

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Arrays, Initialization and Traversal**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyze and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

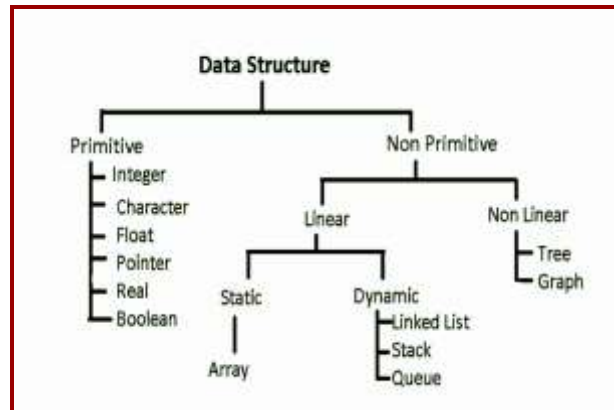
Jan - May, 2022

Dept. of CSE,

PES University

## Data Structures

Data structures are **used** to **store data** in an **organized** and **efficient** manner so that the **retrieval is efficient**. Classification of data structures in C is as below.



When solving problems, it is important to visualize the data related to the problem. Sometimes the data consist of just a single number (as we discussed in unit-1, primitive types). At other times, the data may be a coordinate in a plane that can be represented as a pair of numbers, with one number representing the x-coordinate and the other number representing the y-coordinate. There are also times when we want to work with a set of similar data values, but we do not want to give each value a separate name. For example, we want to read marks of 1000 students and perform several computations. To store marks of thousand students, we do not want to use 1000 locations with 1000 names. To store group of values using a single identifier, we use a data structure called an Array.

**An array is a linear data structure, which is a finite collection of similar data items stored in successive or consecutive memory locations. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. It can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.**

To create an array, define the data type and specify the name of the array followed by square brackets [] with size mentioned in it. **Example: int s[100];**

### **Characteristics/Properties of arrays:**

- Non-primary data or secondary data type
- Memory allocation is contiguous in nature
- Elements need not be unique.
- Demands same /homogenous types of elements
- Random access of elements in array is possible
- Elements are accessed using index/subscript
- Index or subscript starts from 0
- Memory is allocated at compile time.
- Size of the array is fixed at compile time. Returns the number of bytes occupied by the array.
- Cannot change the size at runtime.
- Arrays are assignment incompatible.
- Accessing elements of the array outside the bound will have undefined behaviour at runtime.

### **Types of Array**

Broadly classified into two categories as

#### **Category 1:**

1. Fixed Length Array: Size of the array is fixed at compile time
2. Variable Length Array - Not supported by older few C standards. Better to use dynamic memory allocation functions than variable length array.

#### **Category 2:**

1. One Dimensional Array
2. Multi-Dimensional Array

## One dimensional Array

- One dimensional array is also called as linear array(also called as 1-D array).
- The one dimensional array stores the data elements in a single row or column.

### Array Declaration:

**Syntax:** `Data_type Array_name[Size];` // Declaration syntax

- `Data_type` : Specifies the type of the element that will be contained in the array
- `Size`: Indicates the maximum number of elements that can be stored inside the array
- `Array_name`: Identifier to identify a variable.
- Subscripts in array can be integer constant or integer variable or expression that yields integer
- C performs no bound checking – care should be taken to ensure that the array indices are within the declared limits

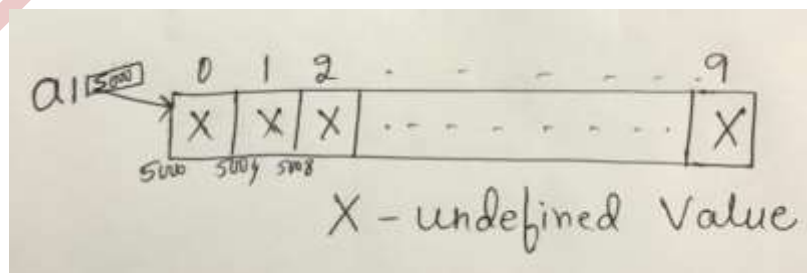
### Examples:

- `float average [6];` // Can contain 6 floating point elements, 0 to 5 are valid array indices
- `char name[20];` // Can contain 20 char elements, 0 to 19 are valid array indices
- `double x[15];` // Can contain 15 elements of type double, 0 to 14 are valid array indices.

Consider `int a1[10];`

Declaration allocates number of bytes in a contiguous manner based on the size specified. All these memory locations are filled with undefined values. If the starting address is 0x5000 and if the size of integer is 4 bytes, refer the diagrams below to understand this declaration clearly. Number of bytes allocated = size specified\*size of integer i.e,  $10 \times 4$ .

`printf("size of array is %d\n", sizeof(a1));` // 40 bytes



Address of the first element is called the Base address of the array. Address of  $i^{\text{th}}$  element of the array can be found using formula: **Address of  $i^{\text{th}}$  element = Base address + (size of each element \* i)**

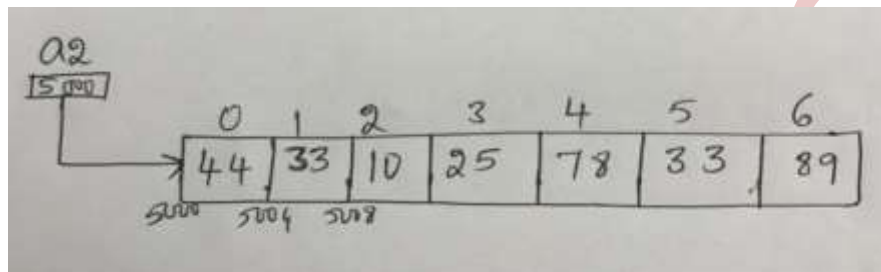
### Array Initialization:

- After an array is declared it must be initialized.
- An Uninitialized array will contain undefined values/junk values.
- An array can be initialized at either compile time or at runtime

Consider `int a2[ ] = {44,33,10,25,78,33,89};`

Above initialization does not specify the size of the array. Size of the array is based on the number of elements stored in it and the size of type of elements stored. So, the above array occupies  $7 \times 4 = 28$  bytes of memory in a contiguous manner.

`printf("size of array is %d\n", sizeof(a2));` //28 bytes



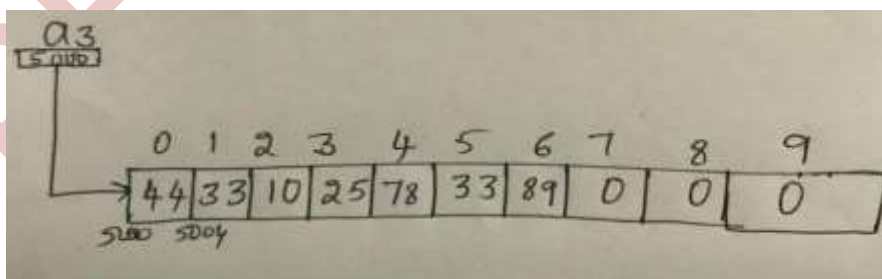
### Partial Initialization:

Consider `int a3[10] = {44,33,10,25,78,33,89};`

Size of the array is specified. But only few elements are stored. Now, the size of the array is  $10 \times 4 = 40$  bytes of memory in a contiguous manner. **All uninitialized memory locations in the array are filled with default value 0.**

`printf("size of array is %d\n", sizeof(a3));` //40 bytes

`printf("%p %p\n", a3, &a3);` // Must be different. But shows same address



### Compile time Array initialization:

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

Syntax: data-type array-name[size] = { list of values };

```
int marks[4]={ 67, 87, 56, 77 }; // integer array initialization
float area[5]={ 23.4, 6.8, 5.5 }; // float array initialization
intarr[] = { 2, 3, 4};
int marks[4]={ 67,87,56,77,5 } //undefined behavior
```

### Variable length array:

Variable length arrays are also known as runtime sized or variable sized arrays. The size of such arrays is defined at run-time.

The variable length arrays are always unsafe. This dynamically creates stack memory based on data size, which can cause overflow of stack frame.

### Designated initializers (C99):

- It is often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values:

```
int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48};
```

- So, we want element 2 to be 29, 9 to be 7 and 14 to be 48 and the other values to be zeros. For large arrays, writing an initializer in this fashion is tedious.
- C99's designated initializers

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- Each number in the brackets is said to be a designator.
- Besides being shorter and easier to read, designated initializers have another advantage: the order in which the elements are listed no longer matters. The previous expression can be written as:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

### Designators must be constant expressions:

- If the array being initialized has length n, each designator must be between 0 and n-1.
- if the length of the array is omitted, a designator can be any non-negative integer.
- In that case, the compiler will deduce the length of the array by looking at the largest designator.

```
int b[] = {[2] = 6, [23]=87};
```

- Because 23 has appeared as a designator, the compiler will decide the length of this array to be 24.
- An initializer can use both the older technique and the later technique.

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8]=6};
```

**Runtime initialization:** Using a loop and input function in c

Example:

```
inta[5];
for(int i=0;i<5;i++)
{
    scanf("%d",&a[i]);
}
```

## Traversal of the Array:

**Accessing each element of the array** is known as traversing the array.

Consider int a1[10];

- How do you access the 5<sup>th</sup> element of the array? **a1[4]**
- How do you display each element of the array?

```
int i;
```

```
for(i = 0; i<10;i++)
```

```
    printf("%d\t",a1[i]);//Using the index operator [ ].
```

```
//Above code prints undefined values as a1 is just declared.
```

Consider `int a2[ ] = {12,22,44,14,77,911};`

```
int i;
for(i = 0; i<10;i++)
    printf("%d\t",a2[i]);//Using the index operator [ ].
```

Above code prints initialized values when `i` is between 0 to 5. When `i` becomes 6, `a2[6]` is outside bound as the size of `a2` is fixed at compile time and it is `6*4`(size of `int` is implementation specific) = 24 bytes

**Anytime accessing elements outside the array bound is an undefined behavior.**

**Coding Example\_1:** Let us consider the program to read 10 elements from the user and display those 10 elements.

```
#include<stdio.h>

int main()
{
    int arr[100];
    int n;
    printf("enter the number of elements\n");
    scanf("%d",&n);
    printf("enter %d elements\n",n);
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]);
        i++;
    }
    printf("entered elements are\n");
    i = 0;
    while(i<n)
    {
        printf("%d\n",arr[i]);
        i++;
    }
}
```



```
return 0;
}
```

### Output:

```
enter the number of elements
3
enter 3 elements
34
56
89
entered elements are
34
56
89

...Program finished with exit code 0
Press ENTER to exit console.
```

**Coding Example\_2:** Let us consider the program to find the sum of the elements of an array.

```
int arr[] = {-1,4,3,1,7};
```

```
int i;
```

```
int sum = 0;
```

```
int n = sizeof(arr)/sizeof(arr[0]);
```

```
for(i = 0; i<n; i++)
```

```
{    sum += arr[i];
```

```
}
```

```
printf("sum of elements of the array is %d\n", sum);
```

Think about writing a function to find the sum of elements of the array. Few questions to think about above codes?

- Should I use while only? for and while are same in C except for syntax.
- Can I use arr[n] while declaration? Variable length array
- Can we write read and display user defined functions to do the same? Yes. To do this, we should understand functions and how to pass array to a function.

Happy Coding using Arrays!!

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Pointers**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyze and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

## Pointers

- Pointer is a variable which contains the address. This address is the location of another object in the memory
- Pointers can be used to access and manipulate data stored in memory.
- Pointer of particular type can point to address any value of that particular type.
- Size of pointer of any type is same /constant in that system.
- Not all pointers actually contain an address

Example: NULL pointer // Value of NULL pointer is 0.

### Pointer can have three kinds of contents in it

1. The address of an object, which can be de referenced.
2. A NULL pointer
3. Undefined value // If p is a pointer to integer, then – int \*p;

**Note:** A **pointer** is a **variable** that **stores** the **memory address** of **another variable** as **its value**. The **address of the variable we are working with** is **assigned to the pointer**

Memory	Address
x = 5	0x0
y = 5	0x1
p = 0x0	0x2
	0x3
	0x4
	0x5
	0x6
	0x7
	0x8
	0x9

```

#include <stdio.h>

int main( int argc, char *argv[] ) {
    int x, y;
    int *p;

    x = 5;
    p = &x;
    y = *p; /* same as y = x */

    return 0;
}

```

We can get the value of the variable the pointer currently points to, by dereferencing pointer by using the \* operator. When used in declaration like int\* ptr, it creates a pointer variable. When not used in declaration, it act as a dereference operator w.r.t pointers

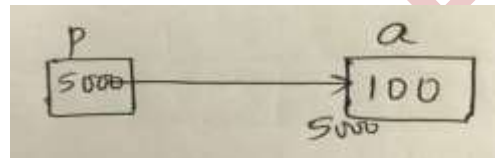
### Pointer Declaration:

Syntax: Data-type \*name;

Example: `int *p;` // Compiler assumes that any address that it holds points to an integer type.  
`p = &sum;` // Memory address of sum variable is stored into p.

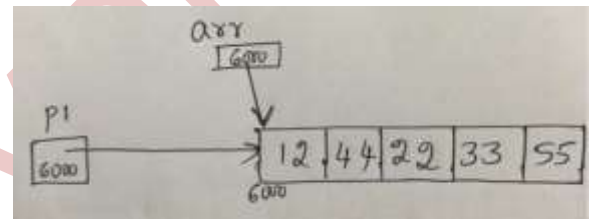
### Coding Example\_1:

```
int *p; // p can point to anything where integer is stored. int* is the type. Not just int.
int a = 100;
p = &a;
printf("a is %d and *p is %d", a, *p);
```



### Coding Example\_2: Pointer pointing to an array

```
Now, int arr[] = {12,44,22,33,55};
int *p1 = arr; // same as int *p1;
p1 = arr; // same as int *p1; p1 = &arr[0];
int arr2[10];
arra2 = arr; // Arrays are assignment incompatible. Compile time Error
```



### Pointer Arithmetic:

Below arithmetic operations are allowed on pointers

- Add an int to a pointer
- Subtract an int from a pointer
- Difference of two pointers when they point to the same array.

**Integer is not same as pointer.** We get warning when we try to compile the code where integer is stored in variable of int\* type.

### Coding Example\_3:

```
int arr[] = {12,33,44};
int *p2 = arr;
printf("before increment %p %d\n", p2, *p2);
```

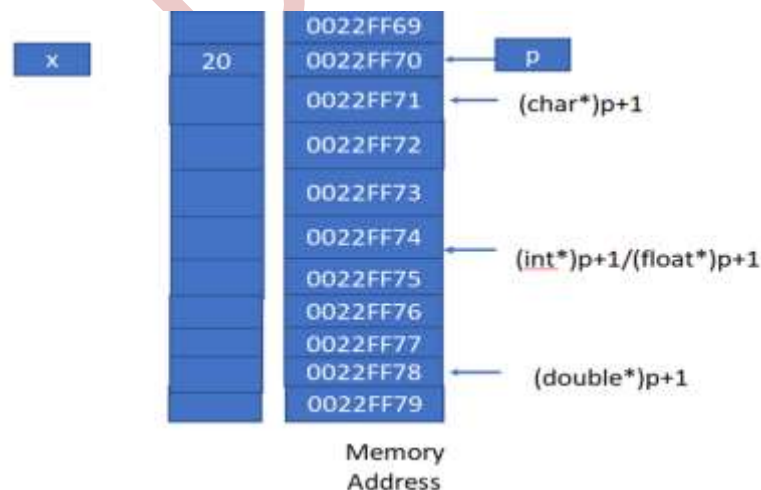
```
p2++; //same as p2 = p2+1
// This means 5000+sizeof(every element)*1 if 5000 is the base address
//increment the pointer by 1. p2 is now pointing to next location.
printf("after increment %p %d\n",p2, *p2);
```

#### Coding Example\_4: Example on Pointer Arithmetic :

```
int *p, x = 20;
p = &x;
printf("p = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

#### Sample output:

```
p = 0022FF70
p+1 = 0022FF74
p+1 = 0022FF71
p+1 = 0022FF74
p+1 = 0022FF78
```



**Coding Example\_5:**

```
int main()
{
    int *p;
    int a = 10;
    p = &a;
    printf("%d\n",(*p)+1);    // 11 ,p is not changed
    printf("before *p++ %p\n",p);    //address of p
    printf("%d\n",*p++);    // same as *p and then p++ i.e 10
    printf("after *p++ %p\n",p);
    //address incremented by the size of type of value stored in it
    return(0);
}
```

**Coding Example\_6:**

```
int main()
{
    int *p;
    int a = 10;
    p = &a;
    printf("%d\n",*p);//10
    printf("%d\n",(*p)++);// 10 value of p is used and then value of p is incremented
    printf("%d\n",*p); // 11
    return 0;
}
```



**Array Traversal using pointers:****Version 1: Index operator can be applied on pointer. Array notation**

```
for(i = 0;i<5;i++)  
    printf("%d \t",p3[i]); // 12 44 22 33 55  
// every iteration added i to p3 .p3 not modified
```

**Version 2: Using pointer notation**

```
for(i = 0;i<5;i++)  
    printf("%d\t",*(p3+i));    // 12  44    22    33    55  
// every iteration i value is added to p3 and content at that address is printed.  
// p3 not modified
```

**Version 3:**

```
for(i = 0;i<5;i++)  
    printf("%d \t",*p3++); // 12 44 22 33 55  
// Use p3, then increment, every iteration p3 is incremented.
```

**Version 4: undefined behavior if you try to access outside bound**

```
for(i = 0;i<5;i++)  
    printf("%d \t",*++p3); // 44  22    33    55    undefined value  
// every iteration p3 is incremented.
```

**Version 5:**

```
for(i = 0;i<5;i++)  
    printf("%d \t",(*p3)++); // 12 13 14 15 16  
// every iteration value at p3 is used and then incremented.
```

**Version 6:**

```
for(i = 0;i<5;i++,p3++)  
    printf("%d \t",*p3); // 12  44  22  33  55  
// every iteration value at p3 is used and then p3 is incremented.
```

**Version 7: p3 and arr has same base address of the array stored in it. But array is a constant pointer. It cannot point to anything in the world.**

```
for(i = 0;i<5;i++)  
    printf("%d\t", *arr++); // Compile Time Error
```

### Arrays and Pointers:

An array during compile time is an actual array but degenerates to a constant pointer during run time. Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

#### Coding Example\_8:

```
int *p1;  
float *f1 ;  
char *c1;  
printf("%d%d%d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all  
int a[] = {22,11,44,5};  
int *p = a;  
a++; // Error constant pointer  
p++; // Fine  
p[1] = 222; // allowed  
a[1] = 222 ; // Fine
```

**Note: If variable i is used in loop for the traversal, a[i], \*(a+i), p[i], \*(p+i), i[a], i[p] are all same.**

### Differences between array and pointer:

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. String literal initialization of a character array – Will be discussed in detail in next lecture



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

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

```
int a[10];  
int *p;  
p=a; //allowed  
a=p; //not allowed
```

5. An arithmetic operation on pointer variable is allowed.

```
int a[10];  
int *p;  
p++; /*allowed*/  
a++; /*not allowed*/
```

**Happy Coding using Arrays and Pointers!!**

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Arrays and function**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

## Array and functions

When array is passed as an argument to a function, arguments are copied to parameters of the function and parameters are always pointers. **Array degenerates to a pointer at runtime.** All the operations that are valid for pointer will be applicable for array too in the body of the function. **Function call happens always at run time.**

**Coding Example\_1: Using functions read n elements into an array and display it. Also include a function to find the sum of all the elements in the array**

**Version 1: n is a global variable**

```
#include<stdio.h>
void read_array(int[]);
int n; // global variable
void display_array(int[]);
int find_sum(int[]);
int main()
{
    int a[100];
    printf("how many elements u want to add\n");
    scanf("%d",&n);
    printf("enter %d elements\n",n);
    read_array(a);
    printf("entered elements are\n");
    display_array(a);
    printf("\nsum is %d\n",find_sum(a));
    return 0;
}
void read_array(int a[])
{
    for(int i= 0; i<n;i++)
    {
```

```
        scanf("%d",&a[i]);
    }
}
void display_array(int a[])
{
    for(int i= 0; i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
int find_sum(int a[])
{
    int sum = 0;
    for(int i= 0; i<n;i++)
    {
        sum = sum+a[i];
    }
    return sum;
}
```

**Version 2: n is local to a main function. Can other functions access n then? It throws Compile time Error**

**Version 3: As the array becomes pointer at runtime, finding the size of the passed argument to any function is same as finding the size of the pointer.**

```
#include<stdio.h>
void read_array(int[]);
int n; // global variable
void display_array(int[]);
int find_sum(int[]);
int main()
{
```

```
int a[100];
printf("how many elements u want to add\n");
scanf("%d",&n);
printf("sizeof a is %d\n",sizeof(a)); // in my system, 400 bytes
printf("enter %d elements\n",n);
read_array(a);
...
return 0;
}
void read_array(int a[])
{
    printf("inside read_array sizeof a is %d\n",sizeof(a));
    //4 bytes or constant value for any pointer
    for(int i= 0; i<n;i++)
    {
        scanf("%d",&a[i]);
    }
}
```

**Version 4: As n is local to main function, send this to functions as an argument**

```
#include<stdio.h>
void read_array(int[],int);
void display_array(int[],int);
int find_sum(int[],int);
void increment(int a[],int n);
int main()
{
    int a[100];
    int n;
    printf("how many elements u want to add\n");
    scanf("%d",&n);
```

```
printf("enter %d elements\n",n);
printf("sizeof a is %d\n",sizeof(a));
read_array(a,n);
printf("entered elements are\n");
display_array(a,n);
printf("\nsum is %d\n",find_sum(a,n));
increment(a,n);
printf("array with updated elements are\n");
display_array(a,n);
return 0;
}
void read_array(int a[],int n)
{
    for(int i= 0; i<n;i++)
    {
        scanf("%d",&a[i]);
    }
}
void display_array(int a[],int n)
{
    for(int i= 0; i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
int find_sum(int a[],int n)
{
    int sum = 0;
    for(int i= 0; i<n;i++)
    {
        sum = sum+a[i];
    }
}
```

```
    }  
    return sum;  
}
```

**Version 5: Using the pointer in the parameter makes more sense as array become pointer at runtime during function call.**

```
#include<stdio.h>  
void read_array(int*,int);  
void display_array(int*,int);  
int find_sum(int*,int); // observe this  
int main()  
{  
    ...  
    void read_array(int *a,int n) // this too  
    {  
        ...  
    }  
    void display_array(int *a,int n)  
    {  
        ...  
    }  
    int find_sum(int *a,int n)  
    {  
        ...  
    }
```

**Version 6:** The display and find\_sum functions should not be allowed to make any changes to the array sent in the argument. But it is possible now.

```
#include<stdio.h>  
void read_array(int*,int);  
void display_array(int*,int);  
int find_sum(int*,int);  
int main()  
{  
    ...  
    void read_array(int *a,int n)  
    {  
        ...  
    }  
  
    void display_array(int *a,int n)
```

```
{    a[4] = 8989;    // allowed    }  
int find_sum(int *a,int n)  
{    ...    }
```

PES University



**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Functions in C**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

## Introduction

In case we want to repeat the set of tasks or instructions, we may have two options:

- Use the same set of statements every time we want to perform the task
- Create a function to perform that task, and just call it every time when we need to perform that task, is usually a good practice and a good programmer always uses functions while writing code in C

Functions **break large computing tasks into smaller ones** and enable people to build on what others have done instead of starting from scratch. In programming, **Function is a subprogram to carry out a specific task**. A function is a self-contained block of code that performs a particular task. Once the function is designed, it can be **treated as a black box**. The inner details of operation are invisible to rest of the program. Functions are useful because of following reasons:

- **To improve the readability** of code.
- **Improves the reusability** of the code, same function can be used in any program rather than writing the same code from scratch.
- **Debugging of the code would be easier** if we use functions, as errors are easily traced.
- **Reduces the size of the code**, redundant set of statements are replaced by function calls.

## Types of functions

C functions can be broadly classified into two categories:

### Library Functions

Functions which are defined by C library. Examples include printf(), scanf(), strcat() etc. We just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

### User-defined Functions

Functions which are defined by the developer at the time of writing program. Developer can make changes in the implementation as and when he/she wants. It reduces the complexity of a big program and optimizes the code, supporting Modular Programming Paradigm

In order to make use of user defined functions, we need to understand three key terms associated with functions: **Function Definition, Function call and Function Declaration.**

## Function Definition

Each function definition has the form

```
return_type function_name(parameters) // parameters optional
{
    // declarations and statements
}
```

A function definition should have a return type. The return type must match with what that function returns at the end of the function execution. If the function is not designed to return anything, then the type must be mentioned as void. Like variables, function name is also an identifier and hence must follow the rules of an identifier. Parameters list is optional. If more than one parameter, it must be comma separated. Each parameter is declared with its type and parameters receive the data sent during the function call. If function has parameters or not, but the parentheses is must. Block of statements inside the function or body of the function must be inside { and }. There is no indentation requirement as far as the syntax of C is considered. For readability purpose, it is a good practice to indent the body of the function.

Function definitions can occur in any order in the source file and the source program can be split into multiple files. One must also notice that only one value can be returned from the function, when called by name. There can also be side effects while using functions in c

```
int sum(int a, int b)
{
    return a+b;
}
```

```
int decrement(int y)
{
    return y-1;
}
```

```
void disp_hello()    // This function doesn't return anything
{
    printf("Hello Friends\n");
}

double use_pow(int x)
{
    return pow(x,3);    // using pow() function is not good practice.
}

int fun1(int a, int)    // Error in function definition.
{ }

int fun2(int a, b)    // Invalid Again.
{ }
```

## Function Call

**Function-name(list of arguments);** // semicolon compulsory and Arguments depends on the number of parameters in the function definition

A function can be called by using **function name followed by list of arguments** (if any) enclosed in parentheses. The function which calls other function is known to be Caller and the function which is getting called is known to be the Callee. **The arguments must match the parameters in the function definition in it's type, order and number. Multiple arguments must be separated by comma. Arguments can be any expression in C. Need not be always just variables. A function call is an expression. When a function is called, Activation record is created. Activation record is another name for Stack Frame. It is composed of:**

- Local variables of the callee
- Return address to the caller
- Location to store return
- value Parameters of the
- callee Temporary variables

**The order in which the arguments are evaluated in a function call is not defined** and is determined by the calling convention(out of the scope of this notes) used by the compiler. It is left to the compiler Writer. **Always arguments are copied to the corresponding parameters.** Then the control is transferred to the called function. Body of the function gets executed. When all the statements are executed, callee returns to the caller. OR when there is return statement, the expression of return is evaluated and then callee returns to the caller. If the return type is void, function must not have return statement inside the function.

### Coding Example\_1:

```
#include<stdio.h>

int main()
{
    int x = 100;
    int y = 10;
    int answer = sum(x,y);
    printf("sum is %d\n",answer);
    answer = decrement(x);
    printf("decremented value is %d\n",answer);
    disp_hello();
    double ans = use_pow(x);
    printf("ans is %lf\n",ans);
    answer = sum(x+6,y);
    printf("answer is %d\n", answer);
    printf("power : %lf\n", use_power(5));
    return 0;
}
```

### Function Declaration/ Prototype

All functions must be declared before they are invoked. The function declaration is as follows.

**return\_type Function\_name (parameters list);** // semicolon compulsory

A function may or may not accept any argument. A function may or may not return any value

directly when called by name. There are different type of functions based on the arguments and return type.

- Function without arguments and without return value
- Function without arguments and with return value
- Function with arguments and without return value
- Function with arguments and with return value

**The parameters list must be separated by commas. The parameter names do not need to be the same in declaration and the function definition. The types must match the type of parameters in the function definition in number and order. Use of identifiers in the declaration is optional. When the declared types do not match with the types in the function definition, compiler will produce an error.**

### Parameter Passing in C

**Parameter passing is always by Value in C.** Argument is copied to the corresponding parameter. The parameter is not copied back to the argument. It is possible to copy back the parameter to argument only if the argument is l- value. Argument is not affected if we change the parameter inside a function.

**Types of parameters:** Actual Parameter/Arguments and Formal Parameters

**Actual parameters** are values or variables containing valid values that are passed to a function when it is invoked or called. **Formal parameters** are the variables defined by the function that receives values when the function is called.

#### Coding Example\_2:

```
void fun1(int a1); // declaration
void main()
{
    int a1 = 100;
    printf("before function call a1 is %d\n", a1); // a1 is 100
    fun1(a1);    // call
```

## Problem Solving With C – UE23CS151B

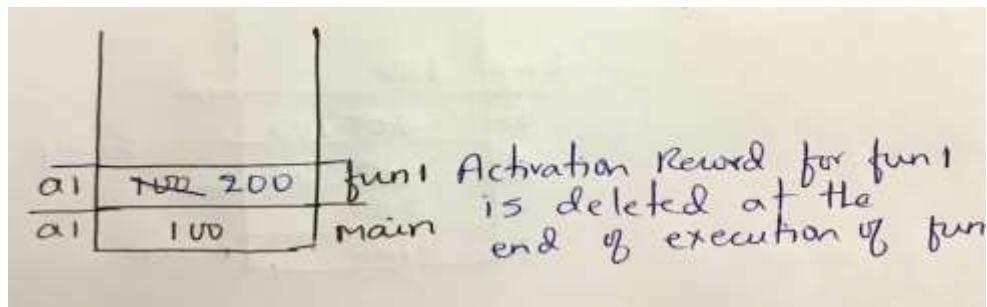
```

printf("after function call a1 is %d\n", a1); // a1 is 100
return 0;
}

void fun1(int a1)
{
    printf("a1 in fun1 before changing %d\n", a1); //100 a1 = 200;
    printf("a1 in fun1 after changing %d\n", a1); //200
}

```

a1 has not changed in main function. The parameter a1 in fun1 is a copy of a1 from main function. Refer to the below diagram to understand activation record creation and deletion to know this program output in detail.



Think about this Question: Is it impossible to change the argument by calling a function? Yes – by passing the l-value

### Coding Example\_3:

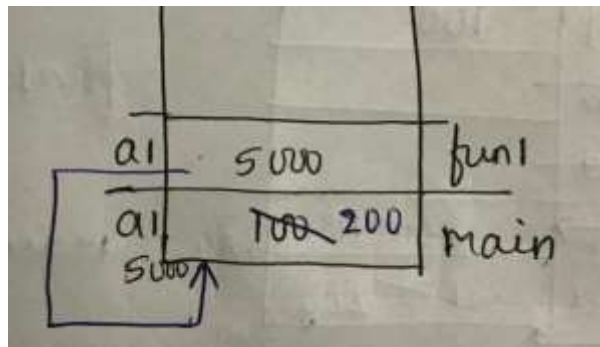
```

void fun1(int *a1);

int main()
{
    int a1 = 100;
    printf("before function call a1 is %d\n", a1); // 100
    fun1(&a1); // call
    printf("after function call a1 is %d\n", a1); // 200
}

```

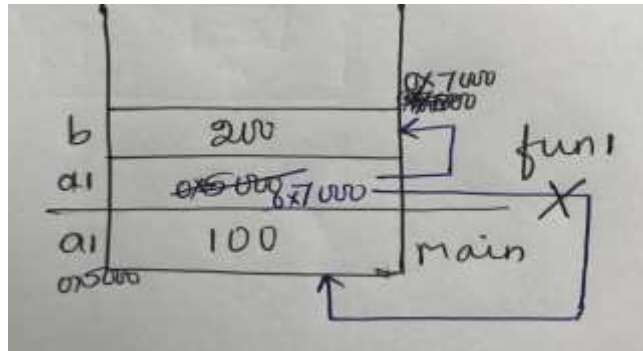
```
void fun1(int *a1)
{
    printf("**a1 in fun1 before changing %d\n", *a1);    //100
    *a1 = 200;
    printf("**a1 in fun1 after changing %d\n", *a1);    //200
}
```



### Coding Example\_3:

```
void fun1(int *a1);
int main()
{
    int a1 = 100;
    printf("before function call a1 is %d\n", a1); // 100
    fun1(&a1);    // call
    printf("after function call a1 is %d\n", a1); // 100
}
void fun1(int *a1)
{
    int b = 200;
    printf("**a1 in fun1 before changing %d\n", *a1);    //100
    a1 = &b;
    printf("**a1 in fun1 after changing %d\n", *a1);    //200
}
```





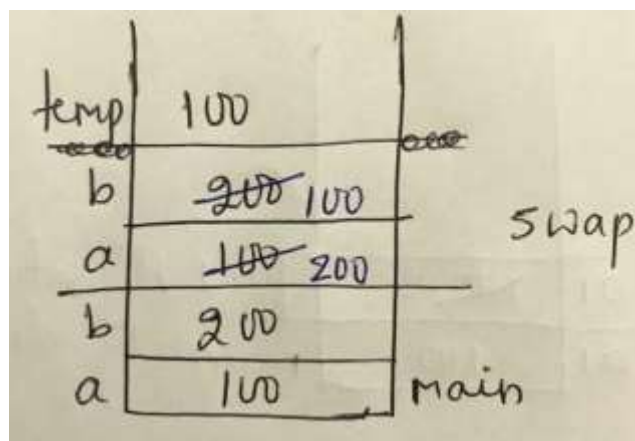
Shall we write the function to swap two numbers and test this function?

#### Coding Example\_4:

**Version – 1:** Is this right version?

```
void swap(int a, int b)
{
    int temp = a; a = b; b = temp;
}

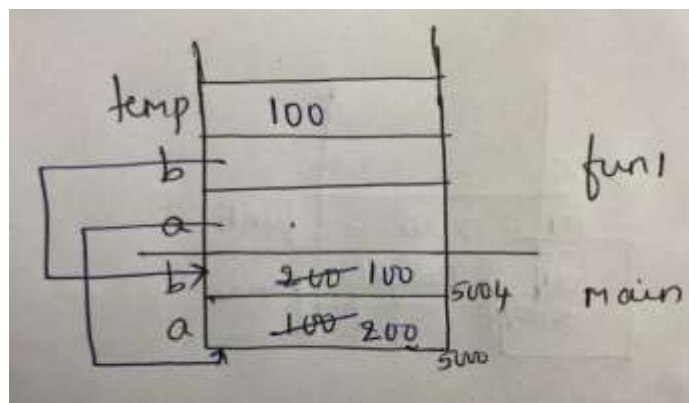
int main()
{
    int a = 100; int b = 200;
    printf("before call a is %d and b is %d\n", a, b);    // a is 100 and b is 200
    swap(a, b);
    printf("after call a is %d and b is %d\n", a, b);    // a is 100 and b is 200
    return 0;
}
```



**Version 2:**

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int a = 100;
    int b = 200;
    printf("before call a is %d and b is %d\n", a, b);    // a is 100 and b is 200
    swap(&a, &b);
    printf("after call a is %d and b is %d\n", a, b);    // a is 100 and b is 200
    return 0;
}
```



**return keyword in C**

Function must do what it is supposed to do. It should not do something extra. For Example, refer to the below code.

```
int add(int a, int b)
{
    return a+b;
}
```

The function `add()` is used to add two numbers. It should not print the sum inside the function. We cannot use this kind of functions repeatedly if it does something extra than what is required. Here comes the usage of `return` keyword inside a function. **Syntax: `return expression;`**

**In 'C', function returns a single value. The expression of return is evaluated and copied to the temporary location by the called function. This temporary location does not have any name. If the return type of the function is not same as the expression of return in the function definition, the expression is cast to the return type before copying to the temporary location. The calling function will pick up the value from this temporary location and use it.**

**Coding Example\_5:**

```
void f1(int);
void f2(int*);
void f3(int);
void f4(int*); i
nt* f5(int* );
int* f6();
int main()
{   int x = 100;
    f1(x);
    printf("x is %d\n", x); // 100
    double y = 6.5;
    f1(y); // observe what happens when double value is passed as argument to integer
parameter?
    printf("y is %lf\n", y); //      6.500000
    int *p = &x; // pointer variable
    f2(p);
    printf("x is %d and *p is %d\n", x, *p);    // 100 100
    f3(*p);
    printf("x is %d and *p is %d\n", x, *p);    // 100 100
    f4(p);
    printf("x is %d and *p is %d\n", x, *p, p);
```

## Problem Solving With C – UE23CS151B

```

int z= 10;
p =f 5(&z);
printf("z is %d and %d\n", *p, z);    // 10    10
p = f6();
printf("*p is %d \n", *p);
return 0;
}

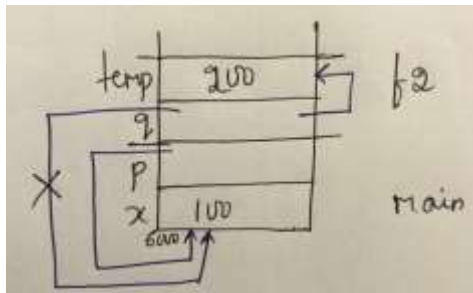
```

```

void f1(int x)
{
    x = 20;
}

void f2(int* q)
{
    int temp = 200;
    q = &temp;
}

```



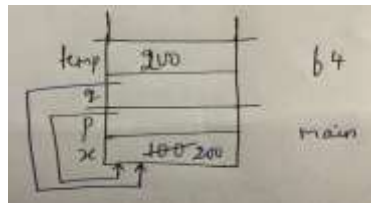
```

void f3(int t)
{
    t = 200;
}

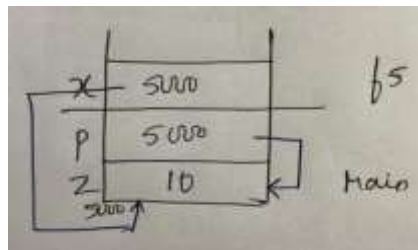
void f4(int* q)
{
    int temp = 200;
    *q = temp;
}

```

}

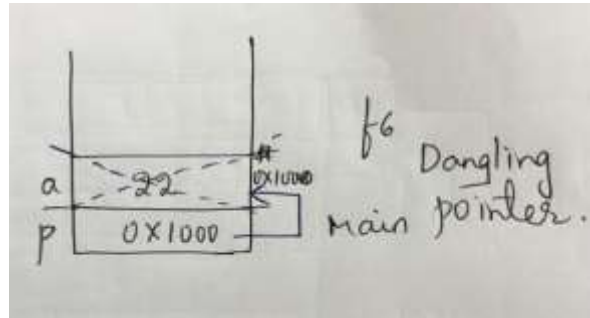


```
int* f5(int* x)
{
    return x;
}
```



When there is function call to f6, Activation Record gets created for that and deleted when the function returns. p points to a variable which is not available after the function execution. This problem is known as Dangling Pointer. The pointer is available. But the location it is not available which is pointed by pointer. Applying dereferencing operator(\*) on a dangling pointer is always undefined behaviour.

```
int* f6()
{
    int a = 22;    // We should never return a pointer to a local variable
    return &a;     //      Compile time Error
}
```



When there is function call to f6, Activation Record gets created for that and deleted when the function returns. p points to a variable which is not available after the function execution. This problem is known as Dangling Pointer. The pointer is available. But the location it is not available which is pointed by pointer. Applying dereferencing operator(\*) on a dangling pointer is always undefined behaviour.

### void data type in c

void is an **empty data type that has no value**. We mostly use void data type in functions when we don't want to return any value to the calling function. We also use void data type in pointer like "void \*p", which means, pointer "p" is not pointing to any non void data types . void \* acts as generic pointer. We will be using void \*, when we are not sure on the data type that this pointer will point to. We can use void \* to refer either integer data or char data. void \* should not be dereferenced without explicit type casting. We use void in functions as "int function\_name (void)", which means, the function does not accept any argument

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Recursion**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

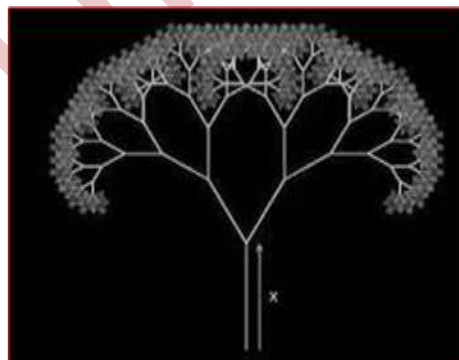
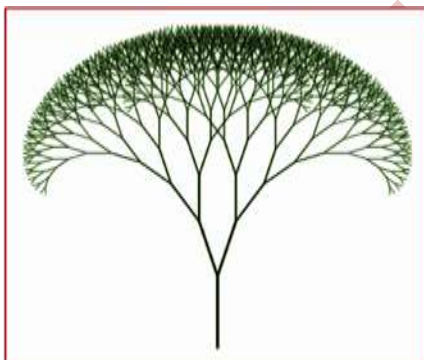
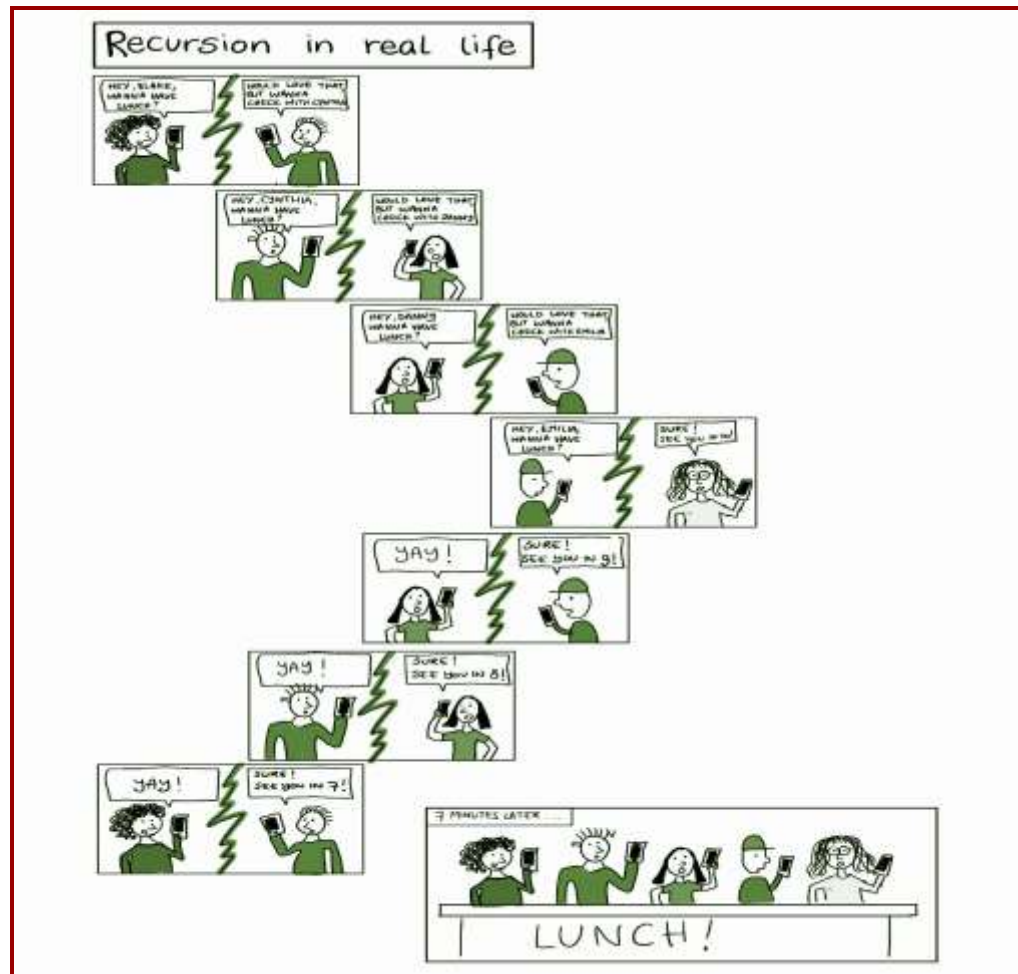
CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

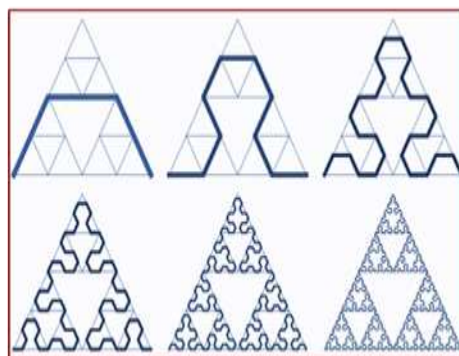
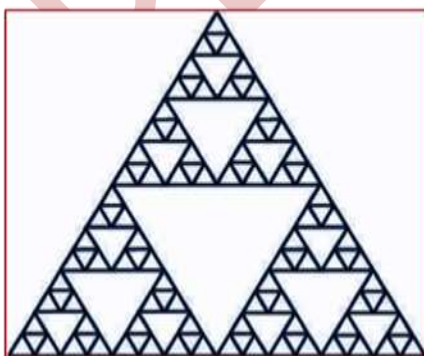
Jan - May, 2022

Dept. of CSE,

PES University



Fractals



The Sierpinski Triangle



## Recursion

A function may call itself directly or indirectly. **The function calling itself with the termination/stop/base condition is known as recursion.** Recursion is used to solve various problems by dividing it into smaller problems. We can **opt if and recursion instead of looping statements.** In recursion, we try to **express the solution to a problem in terms of the problem itself, but of a smaller size.**

### Coding Example\_1:

```
int main()
{
    printf("Hello everyone\n");
    main();
    return 0;
}
```

Execution starts from main() function. Hello everyone gets printed. Then call to main() function. Again, Hello everyone gets printed and these repeats. We have not defined any condition for the program to exit, results in Infinite Recursion. In order to **prevent infinite recursive calls, we need to define proper base condition in a recursive function.**

Let us write Iterative and Recursive functions to find the Factorial of a given number.

### Coding Example\_2:

```
int fact(int n);
int main()
{
    int n = 6;
    printf("factorial of %d is %d\n",fact(n));
    return 0;
}
```

### Iterative Implementation:

```
int fact(int n)
{
    int result = 1;
    int i;
```

```
for (i = 1; i<=n; i++)  
{  
    result = result * i;  
}  
return result;  
}
```

**Recursive Implementation:**

Logic: If n is 0 or n is 1, result is 1 Else, result is  $n*(n-1)!$

```
int fact(int n)  
{  
    if (n==0)  
        return 1;  
    else  
    {  
        return n*fact(n-1);  
    }  
}
```

**Pictorial representation: When n is 5**

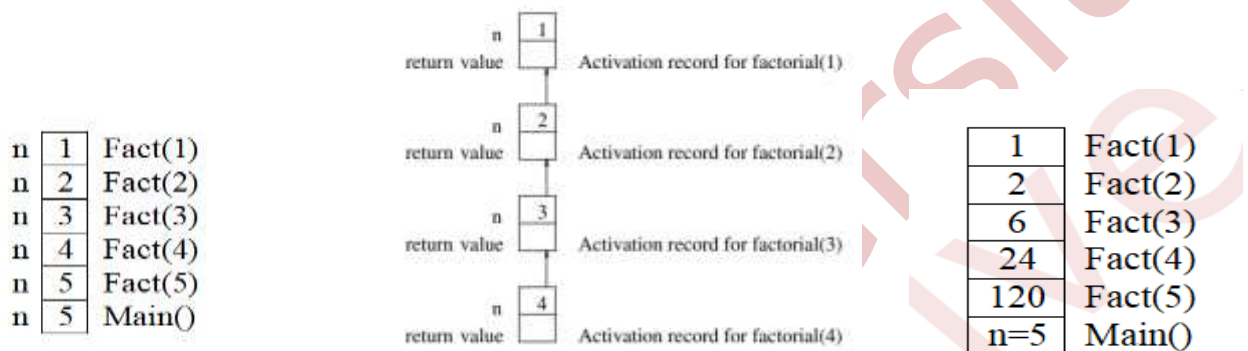
```
=factorial(5)  
=5*factorial(4)  
=5*4*factorial(3)  
=5*4*3*factorial(2)  
=5*4*3*2*factorial(1)  
=5*4*3*2*1*factorial(0)  
=120
```

**Output:** When n is 6

```
factorial of 6 is 720  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

## Activation Record

- Each function call generates an instance of that function containing memory for each parameter, memory for each local variable and memory for return values. This chunk of memory is called as an **activation record**.
- To support recursive function calls, the run-time system treats memory as a stack of activation record.
- For example computing the factorial (n) requires allocation of 'n' activation records on the stack.



Let us try to solve below examples to understand recursive calls better.

**Coding Example\_3: What this code does? Write activation frame for each call and then decide**

```
#include<stdio.h>
int what1(int n)
{
    if(n==0) return 0;
    else return (n%2)+10*what1(n/2);
}
int main()
{
    printf("%d",what1(8));
    return 0;
}
```

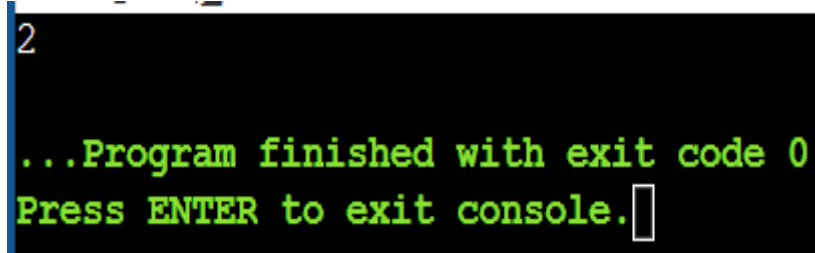
**Output:** This code prints the binary representation of a given number

```
1000
...Program finished with exit code 0
Press ENTER to exit console.█
```

**Coding Example\_4: What this code does?**

```
#include<stdio.h>
int what2_v1(int n);
int what2_v2(int n);
int main()
{
    int n = 10;
    printf("%d",what2_v1(n));
    //printf("%d",what2_v2(n));
    return 0;
}
int what2_v2(int n)
{
    if(!n) return 0;
    else if (!(n%2)) return what2_v2(n/2);
    else
        return 1+what2_v2(n/2);
}
int what2_v1(int n)
{
    if(n==0) return 0;
    else return (n&1)+ what2_v1(n>>1);
}
```

**Output:** Finds the number of 1s in the binary representation of the number



```
2
...Program finished with exit code 0
Press ENTER to exit console.
```

**Coding Example\_4:** *What this code does?*

```
#include <stdio.h>

int what3_v1(int,int);

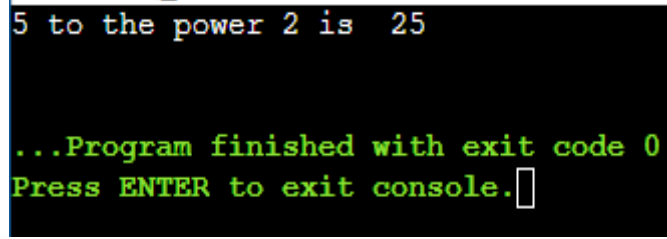
int main()
{
    int a=5,b=2;
    printf("%d to the power %d is  %d\n",a,b,what3_v1(a,b));
    //printf("%d to the power %d is  %d\n",a,b,what3_v2(a,b));
    return 0;
}

int what3_v1(int b,int p)
{
    int result=1;
    if(p==0)
        return result;
    else
        result=b*(what3_v1(b,p-1));
}

int what3_v2(int b, int p)
{
    if(!p)
        return 1;
    else if(!(p % 2))
        return what3_v2(b*b, p/2);
    else
```

```
    return b*what3_v2(b*b,p/2);  
}
```

**Output:** Finds a to the power b



**Coding Example\_5: Last example for today!!**

```
#include <stdio.h>  
  
int what4(int);  
  
int main()  
{  
    int a=5;  
    printf("the sum of numbers upto %d is %d\n",a,what4(a));  
    return 0;  
}  
  
int what4(int n)  
{  
    if (n>0)  
        return n + what4(n - 1);  
    else if (n==0)  
        return 0;  
    else  
        return -1;  
}
```

**Output:** If the given number is n, Finds  $n+(n-1)+(n-2)+ \dots +0$  If n is -ve, returns -1.

```
the sum of numbers upto 5 is 15  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

**Few problems for you to Code:**

- Recursive function to reverse a string
- Recursive function to print from 1 to n in reverse order
- Find the addition, subtraction and multiplication of two numbers using recursion. Write separate recursive functions to perform these.
- Find all combinations of words formed from Mobile Keypad.
- Find the given string is palindrome or not using Recursion.
- Find all permutations of a given string using Recursion

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Storage classes**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

**Team – PSWC,**

**Jan - May, 2022**

**Dept. of CSE,**

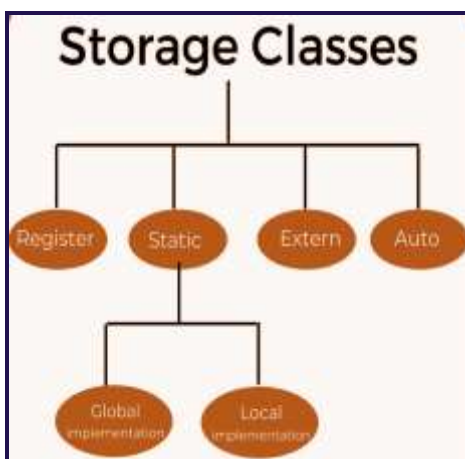
**PES University**



## Storage classes

### Introduction

Storage Classes are used to describe the features of a variable/function. These features basically include the **scope(visibility) and life-time which help us to trace the existence of a particular variable during the runtime of a program**. The following storage classes are most often used in C programming. Storage classes in C are used to determine the **lifetime, visibility, memory location, and initial value of a variable**.



### Automatic Variables

A variable declared inside a function without any storage class specification is by default an automatic variable. They are created when a function is called and are destroyed automatically when the function execution is completed. Automatic variables can also be called local variables because they **are local to a function**. By default, they are assigned to undefined values.

### Coding Example\_1:

```

#include<stdio.h>

int main()
{
    int i=90;      // by default auto because defined inside a function
    auto float j=67.5;
    printf("%d %f\n",i,j);
  
```

```
}  
int f1()  
{  
    int a; // by default auto because declared inside a function f1()  
    // Not accessible outside this function.  
}
```

### External variables

The extern keyword is used before a variable **to inform the compiler that the variable is declared somewhere else**. The extern declaration does not allocate storage for variables. All functions are of type extern. **The default initial value of external integral type is 0 otherwise null. We can only initialize the extern variable globally, i.e., we cannot initialize the external variable within any block or method.**

#### Coding Example\_2:

```
int main()  
{  
    extern int i;    // Information saying declaration exist somewhere else and make it available  
    during linking  
    // if u comment this line, it throws an error: i undeclared  
    printf("%d\n",i);  
}  
int i=23;
```

#### Note:

An external variable can be declared many times but can be initialized at only once.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions

**Coding Example\_3: Two files share the same global variable****Sample.c**

```
#include <stdio.h>

int count ;

extern void write_extern();

int main() {
    count = 5;
    write_extern();
    return 0;
}
```

**Sample1.c**

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
    printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c.

// please check the execution steps below

```
gcc Sample.c Sample1.c -c
```

```
gcc Sample.o Sample1.o
```

**Output - count is 5**

**Static variables**

A static variable tells the compiler to persist the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static is initialized only once and remains into existence till the end of program. A static variable can either be local or global depending upon the place of declaration.

**Scope of local static variable remains inside the function in which it is defined but the life time of local static variable is throughout that program file.** Global static variables remain restricted to scope of file in each they are declared and life time is also restricted to that file only. **All static variables are assigned 0 (zero) as default value.**

#### Coding Example\_4: Demo for Life time of Static variable

```
#include<stdio.h>

int* f1();

int main()
{
    int *res = f1();
    printf("Res is %p %d\n",res,*res);    //same address in all three outputs. Then 11
    res = f1();
    printf("Res is %p %d\n",res,*res);    // 12
    res = f1();
    printf("Res is %p %d\n",res,*res);    // 13
    return 0;
}

int* f1()
{
    static int a= 10;// memory is shared. This line is ignored after first function call
    a++;
    return &a;    // valid because life time of static variable is through out the file execution
}
```

#### Coding Example\_5: Understand the difference between local and global static variable.

```
#include<stdio.h>

void f1();

static int j=20;    // global static variable: cannot be used outside this program file
int k=23;            //global variable: can be used anywhere by linking with this file.
                    //By default has extern with it.

int main( )
```

```
{  
    f1();           // 0 20  
    f1();           // 0 21  
    f1();           // 0 22  
    return 0;  
}  
void f1()  
{  
    int i=0;  
    printf("%d %d\n",i,j);  
    i++;  
    j++;  
}
```

## Register variables

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. It's compiler's choice to put it in a register or not.

Register is an integral part of the processor. It is a very fast memory of computer mainly used to execute the programs and other main operation quite efficiently. They are used to quickly store, accept, transfer, and operate on data based on the instructions that will be immediately used by the CPU. Main operations are fetch, decode and execute. Numerous fast multiple ported memory cells are the atomic part of any register. Generally, compilers themselves do optimizations and put the variables in register. If a free register is not available, these are then stored in the memory only. If & operator is used with a register variable then compiler may give an error or warning (depending upon the compiler used ), because when a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. **The access time of the register variables is faster than the automatic variables.**

Mostly used cpu registers are Accumulator, Flag Register, Address Register (AR), Data Register (DR), Program Counter (PC), Instruction Register (IR), Stack Control Register (SCR), Memory Buffer Register (MBR) and Index register (IR)

**Coding Example\_6:**

```
#include<stdio.h>

int main()
{
    register int i = 10;
    int* a = &i; // error
    printf("%d", *a);
    getchar();
    return 0;
}
```

**Coding Example\_7:** We can store pointers into the register, i.e., a register can store the address of a variable.

```
#include<stdio.h>

int main()
{
    int i = 10;
    register int* a = &i; // fine
    printf("%d", *a);
    getchar();
    return 0;
}
```

**Note:** Static variables cannot be stored into the register since we cannot use more than one storage specifier for the same variable

**Global variables**

The variables declared outside any function are called global variables. They are not limited to any function. **Any function can access and modify global variables.** Global variables are automatically initialized to 0 at the time of declaration.

**Coding Example\_8:**

```
#include<stdio.h>

int i = 100;
int j;
int main()
{
    printf("%d\n",i);           // 100
    i=90;
    printf("%d\n",i);           // 90
    f1();
    printf("%d\n",i);           // 40
    printf("%d\n",j);           // 0 by default, global variable is 0 if just declared
    return 0;
}
int f1()
{
    i = 40;    // Any function can modify the global variable.
}
```

**Think about it!****What if we have `int i = 200;` in a function `f1()` ?****Happy Thinking and Coding!!**

**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Enumerations**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

**Team – PSWC,**

**Jan - May, 2022**

**Dept. of CSE,**

**PES University**



## Enumerations

### Introduction

Enumeration (or enum) is a way of creating user-defined data type in C. It is mainly used to assign names to integral constants. The names make a program easy to read and maintain. It is easier to remember names than numbers. Example scenario: We all know Google.com. We don't remember the IP address of the website Google.com.

The keyword 'enum' is used to declare new enumeration types in C. In Essence, **enumerated types provide a symbolic name to represent one state out of a list of states**. The names are symbols for integer constants, which won't be stored anywhere in the program's memory. **Enums are used to replace #define chains**

An enumeration data type consists of named integer constants as a list. It starts with 0 (zero) by default and the value is incremented by 1 for the sequential identifiers in the list.

**Syntax:** `enum identifier { enumerator-list };` // semicolon compulsory & identifier optional

Examples:

```
enum month { Jan, Feb, Mar }; // Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default
```

```
enum month { Jan = 1, Feb, Mar }; // Feb and Mar variables will be assigned to 2 and 3 respectively by default
```

Note: These values are symbols for integer constants, which won't be stored anywhere in program's memory. It doesn't make sense to take its address.

**Coding Example\_1: Enum variables are automatically assigned values if no value specified.**

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

```
enum months
```

```
{
```

```
    jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
```

```
// symbol jan gets the value 0, all others previous+1
//If we do not explicitly assign values to enum names, the compiler by default assigns
//values starting from 0. jan gets value 0, feb gets 1, mar gets 3 and so on.
// all constants are separated by comma(.).
// no memory allocated for these constants. Available in this file anywhere directly.
}; // compulsory semi-colon
int main()
{
    enum months m; // memory is allocated for m.
    printf("sizeof m is %lu\n",sizeof(m)); // same as the size of integer
    m=apr; // 3 is stored in m
    printf("%d",m); //3
    return 0;
}
```

**Coding Example\_2: We can assign values to some of the symbol names in any order. All unassigned names get value as value of previous name plus one.**

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5, wednesday, thursday = 10, friday, saturday};
int main()
{
    printf("enter the value for monday\n");
    //scanf("%d",&monday); // error: not legal
    // Memory not allocated for constants in enum. So ampersand(&) can't be used with those.
    //Also value already assigned to monday that can't be changed
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday, wednesday, thursday, friday,
                                                saturday);

    return 0;
}
```

**Coding Example\_3: Only integer constants are allowed. Below code results in Error if abc=3.5 is uncommented.**

```
#include<stdio.h>

enum examples
{
    //abc=3.5, bef; // Value is not an integer constant
    abc="Hello", bef;    // Value is not an integer constant
};

int main()
{
    enum examples e1;
    return 0;
}
```

**Coding Example\_4: Valid arithmetic operators are +, - \* and / and %**

```
#include<stdio.h>

enum months
{
    jan,feb,mar,apr,may,jun,july,aug,sep,oct,nov,dec
};

int main()
{
    printf("%d\n",mar-jan);           // valid
    printf("%d\n",mar*jan);           // valid
    printf("%d\n",mar&&feb);          // valid
    //mar++;                          //error : mar is a constant
    //printf("after incrementing %d\n",mar); // error
    enum months m=feb;
    m=(enum months)(m + jan); // m can be changed. m is a variable of type enum months.
    // But all constants in enum cannot be changed its value
    printf("m, after incrementing %d\n",m);
    for(enum months i=jan;i<=dec;i++) // loop to iterate through all constants in enum
}
```

```
        {           printf("%d\n", i);           }  
    return 0;  
}
```

**Coding Example\_5: Enumerated Types are Not Strings .Two enum symbols/names can have same value.**

```
#include <stdio.h>  
enum State {Working = 1, Failed = 0, Freezed = 0};  
int main()  
{  
    printf("%d, %d, %d", Working, Failed, Freezed);    // associated values are printed  
    return 0;  
}
```

**Coding Example\_6: All enum constants must be unique in their scope.**

Below code results in Error: Redclaration of enumerator 'def'.

```
#include<stdio.h>  
enum example1 {def, cdt}; enum  
example2 {abc, def};  
int main()  
{  
    enum example1 e;  
    return 0;  
}
```

**Coding Example\_7: It is not possible to change the constants.**

Uncommenting the statement feb=10, results in error.

```
#include<stdio.h>  
enum months  
{
```

```
    jan,feb=1 1,mar,apr,may=23,jun,july
};
int main()
{
    enum months m;
    //feb=10;           // error
    printf("%d",feb);    // no error           // we can access it directly
    return 0;
}
```

**Coding Example\_8: Storing the symbol of one enum in another enum variable. It is allowed.**

```
#include<stdio.h>
enum nation
{
    India=1, Nepal    };
enum States
{
    Delhi, Katmandu   };
int main()
{
    enum States n=Delhi;
    enum States n1=India;
    // Value 1 is stored in n1 which is of type enum States
    enum nation n2=Katmandu; printf("%d\n",n);
    printf("%d\n",n1);
    printf("%d",n2);
    return 0;
}
```

**Coding Example\_9:**

**One of the short comings of Enumerated Types is that they don't print nicely.**

To print the "String"(symbol) associated with the defined enumerated value, you must use the following cumbersome code

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

```
enum example1
{
    abc=123, bef=345, cdt=555
};
void printing(enum example1 e1);
int main()
{
    enum example1 e1;
    // how to print abc, bef and cdt based on user choice?? printf("enter the number");
    scanf("%d",&e1); //enter any number
    printing(e1);    // user defined function to print the symbol name
    return 0;
}
void printing(enum example1 e1)
{
    switch(e1)
    {
        case abc: printf("abc");break;
        case bef:printf("bef");break;
        case cdt:printf("cdt");break;
        default:printf("no symbol in enum with this value"); break;
    }
}
```

**Coding Example\_10:** The value assigned to enum names must be some integral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value

Below code Results in Error: overflow in enumeration values.

```
#include<limits.h>
#include<stdio.h>
enum examples
{
```

```
abc = INT_MAX, def, cdt
// INT_MAX is defined in limits.h
// def must get INT_MAX+1 value which is an error
// delete def and cdt and compile and run again
};
int main()
{
    enum examples e;
    e=abc;
    printf("%d",e);
    return 0;
}
```

**Coding Example\_11:** Enumerations are actually used to restrict the values to a set of integers within the range of values. But outside the range, if any value is used it is not an error in C standards like C11, C99 and C89. Other languages (ex: Java) doesn't support any value outside the range.

Code to demonstrate why enum support values outside range values in C.

```
#include<stdio.h>
enum design_flags
{
    bold =1, italics=4, underline=8    }df;
int main()
{
    df=bold;
    df=italics|bold; // to set bold and italics. Now df contains 5 which is not defined in enum
    // if you write again df=italics, 5 is replaced with 4. So text is no more bold. It is only italics
    if(df==(bold|italics))
        printf("both\n");
    else if(df==bold)
        printf("bold it is\n");
    else if(df==italics)
```

```
printf("italics it is\n");

if(df&bold)    // whether the text is bold
    printf("bold\n");
if(df & italics) // whether the text is italics
    printf("italics\n");
if(df & underline)    // whether the text is underlined
    printf("underline\n");
return 0;
}
```



**Unit #: 2**

**Unit Name: Counting, Sorting and Searching**

**Topic: Functions - Interface and Implementation**

**Course objectives:**

The objective(s) of this course is to make students

CObj1: Acquire knowledge on how to solve relevant and logical problems using computing machine

CObj2: Map algorithmic solutions to relevant features of C programming language constructs

CObj3: Gain knowledge about C constructs and it's associated eco-system

CObj4: Appreciate and gain knowledge about the issues with C Standards and it's respective behaviors

CObj5: Get insights about testing and debugging C Programs

**Course outcomes:**

At the end of the course, the student will be able to

CO1: Understand and apply algorithmic solutions to counting problems using appropriate C Constructs

CO2: Understand, analyse and apply text processing and string manipulation methods using C Arrays, Pointers and functions

CO3: Understand prioritized scheduling and implement the same using C structures

CO4: Understand and apply sorting techniques using advanced C constructs

CO5: Understand and evaluate portable programming techniques using preprocessor directives and conditional compilation of C Programs

Team – PSWC,

Jan - May, 2022

Dept. of CSE,

PES University

## Interface and Implementation

**An interface specifies only those identifiers that clients may use,** hiding irrelevant representation details and algorithms as much as possible. It helps clients avoid dependencies on the specifics of particular implementations.

Dependency between a client and an implementation — coupling — causes bugs when an implementation changes; these bugs can be particularly hard to fix when the dependencies are buried in hidden or implicit assumptions about an implementation. A well-designed and precisely specified interface reduces coupling.

C has only minimal support for separating interfaces from implementations, but simple conventions can yield most of the benefits of the interface/implementation methodology. **In C, an interface is specified by a header file, which usually has a .h file extension.** This header file declares the macros, types, data structures, variables, and routines that clients may use. A client imports an interface with the C preprocessor `#include` directive.

**An implementation exports an interface.** It defines the variables and functions necessary to provide the facilities specified by the interface. An implementation reveals the representation details and algorithms of its particular rendition of the interface, but, ideally, clients never need to see the implementation details. Clients share object code for implementations, usually by loading them from libraries. An interface can have more than one implementation. As long as the implementation adheres to the interface, it can be changed without affecting clients.

In C, an implementation is provided by one or more .c files. An implementation must provide the facilities specified by the interface it exports. Implementations include the interface .h file to ensure that its definitions are consistent with the interface declarations. Beyond this, however, there are no linguistic mechanisms in C to check implementation compliance

**Interface means Function declaration. Implementation means Function definition.**

**Interface tells us what the function requires while calling.** It does not tell us anything about how that function works. **Implementation always refers to the body of the function.** If any change in the body of the function, it does not affect outside. The one who is using the function, need not worry about the change in implementation.

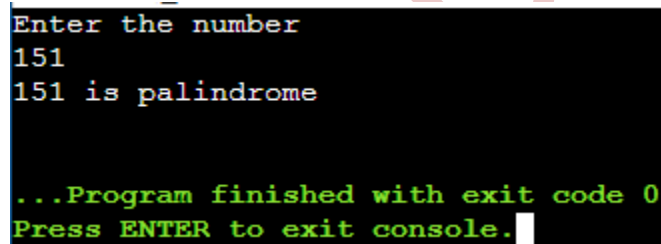
Let us say, requirement is to check whether a given number is a palindrome or not.

**Coding Example\_1: All operations are carried out in the client code itself.**

```
#include<stdio.h>

int main()
{
    int n;
    printf("Enter the number\n");
    scanf("%d", &n);
    int rev = 0;
    int number = n;
    while(n>0)    // can be just while(n). When n is 0, loop terminates.
    {
        rev = rev * 10 + n % 10; n /= 10;
    }
    if(number == rev)
        printf("%d is palindrome\n",number);
    else
        printf("%d is not palindrome\n",number);
}
```

**Output:**



```
Enter the number
151
151 is palindrome

...Program finished with exit code 0
Press ENTER to exit console.
```

**Coding Example\_2: Function to check whether the given number is palindrome or not.**

```
#include<stdio.h>

int is_palin(int);    // interface

int main()
{
    int n;
```

```
printf("Enter the number\n");
scanf("%d", &n);
if(is_palin(n))
    { printf("%d is a palindrome\n", n); }
else
    { printf("%d is not a palindrome\n", n); }
}

int is_palin(int n) // implementation
{
    int rev = 0;
    int number = n;
    while(n>0)
    {
        rev = rev * 10 + n % 10; n /= 10;
    }
    return number == rev;
}
```

You may want to write reverse function which reverses the given number and returns the value. Later this can be used inside is\_palin function. Try it!

**Coding Example\_3:** Separating the client code, server code and the header file.

**palin.h** contains declarations of functions

```
int is_palin(int n);
```

Then we write the client code [where the execution starts – main() in C ] as below.

**client.c** contains the below code.

```
#include<stdio.h>
#include "palin.h" // inclusion of user defined header file
int main()
{
    int n;
```

```
printf("Enter the number\n");
scanf("%d", &n);
if(is_palin(n))
{ printf("%dis a palindrome\n", n); }
else { printf("%d is not a palindrome\n", n); }
}
}
```

**palin.c** - Definitions of User defined functions are saved here

```
int is_palin(int n)    // implementation
{
    int rev = 0;
    int number = n;
    while(n>0)
    { rev = rev * 10 + n % 10; n /= 10; }
    return number == rev;
}
```

Commands to execute above codes are as below.

```
gcc -c client.c      // this creates client.o
gcc -c palin.c       // this creates palin.o
gcc client.o palin.o // this is for linking. We get the loadable image a.exe or a.out
a.exe or ./a.out     // press enter to see the result or output
```

If I make some changes in the implementation file i.e., palin.c, I need to compile only that. Then link palin.o and client.o to get the executable.

If there are one or two implementation files, the changed files can be recompiled and relinked easily. But, if you have many implementation files and you have made modifications to some of these, you need to remember which all files you need to compile again. Or Compile all files and link all object files again. This is waste of time. Rather, we have a facility called **make** which

completes this requirement in an easier way.

### Usage of make command

make is Unix utility that is designed to start execution of a makefile. A makefile is a special file, containing shell commands, that you create and name it with .mk extension preferably. While in the directory containing this makefile, you will type make and the commands in the makefile will be executed. A makefile that works well in one shell may not execute properly in another shell. Because it must be written for the shell which will process the makefile.

**The makefile contains a list of rules.** These rules tell the system when and which command to be executed. These rules are commands to compile(or recompile) a series of files. The rules, which must begin in column 1, are specified in two types of lines. The first line is called a **Dependency line** and the subsequent line(s) are called **Action lines**. **The action line(s) must be indented with a tab.**

The **dependency line** is made of two parts. The first part (before the colon) is Target files and the second part (after the colon) are called as Source files. It is called a dependency line because the first part depends on the second part. Multiple target files must be separated by a space. Multiple source files must also be separated by a space. The

The **action line** will be executed if the target file mentioned in the dependency line is older than any of the source file or if the target file is missing.

Make reads the makefile and creates a **dependency tree** and takes whatever action is necessary. It will not necessarily do all the rules in the makefile as all dependencies may not need updation. It will rebuild target files if they are missing or older than the dependency files. It keeps track of the recent time, the files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date. If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files. Without a makefile, this is an extremely time-consuming task.

**Command to be used in Linux based OS:**

```
make -f filename.mk // -f to read file as a make file
```

**Windows:** you need to download this utility.

If you have installed gcc using mingw package, go to mingw/bin folder in command prompt and type as below.

**mingw-get install mingw32-make** // press enter Command to execute on Windows

```
mingw32-make -f filename.mk // -f to read file as a make file
```

Corresponding Make file for palindrome checking code is as below.

```
a.exe: client.o palin.o
```

```
gcc client.o palin.o
```

```
client.o : client.c palin.h
```

```
gcc -c client.c
```

```
palin.o : palin.c palin.h
```

```
gcc -c palin.c
```

Execute the make file to get the executable either a.exe or a.out. Run the code. Make small changes in the source file or client file. Execute the make file again. New a.exe or a.out will be generated. If you execute the makefile without making any changes to any of the source files or client file, you will get a message on the terminal saying, **a.exe is up to date.**