

# Data Structures

## UNIT 1

Course Instructor

**Dr. Nagarathna N**

Department of CSE(ICB), BMSCE

# UNIT 1

- **Introduction To Data Structure:** Data Management concepts, Data types – primitive and non-primitive, Types of Data Structures- Linear & Non-Linear Data Structures. Structures and pointers
- **Dynamic memory allocation:** allocating a block of memory: Malloc, allocating multiple blocks of memory: Calloc, Releasing the used space: Free Altering the size of memory: Realloc.

# Introduction to Pointers

- Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the type of the data.

```
int x = 10;
```

- When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol x and the relative address in memory where those 2 bytes were set aside.
- Thus, every variable in C has a value and an also a memory location (commonly known as address) associated with it. Some texts use the term *rvalue* and *lvalue* for the value and the address of the variable respectively.
- The rvalue appears on the right side of the assignment statement and cannot be used on the left side of the assignment statement.
- Therefore, writing `10 = k;` is illegal.

# Introduction to Pointers

- A variable has address and value associated with it.
  - The address and value are also denoted as **lvalue** and **rvalue**
  - A pointer is a variable which stores address of another variable
- 
- `int x=10;`
  - `int *ptr;`
  - `ptr=&x;`

# Introduction to Pointers

- A pointer is a variable that contains the memory location of another variable.
- Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element.
- A pointer variable can store only the address of a variable.
- Pointers are not allowed to store actual memory addresses, but only the addresses of variables of a given type.
- Therefore, the statement

# Introduction to Pointers

- Applications of pointers :
  - To pass information back and forth between a function and its reference point.
  - to enable programmers to return multiple data items from a function via function arguments.
  - Provides an alternate way to access individual elements of the array.
  - To pass arrays and strings as function arguments.
  - Enable references to functions, so that functions can be passed as arguments to another function.
  - For dynamic memory allocation.

# Introduction to Pointers

- A pointer is a special variable that is used to store the address of some other variable.
- A pointer can be used to store the address of a single variable, array, structure, union, or even a pointer.
- Using Pointers allows us to –
  - Achieve call by reference (i.e write functions which change their parameters)
  - Handle arrays efficiently.
  - Handle structures (Record) efficiently.
  - Create linked lists, trees, graphs etc.

## Declaring a Pointer Variable

- Actually pointers are nothing but memory addresses.
- A pointer is a variable that contains the memory location 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;
```

```
int x= 10;
```

```
int *ptr;
```

```
ptr = &x;
```

- The '\*' informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.
- The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.



## Declaring a Pointer Variable

- `int *pi;     /* pi is a pointer to an int */`
- `long int *p; /* p is a pointer to a long int */`
- `float* pf;  /* pf is a pointer to a float */`
- `char c, d, *pc; /* c and d are a char and pc is a  
                  pointer to char */`
- `double * pd, e, f; /* pd is pointer to a double, e and f  
                      are double */`
- `char * start;   /* start is a pointer to a char */`
- `char * end;     /* end is a pointer to a char */`

## Pointer Variable

- *The two fundamental operators used with the pointers are:*
- **1. Address operator &**
- **2. Indirection operator \***

# Pointer

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int num, *pnum;
```

```
    pnum=&num;
```

```
    printf("\n Enter the number");
```

```
    scanf("%d", &num);
```

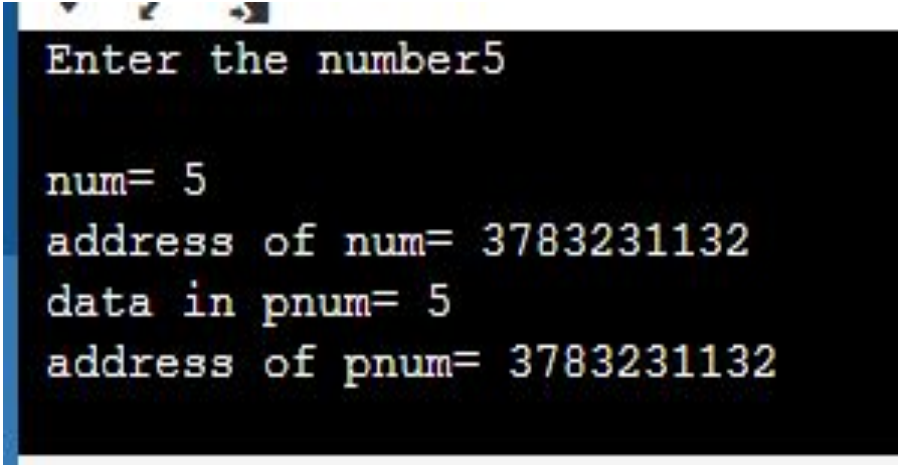
```
    printf("\n num= %d", num);
```

```
    printf("\n address of num= %u", &num);
```

```
    printf("\n data in pnum= %d", *pnum);
```

```
    printf("\n address of pnum= %u", pnum);    return 0;
```

```
}
```



```
Enter the number5
```

```
num= 5
```

```
address of num= 3783231132
```

```
data in pnum= 5
```

```
address of pnum= 3783231132
```

## Pointer Variable

```
#include<stdio.h>
```

```
main()
```

```
{
```

```
int a = 5;
```

```
int *p; /*pointer declaration*/
```

```
p= &a; /*copying address of variable a to the pointer p*/
```

```
*p = 10; /*indirection or use of pointer to change the value of  
variable a*/
```

```
printf("%d ", a);
```

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

```
printf("%d ",*(&a));
```

```
printf("%u ",p); printf("%u ",&a);
```

```
return 0;}
```

## Pointer Variable

```
#include<stdio.h>
main()
{
int a = 5;
int *p; /*pointer declaration*/
p= &a; /*copying address of variable a to the pointer p*/
*p = 10; /*indirection or use of pointer to change the value of variable
a*/
printf("%d ", a);
printf("%d ",*p);
printf("%d ",*(&a));
printf("%u ",p); printf("%u ",&a);
return 0;}
```

**Output: 10 10 10 1286290532 1286290532**

## Pointer Variable

```
int main(){
    int b, a = 5;
    int *p; /*pointer declaration*/
    p= &b; /*copying address of variable b to the pointer p*/
    *p = 10;
    printf("%d ", a);
    printf("%u ",&a);
    printf("%d ",*p);
    printf("%u ",p);
    printf("%d ",b);
    printf("%u ",&b);
    return 0;}
```

**Output: 5 3261754176 10 3261754180 10 3261754180**

## Pointer Variable

```
int *p; /*pointer declaration*/
```

```
p= &a; /*copying address of variable a to the pointer p*/
```

- It is a variable which stores the address of some other variable (a in this case).
- Since p is a variable, the compiler must provide memory to this variable also.
- Any type of pointer gets four/eight bytes in the memory.

```
int *p1;
```

```
float *p2;
```

```
char *p3;
```

- All pointers p1, p2, p3 get 4/8 bytes each.
- Thus we can say that a pointer irrespective of its type is storing the addresses as integer values and each integer requires only 4/8 bytes.



## Pointer Variable

```
#include<stdio.h>
main()
{
    int a,*p1;  a = 5;
    float b,*p2;  b=2.5;
    char c, *p3;  c='a';
    p1= &a;
    p2 = &b;
    p3=&c;
    printf(“%d”,sizeof(p1));
    printf(“%d”,sizeof(p2));
    printf(“%d”,sizeof(p3));
}
```

**output: 8 8 8**

# Pointer Variable

- **Indirection operator \***
- We can **dereference a pointer** i.e. refer to the value of the variable to which it points, by using unary ‘\*’ operator known as the **indirection operator**.

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

```
x=5;
```

```
p=&x;
```

```
*p = 10;
```

- \* is equivalent to writing value at address

## Pointer Illustrations

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int num, *pnum1,*pnum2;
```

```
    pnum1=&num;
```

```
    *pnum1=10;
```

```
    pnum2=pnum1;
```

```
    printf("the value of num is %d \t %d \t %d \n",num,  
    *pnum1,*pnum2);
```

```
    printf("the address of num is %x \t %x \t %x \n",&num,  
    pnum1,pnum2);
```

```
return 0;
```

```
}
```

**Output : ?**

## Pointer Illustrations

```
#include <stdio.h>

int main()
{
    int num, *pnum1,*pnum2;
    pnum1=&num;
    *pnum1=10;
    pnum2=pnum1;
    printf("the value of num is %d \t %d \t %d \n",num,
    *pnum1,*pnum2);
    printf("the address of num is %x \t %x \t %x \n",&num,
    pnum1,pnum2);
    return 0;
}
```

**Output :the value of num is 10 10 10**

## Pointer Illustrations

```
#include <stdio.h>
int main()
{
    int a=3,b=5, *pnum;
    float*pfnum;
    pnum=&a;
    printf("the value of a is %d \n", *pnum);
    pnum=&b;
    printf("the value of b is %d \n",*pnum);
    pfnum= &a; //invalid
    return 0;
}
```

**Output :the value of a is 3**

**the value of b is 5**

## Pointer Illustrations

```
#include <stdio.h>
int main()
{
    int num, *pnum;
    pnum=&num;
    *pnum=10;
    printf("the *pnum is %d \n", *pnum);
    printf("the num is %d \n",num);
    *pnum= *pnum+1;
    printf(" After Increment the *pnum is %d \n", *pnum);
    printf("After Increment the num is %d \n",num);
    return 0;
}
```

**Output: the \*pnum is 10**

**the num is 10**

**After Increment the \*pnum is 11**

**After Increment the num is 11**

**Write a program to test whether a number is positive, negative, or equal to zero**

```
#include <stdio.h>
int main()
{
    int num, *pnum = &num;
    printf("\n Enter any number: ");
    scanf("%d", pnum);
    if(*pnum>0)
        printf("\n The number is positive");
    else
    {
        if(*pnum<0)
            printf("\n The number is negative");
        else
            printf("\n The number is equal to zero");
    }
    return 0;
}
```

# Pointer Arithmetic

```
int main()
{
    float num1,num2,sum=0.0, *nump1,*nump2, *sump;
    nump1=&num1;
    nump2=&num2;
    sump=&sum;
    scanf("%f %f",nump1,nump2);
    *sump= *nump1+ *nump2;
    printf("the sum of %f\t %f\t is %f\n",*nump1,*nump2,*sump);

    return 0;
}
```

**Output : the sum of 2.5 3.4 is 5.9**



- `#include <stdio.h>`
- `int main()`
- `{`
- `char *ch = "Hello World";`
- `printf("%s", ch);`
- `return 0;`
- `}`

# Passing arguments to functions using Pointers

- Pointers provide a mechanism to modify data declared in one function using code written in another function.
- The calling function sends the address of the variables.
- The called function receives them using Pointers.

# Syntax to be followed

- Formal Parameters are Pointers
- Use dereferencing operator(\*) to use these pointer variables
- The actual arguments are address of the variables

## WAP to add two integers using function

```
#include<stdio.h>
void sum(int*,int*,int *);
int main()
{
    int n1,n2,total;
    printf("Enter the first number\n");
    scanf("%d",&n1);
    printf("Enter the second number\n");
    scanf("%d",&n2);
    sum(&n1,&n2,&total);
    printf("Total is %d \n", total);    return 0;
}
void sum(int *a, int *b, int *s)
{
    *s=*a+*b;
}
```

# Regular functions without pointers

```
sum(1,2);
```

OR

```
sum(x,y);
```

```
void sum(int a, int b)
```

```
{
```

```
    int s;
```

```
    s=a+b;
```

```
    printf(“%d”, s);
```

```
}
```

# Arrays

- `int a[10];`
- `int *x;`
- `x=a;`
- `x++` to access the elements in the array.

# Call by Reference

- Since arguments are not copied into a new variable better time and space efficiency
- The called function can change the value of the argument and the change is reflected in the calling function
- The reference can be used to return multiple values

# Swapping two variables

```
void swap_call_by_val(int,int)
void swap_call_by_ref(int *,int *)
int main()
{
    int a=1,b=2,c=3,d=4;
    printf("In main() a=%d,b=%d\n",a,b);
    swap_call_by_val(a,b);
    printf("In main() a=%d,b=%d\n",a,b);
    printf("In main() c=%d,d=%d\n",c,d);
    swap_call_by_ref(&c,&d);
    printf("In main() c=%d,d=%d\n",c,d);
    return 0;
}
```



# Swapping two variables

```
void swap_call_by_val(int a,int b)
```

```
{
```

```
    int temp;
```

```
    temp=a;
```

```
    a=b;
```

```
    b=temp;
```

```
    printf("In function() a=%d,b=%d\n",a,b);
```

```
}
```

```
void swap_call_by_ref(int *c,int *d)
```

```
{
```

```
    int temp;
```

```
    temp=*c;
```

```
    *c=*d;
```

```
    *d=temp;
```

```
    printf("In function() c=%d,d=%d\n",*c,*d);
```

```
}
```

# Generic Pointers

- A generic pointer is a pointer variable that has void as its data type.
- The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type.
- It is declared like a normal pointer variable but using the void keyword as the pointer's data type.

void \*ptr;

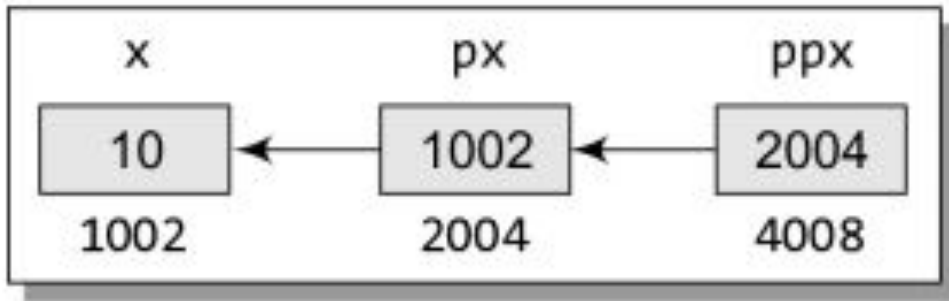
- In C, since you cannot have a variable of type void, the void pointer will therefore not point to any data and, thus, cannot be dereferenced.
- You need to cast a void pointer to another kind of pointer before using it.
- Generic pointers are often used when you want a pointer to point to data of different types at different times

- #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;
- }
- Output
- Generic pointer points to the integer value = 10
- Generic pointer now points to the character = A

# Pointer to Pointers

- you can also use pointers that point to pointers. The pointers in turn point to data or even to other pointers. To declare pointers to pointers, just add an asterisk \* for each level of reference.
- For example, consider the following code:  

```
int x=10;  
int *px, **ppx;  
px = &x;  
ppx = &px;
```
- Let us assume, the memory locations of these variables are as shown in figure.



- Now if we write,  

```
printf("\n %d", **ppx);
```
- Then, it would print **?**, the value of x

# Drawbacks of Pointers

- Although pointers are very useful in C, they are not free from limitations. If used incorrectly, pointers can lead to bugs that are difficult to unearth.
- For example, if you use a pointer to read a memory location but that pointer is pointing to an incorrect location, then you may end up reading a wrong value. An erroneous input always leads to an erroneous output. Thus however efficient your program code may be, the output will always be disastrous. Same is the case when writing a value to a particular memory location

- `#include <stdio.h>`

```
int main() {  
    int x, *px;  
    x=10;  
    px=&x;  
    *px = 20;  
    printf("\n %d", *px);  
    return 0;  
}
```

# Drawbacks of Pointers

```
int x, *px;
```

```
x=10;
```

```
px = x;
```

- Error: It should be `px = &x;`

# Drawbacks of Pointers

```
int x=10, y=20, *px, *py;  
px = &x, py = &y;  
if(px<py)  
    printf("\n x is less than y");  
else  
    printf("\n y is less than x");
```

- What is the error?

# Drawbacks of Pointers

```
int x=10, y=20, *px, *py;
```

```
px = &x, py = &y;
```

```
if(px<py)
```

```
    printf("\n x is less than y");
```

```
else
```

```
    printf("\n y is less than x");
```

- Error: It should be if(\*px< \*py)



# Programs

- Develop a C program to perform arithmetic operations (addition, subtraction, multiplication, division and remainder) on two integers using pointers.

Additional program

- Illustrate pointers in swapping two numbers.

# Basic concepts: Dynamic memory allocation

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

**Array Length = 9**  
**First Index = 0**  
**Last Index = 8**

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C.**

# Dynamic memory allocation

- **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:
  - ❑ malloc()
  - ❑ calloc()
  - ❑ free()
  - ❑ realloc()

# Malloc()

- The “**malloc**” or “**memory allocation**” method in C is used to dynamically allocate a single large block of memory with the specified size.
- It returns a **pointer of type void** which can be cast into a pointer of any form.
- `ptr = (cast-type*) malloc(byte-size)`

```
// Dynamically allocate memory using malloc()  
ptr = (int*)malloc(n * sizeof(int));
```



*Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory.  
And, the pointer ptr holds the address of the first byte in the allocated memory.*

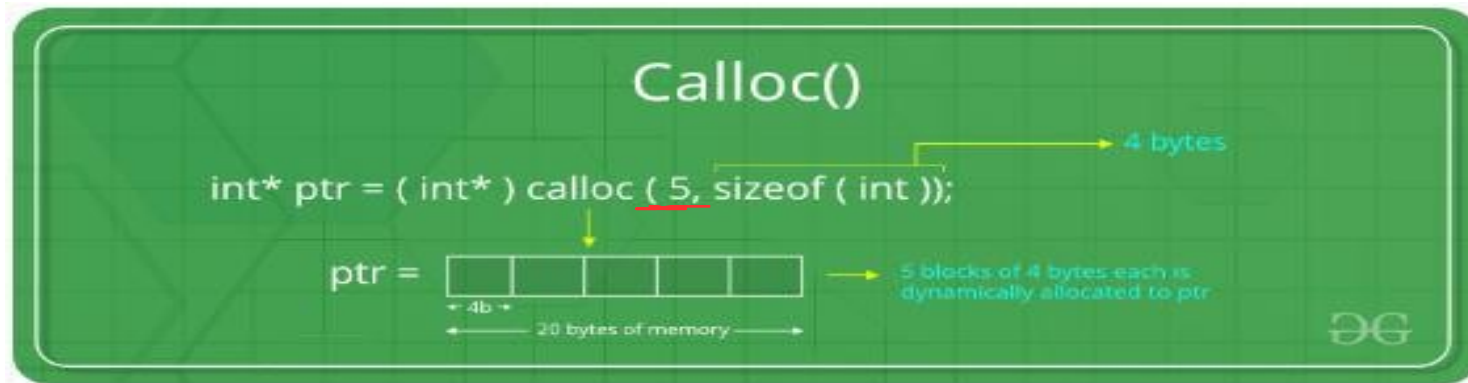
# Calloc()

1. “**calloc**” or “**contiguous allocation**” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

```
ptr = (cast-type*)calloc(n, element-size);
```

here, n is the no. of elements and element-size is the size of each element.

```
// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));
```



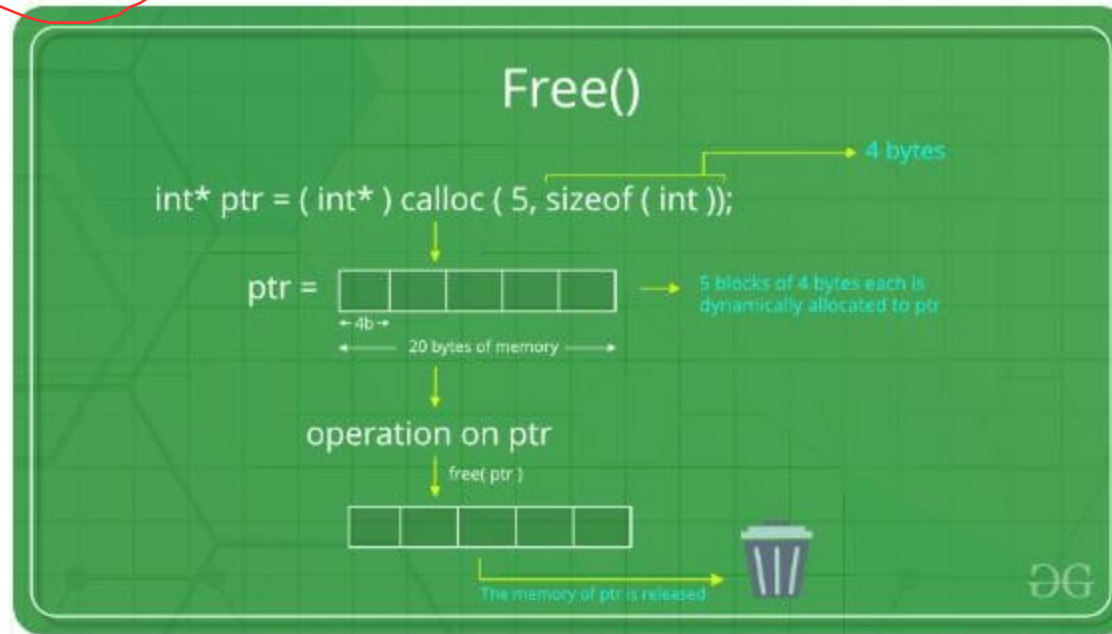
# Calloc()

- Calloc allocates the memory and initializes every byte in the allocated memory to 0.
- In contrast, malloc allocates a memory block of a given size and doesn't initialize the allocated memory
- Calloc() always requires two arguments and malloc() requires only one

# Free()

- “**free**” method in C is used to dynamically de-allocate the memory.
- The memory allocated using functions malloc() and calloc() is not de-allocated on their own.
- Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

```
free(ptr);
```



# Realloc()

“**realloc**” or “**re-allocation**” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

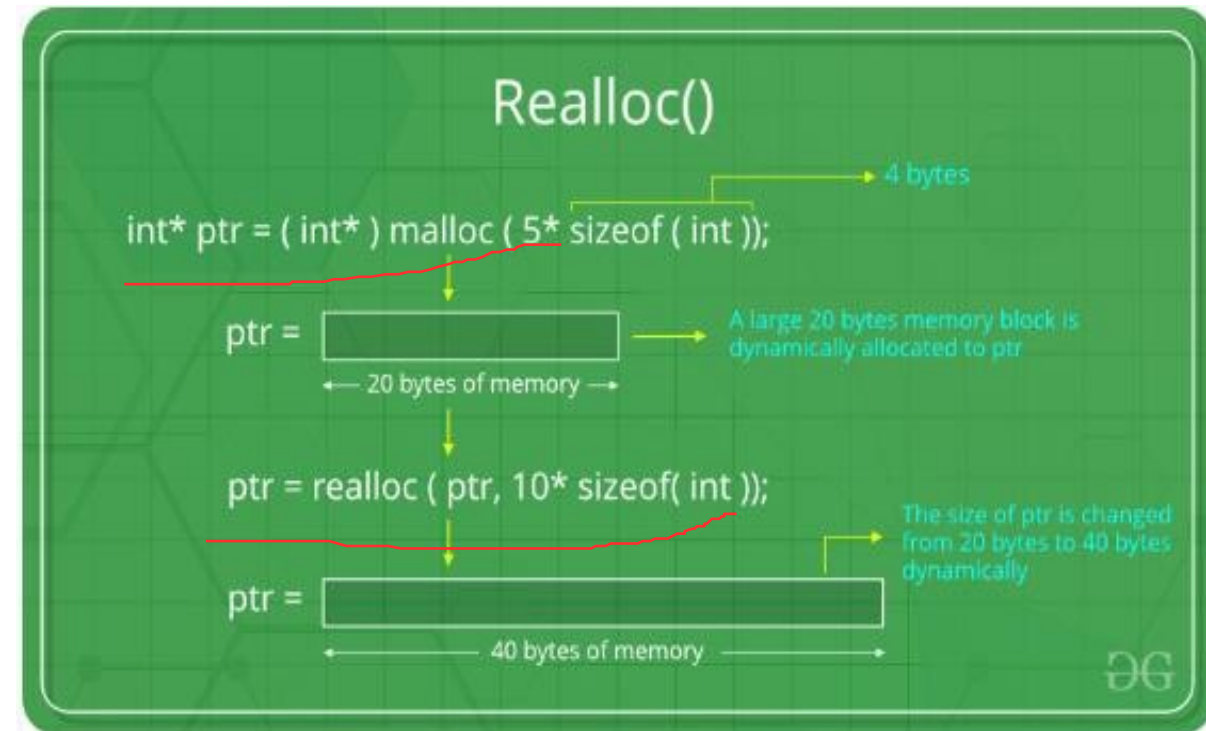
```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.

```
// Dynamically re-allocate memory using realloc()
```

```
ptr = realloc(ptr, n * sizeof(int));
```

Type cast may or may not be required





# Example

```
typedef struct {  
    char * name;  
    int age;  
} person;
```

```
person * myperson = (person *) malloc(sizeof(person));
```

```
myperson->name = "John";  
myperson->age = 27;
```

```
free(myperson);
```

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    //This pointer will hold the base address of the
    //block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory allocated or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else{
        // Memory has been successfully allocated
        printf("Memory successfully allocated using
        calloc.\n");
    }
}

```

```

// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

// Get the new size for the array
n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);

// Dynamically re-allocate memory using realloc()
ptr = (int*)realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

free(ptr);
return 0;
}

```

# Structures

- A structure is in many ways similar to a record. It stores related information about an entity.
- Structure is basically a **user-defined data type** that can store related information (even of different data types) together.
- The major difference between a structure and an array is that an array can store only information of same data type.
- A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types and each has a name that is used to select it from the structure.

# Structures Declaration

- A structure is declared using the keyword `struct` followed by the structure name. All the variables of the structure are declared within the structure. A structure type is generally declared by using the following syntax:

```
struct struct-name {data_type var-name;  
                    data_type var-name;  
                    ..... };
```

For example, if we have to define a structure for a student, then the related information for a student probably would be: `roll_number`, `name`, `course`, and `fees`. This structure can be declared as:

```
struct student { int r_no;  
                 char name[20];  
                 char course[20];  
                 float fees; };
```

# Structures Declaration

```
struct student { int r_no; char name[20];  
                char course[20];  
                float fees; };
```

- The structure has become a user-defined data type.
- Each variable name declared within a structure is called a member of the structure.
- The structure declaration, however, does not allocate any memory or consume storage space. It just gives a **template** that conveys to the C compiler how the structure would be laid out in the memory and also gives the details of member names.

# Structures Declaration

**Like any other data type, memory is allocated for the structure when we declare a variable of the structure.**

For example, we can define a variable of student by writing:

```
struct student stud1;
```

Here, struct student is a data type and stud1 is a variable.

Look at another way of declaring variables. In the following syntax, the variables are declared at the time of structure declaration.

```
struct student { int r_no;  
                char name[20];  
                char course[20];  
                float fees; } stud1, stud2;
```

# Structures Declaration

- Look at another way of declaring variables. In the following syntax, the variables are declared at the time of structure declaration.

```
struct student { int r_no;  
                char name[20];  
                char course[20];  
                float fees; } stud1, stud2;
```

- In this declaration we declare two variables stud1 and stud2 of the structure student. So if you want to declare more than one variable of the structure, then separate the variables using a comma.
- **When we declare variables of the structure, separate memory is allocated for each variable.**
- **NOTE:** care should be taken to ensure that the name of structure and the name of a structure member should not be the same. Moreover, structure name and its variable name should also be different.

# Structures Declaration - typedef

- When we precede a struct name with the typedef keyword, then the struct becomes a new type.

```
typedef struct student { int r_no;  
                        char name[20];  
                        char course[20];  
                        float fees; };
```

- Now that you have preceded the structure's name with the typedef keyword, **student becomes a new data type**.
- Therefore, now you can straightaway declare the variables of this new data type as you declare the variables of type int, float, char, double, etc.
- To declare a variable of structure student, you may write  
student stud1;
- Note that we have not written struct student stud1.



# Initialization of Structures

- The general syntax to initialize a structure variable is as follows:

```
struct struct_name { data_type member_name1;  
                    data_type member_name2;  
                    data_type member_name3;  
                    .....} struct_var = {const1, const2, const3,...};
```

**OR**

```
struct struct_name { data_type member_name1;  
                    data_type member_name2;  
                    data_type member_name3;  
                    ..... };  
struct struct_name struct_var = {constant1, constant2, constant 3,...};
```

# Initialization of Structures

```
struct student { int r_no;  
    char name[20];  
    char course[20];  
    float fees; } stud1 = {01, "Rahul", "BCA", 45000};
```

Or, by writing,

```
struct student stud1 = {01, "Rahul", "BCA", 45000};
```

```
struct student stud1  
= {01, "Rahul", "BCA", 45000};
```

01	Rahul	BCA	45000
r_no	name	course	fees

```
struct student stud2 = {07, "Rajiv"};
```

07	Rajiv	\0	0.0
r_no	name	course	fees

# Accessing the Members of a Structure

- A structure member variable is generally accessed using a '.' (dot) operator.
- The syntax of accessing a structure or a member of a structure can be given as:  
`struct_var.member_name`
- The dot operator is used to select a particular member of the structure.
- For example, to assign values to the individual data members of the structure variable `stud1`:

```
stud1.r_no = 01; stud1.name = "Rahul"; stud1.course = "BCA"; stud1.fees = 45000;
```

- To input values for data members of the structure variable `stud1`, we may write  

```
scanf("%d", &stud1.r_no);  
scanf("%s", stud1.name);
```
- Similarly, to print the values of structure variable `stud1`, we may write  

```
printf("%s", stud1.course);  
printf("%f", stud1.fees);
```
- Of all the operators `→`, `.`, `( )`, and `[]` have the highest priority. This is evident from the following statement  
`stud1.fees++` will be interpreted as `(stud1.fees)++`.

# Problem

- Write a program using structures to read and display the information about a student.

```
#include <stdio.h>
#include <conio.h>
int main()
{
    struct student
    {
        int roll_no;
        char name[80];
        float fees;
        char DOB[80];
    };
    struct student stud1;
    clrscr();
    printf("\n Enter the roll number : ");
    scanf("%d", &stud1.roll_no);
    printf("\n Enter the name : ");
    scanf("%s", stud1.name);
    printf("\n Enter the fees : ");
    scanf("%f", &stud1.fees);
    printf("\n Enter the DOB : ");
    scanf("%s", stud1.DOB);
```

```
printf("\n *****STUDENT'S DETAILS *****");  
    printf("\n ROLL No. = %d", stud1.roll_no);  
    printf("\n NAME = %s", stud1.name);  
    printf("\n FEES = %f", stud1.fees);  
    printf("\n DOB = %s", stud1.DOB);  
    getch();  
    return 0;  
}
```

#### Output

Enter the roll number : 01

Enter the name : Rahul

Enter the fees : 45000

Enter the DOB : 25-09-1991

\*\*\*\*\*STUDENT'S DETAILS \*\*\*\*\*

ROLL No. = 01

NAME = Rahul

FEES = 45000.00

DOB = 25-09-1991

## Problem2

- Write a program to read, display, add, and subtract two complex numbers.
- `#include <stdio.h>`
- `#include <conio.h>`
- `int main()`
- `{`
- `typedef struct complex`
- `{`
- `int real;`
- `int imag;`
- `}COMPLEX;`
- `COMPLEX c1, c2, sum_c, sub_c;`

# NESTED STRUCTURES

typedef struct

```
{    char first_name[20];  
    char mid_name[20];  
    char last_name[20];  
}NAME;
```

typedef struct

```
{    int dd;  
    int mm;  
    int yy;  
}DATE;
```

typedef struct

```
{    int r_no;  
    NAME name;  
    char course[20];  
    DATE DOB;  
    float fees;  
} student;
```



# NESTED STRUCTURES

```
student stud1;  
stud1.r_no = 01;  
stud1.name.first_name = "Rema";  
stud1.name.mid_name = "Raj";  
stud1.name.last_name = "Thareja";  
stud1.course = "BCA";  
stud1.DOB.dd = 15;  
stud1.DOB.mm = 09;  
stud1.DOB.yy = 1990;  
stud1.fees = 45000;
```

## Problem3

- Write a program to read and display the information of a student using a nested structure.

# ARRAYS OF STRUCTURES

```
struct struct_name  
{  
    data_type member_name1;  
    data_type member_name2;  
    data_type member_name3;  
    .....  
};
```

```
struct struct_name struct_var[index];
```

Consider the given structure definition.

```
struct student  
{  
    int r_no;  
    char name[20];  
    char course[20];  
    float fees;  
};
```

A student array can be declared by writing,

```
struct student stud[30];
```

# ARRAYS OF STRUCTURES

To assign values to the *i*th student of the class, we will write

```
stud[i].r_no = 09;
```

```
stud[i].name = "RASHI";
```

Structures and Unions 147

```
stud[i].course = "BE";
```

```
stud[i].fees = 60000;
```

In order to initialize the array of structure variables at the time of declaration, we can write as

follows:

```
struct student stud[3] = { {01, "Aman", "BCA", 45000}, {02, "Aryan", "BCA",  
60000},
```

```
{03, "John", "BCA", 45000} };
```

## Problem4

- Write a program to read and display the information of all the students in a class. Then edit the details of the  $i^{\text{th}}$  student and redisplay the entire information.

# UNIT 1

- **Introduction To Data Structure:** Data Management concepts, Data types – primitive and non-primitive, Types of Data Structures- Linear & Non-Linear Data Structures. Structures and pointers
- **Dynamic memory allocation:** allocating a block of memory: Malloc, allocating multiple blocks of memory: Calloc, Releasing the used space: Free Altering the size of memory: Realloc.

# Basic Terminology

- Our aim is to design good programs,
- where a good program is defined as a program that
  - runs correctly
  - is easy to read and understand
  - is easy to debug and
  - is easy to modify
- A program should undoubtedly give correct results, but along with that it should also run efficiently.
- A program is said to be efficient when it executes in minimum time and with minimum memory space.

# Basic Terminology

- In order to write efficient programs we need to apply certain data management concepts.
- Data structure is a crucial part of data management.
- A data structure is basically a group of data elements that are put together under one name, and which defines a particular way of **storing and organizing** data in a computer so that it can be used efficiently.



# INTRODUCTION

- A **data structure** is a data organization, management, and storage format that enables efficient access and modification.
- Data structures serve as the basis for abstract data types(ADT). The ADT defines the logical form of the data type.
- The data structure implements the physical form of the data type.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

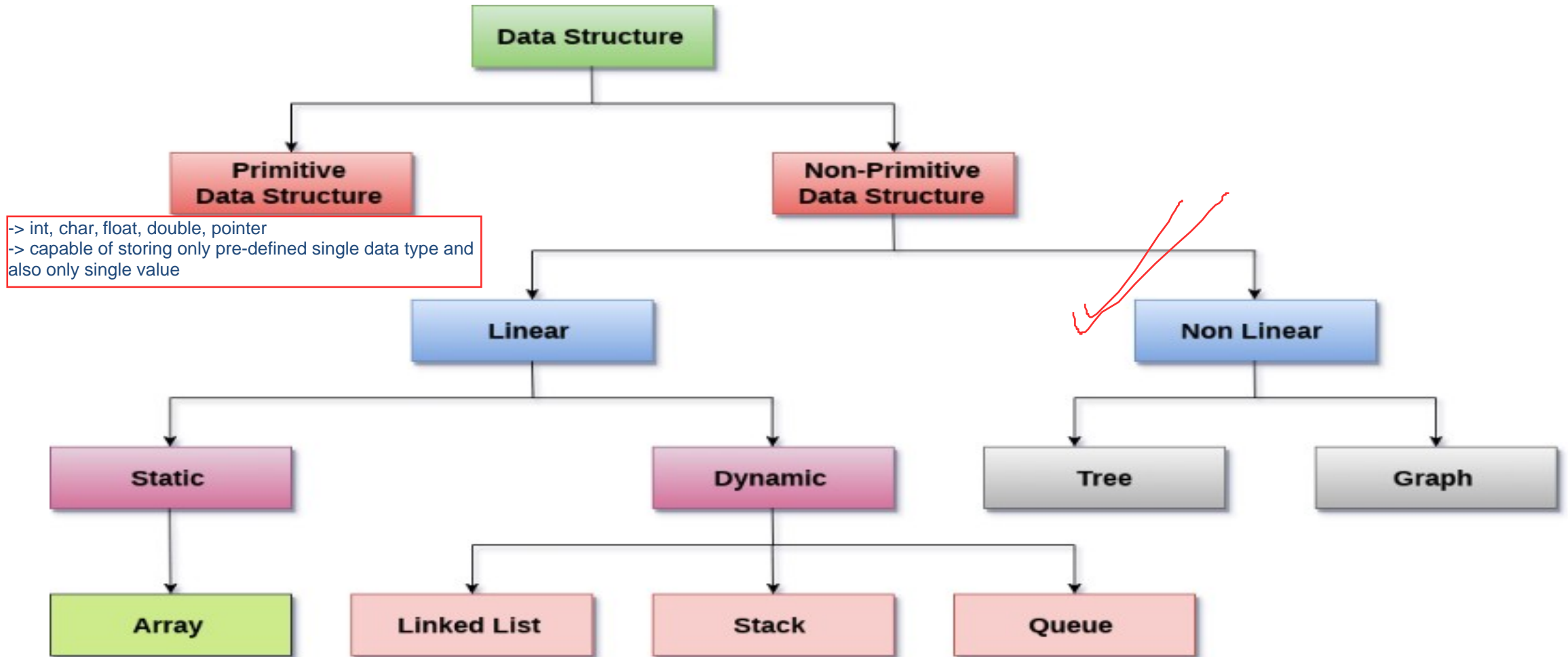
# What is a Data Type?

1. It defines a certain domain of values
  2. It defines operations allowed on those values
- Example: int type
    - Takes only integer values
    - Operations: addition, subtraction, multiplication, etc.

# Abstract Data Types (ADT)

- ADTs are like user defined data types which defines operations on values using functions without specifying what is there inside the function (black box) and how the operations are performed.
- Example: Stack ADT
- A stack consists of elements of some type arranged in a sequential order.
- Operations:
- Initialize() – initializing it to be empty
- Push()- Insert an element into the stack
- Pop() – delete an element from the stack
- isEmpty() – checks if stack is empty
- isFull() – checks if stack is full.
- There are multiple ways to implement an ADT. Example Stack ADT can be implemented using arrays or linked list.

# Data structure classification



# Advantage of Data structures

- Static DS – the memory is allocated at compile time, therefore maximum size is fixed.
  - Advantage: Fast access
  - Disadvantage: Slower insertion and deletion
- Dynamic DS – the memory is allocated at run time, therefore maximum size is flexible.
  - Advantage: Faster insertion and deletion
  - Disadvantage: Slower access

## Advantage of Data structures

- Efficiency – proper choice of data structures makes program efficient in terms of space and time.
- Reusability- one implementation can be used by multiple client program.
- Abstraction- data structure is specified by an ADT which provides a level of abstraction. The client program doesn't have to worry about the implementation details.


# Operations on Data Structures

- Traversing
- Insertion
- Deletion
- Searching
- Sorting
- Merging

# Example

- Which data structure is used in implementing redo and undo feature?
- Which data structure is used to store bitmap image?
- Storing the friendship information on social networking site.



A red hand-drawn scribble, resembling a stylized 'v' or a checkmark, is positioned to the left of the word 'Example'.

# Example

- Which data structure is used in implementing redo and undo feature? STACK
- Which data structure is used to store bitmap image? ARRAY
- Storing the friendship information on social networking site. GRAPH