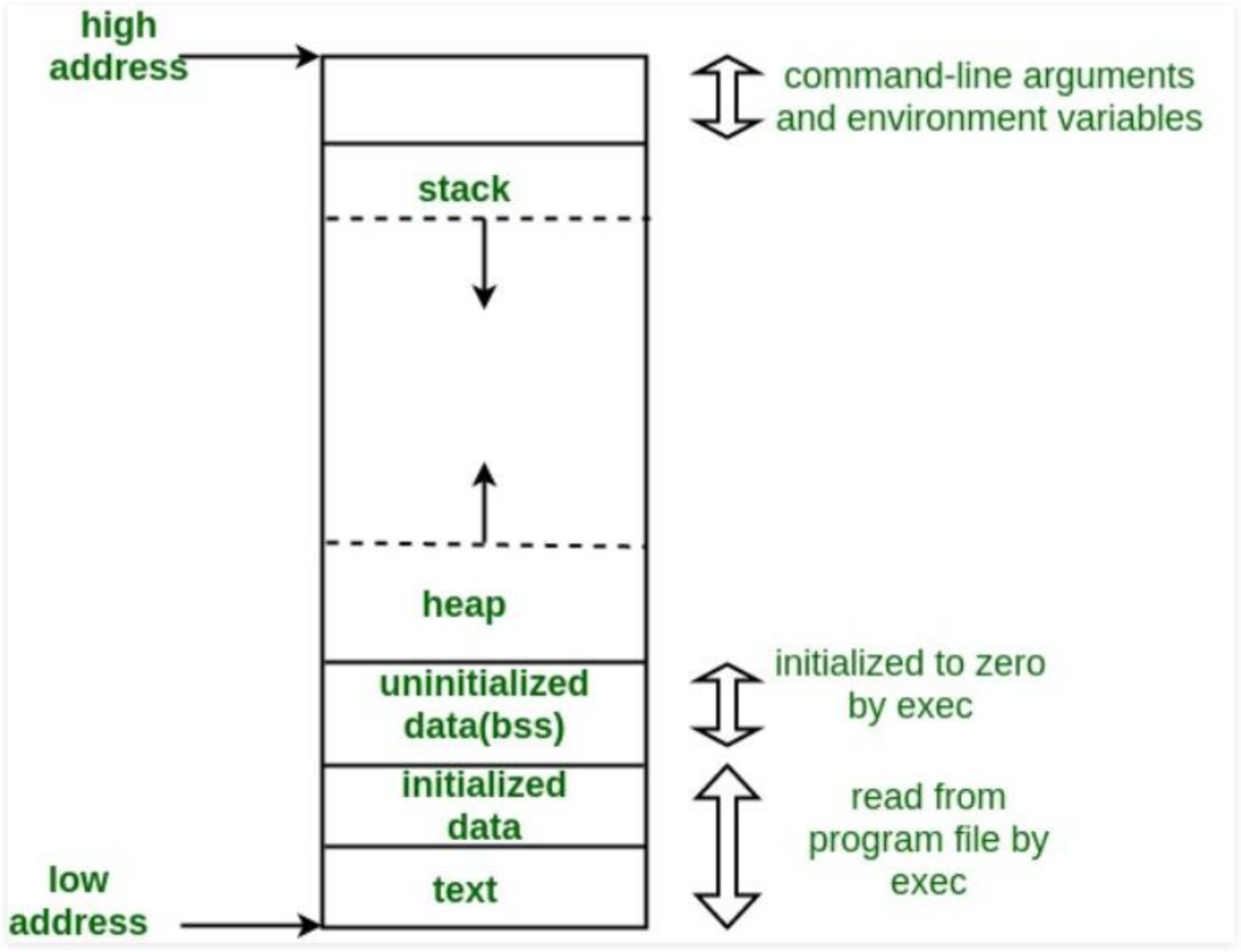
**************************************C**

//C combines the features of both high level and low level languages.
//It can be used for low-level programming, such as scripting for drivers and kernels
//and it also supports functions of high level programming languages,
// such as scripting for software application.

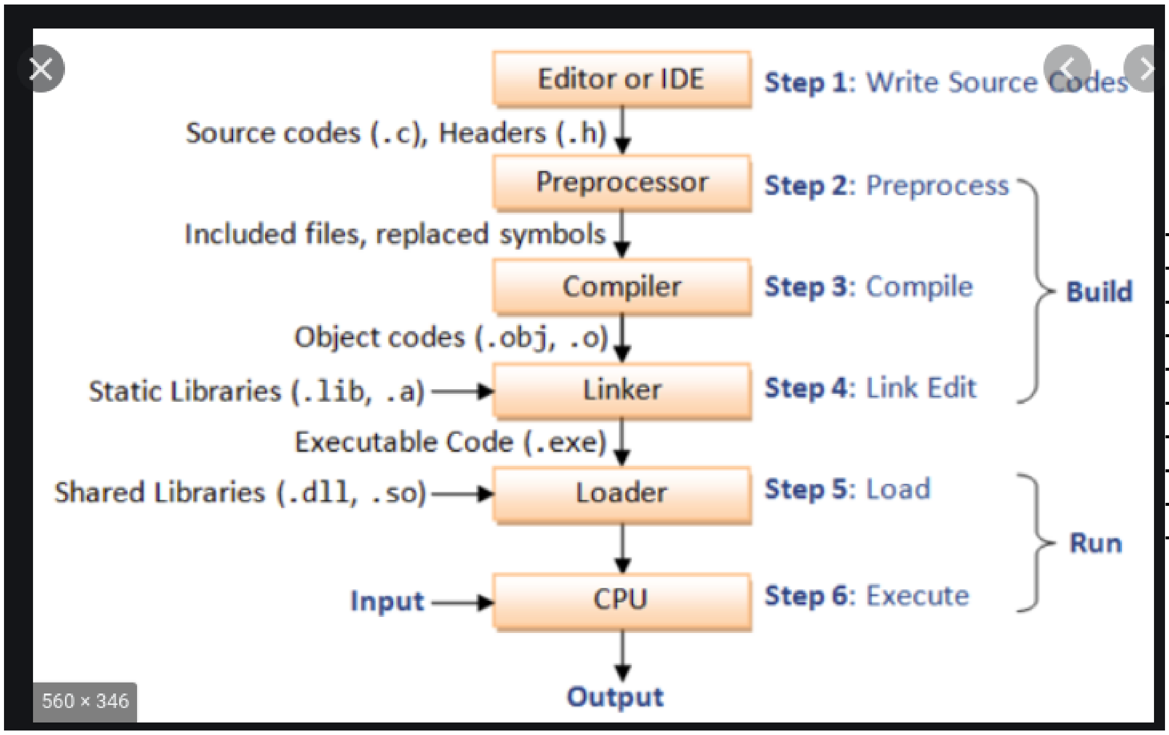
///*Compilation steps in c ******//**
// .c file is processed and .i file is generated and can be checked by gcc with -E option. After preprocessing, compiler compiles for syntax error and
// now asm file is generated. Assembler converts asm to obj file. from obj .out file is generated. object file contains 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.

C memory layout:


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Pre-processing

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



//**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.

//**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.

//*********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 and array************//**

//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.
//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].
//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]
//array of pointers..
//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 *

// programs: array reversing, add element in array, delete element from array, traversing, binary searching,
//linear searching, sorting (selection, bubble and quick sort)

//*************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.

//******Dynamic memory allocation******//**

// char *ptr = (char*)malloc(sizeof(char)*n)

//int *ptr = (int*) calloc(10 ,sizeof (int));

//calloc allocates contiguous memory block, while malloc does not. malloc does not initialize storage, while calloc does.

//*****Structure vs union************//**

//structure and union both are user data types and can store different data types in it but size of structure will be sum of all data types, while size

// of union will be size of largest size data type. In union, value for last updated datatypes will overwrite

//previous values so other values will become garbage value.

//////////////////////////////////////******Enums***********///**

//It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

//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};

//*****C – 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 in 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. To 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 in C************//**

//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)
```

//******Function pointers*****//**

```
//void fun(int a)
// fun_ptr is a pointer to function fun()
//    void (*fun_ptr)(int) = &fun;
//// Invoking fun() using fun_ptr
//   (*fun_ptr)(10);
// function pointer points to code
```

//*******Bitwise operators*********//**

//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

//**********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.

*************************bits_ops****************************

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

#if 0

// Function to set the kth bit of n
int setBit(int n, int k)
{
   return (n | (1 << (k - 1)));
}

// Function to clear the kth bit of n
```

```
int clearBit(int n, int k)
{
   return (n & (~(1 << (k - 1))));
}

// Driver code
int main()
{
   int n = 5, k = 1;

   printf("%d with %d-th bit Set: %d\n",
          n, k, setBit(n, k));
   printf("%d with %d-th bit Cleared: %d\n",
          n, k, clearBit(n, k));

   return 0;
}
#endif

void convertBin(int num)
{
            // Size of an integer is assumed to be 32 bits
   for (int i = 31; i >= 0; i--) {
       int k = 1 << i;
       if (k & num)
           std::cout << "1";
       else
           std::cout << "0";
   }
}

int main(int argc, char* argv[])
{

            convertBin(3);
            return 0;
}

******************************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;}
```

C_string_operations
*

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int findstrlen(const char *str)
{
        int len=0;
        while(*str)
        {
                len++;
                str++;
        }
        return len;
}

char* mystrcat(char* dest, const char*src)
{
        int i = 0, j =0;
        int len = findstrlen(dest);
        while(dest[i] != '\0')
        {
                printf("string before concating is %c\n", dest[i]);
                i++;

        }

        while(src[j] != '\0')
        {
                 dest[i] = src[j];
                 i++;
                 j++;

        }
        dest[i] = '\0';
        printf("concanated string is %s\n", dest);
        return dest;
}

// Function to implement strcmp function
int strcmp(const char *X, const char *Y)
{
        while(*X)
        {
                // if characters differ or end of second string is reached
                if (*X != *Y)
                        break;

                // move to next pair of characters
                X++;
                Y++;
        }

        // return the ASCII difference after converting char* to unsigned char*
        return *(const unsigned char*)X - *(const unsigned char*)Y;
}

char* mystrcpy(char* dest, const char* src)
{
	int i = 0;
	while(src[i] != '\0')
	{
		dest[i] = src[i];
		i++;
	}
	dest[i] = '\0';
	return dest;
}

char *reverseString(char *str)
{
	int j =0;
	char *ptr;
	char temp;
	int len = findstrlen(str);
	ptr = (char*)malloc(sizeof(char)*len);

	/*for(int i=len-1; i>=0 ; i--)
	{
		ptr[j++] = str[i];

	}*/
	for(int i=0; i< len/2 ; i++)
	{
		//ptr[j++] = str[i];
		temp = str[i];
		str[i] = str[len-1-i];
		str[len-1-i] = temp;

	}
	printf("reverse string is %s\n", str);
	return str;
}
int main(int argc, char* argv[])
{

	char* str1, *str2, *str3;
	char arr[] = "Ankit";
	char arr1[10];
	int match;
	str1 = reverseString(arr);
	printf("string value is =%s\n", str1);
	str2 = mystrcpy(arr1, "shukla");
	printf("copied string value is =%s\n", str2);
	str3 = mystrcat(arr, "shukla");
	printf("concanated string value is =%s\n", str3);
	return 0;
}
```