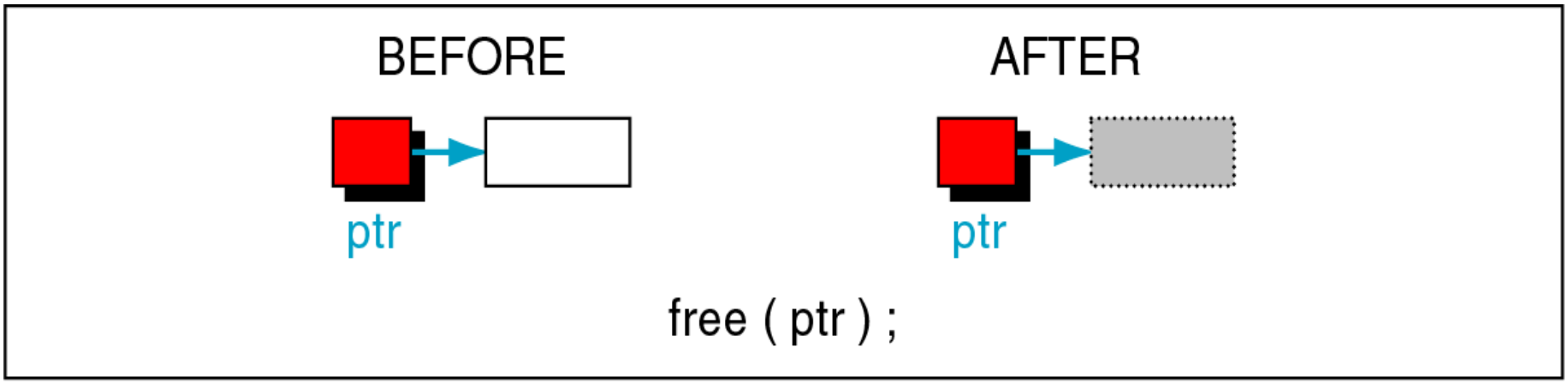
realloc

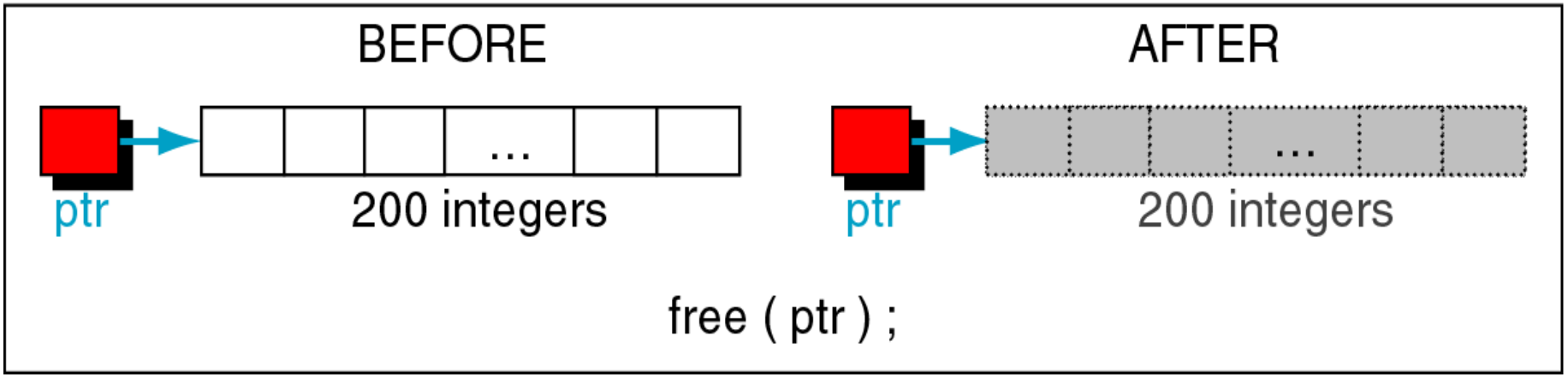

BEFORE
ptr
18
55
33
121
64
1
90
31
5
77
10 integers
ptr = (int *)realloc (ptr, 15 * sizeof(int));
new elements
not initialized
ptr
18
55
33
121
64
1
90
31
5
77
?
?
?
?
?
15 integers
AFTER

Releasing Memory (free)

- When memory locations allocated by *malloc, calloc or realloc* are no longer needed, they should be freed using the predefined function ***free***.
  - ***void free(void* ptr)***
- Below shows the example where first one releases a single element allocated with malloc
- Second example shows 200 elements were allocated with calloc . When free the pointer 200 elements are returned to the heap.

free


BEFORE
AFTER
ptr
ptr
free (ptr) ;


BEFORE
AFTER
ptr
200 integers
ptr
200 integers
free (ptr) ;

Difference between malloc and calloc

S.no	malloc()	calloc()
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block.	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory.
	Total = 80 bytes	Total = 1600 bytes
3	malloc () doesn't initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this function returns void pointer int *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

Resizing and Releasing Memory

- When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

- Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**.

Example-1

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

Enter number of elements: 3
Enter elements of array: 2 7 1
Sum=10

Dynamic-ex1.c

Example - 2

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));
   if(ptr==NULL)
   {
      printf("Error! memory not allocated.");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
   return 0;
}
```

Enter number of elements: 3
Enter elements of array: 2 1 3
Sum=6

Dynamic-ex2.c

Example - 3

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int *ptr,i,n1,n2;
   printf("Enter size of array: ");
   scanf("%d",&n1);
   ptr=(int*)malloc(n1*sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i=0;i<n1;++i)
       printf("%u\t",ptr+i);
   printf("\nEnter new size of array: ");
   scanf("%d",&n2);
   ptr=realloc(ptr,n2);
   for(i=0;i<n2;++i)
       printf("%u\t",ptr+i);
   return 0;
}
```

```
Enter size of array: 3
Address of previously allocated memory: 7474944
7474948 7474952
Enter new size of array: 5
7474944 7474948 7474952 7474956 7474960
```


Dynamic-ex3.c

Example - 4

```
#include <stdio.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 200 * sizeof(char) );
    if( description == NULL )
    {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
        {
        strcpy( description, "Zara ali a DPS student in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
}
```

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```


Dynamic-ex4.c

Example - 5

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 30 * sizeof(char) );
    if( description == NULL )
    {
            fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
            strcpy( description, "Zara ali a DPS student.");
    }
```

Example – 5 Cont---

```
    /* suppose you want to store bigger description */
    description = realloc( description, 100 * sizeof(char) );
    if( description == NULL )
    {
            fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else
    {
            strcat( description, "She is in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
    /* release memory using free() function */
    free(description);
}
```

Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th

Dynamic-ex5.c

Memory Leaks

- A memory leak occurs when allocated memory is never used again but is not freed.
- I can happen when
  - The memory's address is lost
  - The free function is never invoked though it should be
- The problem with memory leak is that the memory cannot be reclaimed and use later. The amount of memory available t the heap manager will be decreased.
- If the memory is repeatedly allocated and then lost, then the program may terminate when more memory is needed but malloc cannot allocate it because it ran out of memory.

Example

```
char *chunk;
while(10
{
        chunk=(char*) malloc (1000000);
        printf("allocating\n");
        }
```

- The variable chunk is assigned memory from heap. However this memory is not freed before another block of memory is assigned to it.
- Eventfully the application will run out of memory and terminate abnormally.

POINTER ARITHMETIC

Pointer Arithmetic

- This section introduces the concept of pointer arithmetic, and this will form one of the very important building blocks in understanding the functionality of pointers.

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or --)
  - Add an integer to a pointer(+ or += , - or -=)
  - Pointers may be subtracted from each other
  - All these operations meaningless unless performed on an array

  - NOTE: Division and Multiplication are not allowed.

Pointer Arithmetic

- *int a,b,*p,*q*
- *p=-q /* illegal use of pointers*/*
- *p<<=1 /* illegal use of pointers*/*
- *p=p-b /*valid*/*
- *p=p-q /* nonportable pointer conversion*/*
- *p=(int*) p-q /*valid*/*
- *p=p-q-a /*valid*/*
- *p=p+a /*valid*/*
- *p=p+q /* invalid pointer addition*/*
- *p=p*q /* illegal use of pointers*/*
- *p=p/q /* illegal use of pointers*/*
- *p=p/a /* illegal use of pointers*/*

Pointer Increment – Example - 1

```
#include<stdio.h>
void main()
{
    int n;
    int *pn;
    pn=&n;
    int *pn1;
    pn1=pn+1;
    printf("%d  %d\n", pn,pn1);

    double d;
    double *pd;
    pd=&d;
    double *pd1;
    pd1=pd+1;
    printf("%d  %d\n", pd,pd1);
}
```

```
2686788  2686792
2686768  2686776
```

Arithmetic-ex1.c

Incrementing Pointer :

- Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.
- Incrementing Pointer Variable Depends Upon data type of the Pointer variable

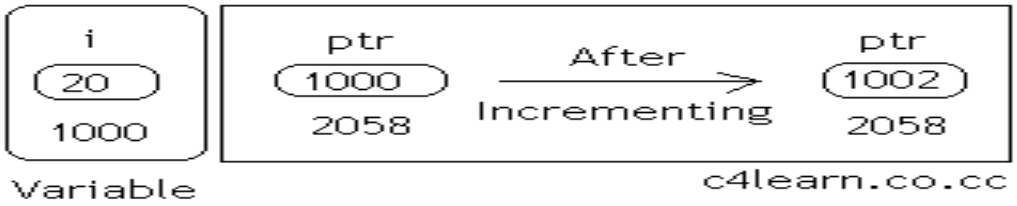
Three Rules should be used to increment pointer –

```
Address + 1 = Address

Address++   = Address

++Address   = Address
```

Pictorial Representation :



Data Type	Older Address stored in pointer	Next Address stored in pointer after incrementing (ptr++)
int	1000	1002
float	1000	1004
char	1000	1001

Example – 2

```
#include<stdio.h>
int main()
{
    int *ptr=(int *)1000;
   printf("Old Value of ptr : %u",ptr);
   ptr=ptr+1;
   printf("New Value of ptr : %u",ptr);
   return 0;
}
```

Old Value of ptr : 1000
New Value of ptr : 1004

Arithmetic-ex2.c

Difference between two integer Pointers – Example - 3

```
#include<stdio.h>

int main(){

float *ptr1=(float *)1000;
float *ptr2=(float *)2000;

printf("\nDifference : %d\n",ptr2-ptr1);

return 0;
}
```

Difference : 250

Arithmetic-ex3.c

Explanation

- Ptr1 and Ptr2 are two pointers which holds memory address of Float Variable.
- Ptr2-Ptr1 will gives us number of floating point numbers that can be stored.
- ptr2 - ptr1 = (2000 - 1000) / sizeof(float)
- = 1000 / 4

Pointer Division – Example - 4

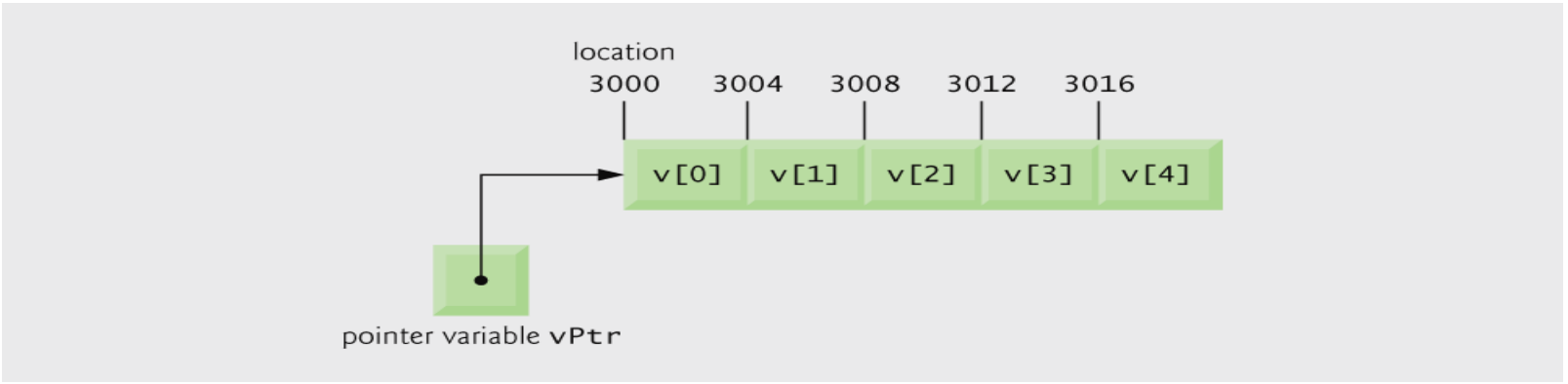
```
#include<stdio.h>
int main()
{
int *ptr1,*ptr2;
ptr1 = (int *)1000;
ptr2 = ptr1/4;
return(0);
}
```

Illegal Use of operator : INVALID

Arithmetic-ex4.c

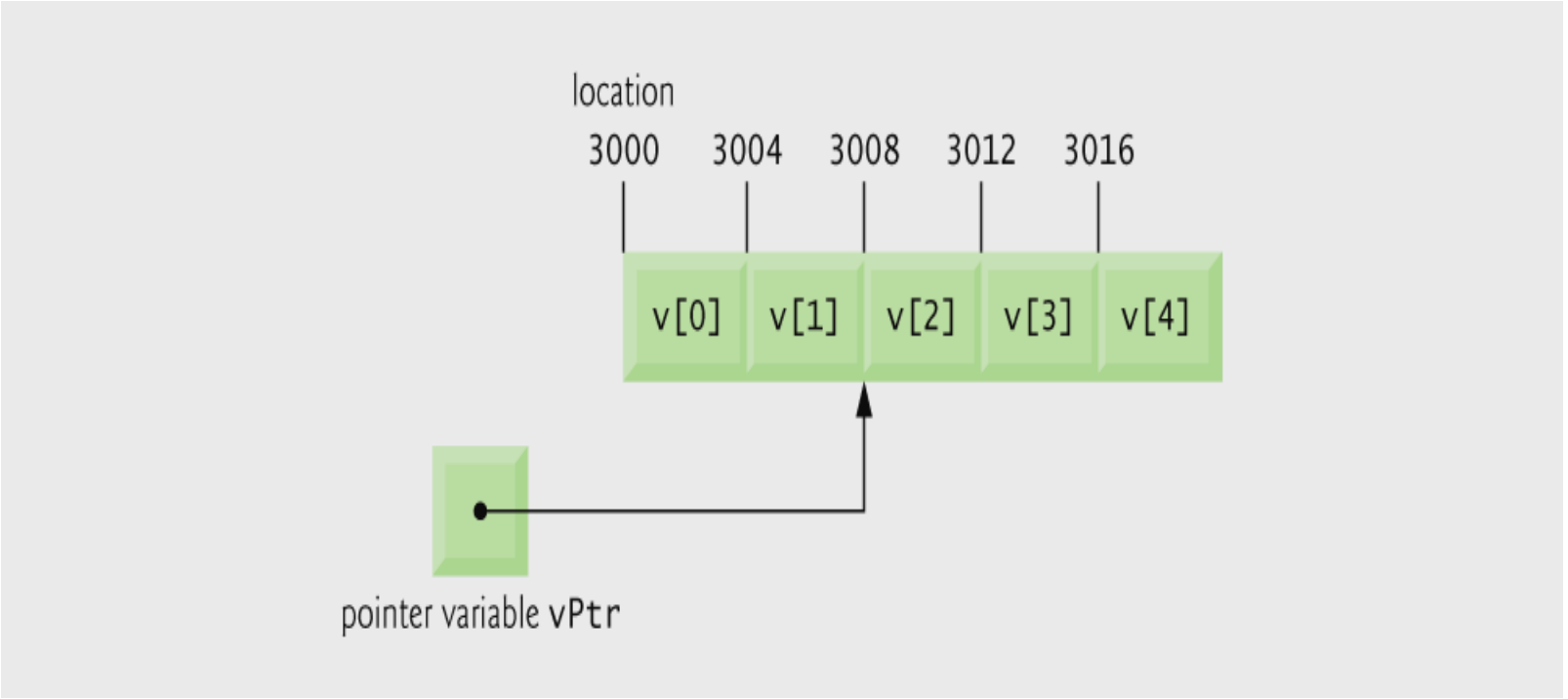
Pointer Expressions and Pointer Arithmetic-Arrays

- 5 element `int` array on machine with 4 byte `int`s
  - `vPtr` points to first element `v[ 0 ]`
    - at location `3000` (`vPtr = 3000`)
  - `vPtr += 2;` sets `vPtr` to `3008`
    - `vPtr` points to `v[ 2 ]` (incremented by 2), but the machine has 4 byte `int`s, so it points to address `3008`



Array `v` and a pointer variable `vPtr` that points to `v`.

The pointer `vPtr` after pointer arithmetic



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - *Returns number of elements from one to the other*. If

    ```
    vPtr2 = v[ 2 ];
    vPtr = v[ 0 ];
    ```

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (`<`, `==` , `>`)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0`

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

```
bPtr = b;
```

    - The array name (`b`) is actually the address of first element of the array `b[ 5 ]`

```
bPtr = &b[ 0 ]
```

    - Explicitly assigns `bPtr` to address of first element of `b`

The Relationship Between Pointers and Arrays

- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer/offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer/subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself

    `*( b + 3 )`

Example - 5

```
1
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main( void )
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b;                /* set bPtr to point to array b */
10    int i;                        /* counter */
11    int offset;                   /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArr                h" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
24
25    /* loop through array b */
26    for ( offset = 0; offset < 4; offset++ ) {
27       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28    } /* end for */
29
```

Array subscript notation

Pointer/offset notation

```
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```

Pointer subscript notation

Pointer offset notation

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40
```

Arithmetic-ex5.c

(continued on next slide…)

(continued from previous slide…)

```
Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

```c

   Copying a string using array notation and pointer notation. */
#include <stdio.h>

void copy1( char * const s1, const char * const s2 ); /* prototype */
void copy2( char *s1, const char *s2 ); /* prototype */

int main( void )
{
   char string1[ 10 ];           /* create array string1 */
   char *string2 = "Hello";      /* create a pointer to a string */
   char string3[ 10 ];           /* create array string3 */
   char string4[] = "Good Bye"; /* create a pointer to a string */

   copy1( string1, string2 );
   printf( "string1 = %s\n", string1 );

   copy2( string3, string4 );
   printf( "string3 = %s\n", string3 );

   return 0; /* indicates successful termination */

} /* end main */

```

```
25 /* copy s2 to s1 using array notation */
26 void copy1( char * const s1, const char * const s2 )
27 {
28    int i; /* counter */
29
30    /* loop through strings */
31    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
32       ; /* do nothing in body */
33    } /* end for */
34
35 } /* end function copy1 */
36
37 /* copy s2 to s1 using pointer notatio
38 void copy2( char *s1, const char *s2 )
39 {
40    /* loop through strings */
41    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42       ; /* do nothing in body */
43    } /* end for */
44
45 } /* end function copy2 */
```

Condition of **`for`** loop actually performs an action

```
string1 = Hello
string3 = Good Bye
```

ARRAY OF POINTERS

Introduction

- An array of pointer is similar to an array of ant predefined data type
- As a pointer variable always contain an address, an array of pointer is a collection of addresses.
- These can be address of ordinary isolated variable or of array elements.
- The elements of an array of pointers are stored in the memory just like elements of any other kind of array.
- Example is given below..

Arraypointer-ex1.c

Example - 1

```
#include <stdio.h>
 const int MAX = 3;
 int main () {
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX]; //array of pointers
    for ( i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
      for ( i = 0; i < MAX; i++) {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
      return 0;
}
```

Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Example - 2

```
#include<stdio.h>
void main()
{
   int arr[3]={1,2,3};
   int i, *ptr[3];
   for(i=0;i<3;i++)
         ptr[i]=arr+i;
   for(i=0;i<3;i++)
         printf("%p  %d\n", ptr[i],*ptr[i]);
}
```

```
0028FF30  1
0028FF34  2
0028FF38  3
```

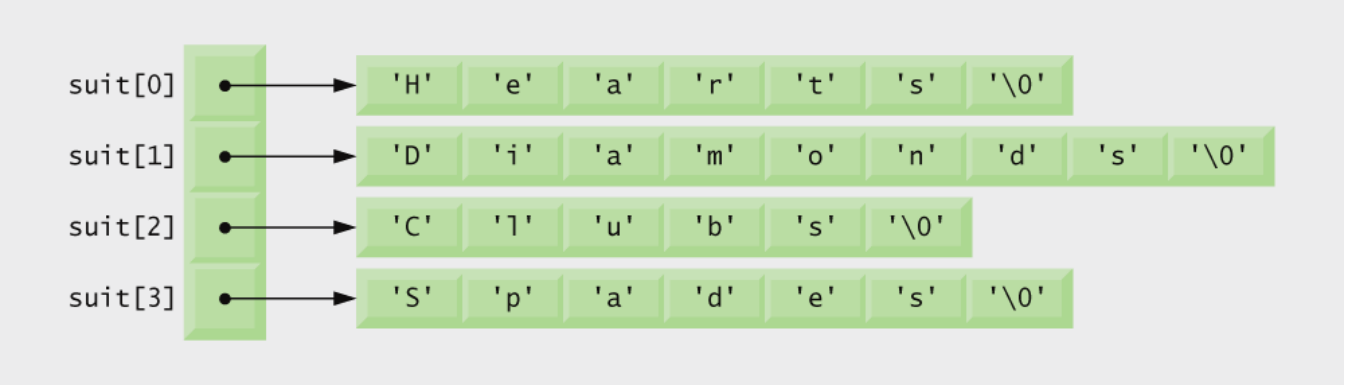
Arraypointer-ex2.c

Arrays of Pointers - Strings

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - `char *` – each element of `suit` is a pointer to a `char`
  - The strings are not actually stored in the array `suit`, only pointers to the strings are stored



  - `suit` array has a fixed size, but strings can be of any size

Case study: Roman numeral equivalents – Example - 3

```
#include <stdio.h>

void main( void ) {
        int decimal_number = 101, a = 0, b = 0;
    const char *x[11] = {"", "x", "xx", "xxx", "xl", "l", "lx", "lxx", "lxxx", "xc", "c"};
    const char *y[10] = {"", "i", "ii", "iii", "iv", "v", "vi", "vii", "viii", "ix"};

    while ((decimal_number > 100) || (decimal_number < 0)) {
            printf("Enter the decimal numbers in the range 1 to 100:\n");
            scanf("%d", &decimal_number);
    }

    a = decimal_number/10;
    b = decimal_number%10;

    printf("The equivalent roman is %s%s\n", x[a], y[b]);
}
```

Enter the decimal numbers in the range 1 to 100:
15
The equivalent roman is xv

Arraypointer-ex3.c

Review

- Dynamic Memory allocation
- Pointer Arithmetic
- Array of pointers

Try it Yourself

- Write a program using dynamic memory allocation to get a student's mark and display it back.
- Write a program to do the following

The process for finding a prime is quite simple. First, you know by inspection that 2, 3, and 5 are the first three prime numbers, because they aren't divisible by anything other than 1 and themselves. Because all the other prime numbers must be odd (otherwise they would be divisible by 2), you can work out the next number to check by starting at the last prime you have and adding 2. When you've checked out that number, you add another 2 to get the next to be checked, and so on. to check whether a number is actually prime rather than just odd, you could divide by all the odd numbers less than the number that you're checking, but you don't need to do as much work as that. if a number is *not prime, it must be* divisible by one of the primes lower than the number you're checking. Because you'll obtain the primes in sequence, it will be sufficient to check a candidate by testing whether any of the primes you've already found is an exact divisor. Store all primes in an array using pointers and pointer arithmetic.

Try it Yourself

- Find Largest Element Using Dynamic Memory Allocation and pointer arithmetic to access the array
- Write a C program to find the sum of two 1D matrices using dynamic memory and pointer arithmetic.

Try it Yourself –Ans - Student ma


```
#include<stdio.h>
#include<stdlib.h>
 void main()
{
   int no, *pt,i;
   clrscr();

   printf("Enter no of Students :");
   scanf("%d",&no);
   pt=(int *)malloc(no*2);
   if(pt== NULL)
   {
      printf("\n\nMemory allocation failed!");
     exit(0);
   }
```

Student mark

```
    printf("* * * * Enter roll no of students. * * * *\n");
    for (i=0;i<no;i++)
    {
        printf("-->");
        scanf("%d",(pt+i));
    }

    printf("\n* * * * Entered roll no. * * * *\n");
    for (i=0;i<no;i++)
    {
        printf("%d, ",*(pt+i));
    }
}
```