Programming in C

Presented by: Nimesh Shiwakoti

Function

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

advantages of C functions.

By using functions, we can avoid rewriting same logic/code again and again in a program.

We can call C functions any number of times in a program and from any place in a program.

We can track a large C program easily when it is divided into multiple functions.

Reusability is the main achievement of C functions.

However, Function calling is always a overhead in a C program.

Types of Function in C

1. Library functions in C
2. User defined function in C

Library functions

- Library functions in C language are inbuilt functions which are grouped together and placed in a common place called library.
- Each library function in C performs specific operation.
- We can make use of these library functions to get the pre-defined output instead of writing our own code to get those outputs.
- These library functions are created by the persons who designed and created C compilers.
- All C standard library functions are declared in many header files which are saved as file_name.h.
- Actually, function declaration, definition for macros are given in all header files.
- We are including these header files in our C program using “#include<file_name.h>” command to make use of the functions those are declared in the header files.
- When we include header files in our C program using “#include<filename.h>” command, all C code of the header files are included in C program. Then, this C program is compiled by compiler and executed.

Advantages of using C library functions

- Simple and easy to use
- **The functions are optimized for performance**
- **It saves considerable development time**
- **The functions are portable**

Stdio.h –standard input output functions

clearerr()	clrmemf()	fclose()	fdelrec()	feof()
ferror()	fflush()	fgetc()	fgetpos()	fgets()
fldata()	flocate()	fopen()	fprintf()	fputc()
fputs()	fread()	freopen()	fscanf()	fseek()
fseeko()	fsetpos()	ftell()	ftello()	fupdate()
fwrite()	getc()	getchar()	gets()	perror()
printf()	putc()	putchar()	puts()	remove()
rename()	rewind()	scanf()	setbuf()	setvbuf()
sprintf()	sscanf()	svc99()	tmpfile()	tmpnam()
ungetc()	vfprintf()	vprintf()	vsprintf()	

conio.h **Console-Input-Output functions**

Functions	Description
clrscr( )	used to clear the output screen
getch()	It reads character from the keyboard
getche()	reads character from keyboard and echoes to o/p screen
textcolor()	used to change the text color
textbackground()	used to change text background

math.h — Floating-point math functions

absf()	absl()	acos()	acosf()	acoshf()
acoshl()	acosl()	asin()	asinf()	asinhf()
asinhl()	asinl()	atan()	atan2()	atan2f()
atan2l()	atanf()	atanl()	cbrtf()	cbrtl()
ceil()	ceilf()	ceill()	copysign()	copysignf()
copysignl()	cos()	cosf()	cosh()	coshf()
coshl()	cosl()	exp()	expf()	expl()
expm1f()	expm1l()	exp2()	exp2f()	exp2l()
fabsf()	fabsl()	floor()	floorf()	floorl()
fma()	fmaf()	fmal()	fmax()	fmaxf()
fmaxl()	fmin()	fminf()	fminl()	fmod()
fmodf()	fmodl()	frexp()	frexpf()	frexpl()
hypotf()	hypotl()	ilogbf()	ilogbl()	ldexp()
ldexpf()	ldexpl()	lgammaf()	lgammal()	llrint()
llrintf()	llrintl()	llround()	llroundf()	llroundl()
log()	logbf()	logbl()	logf()	logl()
log1pf()	log1pl()	log10()	log10f()	log10l()
lrint()	lrintf()	lrintl()	lround()	lroundf()
lroundl()	modf()	modff()	modfl()	nan()
nanf()	nanl()	nearbyint()	nearbyintf()	nearbyintl()
nextafterf()	nextafterl()	nexttoward()	nexttowardf()	nexttowardl()
pow()	powf()	powl()	remainderf()	remainderl()
remquo()	remquof()	remquol( )	rintf()	rintl()
round()	roundf()	roundl()	scalbln()	scalblnf()
scalblnl()	sin()	sinf()	sinh()	sinhf()
sinhl()	sinl()	sqrt()	sqrtf()	sqrtl()
tan()	tanf()	tanh()	tanhf()	tanhl()
tanl()	tgamma()	tgammaf()	tgammal()	

string.h — String manipulation functions

memchr()[	memcmp()	memcpy()	memmove()	memset()
strcat()	strchr()	strcmp()	strcoll()	strcpy()
strcspn()	strerror()	strlen()	strncat()	strncmp()
strncpy()	strpbrk()	strrchr()	strspn()	strstr()
strtok()	strxfrm()			

User defined Function in C

C programming language allows coders to define functions to perform special tasks. As functions are defined by users, they are called user-defined functions. user-defined functions **have contained the block of statements which are written by the user to perform a task**.

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list ) {
   body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

Return Type − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

- A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.
- A function declaration has the following parts −
- return_type function_name(parameter list);
- For the above defined function max(), the function declaration is as follows −
- int max(int num1, int num2);
- Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −
- int max(int, int);

Type of User-defined Functions in C

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

An argument is referred to **the values that are passed within a function when the function is called**

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>

void greatNum();      // function declaration

int main()
{
   greatNum();       // function call
   return 0;
}

void greatNum()       // function definition
{
   int i, j;
   printf("Enter 2 numbers that you want to
compare...");
   scanf("%d%d", &i, &j);
   if(i > j) {
      printf("The greater number is: %d", i);
   }
   else {
      printf("The greater number is: %d", j);
   }
}
```

Function with no arguments and a return value

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>

int greatNum();      // function declaration

int main()
{
   int result;
   result = greatNum();       // function call
   printf("The greater number is: %d", result);
   return 0;
}

int greatNum()       // function definition
{
   int i, j, greaterNum;
   printf("Enter 2 numbers that you want to
compare...");
   scanf("%d%d", &i, &j);
   if(i > j) {
      greaterNum = i;
   }
   else {
      greaterNum = j;
   }
   // returning the result
   return greaterNum;
}
```

Function with arguments and no return value

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>

void greatNum(int a, int b);      // function declaration

int main()
{
   int i, j;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   greatNum(i, j);       // function call
   return 0;
}
```

```
void greatNum(int x, int y)       // function definition
{
   if(x > y) {
      printf("The greater number is: %d", x);
   }
   else {
      printf("The greater number is: %d", y);
   }
}
```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>

int greatNum(int a, int b);      // function declaration

int main()
{
   int i, j, result;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   result = greatNum(i, j); // function call
   printf("The greater number is: %d", result);
   return 0;
}
```

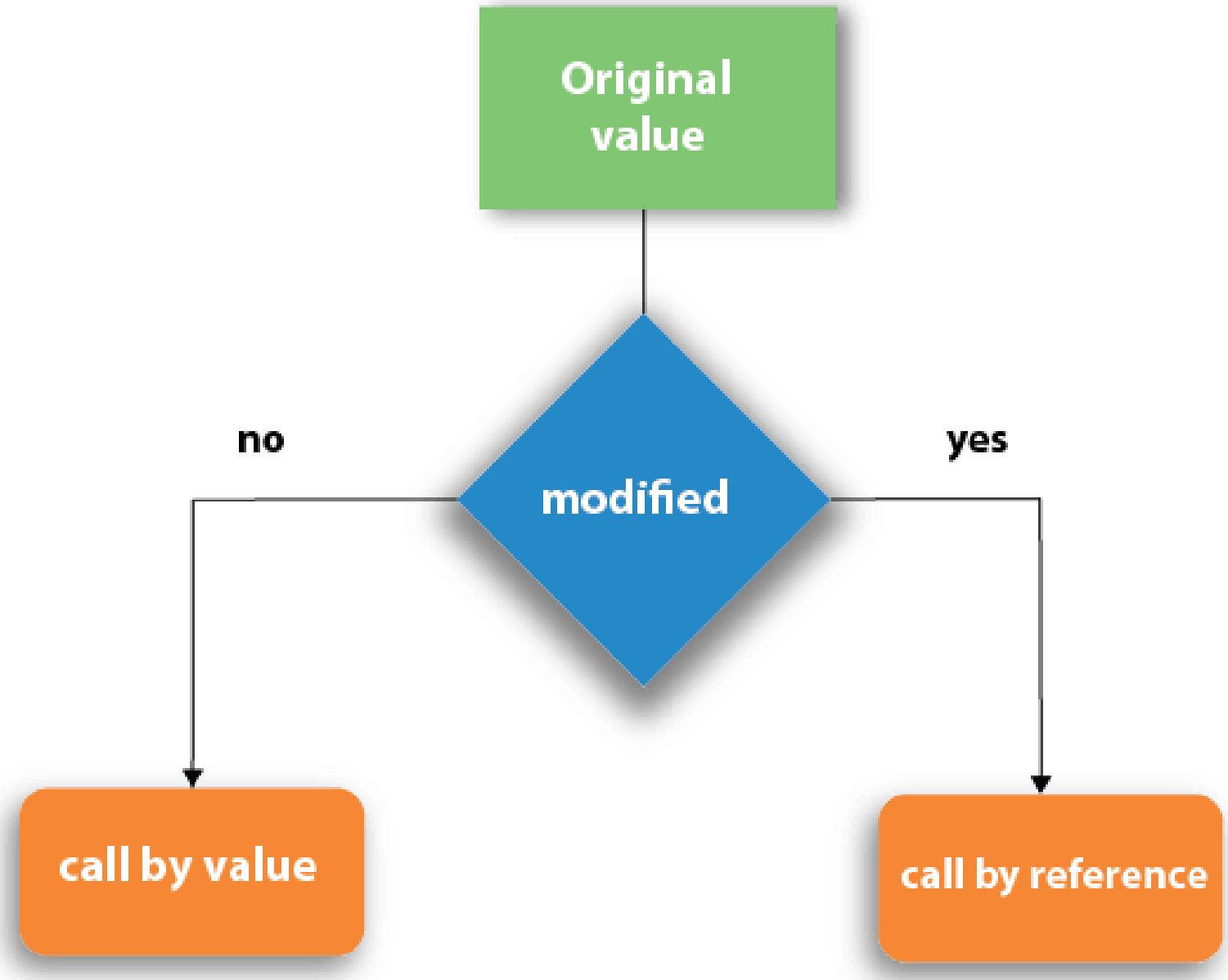
```
int greatNum(int x, int y)      // function definition
{
   if(x > y) {
      return x;
   }
   else {
      return y;
   }
}
```

Accessing function by passing values

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***. Also, the arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.

Formal Parameter : A variable and its type as they appear in the body of the function or method.

Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment


Original value
modified
no
yes
call by value
call by reference

Important methods of Parameter Passing

Call By Value : The call by value method of passing arguments to **a function copies the actual value of an argument into the formal parameter of the function**. In this case, changes made to the parameter inside the function have no effect on the argument.

```
#include<stdio.h>
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference in C

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

call by value	call by reference
This method copy original value into function as a arguments.	This method copy address of arguments into function as a arguments.
Changes made to the parameter inside the function have no effect on the argument.	Changes made to the parameter affect the argument. Because address is used to access the actual argument.
Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Storage Classes in C

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.

C language uses 4 storage classes, namely:

Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1. auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared. However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables resides. They are assigned a garbage value by default whenever they are declared.

2. external: Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

3.**static**: This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

4.**register**: This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

```c
// A C program to demonstrate different storage
// classes
#include <stdio.h>
// declaring the variable which is to be made extern
// an initial value can also be initialized to x
int x;
void autoStorageClass()
{
	printf("\nDemonstrating auto class\n\n");
	// declaring an auto variable (simply
	// writing "int a=32;" works as well)
	auto int a = 32;
	// printing the auto variable 'a'
	printf("Value of the variable 'a'"
	" declared as auto: %d\n",a);
	printf("--------------------------------");
}
void registerStorageClass()
{
	printf("\nDemonstrating register class\n\n");
	// declaring a register variable
	register char b = 'G';
	// printing the register variable 'b'
	printf("Value of the variable 'b'"
	" declared as register: %d\n",b);
	printf("--------------------------------");
}
void externStorageClass()
{	printf("\nDemonstrating extern class\n\n");
	// telling the compiler that the variable
	// x is an extern variable and has been
	// defined elsewhere (above the main
	// function)
	extern int x;
	// printing the extern variables 'x'
	printf("Value of the variable 'x'"
	" declared as extern: %d\n",x);
	// value of extern variable x modified
	x = 2;
	// printing the modified values of
	// extern variables 'x'
	printf("Modified value of the variable 'x'"
	" declared as extern: %d\n",
	x);
	printf("--------------------------------");
}
void staticStorageClass()
{
	int i = 0;
	printf("\nDemonstrating static class\n\n");
	// using a static variable 'y'
	printf("Declaring 'y' as static inside the loop.\n"
		"But this declaration will occur only"
		" once as 'y' is static.\n"
		"If not, then every time the value of 'y' "
		"will be the declared value 5"
		" as in the case of variable 'p'\n");

	printf("\nLoop started:\n");
	for (i = 1; i < 5; i++) {
		// Declaring the static variable 'y'
		static int y = 5;
		// Declare a non-static variable 'p'
		int p = 10;
		// Incrementing the value of y and p by 1
		y++;
		p++;
		// printing value of y at each iteration
		printf("\nThe value of 'y', “ "declared as static, in %d "
		"iteration is %d\n",i, y);
		// printing value of p at each iteration
		printf("The value of non-static variable 'p', "
		"in %d iteration is %d\n", i, p);
	}
	printf("\nLoop ended:\n");
	printf("--------------------------------");
}
int main()
{

	printf("A program to demonstrate"
			" Storage Classes in C\n\n");
	// To demonstrate auto Storage Class
	autoStorageClass();
	// To demonstrate register Storage Class
	registerStorageClass();
	// To demonstrate extern Storage Class
	externStorageClass();
	// To demonstrate static Storage Class
	staticStorageClass();
	// exiting
	printf("\n\nStorage Classes demonstrated");
	return 0;
}
```

Output: A program to demonstrate Storage Classes in C
Demonstrating auto class
Value of the variable ‘a’ declared as auto: 32
————————————
Demonstrating register class
Value of the variable ‘b’ declared as register: 71
————————————
Demonstrating extern class
Value of the variable ‘x’ declared as extern: 0
Modified value of the variable ‘x’ declared as extern: 2
————————————
Demonstrating static class
Declaring ‘y’ as static inside the loop.
But this declaration will occur only once as ‘y’ is static.
If not, then every time the value of ‘y’ will be the declared value 5 as in the case of variable ‘p’
Loop started:
The value of ‘y’, declared as static, in 1 iteration is 6
The value of non-static variable ‘p’, in 1 iteration is 11
The value of ‘y’, declared as static, in 2 iteration is 7
The value of non-static variable ‘p’, in 2 iteration is 11
The value of ‘y’, declared as static, in 3 iteration is 8
The value of non-static variable ‘p’, in 3 iteration is 11
The value of ‘y’, declared as static, in 4 iteration is 9
The value of non-static variable ‘p’, in 4 iteration is 11
Loop ended:
————————————
Storage Classes demonstrated

Recursive function

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. A recursive function is a function defined in terms of itself via self-calling expressions. This means that the function will continue to call itself and repeat its behavior until some condition is satisfied to return a value.

Program to find factorial of a given number recursively to illustrate recursive function

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

int fact(int n); /* Function Definition */

void main()
{
 int num, res;
 clrscr();
 printf("Enter positive integer: ");
 scanf("%d",&num);
 res = fact(num); /* Normal Function Call */
 printf("%d! = %d" ,num ,res);
 getch();
}
int fact(int n) /* Function Definition */
{
 int f=1;
 if(n <= 0)
 {
  return(1);
 }
 else
 {
  f = n * fact(n-1); /* Recursive Function Call as fact() calls itself */
  return(f);
 }
}
```

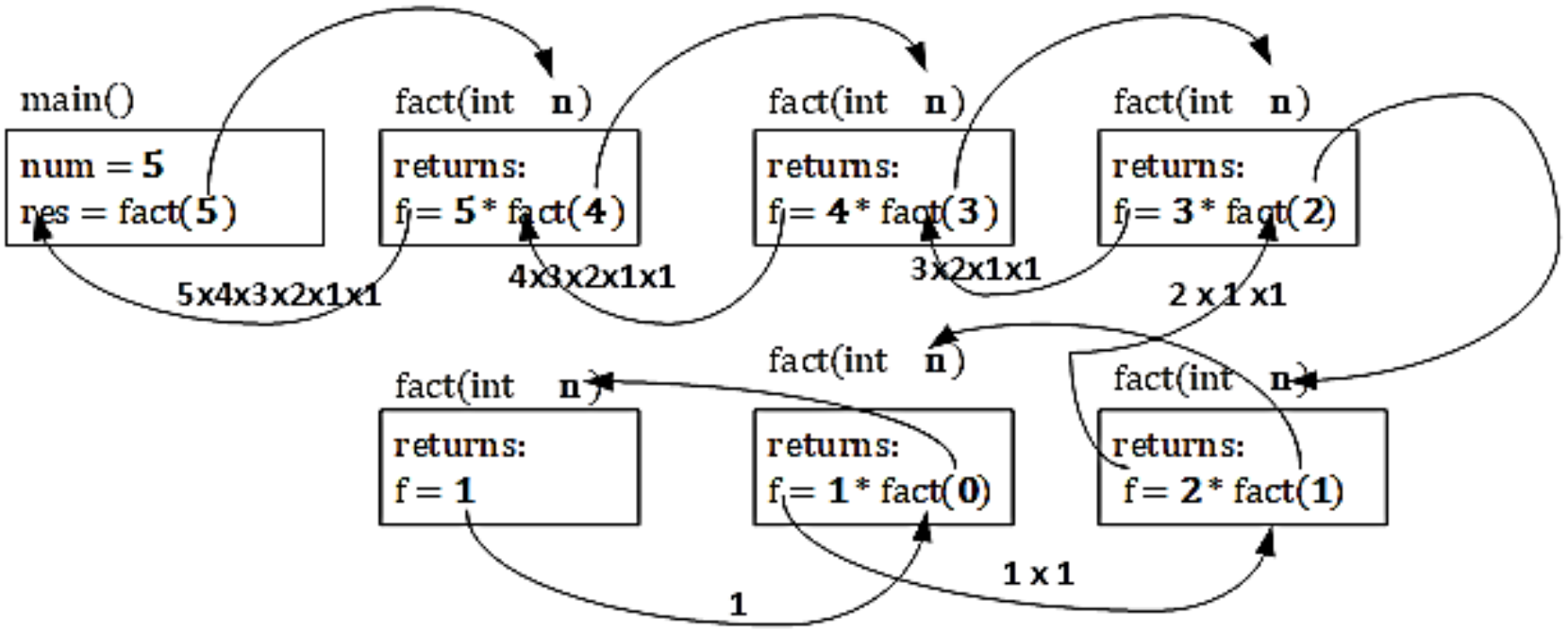
Output

Enter positive integer:

5↲

5! = 120

Working of above recursive example can be illustrated using following diagrams:



Fibonacci series is a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.

The following example generates the Fibonacci series for a given number using a recursive function −

```
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

output

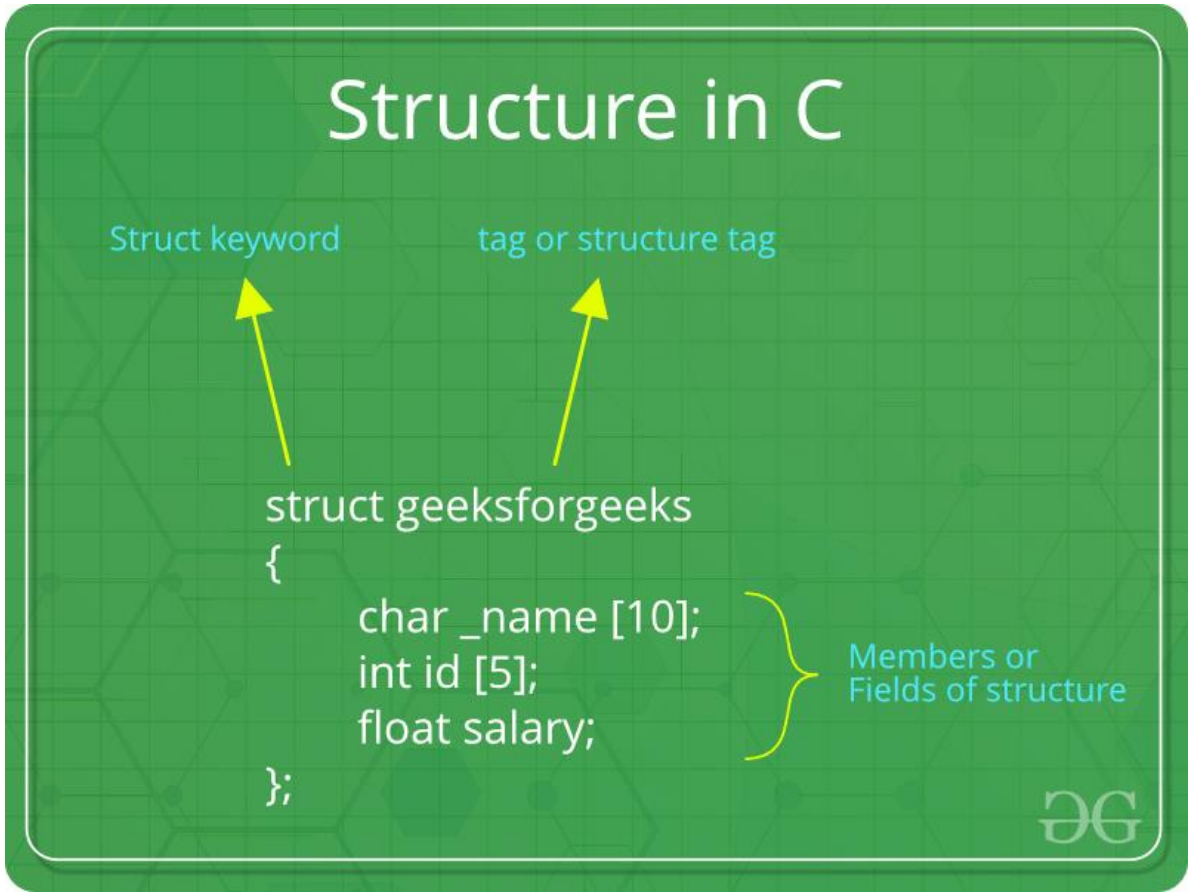
```
0

1

1

2

3

5

8

13

21

34
```

Structure and Union

What is a structure?

A structure is a user defined data type in C. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.



Defining a structure

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived(secondary) data types

Syntax:

```
struct [structure_tag]
{
   //member variable 1
   //member variable 2
   //member variable 3
   ...
}[structure_variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure

```
struct Books {
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
} book;
```

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    // F for female and M for male
    char gender;
};
```

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived data types

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. Student is the name of the structure and is called as the structure tag.

Declaring Structure Variables

It is possible to declare variables of a structure, either along with structure definition or after the structure is defined. Structure variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
}S1, S2;
```

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable using a dot . operator also called period or member access operator. **For example:**

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

struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

int main()
{
   struct Student s1;

   /*
      s1 is a variable of Student type and
      age is a member of Student
   */
   s1.age = 18;
   /*
      using string function to add name
   */
   strcpy(s1.name, "Viraaj");
   /*
      displaying the stored values
   */
   printf("Name of Student 1: %s\n", s1.name);
   printf("Age of Student 1: %d\n", s1.age);

   return 0;
}
```

Output

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
   float height;
   int weight;
   int age;
}p1={180.1,65,30};
```

or

```
struct Patient p1 = { 180.75 , 73, 23 }; //initialization
```

or,

```
struct Patient p1;
p1.height = 180.75;     /*initialization of each member separately */
p1.weight = 73;
p1.age = 23;
```

Size of structure

In C language, sizeof() operator is used to calculate the size of **structure**, **variables**, **pointers** or data types, data types could be pre-defined or user-defined. Using the sizeof() operator we can calculate the size of the structure straightforward to pass it as a parameter.

```
#include <stdio.h>

struct A {
    int a;
    int* b;
    char c;
    char* d;
};

int main()
{
    struct A a1;
    printf("Size of struct A: %lu\n", sizeof(struct A));
    printf("Size of object a1: %lu\n", sizeof(a1));
    return 0;
}
```

Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable. Example : struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#include<stdio.h>

struct Employee
{
   char ename[10];
   int sal;
};

struct Employee emp[5];
int i, j;
void ask()
{
   for(i = 0; i < 3; i++)
   {
      printf("\nEnter %dst Employee record:\n", i+1);
      printf("\nEmployee name:\t");
      scanf("%s", emp[i].ename);
      printf("\nEnter Salary:\t");
      scanf("%d", &emp[i].sal);
   }
   printf("\nDisplaying Employee record:\n");
   for(i = 0; i < 3; i++)
   {
      printf("\nEmployee name is %s", emp[i].ename);
      printf("\nSlary is %d", emp[i].sal);
   }
}
void main()
{
   ask();
}
```

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

output

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700

Union in C

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

Advantages of union

Here, are pros/benefits for using union:

1. It occupies less memory compared to structure.
2. When you use union, only the last variable can be directly accessed.
3. Union is used when you have to use the same memory location for two or more data members.
4. It enables you to hold data of only one data member.
5. Its allocated space is equal to maximum size of the data member.

Structure	Union
You can use a struct keyword to define a structure.	You can use a union keyword to define a union.
Every member within structure is assigned a unique memory location.	In union, a memory location is shared by all the data members.
Changing the value of one data member will not affect other data members in structure.	Changing the value of one data member will change the value of other data members in union.
It enables you to initialize several members at once.	It enables you to initialize only the first member of union.
The total size of the structure is the sum of the size of every data member.	The total size of the union is the size of the largest data member.
It is mainly used for storing various data types.	It is mainly used for storing one of the many data types that are available.
It occupies space for each and every member written in inner parameters.	It occupies space for a member having the highest size written in inner parameters.
You can retrieve any member at a time.	You can access one member at a time in the union.
It supports flexible array.	It does not support a flexible array.

following example displays the total memory size occupied by the above union −

Live Demo

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

union Data {
   int i;
   float f;
   char str[20];
};

void main( ) {
clrscr();
   union Data data;
   printf( "Memory size occupied by data : %d\n", sizeof(data));
   getch();
}
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program −

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

union Data {
   int i;
   float f;
   char str[20];
};

void main( ) {
   union Data data;
clrscr ();

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Ouptut

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

- Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.
- Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

ARRAY	STRUCTURE
Array refers to a collection consisting of elements of homogeneous data type.	Structure refers to a collection consisting of elements of heterogeneous data type.
Array uses subscripts or “[ ]” (square bracket) for element access	Structure uses “.” (Dot operator) for element access
Array is pointer as it points to the first element of the collection.	Structure is not a pointer
Instantiation of Array objects is not possible.	Instantiation of Structure objects is possible.
Array size is fixed and is basically the number of elements multiplied by the size of an element.	Structure size is not fixed as each element of Structure can be of different type and size.
Bit filed is not possible in an Array.	Bit filed is possible in an Structure.
Array declaration is done simply using [] and not any keyword.	Structure declaration is done with the help of “struct” keyword.
Arrays is a non-primitive datatype	Structure is a user-defined datatype.
Array traversal and searching is easy and fast.	Structure traversal and searching is complex and slow.
data_type array_name[size];	struct sruct_name{ data_type1 ele1; data_type2 ele2; };
Array elements are stored in continuous memory locations.	Structure elements may or may not be stored in a continuous memory location.
Array elements are accessed by their index number using subscripts.	Structure elements are accessed by their names using dot operator.

Pointers in C

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

```
int n = 10;
int* p = &n;
 // Variable p of type pointer is pointing to the address of the variable n of the type integer.
```

Pointer declaration in C

The pointer in c language can be declared using * (asterisk symbol).

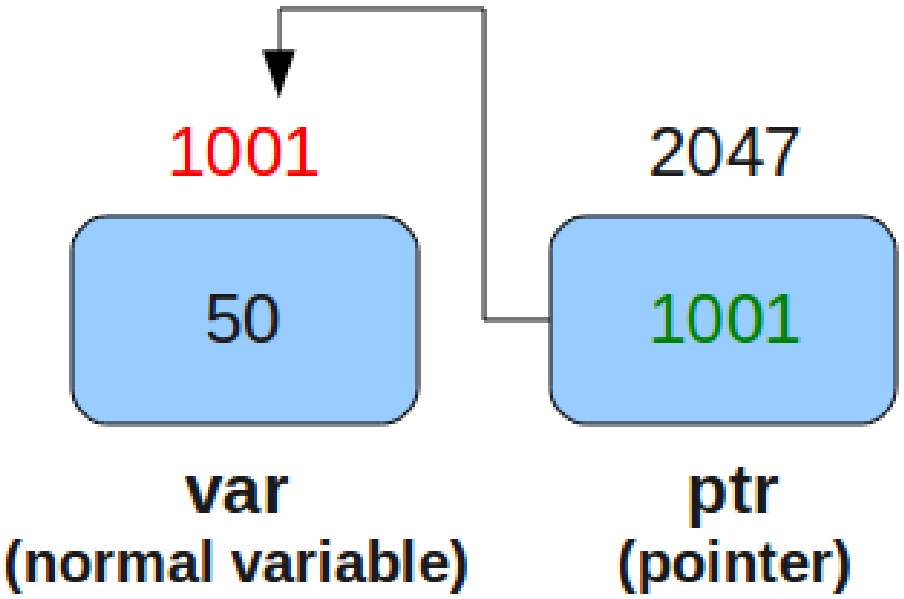
The general syntax of pointer declaration is,

datatype *pointer_name;

Note: The data type of the pointer and the variable to which the pointer variable is pointing must be the same.

Examples:

- **int** *a;//pointer to int
- **char** *c;//pointer to char



Advantages of Using Pointers

- Less time in program execution
- Working on the original variable
- With the help of pointers, we can create data structures (linked-list, stack, queue).
- Returning more than one values from functions
- Searching and sorting large data very easily
- Dynamically memory allocation

Disadvantages of Using Pointers

- Sometimes by creating pointers, such errors come in the program, which is very difficult to diagnose.
- Sometimes pointer leaks in memory are also created.
- If extra memory is not found then a program crash can also occur.

NULL pointer in C

The pointer variable which is initialized with the null value is called the Null Pointer. Null Pointer doesn't point to any memory location until we are not assigning the address. The size of the Null pointer also is 2 bytes according to DOS Compiler.

int * pInt = NULL;

Some uses of the null pointer are:

a) To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.

b) To pass a null pointer to a function argument when we don't want to pass any valid memory address.

c) To check for null pointer before accessing any pointer variable. So that, we can perform error handling in pointer related code e.g. dereference pointer variable only if it's not NULL.

```
#include <stdio.h>
#include<conio.h>
void main() {
   int *p= NULL;//initialize the pointer as null.
   printf("The value of pointer is %u",p);
   getch();
}
```

C Pointers Operators that are used with Pointers

“Address of”(&) Operator

We have already seen in the first example that we can display the address of a variable using ampersand sign. I have used &num to access the address of variable num. The & operator is also known as “Address of” Operator.

printf("Address of var is: %p", &num);

//The %p format specifier is **used for printing the value of a pointer in C**.

“Value at Address”(*) Operator

The * Operator is also known as **Value at address** operator. It gives the value stored at a particular address. The ‘value at address’ operator is also called ‘indirection’ operator.

*q = *m;*

```
\* Pointer to initialize and print the
 value and address of variable. *\
#include<conio.h>
# include < stdio.h >
void  main( )
{
int  a = 25 ;
int  *b ;
b = &a ;
printf("\n Address of a = %u ",  & a) ;
printf("\n Address of a = %u ",  b) ;
printf("\n Address of b = %u ",  & b) ;
printf("\n Value of b = %u ",  b) ;
printf("\n Value of a = %d ",  a) ;
printf("\n Value of a = %d ",  *( &a ) )) ;
printf("\n Value of a = %d ",  *b) ;
getch();
}
```

Output of the program :

```
Address of a = 12345
Address of a = 12345
Address of b = 12345
Value of b = 12345
Value of a = 5
Value of a = 5
Value of a = 5
```

```
#include <stdio.h>
#include<conio.h>
void main()
{
  /* Pointer of integer type, this can hold the
   * address of a integer type variable.
   */
  int *p;

  int var = 10;

  /* Assigning the address of variable var
to the pointer
   * p. The p can hold the address of var
because var is
   * an integer type variable.
   */
  p= &var;

  printf("Value of variable var is: %d", var);
  printf("\nValue of variable var is: %d", *p);
  printf("\nAddress of variable var is: %p", &var);
  printf("\nAddress of variable var is: %p", p);
  printf("\nAddress of pointer p is: %p", &p);
  getch();
}
```

Output:

```
Value of variable var is: 10
Value of variable var is: 10
Address of variable var is: 0x7fff5ed98c4c
Address of variable var is: 0x7fff5ed98c4c
Address of pointer p is: 0x7fff5ed98c50
```

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Incrementing Pointer in C

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

new_address= current_address + i * size_of(data type)

Where i is the number by which the pointer get increased.

For 32-bit int variable, it will be incremented by 2 bytes.

For 64-bit int variable, it will be incremented by 4 bytes.

```
#include<stdio.h>
#include<conio.h>
void main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After increment: Address of p variable is %u \n",p); // in our case, p w
ill get incremented by 4 bytes.
getch();
}
```

Output

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

Traversing an array by using pointer

```
#include<stdio.h>
void main ()
{
    int arr[5] = {1, 2, 3, 4, 5};
    int *p = arr;
    int i;
    printf("printing array elements...\n");
    for(i = 0; i< 5; i++)
    {
        printf("%d  ",*(p+i));
    }
}
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

new_address= current_address - i * size_of(data type)

For 32-bit int variable, it will be decremented by 2 bytes.

For 64-bit int variable, it will be decremented by 4 bytes.

```
#include<stdio.h>
#include<conio.h>
void main ()  {
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After decrement: Address of p variable is %u \n",p);
// P will now point to the immidiate previous location.
getch();
}
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:.
new_address= current_address + (number * size_of(data type))

For 32-bit int variable, it will add 2 * number.

For 64-bit int variable, it will add 4 * number.

```
#include<stdio.h>
#include<conio.h>
void main ()  {
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
getch();
}
```

output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

as you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

new_address= current_address - (number * size_of(data type))

For 32-bit int variable, it will subtract 2 * number.

For 64-bit int variable, it will subtract 4 * number.

```
#include<stdio.h>
#include<conio.h>
void main ()
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
getch();
}
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

Consider the following example to subtract one pointer from an another.

```
#include<stdio.h>
#include<conio.h>
void main ()
{
   int i = 100;
   int *p = &i;
   int *temp;
   temp = p;
   p = p + 3;
   printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);
getch();
}
```

Output

Pointer Subtraction: 1030585080 - 1030585068 = 3

Pointer Comparison in C

In C language pointers can be compared if the two pointers are pointing to the same array.

All relational operators can be used for pointer comparison, but a pointer cannot Multiplied or Divided.

Below is a program on pointer comparison for same type of pointer:

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

void main()
{
   int *ptrA,*ptrB;
   ptrA = (int *)1;
   ptrB = (int *)2;

   if(ptr2 > ptr1)
      printf("PtrB is greater than ptrA");

getch();
}
```

Below is a program on pointer comparison for different type of pointer:

```
#include <stdio.h>
#include<conio.h>
void main(){
{
   int *ptrA;
   float *ptrB;

   ptrA = (int *)1000;  //type casting pointer as interger
   ptrB = (float *)2000; // type casting pointer as float

   if(ptrB > ptrA)
     printf("PtrB is greater than ptrA");

          getch();
}
```

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

Address + Address = illegal

Address * Address = illegal

Address % Address = illegal

Address / Address = illegal

Address & Address = illegal

Address ^ Address = illegal

Address | Address = illegal

~Address = illegal

```
//sum of two numbers using pointers
#include <stdio.h>
#include<conio.h>
void main()
{
   int first, second, *p, *q, sum;
clrscr();
   printf("Enter two integers to add\n");
   scanf("%d%d", &first, &second);
   p = &first;
   q = &second;
   sum = *p + *q;
   printf("Sum of the numbers = %d\n", sum);
   getch();
}
```

```
#include <stdio.h>
#include<conio.h>
void main() {
   int data[5];
          clrscr();
   printf("Enter elements: ");
   for (int i = 0; i < 5; ++i)
      scanf("%d", data + i);

   printf("You entered: \n");
   for (int i = 0; i < 5; ++i)
      printf("%d\n", *(data + i));
 getch();
}
```

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

There are 2 kinds of files in which data can be stored in 2 ways either in characters coded in their ASCII character set or in binary format. They are

- **Text Files.**

A Text file contains only the text information like alphabets ,digits and special symbols. The ASCII code of these characters are stored in these files.It uses only 7 bits allowing 8 bit to be zero.

- **Binary Files**

A binary file is a file that uses all 8 bits of a byte for storing the information .It is the form which can be interpreted and understood by the computer.

The only difference between the text file and binary file is the data contain in text file can be recognized by the word processor while binary file data can’t be recognized by a word processor.

EOF

EOF in any programming language is for End Of File.

So if you are trying to display the contents of file and want to stop when the fil

The EOF function **returns False until the end of the file has been reached**. It returns true when the file ends.

getc() and feof() // function for EOF in C

Types of File access in C

depending up on the method of accessing the data stored ,there are two types of files.

- **Sequential file**

In this type of files data is kept in sequential order if we want to read the last record of the file, we need to read all records before that record so it takes more time.

- **Random access file**

In this type of files data can be read and modified randomly .If we want to read the last record we can read it directly. It takes less time when compared to sequential file.

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

fopen() function

The **fopen()** function in C is a library function that is used to open a file to perform various operations which include reading, writing etc. along with various modes. If the file exists then the particular file is opened else a new file is created.

Syntax: FILE *fptr;

fptr = fopen("fileName","mode");

For example,

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

Sr.No.	Mode & Description
1	**"r"** Opens a file for reading. The file must exist.
2	**"w"** Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.
3	**"a"** Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.
4	**"r+"** Opens a file to update both reading and writing. The file must exist.
5	**"w+"** Creates an empty file for both reading and writing.
6	**"a+"** Opens a file for reading and appending.

fclose()

- Declaration: int **fclose**(FILE *fp);
- fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.
  fclose (fp);

Example:

fclose(fptr);

fprintf()

The fprintf() function is used to write set of characters into file.

Declaration: int fprintf(FILE *fp, const char *format, …)

fprintf() function is used to write formatted data into a file. In a C program, we use fprinf() as below.
fprintf (fp, “%s %d”, “var1”, var2);

Where, fp is file pointer to the data type “FILE”.
var1 – string variable
var2 – Integer variable
This is for example only. You can use any specifiers with any data type as we use in normal printf() function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i, n=2;
   char str[50];
     //open file sample.txt in write mode
   FILE *fptr = fopen("sample.txt", "w");
   if (fptr == NULL)
   {
      printf("Could not open file");
      return 0;
   }
  for (i = 0; i < n; i++)
   {
      puts("Enter a name");
      scanf("%s", str);
      fprintf(fptr,"%d.%s\n", i, str);
   }
   fclose(fptr);

   getch();
}
```

fscanf()

fscanf() function is used to read formatted data from a file. In a C program, we use fscanf() as below.

Declaration: int fscanf(FILE *fp, const char *format, …)

fscanf (fp, “%d”, &age);
Where, fp is file pointer to the data type “FILE”.
Age – Integer variable
This is for example only. You can use any specifiers with any data type as we use in normal scanf() function.

```
/*c program demonstrating fscanf and its usage*/
#include<stdio.h>
#include<conio.h>
void main()
{
	FILE* ptr = fopen("abc.txt","r");
	if (ptr==NULL)
	{
		printf("no such file.");
		return 0;
	}

	/* Assuming that abc.txt has content in below
	format
	NAME AGE CITY
	abc	12 hyderbad
	bef	25 delhi
	cce	65 bangalore */
	char buf[100];
	while (fscanf(ptr,"%*s %*s %s ",buf)==1)
		printf("%s\n", buf);

	getch();
}
```

putw()

putw function is used to write an integer into a file. In a C program, we can write integer value in a file as below.

Declaration: int putw(int number, FILE *fp);
putw(i, fp);

where
i – integer value
fp – file pointer

getw()

getw function reads an integer value from a file pointed by fp. In a C program, we can read integer value from a file as below.

Declaration: int getw(FILE *fp);

getw(fp);

```
#include<stdio.h>
#include<conio.h>
void main( ){
   FILE *fp;
   int i;
   fp = fopen ("num.txt", "w");
   for (i =1; i<= 10; i++){
      putw (i, fp);
   }
   fclose (fp);
   fp =fopen ("num.txt", "r");
   printf ("file content is\n");
   for (i =1; i<= 10; i++){
      i= getw(fp);
      printf ("%d",i);
      printf("\n");
   }
   fclose (fp);
   getch();
}
```

fgetc()

fgetc functions is used to read a character from a file. It reads single character at a time. In a C program, we use fgetc() function as below.
fgetc (fp);

Declaration: int **fgetc**(FILE *fp)

where,
fp – file pointer

fputc()

fputc functions is used to write a character into a file. In a C program, we use fputc() function as below.

Declaration: int fputc(int char, FILE *fp)
fputc(ch, fp);

where,
ch – character value
fp – file pointer

```
#include<stdio.h>
#include<conio.h>
void main() {
   FILE *f;
   char s;
   clrscr();
   f=fopen("new.txt","r");
   while((s=fgetc(f))!=EOF) {
      printf("%c",s);
   }
   fclose(f);
   getch();
}
```

```
#include<stdio.h>
#include<conio.h>
void main() {
   FILE *f;
   f = fopen("new.txt", "w");
   fputc('a',f);
   fclose(f);
 getch();
}
```

```
**
 * C program to create a file and write data into file.
 */
#include <stdio.h>
#include <stdlib.h>
#define DATA_SIZE 1000
int main()
{
   /* Variable to store user content */
   char data[DATA_SIZE];
   /* File pointer to hold reference to our file */
   FILE * fPtr;
   /*
    * Open file in w (write) mode.
    * "data/file1.txt" is complete path to create file
    */
   fPtr = fopen("data/file1.txt", "w");

   /* fopen() return NULL if last operation was
unsuccessful */
   if(fPtr == NULL)
   {
      /* File not created hence exit */
      printf("Unable to create file.\n");
      exit(EXIT_FAILURE);
   }
   /* Input contents from user to store in file */
   printf("Enter contents to store in file : \n");
   fgets(data, DATA_SIZE, stdin);

   /* Write data to file */
   fputs(data, fPtr);
   /* Close file to save file data */
   fclose(fPtr);
   /* Success message */
   printf("File created and saved successfully. :) \n");
getch();
}
```

Write a C program to read name and marks of n number of students from user and store them in a file

```
#include <stdio.h>
#include<conio.h>
void main() {
	char name[50];
	int marks,i,n;
	printf("Enter number of students: ");
	scanf("%d",&n);
	FILE *fptr;
	fptr=(fopen("C:\\student.txt","w"));
	if(fptr==NULL) {
		printf("Error!");
		exit(1);
	}
	for (i=0;i<n;++i) {
		printf("For student%d\nEnter name: ",i+1);
		scanf("%s",name);
		printf("Enter marks: ");
		scanf("%d",&marks);
		fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
	}
	fclose(fptr);
	getch();
}
```

Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.

```
#include <stdio.h>
#include<conio.h>
#include<stdlib.h>
void main() {
char name[50];
int marks,i,n;
printf("Enter number of students: ");
scanf("%d",&n);
FILE *fptr;
fptr=(fopen("C:\\student.txt","a"));
if(fptr==NULL) {
printf("Error!");
exit(1);// this function is in stdlib.h
}
for (i=0;i<n;++i) {
printf("For student%d\nEnter name: ",i+1);
scanf("%s",name);
printf("Enter marks: ");
scanf("%d",&marks);
fprintf(fptr,"\nName: %s \nMarks=%d \n",name,marks);
}
fclose(fptr);
getch();
}
```

C:\Users\user\Desktop

C program to rename a file

```
#include <stdio.h>
#include<conio.h>
int main () {
   int ret;
   char oldname[] = "file.txt";
   char newname[] = "newfile.txt";

   ret = rename(oldname, newname);

   if(ret == 0) {
      printf("File renamed successfully");
   } else {
      printf("Error: unable to rename the file");
   }

   getch();
}
```