CSE101-Lec# 18,19

- Pointers in C

Introduction-Pointer declaration and Initialization

- A pointer is a variable that holds the address of another variable.
- The general syntax of declaring pointer variable is

  data_type *ptr_name;

  Here, data_type is the data type of the value that the pointer will point to. For example:

  int *pnum; char *pch; float *pfnum; //Pointer declaration

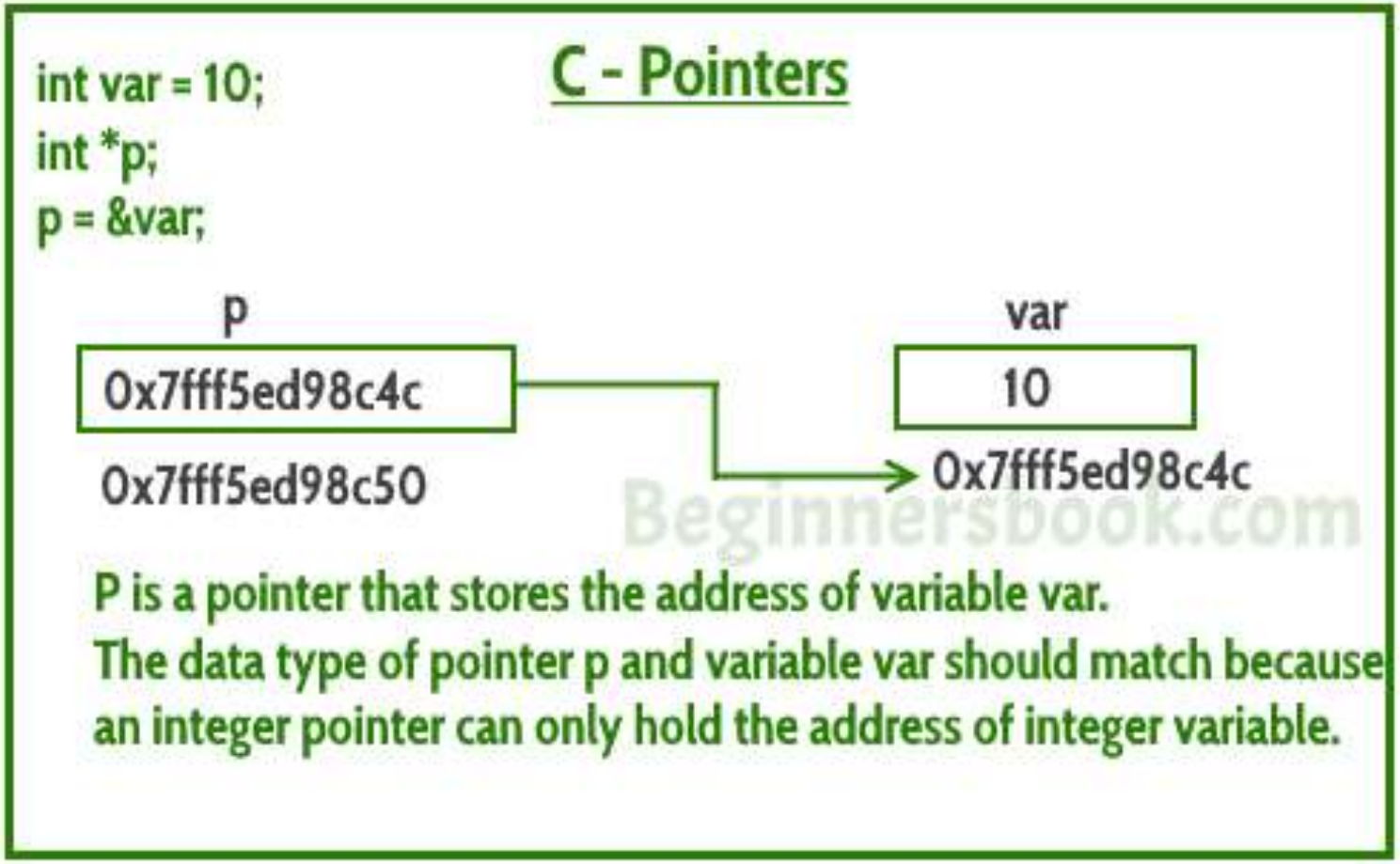
  int x= 10;

  int *ptr = &x; //Pointer initialization[When some variable's address is assigned to pointer, it is said to be initialized]

  The '*' informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable. ['*' is also known as indirection/ or deferencing/ or value at address operator]

  The & operator retrieves the address of x, and copies that to the contents of the pointer ptr. ['&' is also known as address of operator]

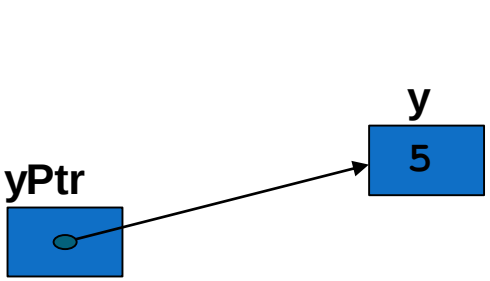
Understanding pointers

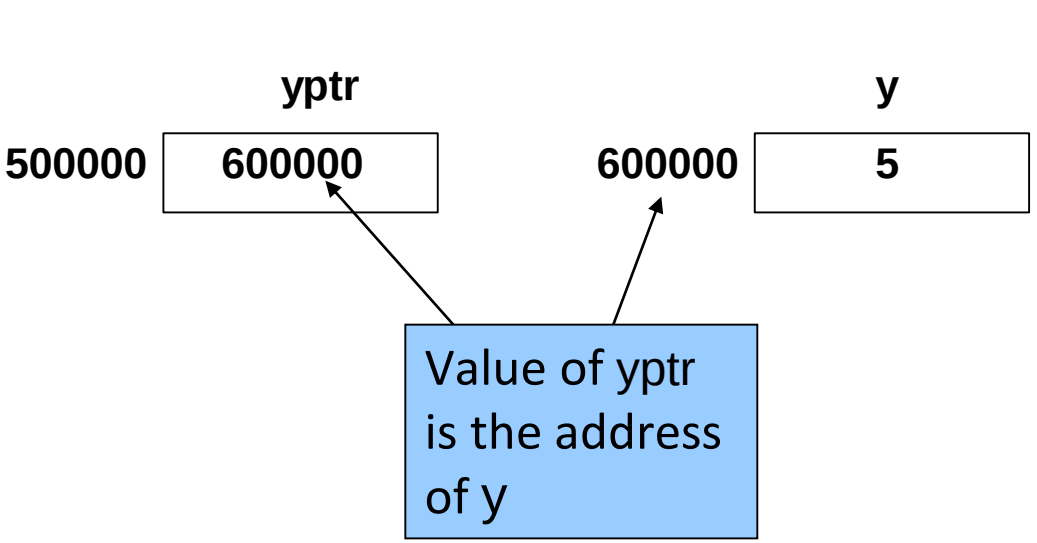


Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr "points to" y
```





Pointer Operators

- * (indirection/dereferencing operator)
  - Returns the value of the variable that it points to.
  - *yptr returns value of y (because yptr points to y)
  - * can be used for assignment

```
*yptr = 7;  /* changes y to 7 */
```

Example Code

```c
#include <stdio.h>

int main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

This program demonstrates the use of the pointer operators: & and *

Output

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Key points related to pointers

- ***Data type of the pointer variable and variable whose address it will store must be of same type***

Example:

int x=10;

float y=2.0;

int *px=&y;//Invalid, as px is of integer type and y is of float type

int *ptr=&x;//Valid as both ptr and x are of same types

- ***Any number of pointers can point to the same address***

Example:

int x=12;

int *p1=&x,*p2=&x,*p3=&x;// All the three pointers are pointing towards x

- ***Memory taken by any kind of pointer(i.e int, float, char, double…) as always equivalent to the memory taken by unsigned integer, as pointer will always store address of a variable(which is always unsigned integer), so the type of pointer will not make any difference***

Example-size taken by different type of pointers

```
#include<stdio.h>
int main()
{
        int *pnum;
        char *pch;
        float *pfnum;
        double *pdnum;
        long *plnum;
        printf("\n Size of integer pointer=%d",sizeof(pnum));
        printf("\n Size of character pointer=%d",sizeof(pch));
        printf("\n Size of float pointer=%d",sizeof(pfnum));
        printf("\n Size of double pointer=%d",sizeof(pdnum));
        printf("\n Size of long pointer=%d",sizeof(plnum));
        return 0;
}
//All will give the same answer(equivalent to size taken by unsigned integer for a particular compiler)
```

Program example-Finding area of circle using pointers

```
#include<stdio.h>
int main()
{
        double radius,area=0.0;
        double *pradius=&radius,*parea=&area;
        printf("\n Enter the radius of the circle:");
        scanf("%lf",pradius);
        *parea=3.14*(*pradius)*(*pradius);
        printf("\n The area of the circle with radius %.2lf = %.2lf",*pradius,*parea);
        return 0;
}
```

Program example-Factorial of a number using pointer

```
#include<stdio.h>
int main()
{
        int i,n,fact=1;
        int *pn,*pfact;
        pn=&n;
        pfact=&fact;
        printf("\n Enter number:");
        scanf("%d",pn);
        for(i=1;i<=*pn;i++)
        {
                *pfact=*pfact*i;
        }
        printf("\n Factorial of number is:%d",*pfact);
        return 0;
}
```

Program example-Reverse of a number using pointers

```
#include <stdio.h>
int main()
{
   int n, reversedNumber = 0, remainder;
   int *pn,*prn,*pr;
   pn=&n;
   prn=&reversedNumber;
   pr=&remainder;
   printf("Enter an integer: ");
   scanf("%d", pn);
   while(*pn != 0)
   {
      *pr = *pn%10;
      *prn = *prn*10 + *pr;
      *pn = *pn/10;
   }
   printf("Reversed Number = %d",*prn);

   return 0;
}
```

Types of pointers

- Null pointer
- Wild pointer
- Generic pointer(or void) pointer
- Constant pointer
- Dangling pointer

Null pointer

- A Null Pointer is a pointer that does not point to any memory location
- It is used to initialize a pointer variable when the pointer does not point to a valid memory address.
- So, if we don't know in the initial phases, where the pointer will point? , it is better to initialize pointer with NULL address

To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

or

```
int *ptr=0;
```

We can overwrite the NULL address hold by NULL pointer with some valid address also, in the later stages of program

Note: It is invalid to dereference a null pointer.

Example

```
#include<stdio.h>
int main()
{
        int *ptr=NULL;
        int a=10;
        printf("%u",ptr);// 0 will be displayed
        printf(“%d”,*ptr);//Invalid(Dereferencing), as ptr is NULL at this point.
        ptr=&a;
    printf("\n%d",*ptr);//Now it is allowed, as NULL pointer has starting pointing somewhere
        return 0;
}
```

Wild pointer

- Pointer which are not initialized during its definition holding some junk value(or Garbage address) are Wild pointer.
- Example of wild pointer:

```
int *ptr;
```

- Every pointer when it is not initialized is defined as a wild pointer.
- As pointer get initialized, start pointing to some variable its defined as pointer, not a wild one.

Example

```
#include<stdio.h>
int main()
{
        int *ptr;//Wild pointer
        int a=10;
        //printf("%u",ptr);//Gives garbage address value
        //printf("\n%d",*ptr);//Gives garbage value stored in the garbage address
        ptr=&a;//Now ptr is not a wild pointer
        printf("\n%d",*ptr);//
        return 0;
}
```

Void pointer

- Is a pointer that can hold the address of variables of different data types at different times also called generic pointer.
- The syntax for declaring a void pointer is

  void *pointer_name;
- Here, the keyword **void** represents that the pointer can point to value of any data type.
- But before accessing the value through generic pointer by dereferencing it, it must be properly **typecasted**.
- To Print value stored in pointer variable:

  ***(data_type*) pointer_name;**

Limitations of void pointers:

- void pointers cannot be directly dereferences. They need to be appropriately typecasted.
- Pointer arithmetic cannot be performed on void pointers.

Example

```
#include<stdio.h>
int main()
{
        int x=10;
        char ch='A';
        void *gp;
        gp=&x;
        printf("\n Generic pointer points to the integer value=%d",*(int*)gp);
        gp=&ch;
        printf("\n Generic pointer now points to the character
%c",*(char*)gp);
        return 0;
}
```

Constant Pointers

- A constant pointer, **`ptr`**, is a pointer that is initialized with an address, and cannot point to anything else.
- But we can use **`ptr`** to change the contents of variable pointing to
- Example

```
int value = 22;
int * const ptr = &value;
```

Constant Pointer

- Example:

  int * const ptr2

  indicates that ptr2 is a pointer which is constant. This means that ptr2 cannot be made to point to another integer.

- However the integer pointed by ptr2 can be changed.

Example

```
#include<stdio.h>
int main()
{
   int var1 = 60, var2 = 70;
   int *const ptr = &var1;
   printf("\n%d",*ptr);
   //ptr = &var2; //Invalid-Error will arise
   //printf("%d\n", *ptr);
   return 0;
}
```

Dangling pointer

- It is a type of pointer which point towards such a memory location which is already deleted/ or deallocated.
- It is a problem associated with pointers, where in a pointer is unnecessarily pointing towards deleted memory location
- It can be resolved through assigning NULL address once, the memory has been deallocated

Dangling pointer-Example 1[Compile time case]
When local variable goes out of scope

```
#include<stdio.h>
int main()
{
   int *ptr;
   {
      int val=23;
      ptr=&val;
      printf("\n%d",*ptr);// 23 is printed
      printf("\n%u",ptr);// Address of val is printed
   }
   printf("\n%u",ptr);// Same address is printed, even val is destroyed, hence ptr is dangling pointer
   ptr=NULL;//Solution
   printf("\n%u",ptr);// Now ptr is not a dangling pointer[0 address value is printed]
   return 0;
}
```

Dangling pointer-Example 2[Runtime/or Dynamic memory allocation case]
When free() function is called

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{

        int n=1;
        int *ptr = (int *)malloc(n*sizeof(int));
        *ptr=6;
   printf("%d",*ptr);//6 is printed
   printf("\n%d",ptr);//Printing address hold by pointer before deallocation
   free(ptr);
   printf("\n%d",ptr);//Same address will be printed(Dangling pointer)
  //SOLUTION
  ptr = NULL;//Pointer is now changed to NULL pointer
  printf("\n%d",ptr);//0 will be printed
  return 0;
}
```

Example-1-Passing pointer to a function(or call by reference)

```
//Passing arguments to function using pointers
#include<stdio.h>
void sum(int *a,int *b,int *t);
int main()
{
        int num1,num2,total;
        printf("\n Enter the first number:");
        scanf("%d",&num1);
        printf("\n Enter the second number:");
        scanf("%d",&num2);
        sum(&num1,&num2,&total);
        printf("\n Total=%d",total);
        return 0;
}
void sum(int *a,int *b,int *t)
{
        *t=*a+*b;
}
```

Example-2-Passing pointer to a function(or call by reference)

```
#include<stdio.h>
void read(float *b,float *h);
void calculate_area(float *b,float *h,float *a);
int main()
{
        float base,height,area;
        read(&base,&height);
        calculate_area(&base,&height,&area);
        printf("\n Area is :%f",area);
        return 0;
}
void read(float *b,float *h)
{
        printf("\n Enter the base of the triangle:");
        scanf("%f",b);
        printf("\n Enter the height of the triangle:");
        scanf("%f",h);
}
void calculate_area(float *b,float *h,float *a)
{
        *a=0.5*(*b)*(*h);
}
```

Choose the best Answer

Prior to pointer variable

A.It should be declared.

B.It should be initialized.

C.It should be both declared and initialized.

D.None of these.

MCQ

Comment on the following pointer declaration

int *ptr, p;

A.ptr is a pointer to integer, p is not.

B.ptr and p, both are pointers to integer.

C.ptr is pointer to integer, p may or may not be.

D.ptr and p both are not pointers to integer.

MCQ

The operator used to get value at address stored in a pointer variable is

A. *

B. &

C. &&

D. ||

MCQ

A pointer is

A. A keyword used to create variables

B. A variable that stores address of an instruction

C. A variable that stores address of other variable

D. All of the above

CSE101-Lec 20

Pointer arithmetic and expressions

Pointer and One dimensional array(or Pointer to 1D array)

Pointer arithmetic

- ***A limited set of arithmetic operations can be performed on pointers. A pointer may be:***
  - incremented (++), e.g. ptr++, ++ptr
  - decremented (--), e.g. ptr--, --ptr
  - an integer may be added to a pointer (+ or +=), e.g. ptr+2, ptr=ptr+2
  - an integer may be subtracted from a pointer (– or -=), e.g. ptr-2, ptr=ptr-2
  - We can subtract two pointers, if they are pointing towards same array
  - We can compare two pointers, if they are pointing towards same array
- ***Following set of operations are not applicable on pointers***
- We cannot add two pointers(addresses)
- We cannot multiply, divide and modulo two pointers(addresses)
- We cannot multiply, divide, modulo any constant from pointer(address)

Pointer arithmetic-Example

```
#include<stdio.h>
int main()
{
        int arr[]={1,2,3,4,5,6,7,8,9};
        int *p1,*p2;
        p1=arr;
        p1++;// p1 will point towards next memory location
        printf("\n%d",*p1);//2 will be displayed
        p1--;//p1 will point towards previous memory location
        printf("\n%d",*p1);// 1 will be displayed
        p1=p1+2;// Adding a constant to pointer(p1 will point towards 3rd element)
        printf("\n%d",*p1);// 3 will be displayed
        p1=p1-2;//Subtracting a constant from a pointer(P1 will point towards first element)
        printf("\n%d",*p1);// 1 will be displayed
        p2=&arr[4];
        printf("\n%d",p2-p1);//Subtracting two pointers(Returns 4(no. of elements b/w+1)(Pointers pointing to the same array)
//Comparing two pointers
        while(p1<=p2)
        {

        printf("\n%d",*p1);//Comparison of two pointers (Pointers pointing to the same array)
                p1++;
        }
        //Following are the invalid arithmetic operations(Not allowed on pointers)
        //printf("\n%d",p1+p2);//Invalid arithmetic
  //printf("\n%d",p1/p2);//Invalid arithmetic
  //printf("\n%d",p1*p2);//Invalid arithmetic
        //printf("\n%d",p1%p2);//Invalid arithmetic
  //printf("\n%d",p1*2);//Invalid arithmetic
  //printf("\n%d",p1/2);//Invalid arithmetic
  //printf("\n%d",p1%2);//Invalid arithmetic
        return 0;
}
```

Pointer expressions

- We can perform rich set of operations like: arithmetic, relational, assignment, conditional, unary, bitwise on pointer variables
- Examples:

```
*ptr1 + *ptr2
*ptr1 * *ptr2
*ptr1 + *ptr2 - *ptr3
*ptr1 > *ptr2
*ptr1 < *ptr2
*a=10
*b+=20
*z=3.5
*s=4.56743
c = (*ptr1 > *ptr2) ? *ptr1 : *ptr2;
(*ptr1)++
(*ptr1)--
*ptr1 & *ptr2
*ptr1 | *ptr2
*ptr1 ^ *ptr2
```

All these are the valid pointer expressions, and here we are working on values(not on addresses)

Pointer to an array(1D)

- A pointer can point towards an array using following notation:

Consider:

```
int a[]={1,2,3,4,5};
int *p=a; // pointer p starts pointing towards first element of array
```

Or

```
int *p=&a[0];
```

Now we can access elements of given array via pointer, such as:

```
int i;
for(i=0;i<5;i++)
{
printf("\n%d",*(p+i));
}
```

The Relationship Between Pointers and Arrays

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

```
bPtr = b;
```

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

```
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 `3` 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)`

- Array name itself is an address or pointer. It points to the first element(0^{th} element) of array.
- The arrays are accessed by pointers in same way as we access arrays using array name.
- Consider an array `b[5]` and a pointer `bPtr`:
  - `bPtr[3]` is same as `b[3]`

Example-Different notations with pointer to an array

```
#include<stdio.h>
int main()
{
        int a[]={1,2,3,4,5};
        int *p=a;
        // Different notations with pointer to an array for displaying second element
   // Same terminology can be used to display any element
   // All will display 2 on screen
        printf("\n%d",*(p+1));
        printf("\n%d",*(a+1));
        printf("\n%d",p[1]);
        printf("\n%d",1[p]);
        printf("\n%d",1[a]);
        return 0;
}
```

Pointer to an array with pointer arithmetic

```
#include<stdio.h>
int main()
{
        int arr[]={1,2,3,4,5};
        int i;
        int *p;
        p=arr;
        printf("\n First value is:%d",*p);
        p=p+1;
        printf("\n Second value is:%d",*p);
        *p=45;
        p=p+2;
        *p=-2;
        printf("\n Modified array is:");
        for(i=0;i<5;i++)
        {
                printf("\n%d",arr[i]);//We can also write i[arr]
        }
        p=arr;
        *(p+1)=0;
        *(p-1)=1;
        printf("\n Modified array is:");
        for(i=0;i<5;i++)
        {
                printf("\n%d",*(p+i));//We can also write *(i+arr)
        }
        return 0;
}
```

Program example 1-WAP to read and display elements of 1D array using pointer to an array

```
#include<stdio.h>
int main()
{
        int i,n;
        int a[10],*parr=a;
        printf("\n Enter the number of elements:");
        scanf("%d",&n);
        printf("\n Enter the elements:");
        for(i=0;i<n;i++)
        {
                scanf("%d",parr+i);
        }
        printf("\n Entered array elements are:");
        for(i=0;i<n;i++)
        {
                printf("\t %d",*(parr+i));
        }
        return 0;
}
```

Program example 2-WAP to find the sum and mean of 1D array elements using pointer to an array

```
#include<stdio.h>
int main()
{
        int i,n,arr[20],sum=0;
        int *pn=&n,*parr=arr,*psum=&sum;
        float mean=0.0,*pmean=&mean;
        printf("\n Enter the number of elements in the array:");
        scanf("%d",pn);
        for(i=0;i<*pn;i++)
        {
                printf("\n Enter the number:");
                scanf("%d",(parr+i));
        }
        for(i=0;i<*pn;i++)
        {
                *psum=*psum+*(arr+i);
        }
        *pmean=*psum/ *pn;
        printf("\n The numbers you entered are:");
        for(i=0;i<*pn;i++)
        printf("\n%d",*(arr+i));
        printf("\n The sum is:%d",*psum);
        printf("\n The mean is:%f",*pmean);
        return 0;
}
```

Pointer vs Array

1) the sizeof operator
   sizeof(array) returns the amount of memory used by all elements in array
   sizeof(pointer) only returns the amount of memory used by the pointer variable itself

2) the & operator
   &array is an alias for &array[0] and returns the address of the first element in array
   &pointer returns the address of pointer

3) a string literal initialization of a character array

char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’

char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)

4) Pointer variable can be assigned a value whereas array variable cannot be.

```
int a[10];
int *p;
p=a; /*legal*/
a=p; /*illegal*/
```

5) Arithmetic on pointer variable is allowed.

```
p++; /*Legal*/
a++; /*illegal*/
```

Output of following program?

```
# include <stdio.h>
void fun(int *ptr)
{
    *ptr = 30;
}
 int main()
{
  int y = 20;
  fun(&y);
  printf("%d", y);
  return 0;
}
```

a)30 b)20 c)compiler error d)runtime error

What is the output of this C code?

```
int main()
  {
  int i = 10;
  void *p = &i;
  printf("%d\n", (int)*p);
  return 0;
  }
```

A. Compile time error

B. Segmentation fault/runtime crash

C. 10

D. Undefined behaviour

MCQ

Which of the following does not initialize ptr to null (assuming variable declaration of a as int a=0)?

A. int *ptr = &a;

B. int *ptr = &a – &a;

C. int *ptr = a – a;

D. All of the mentioned

What is the output of this C code?

```
int x = 0;
void main()
{
int *ptr = &x;
printf("%p\n", ptr);
x++;
printf("%p\n ", ptr);
}
```

A. Same address

B. Different address

C. Compile time error

D. Varies

CSE101-Lec#21

- Dynamic memory management

Outline

- Dynamic Memory management
  - `malloc()`
  - `calloc()`
  - `realloc()`
  - `free()`

Dynamic Memory Allocation

- The statement:

```
int marks[100];
```

  allocates block of memory to 100 elements of type int and memory is also contiguous. If one `int` requires 4 bytes of memory , a total of 400 bytes are allocated.

Why this approach of declaring array is not useful?

- This may lead to wastage of memory if all allocated memory is not utilized.

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
- So, it will allocate only that much of memory which is actually required by the program.
- Hence memory wastage can be avoided/ or if more memory is required that can also be allocated.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

- The name malloc stands for "memory allocation".
- The `malloc()` function allocates a block of memory of specified size from the memory heap.
- Syntax:

```
void * malloc(size);
```

- Here size is the number of bytes of storage to be allocated.
- If memory is allocated successfully , it returns a **pointer to first location** of newly allocated block of memory.
- If memory is not allocated i.e. no enough space exists for new block or some other reason, returns **NULL**.

malloc()

- Return type of `malloc()` is void pointer , it has to be **cast** to the type of data being dealt with.
- memory allocated by `malloc()` by default contain the garbage values.
- Example:

```
int *p;
p=(int*)malloc(n*sizeof(int));
```

- In the above example, p is **pointer** of type integer
- `int*` tells to what type it will be pointing. `int` tells that the `malloc()` function is type casted to return the address of integer variable.
- `n` is the number of elements

Program example-malloc()

```
#include<stdio.h>
#include<stdlib.h>
int main( )
{
int *p,n,i;
printf ("Enter the number of integers to be entered ") ;
scanf("%d" ,&n) ;
p=(int*)malloc(n*sizeof(int));//malloc() returns void* so we need to typecast with the specific
data type
if(p==NULL)
{
printf("Memory not available\n");
exit(1);
}
else
{
printf("\n Mmeory allocation was successful");
printf ("\nEnter integer values ") ;
for(i=0;i<n;i++)
{
scanf("%d",p+i);//In place of p+i we can write &p[i](treating it as  ID array)
}
for(i=0;i<n;i++)
printf("\n%d",*(p+i));//In place of *(p+i) we can write p[i](treating it as ID array)
}
return 0;
```

calloc()

- The name calloc stands for "contiguous allocation".
- It provides access to memory, which is available for dynamic allocation of variable-sized blocks of memory.
- Syntax:

```
void *calloc(size_t nitems, size_t size);
```

- calloc is similar to malloc, but the main **difference** is that the values stored in the allocated memory space is **zero** by default. With malloc, the allocated memory could have any garbage value.
- `calloc()` requires **two arguments**.
  1. The **first** is the number of variables you'd like to allocate memory for.
  2. The **second** is the size of each variable.

calloc()

- If memory is allocated successfully, function `calloc()` returns a pointer to the first location of newly allocated block of memory otherwise returns NULL
- Memory allocated by `calloc()` by default contains the zero values.
- E.g. If we want to allocate memory for storing n integer numbers in contiguous memory locations

```
int *p;
p=(int*)calloc(n, sizeof(int));
```

Program example-calloc()

```
#include<stdio.h>
#include<stdlib.h>
int main( )
{

int *p,n,i;
printf("Enter the number of blocks we want to reserve:") ;
scanf("%d",&n) ;
p=(int*)calloc(n,sizeof(int));//malloc() returns void* so we need to typecast with the specific data type
if(p==NULL)
{
printf("Memory not available\n");
exit(1);
}
else
{
printf("\n Memory allocation successful");
printf ("\nEnter integer values: ") ;
for(i=0;i<n;i++)
{
scanf("%d",p+i);
}
printf("\n Entered values are:");
for(i=0;i<n;i++)
printf("\n%d",*(p+i));
return 0;
}
}
```

Difference between `malloc()` and `calloc()`

	`calloc()`	`malloc()`
Function:	Allocates a region of memory large enough to hold "n elements" of "size" bytes each.	Allocates "size" bytes of memory.
Syntax:	`void *calloc (number_of_blocks, size_in_bytes);`	`void *malloc (size_in_bytes);`
No. of arguments:	2	1
Contents of allocated memory:	The allocated region is initialized to zero.	The contents of allocated memory are not changed. i.e., the memory contains garbage values.
Return value:	`void pointer (void *).` If the allocation succeeds, a pointer to the block of memory is returned.	`void pointer (void *).` If the allocation succeeds, a pointer to the block of memory is returned.

realloc()

- Now suppose you've allocated a certain number of bytes for an array but later find that you want to add values to it. You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using `realloc()`, without losing your data.
- `realloc()` takes **two** arguments.
  1. The **first** is the pointer referencing the memory.
  2. The **second** is the total number of bytes you want to reallocate.
- Passing zero as the second argument is the equivalent of calling free.
- Syntax:

```
void *realloc(pointerToObject, newsize);
```

realloc()

- If memory is allocated successfully, function `realloc()` returns a pointer to the first location of newly allocated block of memory which may be at same site or at new site and copy the contents from previous location to a new location if required , otherwise returns NULL.

Program example-realloc()

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int *ptr,n,m,i;
        printf("\n Enter initial value of n:");
        scanf("%d",&n);
        ptr=(int *)calloc(n,sizeof(int));
        if(ptr==NULL)
        {
        printf("\n Memory allocation failure(calloc())");
                exit(1);
        }
        else
        {
        printf("\n Memory allocation successful");
        printf("\n Enter values as per initial requirement:");
                for(i=0;i<n;i++)
                {
                scanf("%d",ptr+i);
                }
                printf("\n Entered values are:");
                for(i=0;i<n;i++)
                {
               printf("\n%d",*(ptr+i));
                }
                m=n;
        printf("\n Enter new value of n for reallocation:");
                scanf("%d",&n);
                ptr=(int *)realloc(ptr,n*sizeof(int));
```

-

```
        if(ptr==NULL)
                {
printf("\n Memory allocation failure while realloation");
                exit(2);
                }
                else
                {
                printf("\n Memory reallocated successfully");
                printf("\n Enter new values as per requirement");
                for(i=m;i<n;i++)
                {
                scanf("%d",ptr+i);
                }
                printf("\n All values entered are(old+new):");
                for(i=0;i<n;i++)
                {
                printf("\n%d",*(ptr+i));
                }
                }
                free(ptr);
                printf("\n Memory deallocated");
                }
                return 0;
}
```

free()

- Deallocates a memory block allocated by previous call to `malloc(), calloc()` `or realloc()` and return it to memory to be used for other purposes.
- Syntax:

```
void *free(void *block);
```

- The argument of function `free()` is the pointer to block of memory which is to be freed.

free()

- The realloc() function can behave the same as free() function provided the second argument passed to realloc() is 0.

```
free(ptr);
```

which is equivalent to

```
realloc(ptr,0);
```

Program example-free()

```
#include<stdio.h>
#include<stdlib.h>
int main( )
{
int *p,n,i;
printf ("Enter the number of integers to be entered ") ;
scanf("%d" ,&n) ;
p=(int*)malloc(n*sizeof(int));//malloc() returns void* so we need to typecast with the specific
data type
if(p==NULL)
{
printf("Memory not available\n");
exit(1);
}
else
{
printf("\n Mmeory allocation was successful");
printf ("\nEnter integer values ") ;
for(i=0;i<n;i++)
{
scanf("%d",p+i);//In place of p+i we can write &p[i](treating it as  ID array)
}
for(i=0;i<n;i++)
printf("\n%d",*(p+i));//In place of *(p+i) we can write p[i](treating it as ID array)
}
free(p);
printf("\n Memory deallocated");
return 0;
```

Memory Leak

- A condition caused by a program that does not free up the extra memory it allocates.
- It occurs when the dynamically allocated memory is no longer needed but it is not freed.
- If we continuously keep on allocating the memory without freeing it for reuse, the entire heap storage will be exhausted.
- In such circumstances, the memory allocation functions will start failing and program will start behaving unexpectedly

Program example-Memory leak

```
#include<stdio.h>
int main()
{
        int *p;
        p=(int*)malloc(1*sizeof(int));
        *p=6;
        printf("%d",*p);
        //Memory was not deallocated, hence memory leak may arise
        //Solution
        //free(ptr);
        return 0;
}
```

MCQ

Among 4 header files, which should be included to use the memory allocation functions like malloc(), calloc(), realloc() and free()?

A. #include<string.h>
B. #include<stdlib.h>
C. #include<memory.h>
D. Both b and c

MCQ

Which function is used to delete the allocated memory space?

A. Dealloc()
B. free()
C. Both A and B
D. None of the above

MCQ

Which of the following statement is correct prototype of the malloc() function in c ?

A. int* malloc(int);
B. Char* malloc(char);
C. unsigned int* malloc(unsigned int);
D. void* malloc(size_t);

MCQ

Suppose we have a one-dimensional array, named 'x', which contains 10 integers. Which of the following is the correct way to allocate memory dynamically to the array 'x' using malloc()?

(A) x=(int*)malloc(10);

(B) x=(int*)malloc(10,sizeof(int));

(C) x=malloc(int 10,sizeof(int));

(D) x=(int*)malloc(10*sizeof(int));

MCQ

The number of arguments taken as input which allocating memory dynamically using malloc() is _____

(A) 0

(B) 1

(C) 2

(D) 3