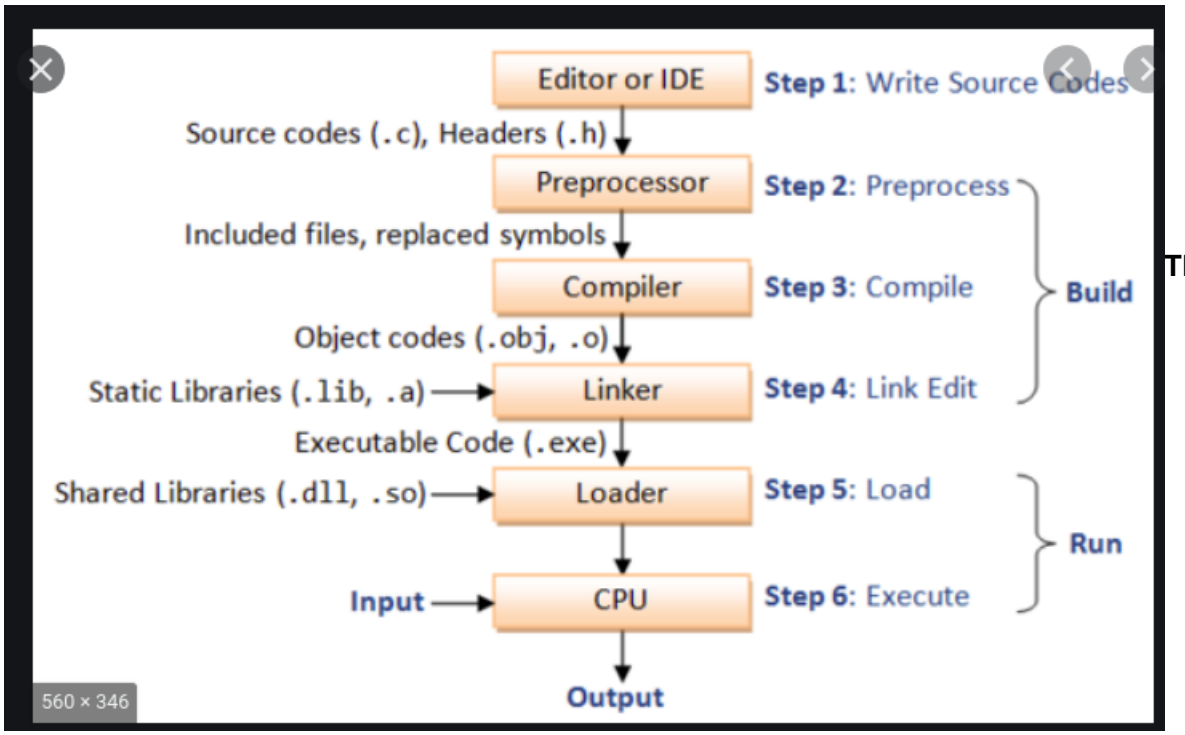
1) Basics:
C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system. The main features of the C language include low-level memory access.

Compilation steps in c:
// .c file is processed and .i file is generated and can be checked by gcc with -E option. After preprocessing, compiler compiles for syntax error and
// now asm file is generated. Assembler converts asm to obj file. from obj .out file is generated. object file contains different memory sections:
// data section(read only/read & write for global initialized variables). //BSS: (read only/read & write for global uninitialized variables)
//stack: for local vars of functions.

Pre processing

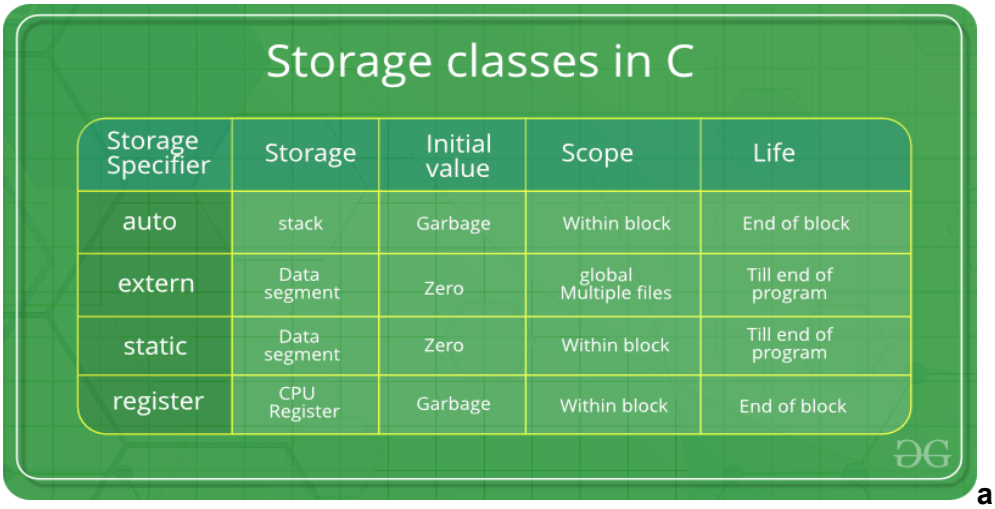
- Removal of Comments
- Expansion of Macros
- Expansion of the included files.
- Conditional compilation



This command will generate all files: g++.exe test1.cpp -o test1 -save-temps

---

2) **Storage class in c**



auto:
This is the default storage class for all the variables declared inside a function or a block. It is stored in stack.
static : it is accessible in same file only not in multiple files. It is stored in data section (read only if const. else writable of data section). static variables cannot be reinitialized.
Can we declare static variable in header file?? static means that the variable is only used within your compilation unit and will not be exposed to the linker, so if you have a static int in a header file and include it from two separate .c files, you will have two discrete copies of that int, which is most likely not at all what you want**.**
Extern:
globally declared variables and functions can be declared as extern in other files or same file. declaration can be done multiple times but definition must be once.
//extern int var = 0; -> variable is declared and defined both so errors.

//extern int var; --> variable is decalared extern but no definition is found, even no header file inclusion means no definition is present so compile error.
//int main(void)
//{
// var = 10;
// return 0;
//}
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.

//Declaration and definition: int a; --> is declaration and definition both.... extern int a;-> only declaration.

Volatile: The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Usage of volatile:

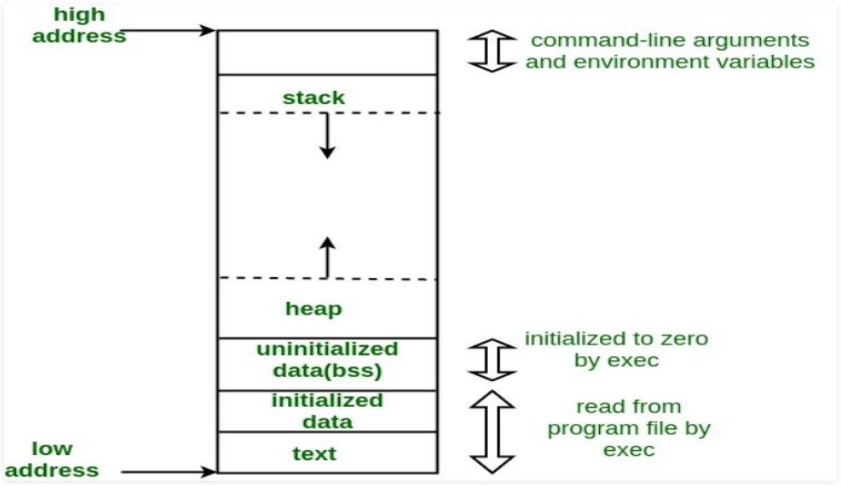
1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program.

2) *Global variables within a multi-threaded application:* There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread.

---

3) Memory layout:

Text segment (i.e. instructions): Machine code of compiled program.
Initialized data segment: Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
Uninitialized data segment (bss): Uninitialized data segment often called the “bss” segment, named after an ancient assembler operator, that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
Heap: Heap is the segment where dynamic memory allocation usually takes place.
The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
Stack: Where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack.



Virtual memory get available to PCB for allocating space at run time. While heap can grow to all available memory, mostly stack can't be grows whenever required.

/* code with memory leak */

```
int main(void)
{
        int *ptr = (int*)malloc(10);

        return 0;
}
```

//Check the leak summary with valgrind tool. It shows memory leak of 10 bytes, which is highlighed in red colour.

```
valgrind –leak-check=full ./free
==1238== LEAK SUMMARY:
==1238==    definitely lost: 10 bytes in 1 blocks.
```

```
int main(void)

{

        int *ptr = (int*) malloc(10);

        /* we are calling realloc with size = 0 */

        realloc(ptr, 0);

        return 0;

}
```

Difference Between malloc() and calloc(): The name **malloc** and **calloc()** are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from the heap segment.

Malloc: Allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn't initialize the allocated memory. If we try to access the content of memory block(before initializing) then we'll get segmentation fault error(or maybe garbage values). **`void* malloc(size_t size);`**

calloc: Allocates the memory and also initializes the allocated memory block to zero. If we try to access the content of these blocks then we'll get 0. **`void* calloc(size_t num, size_t size);`**

How does free() know the size of memory to be deallocated: When memory allocation is done, the actual heap space allocated is one word larger than the requested memory. The extra word is used to store the size of the allocation and is later used by free( )

---

4) Functions:

- **Parameter Passing to functions:**

- **Pass by Value:** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

- **Pass by Reference** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

- **Functions that are executed before and after main() in C:**

```
/* Apply the constructor attribute to myStartupFun() so that it is ex
ecuted before main() */
      void myStartupFun (void) __attribute__ ((constructor));

      /* Apply the destructor attribute to myCleanupFun() so that it
      is executed after main() */
      void myCleanupFun (void) __attribute__ ((destructor));

      /* implementation of myStartupFun */
      void myStartupFun (void)
      {
            printf ("startup code before main()\n");
      }
      /* implementation of myCleanupFun */
      void myCleanupFun (void)
      {
            printf ("cleanup code after main()\n");
      }
      int main (void)
      {
            printf ("hello\n");
            return 0;
      }
```

- **return statement vs exit() in main():**When *exit(0)* is used to exit from program, destructors for locally scoped non-static objects are not called. But destructors are called if return 0 is used. Can be checked using c++ program.

- **What is evaluation order of function parameters in C?:** It is compiler dependent in C. It is never safe to depend on the order of evaluation of side effects.

- **Does C support function overloading?** No but generic function can be implemented by using void*.

  int foo(void * arg1, int arg2);

- **Function pointers**

```
void fun(int a)
```

```
{
}
int main()
{
// fun_ptr is a pointer to function fun()
void (*fun_ptr)(int) = &fun;

    // Invoking fun() using fun_ptr
(*fun_ptr)(10);
}
// fun_ptr_arr is an array of function pointers
void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
```

- **call back functions:**

```
typedef int callback_t;

callback_t add(int x, int y)
{
      return(x+y);
}
callback_t sub(int x, int y)
{
      return((x>y)?(x-y):(y-x));
}

//callback functions:array of function pointers
callback_t (*fun[])(int,int) = {add,sub};

int main()
{
      printf("value after addition is =%d\n",(*fun[0])(5,4));
      printf("value after subtraction is =%d\n",(*fun[1])(5,4));
      return 0;
}
```

- **Generic function:**We can write generic function using void pointer:

```
#include<stdio.h>

enum type {
    TYPE_CHAR,
    TYPE_INT,
    TYPE_FLOAT
};

void* add(enum type t, void *vp1, void *vp2)
{
      int k, l;
    switch (t) {
        case TYPE_CHAR:
        break;
```

```
        case TYPE_INT:
        k = (*(int *)vp1);
            l = (*(int *)vp2);
            return &(k+l);

            case TYPE_FLOAT:
        break;
    }
}

int main()
{
      int i=5;
      int j = 6;

      int ret = add(TYPE_INT, &i, &j);
      printf("%d\n",ret);
      return 0;
}
```

- **Control statements:**

Switch:

- switch is alternate way for lot of if-elseif. every case must have break but not in "default".
- The expression provided in the switch should result in a constant value otherwise it would not be valid.
- Duplicate case values are not allowed.
- The default statement is optional.Even if the switch case statement do not have a default statement,it would run without any problem.
- **range in switch case:**

```
switch (arr[i])
{
      case 1 ... 6:
      cout << arr[i] <<" in range 1 to 6\n";
      break;
      default:
      cout << arr[i] <<" not in range\n";
      break;
}
```

-
-

If-elseif-else:

- if takes zero and non zero values. If can take fractional also ie. 3.5(non zero)
- if-elseif-else: if is true then control will not go to elseif and else. if is not true then elseif will be checked.

- **Array:** An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed

randomly using indices of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type**.**

- **Array declaration by specifying size**
```
int arr1[10];
```
- **declare an array of user specified size**
```
int n = 10;
int arr2[n];
```
- **Array declaration by initializing elements**
```
int arr[] = { 10, 20, 30, 40 }
```
- **Array declaration by specifying size and initializing** elements
```
int arr[6] = { 10, 20, 30, 40 }
```
- **Strings in C:**
  Initializing a String:
  //char str[] = "GeeksforGeeks";
  //char str[50] = "GeeksforGeeks";
  //char str[] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};
  //char str[14] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};

  //while using scanf to read input from user for char array. we do not use '&'. ex:// reading string: scanf("%s",str); --> because str without '[' and ']' is base
  //address of string.

  /puts(str); --> print string to console(standard output)
  //int fputs(const char *str, FILE *stream)--> writes a string to the specified stream up
  // c = fgetc(fp); --> reads character from file stream.
  //char* str2 = (char*)malloc(sizeof(char) * size); -->allocating dynamic memory to char*.
  //const char* str = "This is GeeksForGeeks"; --> if u don't use const it will throw warning. beacause rhs is constant.
  //// const char* str = "This is GeeksForGeeks"; and then str[1] = 'o'; --> gives compile time error because trying to modify read only value.
  //char str[] = "Hello"; and then str[1] = 'o'; it's correct.
  // str1[] = "Hi"; str2[] = "Bye"; we can not assign str1 = str2 (or str2 = str1) --> constants can not be assigned...
  // char*str1 = "hi"; char*str2 ="bye"; str1 = str2(correct)
  //const char *p ----> p is pointer, pointing to const char*(means char* value can't be modified....
  //char * const p ----> const p is pointing to char*(p can not point to other string)
  //const char * const p ----> both r constant means string can not be modified and also p can not point to other string.

  // programs: strlen, strcpy, strcat, strcmp, string reversing

---

Pointers:

Ex:

```
// Declare an array
int val[3] = { 5, 10, 15};

// Declare pointer variable
```

```
int *ptr;

// Assign address of val[0] to ptr.
// We can use ptr=&val[0];(both are same)
ptr = val ;
cout << "Elements of the array are: ";
cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
```

- **Dangling pointer: A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.**
- **`int *ptr = (int *)malloc(sizeof(int));`**

```
// After below free call, ptr becomes a
// dangling pointer
free(ptr);

// No more a dangling pointer
ptr = NULL;
```

Ex: Dangling pointer:

```
// The pointer pointing to local variable becomes
// dangling when local variable is not static.
int *fun()
{
        // x is local variable and goes out of
        // scope after an execution of fun() is
        // over.
        int x = 5;  //if we take it static then, no dangling pointer...

        return &x;
}

// Driver Code
int main()
{
        int *p = fun();
        fflush(stdin);

        // p points to something which is not
        // valid anymore
        printf("%d", *p);
        return 0;
}
```

- **Void pointer:** Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type.

void pointers cannot be dereferenced. It can however be done using typecasting the void pointer. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

- **Wild pointer:** A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.
- **Pointer variable can be assigned a value whereas array variable cannot be:**

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

- **Arithmetic on pointer variable is allowed.**

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

- **Near pointer is used to store 16 bit addresses means within current segment on a 16 bit machine. The limitation is that we can only access 64kb of data at a time.**
- **A far pointer is typically 32 bit that can access memory outside current segment. To use this, compiler allocates a segment register to store segment address.**
- **huge pointer is also typically 32 bit and can access outside segment. In case of far pointers, a segment is fixed. In far pointer, the segment part cannot be modified, but in Huge it can be.**
- **Pointer to integer variable:**
  int *p;
  int x =10;
  p=&x; //p is pointing to x means p is having address of variable x. *p will give value at x and p will have x's address.

- **Pointer to an array:**
  //int *p; int arr[] = {1,2,3,4}; p=arr; -> p is pointing to arr base address. p[0] or *p will give value of arr[0]. p[1] or *(p+1) will give value of arr[1]. ***p*: is pointer to 0^{th} element of the array *arr. The base type of p is int.*** ***`Assign address of arr[0] to ptr or arr is same.`***

- **Pointer to an array of 4 integers**
  //int arr[] = {1,2,3,4}; int (*p)[4]--> This pointer to an array of 4 integers. p= &arr; p[0][0]-> arr[0], p[0][1] is equal to arr[1]. ***ptr* is a pointer that points to the whole array *arr*. so if we write p++, then the pointer *ptr* will be shifted forward by 16 bytes.**

- **More pointer concepts:**
  /Ex: const int a =10; const int *p; p = &a;
  //*p = 20;--> a cannot be modified through p(since p is holding address of int and it should be const)
  // a =10; --> a is constant, cannot be modified
  //int *const p -> p cannot hold any other address.
  //const int *constp-> p cannot hold any other address and a cannot be modified by p.

//int *p; --> int *

Enum, Struct and Union:

- **Enumeration (or enum):** is a 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.

```
enum week{Mon, Tue, Wed};
enum week day;
      Or
enum week{Mon, Tue, Wed}day;
```

//enum State {Working = 1, Failed = 0};
//Two enum names can have same value: enum State {Working = 1, Failed = 0, Freezed = 0};
// If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0
//We can assign values to some name in any order. All unassigned names get value as value of previous name plus one: enum day {sunday = 1, monday, tuesday = 5,
//wednesday, thursday = 10, friday, saturday};

- **Structure:** A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

```
// A variable declaration with structure declaration.

struct address
{
    char name[50];
    char street[100];
    char city[50];
    char state[20];
    int pin;
};
```

  - **Structure members cannot be initialized with declaration:**

```
struct Point
{
    int x = 0; // COMPILER ERROR: cannot initialize
members here
    int y = 0; // COMPILER ERROR: cannot initialize
members here
};
```

  - **Structure members can be initialized using curly braces '{}'**

```
struct Point p1 = {0, 1};
```

  - **Structure members are accessed using dot (.) operator:**

```
Accessing members of point p1
p1.x = 20;
```

  - **array of structures:**

```
struct Point arr[10];
```

- **Union:** is a user defined data type. In union, all members share the same memory location.

both x and y share the same location. If we change x, we can see the changes being reflected in y:

```
union test
{
      int x, y;
};

int main()
{
      // A union variable t
      union test t;
      t.x = 2; // t.y also gets value 2
                        printf("After making x = 2:\n x
      = %d, y = %d\n\n",
      t.x, t.y);
}
```

- Size of a union is taken according the size of largest member in union.

- **Structure Padding:**

To align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.

Architecture of a computer processor is such a way that it can read 1 word (4 bytes 32 bit processor) from memory at a time.

To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member's address.

Ex: struct student
{
int id1;
int id2;
char a;
char b;
float percentage;
} sizeof struct student should be(4+4+1+1+4) = 14 but it's wrong. size will be(4+4+4(for two chars)+4)= 16 due to structure alignment.

//Structure padding takes extra memory sapce so need to avoid structure padding. avoid structure padding, we use structure packing.

struct stud
{
int x;
char y;

```
        int z;
    };      >size of struct stud will be (4+4+4) = 12 (due to structure padding)
```

- **Structure packing:**

```
#pragma pack(1)
struct stud
{
        int x;
        char y;
        int z;
};      >size of struct stud will be (4+1+4) = 9 (due to structure packing). pragma pack(1) will
not add 4 bytes for char data type but only 1 byte. If we use pragma pack(2)..it will add 2 bytes for
char data type.
```

- **Bit Fields:**

```
//The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.
//struct date {
// d has value between 1 and 31, so 5 bits
// are sufficient
//    unsigned int d: 5;

 // m has value between 1 and 12, so 4 bits
// are sufficient
//    unsigned int m : 4;

 //   unsigned int y;
//};   sizeof struct date = 8 bytes(4 bytes for int d and int m and 4 bytes for int y)
```

- **Input/Output:**

printf() : It returns total number of Characters Printed, Or negative value if an output error or an encoding error.

scanf() : It returns **total number of Inputs Scanned successfully**, or EOF if input failure occurs before the first receiving argument was assigned.

return type of getchar(), fgetc() and getc() is int (not char). So it is recommended To assign the returned values of these functions to an integer type variable.

```
int in;
while ((in = getchar()) != EOF)
{
      putchar(in);
}
```

Operators:

Arithmetic Operators (+, -, *, /, %, post-increment, pre-increment, post-decrement, pre-decrement)
Relational Operators (==, !=, >, <, >= & <=) Logical Operators (&&, || and !)
Bitwise Operators (&, |, ^, ~, >> and <<)
Assignment Operators (=, +=, -=, *=, etc)

Logical Operators:
Logical AND operator: The '&&' operator returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. For example, a && b returns true when both a and b are true (i.e. non-zero).
Logical OR operator: The '||' operator returns true even if one (or both) of the conditions under consideration is satisfied. Otherwise, it returns false. For example, a || b returns true if one of a or b or both are true (i.e. non-zero). Of course, it returns true when both a and b are true.
Logical NOT operator: The '!' operator returns true the condition in consideration is not satisfied. Otherwise it returns false. For example, !a returns true if a is false, i.e. when a=0.

Bitwise operator:

The & (bitwise AND) in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

The | (bitwise OR) in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

The ^ (bitwise XOR) in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

The << (left shift) in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.

The >> (right shift) in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

//Bitwise and(&) is used to reset bit(1->0). If bit is 1 & 0 will reset bit.
//Bitwise or(|) is used to enable bit(0->1). 0|1 will set bit.
//& is used to check if bit is enabled or disabled by performing & with 1 to that bit.

//Bitwise or(|) operations: 1|1 = 1, 0|0 = 0, 1|0 = 1, 0|1 = 1
//Bitwise xor(^) operations: 1^1 = 0, 0^0 = 0, 1^0 = 1, 0^1 = 1
//Bitwise and(&) operations: 1&1 = 1, 0&0 = 0, 1&0 = 0, 0&1 = 0

precedence of Prefix ++ (or Prefix –) has same priority than dereference (*) operator, and precedence of Postfix ++ (or Postfix –) is higher than both Prefix ++ and *.
If p is a pointer then *p++ is equivalent to *(p++) and ++*p is equivalent to ++(*p) (both Prefix ++ and * are right

Ex:
```
// Program 1
#include<stdio.h>
int main()
```

```
{
char arr[] = "geeksforgeeks";
char *p = arr;
++*p;
printf(" %c", *p);
getchar();
return 0;
}
```
o/p: h

Ex:
```
// Program 2
#include<stdio.h>
int main()
{
char arr[] = "geeksforgeeks";
char *p = arr;
*p++;
printf(" %c", *p);
getchar();
return 0;
}
```
O/P: e

Data Type:
Char: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
int: As the name suggests, an int variable is used to store an integer.
float: It is used to store decimal numbers (numbers with floating point value) with single precision.
double: It is used to store decimal numbers (numbers with floating point value) with double precision.

```
//**********File handling**********//
//Reading from a file:
//FILE * filePointer;
//filePointer = fopen(“fileName.txt”, “r”);
//fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);


//Writing a file:
//FILE *filePointer ;
//filePointer = fopen(“fileName.txt”, “w”);
//fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);

//Closing a file
```

```
//FILE *filePointer ;
//filePointer= fopen(“fileName.txt”, “w”);
//---------- Some file Operations -------
//fclose(filePointer)

//fputs can be used for writing string to file:
//char dataToBeWritten[50]  = "GeeksforGeeks-A Computer Science Portal for Geeks";
//fputs(dataToBeWritten, filePointer) ;

//fgets can be used for reading from file:
//while( fgets ( dataToBeRead, 50, filePointer ) != NULL )
```

---

macros vs functions//

```
//By using macro, You get a nice performance as the preprocessor takes care of expanding the code wherever you have used the macro,
// so there is no stack or function call needed for this but You cannot return a parameter.
// You cannot debug a macro. and If the macro is used at multiple places, this will increase your binary size.You cannot do recursion in macro.
```

```
//Is it fine to write “void main()” or “main()”:
//The int returned by main() is a way for a program to return a value to “the system” that invokes it
```

//Command line arguments//

```
//argc (ARGument Count) is int and stores number of command-line arguments passed by the
//user including the name of the program. So if we pass a value to a program,
//value of argc would be 2 (one for argument and one for program name)
//The value of argc should be non negative.
//argv(ARGument Vector) is array of character pointers listing all the arguments.
//If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
//Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.
```

/////////return statement vs exit() in main()

```
//In C++, what is the difference between exit(0) and return 0 ?
//When exit(0) is used to exit from program, destructors for locally scoped
//non-static objects are not called. But destructors are called if return 0 is used.
```

**********file_handling_C**********

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

int main( )
{

   // Declare the file pointer
   FILE *filePointer ;
```

```
    // Get the data to be written in file
    char dataToBeWritten[50]
        = "GeeksforGeeks-A Computer Science Portal for Geeks";

    // Open the existing file GfgTest.c using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("GfgTest.c", "w") ;

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if ( filePointer == NULL )
    {
        printf( "GfgTest.c file failed to open." ) ;
    }
    else
    {

        printf("The file is now opened.\n") ;

        // Write the dataToBeWritten into the file
        if ( strlen (  dataToBeWritten  ) > 0 )
        {

            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer) ;
            fputs("\n", filePointer) ;
        }

        // Closing the file using fclose()
        fclose(filePointer) ;

        printf("Data successfully written in file GfgTest.c\n");
        printf("The file is now closed.") ;
    }
    return 0;
}

**********function_pointer**********
#include <stdio.h>
int sum(int num1, int num2);
int sub(int num1, int num2);
int mult(int num1, int num2);
int div(int num1, int num2);


/*void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}
void subtract(int a, int b)
```

```
{
    printf("Subtraction is %d\n", a-b);
}
void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{
    // fun_ptr_arr is an array of function pointers
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    (*fun_ptr_arr[ch])(a, b);

    return 0;
} */

int main()
{  int x, y, choice, result;
  int (*ope[4])(int, int);
  ope[0] = sum;
  ope[1] = sub;
  ope[2] = mult;
  ope[3] = div;
  printf("Enter two integer numbers: ");
  scanf("%d%d", &x, &y);
  printf("Enter 0 to sum, 1 to subtract, 2 to multiply, or 3 to divide: ");
  scanf("%d", &choice);
  result = ope[choice](x, y);
  printf("%d", result);
return 0;}

int sum(int x, int y) {return(x + y);}
int sub(int x, int y) {return(x - y);}
int mult(int x, int y) {return(x * y);}
int div(int x, int y) {if (y != 0) return (x / y); else  return 0;}
```

Strtok use to tokenize the string:

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

```c
#include <string.h>

int main() {
   char string[50] = "Hello! We are learning about strtok";
   // Extract the first token
   char * token = strtok(string, " ");
   // loop through the string to extract all other tokens
   while( token != NULL ) {
      printf( " %s\n", token ); //printing each token
      token = strtok(NULL, " ");
   }
   return 0;
}
```

strtok() keeps some data inside of itself by using static variables. This way, strtok() can continue searching from the point it left off at during the previous call. To signal strtok() that you want to keep searching the same string, you pass a NULL pointer as its first argument. strtok() checks whether the first argument is NULL and if it is, it uses its currently stored data. If the first parameter is not null, it is treated as a new search and all internal data is reset.