

# Unit-4

## Syllabus

**Dynamic Memory Allocation:** malloc, calloc, realloc, free function.

**File Management:** Defining and Opening a File, Closing a File, Input/ Output Operations in Files, Random Access to Files, Error Handling.

Introduction to Graphics Programming

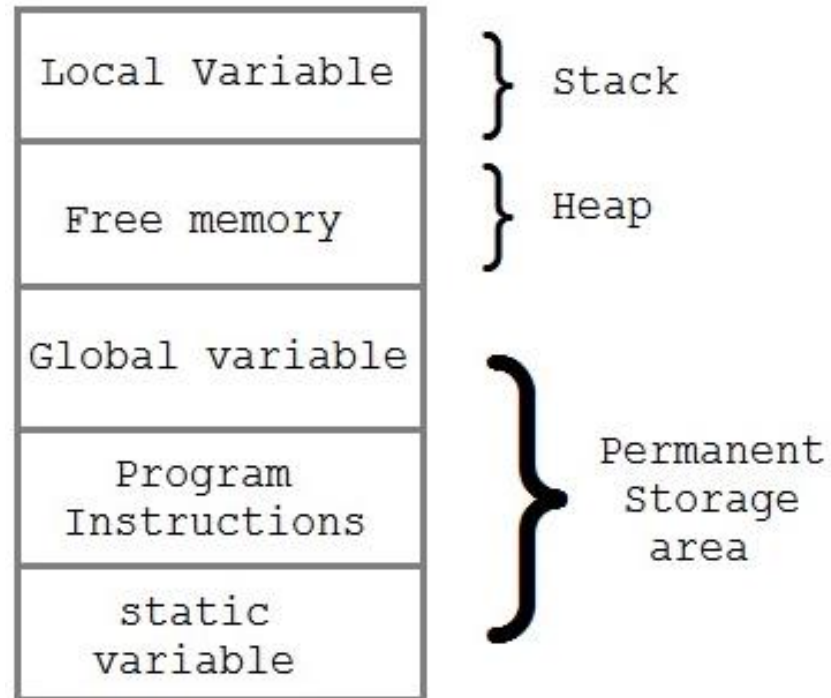
The Pre-processor Directives, Macros,

Command Line Arguments,.

S. K. Saroj Sir

# Memory Allocation Process

- Global variables, static variables and program instructions get their memory in permanent storage area whereas local variables are stored in a memory area called Stack.
- The memory space between these two regions is known as Heap area
- The size of heap keep changing
- Heap area is used for dynamic memory allocation during execution of the program



# Dynamic memory allocation

Since C is a structured language, it has some fixed rules for programming such as:

1. clrscr( ) is always written after variables declaration.
2. An array is collection of items stored at continuous memory locations. Size of an array is fixed at compile time; we can not modify size of array during run time etc.

Suppose array size is 9.

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So, there is a requirement to lessen the length (size) of the array from 9 to 5
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So, the length (size) of the array needs to be changed from 9 to 12
- Dynamic memory allocation is a procedure in which the size of a data structure (like Array, link list) is changed during the runtime
- C provides 4 library functions to achieve these tasks. These functions are defined in <alloc.h> header file

# Dynamic memory allocation

These functions are:

- `malloc()`
- `calloc()`
- `realloc()`
- `free()`

# malloc( )

- “**malloc**” stands for “**memory allocation**”. This method is used to dynamically allocate a single large block of memory with the specified size
- It returns a pointer of type void which pointing to the first byte of the allocated space and can be cast into a pointer of any form
- It initializes each block with default garbage value

Syntax:

```
void *malloc(byte-size);
```

```
pointer_variable = (cast-type *)malloc(byte-size);
```

Example:

```
void *malloc(5*sizeof(int));
```

```
int *x;
```

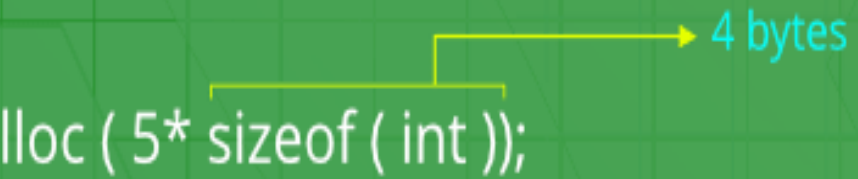
```
x = (int *)malloc(5*sizeof(int));
```

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

# malloc( )

## Malloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```



ptr =



← 20 bytes of memory →

→ A large 20 bytes memory block is dynamically allocated to ptr



# malloc( )

Program:

```
#include <stdlib.h>
void main()
{
    int *ptr;
    int n=5, i;
    printf("Enter number of elements: %d", n);
    ptr = (int*)malloc(n*sizeof(int));

    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using malloc.\n");
        for (i = 0; i < n; ++i)
            ptr[i] = i + 10;
    }

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

# calloc( )

- “**calloc**” stands for “**contiguous allocation**”. This method is used to dynamically allocate the **specified number of blocks** of memory of the **specified type**
- It initializes each block with a default value ‘0’

Syntax:

```
void *calloc(number of elements, element size);
```

```
pointer_variable = (cast-type *)calloc(number of elements, element size);
```

Example:

```
void *calloc(5, sizeof(int));
```

```
int *x;
```

```
x = (int *)calloc(5, sizeof(int));
```

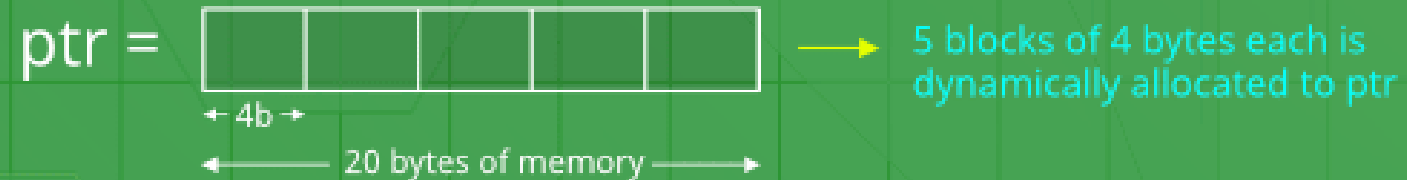
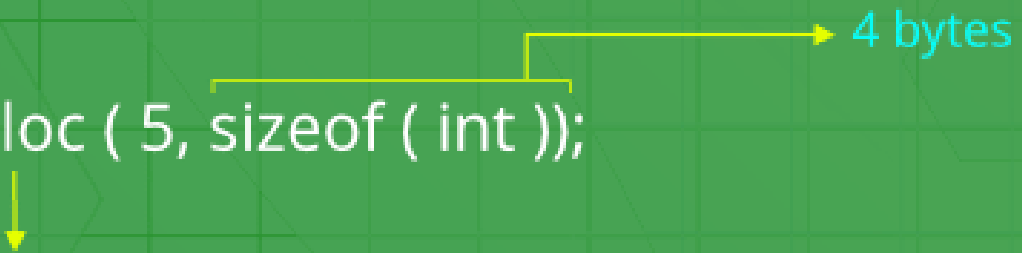
- Since the size of int is 4 bytes, this statement will allocate 20 bytes of memory and, the pointer ptr holds the address of the first byte in the allocated memory



# calloc( )

## Calloc()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ) );
```



# calloc( )

Program:

```
#include <stdlib.h>
void main()
{
    int *ptr;
    int n=5, i;
    printf("Enter number of elements: %d", n);
    ptr = (int*)calloc(n, sizeof(int));

    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using calloc.\n");
        for (i = 0; i < n; ++i)
            ptr[i] = i + 10;
    }

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

## **realloc( )**

- “**realloc**” stands for “**re-allocation**”. This method is used to dynamically change the memory allocation of a previously allocated memory
- In other words, if the memory previously allocated with the help of malloc( ) or calloc( ) is insufficient, realloc( ) can be used to dynamically re-allocate memory
- Re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value
- If space is insufficient, allocation fails and returns a NULL pointer

Syntax:

```
void *realloc(pointer_variable, newSize);  
pointer_variable = realloc(pointer_variable, newSize);
```

Example:

```
ptr = realloc(ptr, 10*sizeof(int));
```

# realloc( )

## Realloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

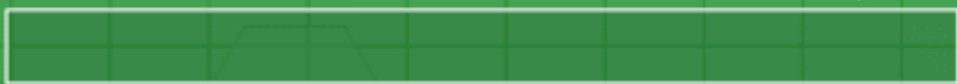
4 bytes

ptr = 

← 20 bytes of memory →

A large 20 bytes memory block is dynamically allocated to ptr

```
ptr = realloc ( ptr, 10* sizeof( int ));
```

ptr = 

← 40 bytes of memory →

The size of ptr is changed from 20 bytes to 40 bytes dynamically

# realloc( )

Program:

```
#include <stdlib.h>

void main()
{
    int *ptr;
    int n=5, i;
    printf("Enter number of elements: %d", n);
    ptr = (int*)calloc(n, sizeof(int));
    if (ptr == NULL)
    {
        printf("Memory not allocated");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using calloc");
        for (i = 0; i < n; ++i)
            ptr[i] = i + 10;
    }
    printf("The elements of the array are:");
    for (i = 0; i < n; ++i)
        printf("%d", ptr[i]);
}
```

# realloc( )

Program:

```
n = 10;
printf("Enter the new size of the array: %d", n);
ptr = realloc(ptr, n*sizeof(int));
for (i = 5; i < n; ++i)
    ptr[i] = i + 10;
printf("The elements of the array are:");
for (i = 0; i < n; ++i)
    printf("%d", ptr[i]);
}
```

## **free( )**

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

Syntax:

```
void *free(pointer_variable);
```

Example:

```
void *free(ptr);
```

# free( )

## Free()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

4 bytes



operation on ptr

free( ptr )



The memory of ptr is released





# free( )

```
#include <stdlib.h>

void main()
{
    int *ptr, *ptr1;
    int n=5, i;
    printf("Enter number of elements: %d", n);
    ptr = (int*)malloc(n*sizeof(int));
    ptr1 = (int*)calloc(n, sizeof(int));
    if (ptr == NULL || ptr1 == NULL)
    {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else
    {
        printf("Memory successfully allocated using malloc");
        free(ptr);
        printf("Malloc Memory successfully freed");
        printf("Memory successfully allocated using calloc");
        free(ptr1);
        printf("Calloc Memory successfully freed");
    }
}
```

# structure and dynamic memory allocation

```
#include <stdio.h>
#include <alloc.h>
void main()
{
    struct cricket
    {
        char P_name[20];
        int P_run;
        float P_strikerate;
    };
    struct cricket *i;
    i=(struct cricket *)malloc(sizeof(struct cricket));
    if(i==NULL)
    {
        printf("memory is not allocated");
        exit(1);
    }
    strcpy(i->P_name, "Pant");
    i->P_run=112;
    i->P_strikerate=190.50;
    printf("%s%d%f", i->P_name, i->P_run, i->P_strikerate);
}
```

# structure and dynamic memory allocation

```
#include <alloc.h>
void main()
{
    struct cricket
    {
        char P_name[20];
        int P_run;
        float P_strikerate;
    };
    struct cricket *c;
    int i;
    c=(struct cricket *)malloc(11*sizeof(struct cricket));
    if(c==NULL)
    {
        printf("memory is not allocated");
        exit(1);
    }
    for(i=0; i<11; i++)
    {
        scanf("%s%d%f", c->P_name, &c->P_run, &c->P_strikerate);
        c++;
    }
    for(i=0; i<11; i++)
    {
        printf("%s%d%f", c->P_name, c->P_run, c->P_strikerate);
        c++;
    }
}
```

---

# function

int \*p(): Here p is a function that has no arguments and returns an integer pointer

```
#include<stdio.h>

int *p()
{
    int a = 6, b = 3;
    int c = a + b;
    int * t = &c;
    return t;
}

int main()
{
    int *a;
    a=p();
    Printf("%d", *a);
}
```

# function

- `int (*p)()`: Here `p` is a pointer to function or function pointer which can store the address of a function which has no argument and returning an integer

```
#include <stdio.h>

int abc()
{
    int a = 5, b = 9;
    return a + b;
}

int main()
{
    int (*p)();    //p is a pointer to function or function pointer
    p = abc;
    printf("%d", *a);
}
```

Ans:- 14

# array

- `int *p[10]`: Here `p` is an array that stores addresses of 10 elements who are integer types

```
#include <stdio.h>
void main()
{
    int a = 1, b = 2, c = 3;
    int *p[3];          --> // p is an array of the size 3 which can store integer pointers or
    p[0] = &a;           // p is an array that stores addresses of all elements who are integer types.
    p[1] = &b;
    p[2] = &c;
    for (int i = 0; i < 3; i++)
        printf("%d", *p[i]);
}
```

Ans:- 1 2 3

# array

- `int (*p)[10]`: Here `p` is a pointer to array or array pointer which can store the address of an array which has dimension 10 and data type `int`

```
#include <stdio.h>
void main()
{
    int a[3] = { 1, 2, 3 };
    int (*p)[3];          --> // p is the pointer which can point to an array of three integers or
    p = &a;                // p is a pointer to array or array pointer which can store the address of
    for(int i = 0; i < 3; i++) // an array which has dimension 3 and data type int
    {
        printf("%d", *p);
        p++;
    }
}
```

Ans:- 1 2 3

# Test yourself

- `int *p();`
- `int *p(int);`
- `int *p(int, int);`
- `int *p(int, char);`
- `int *p(float, char, int);`
- `float *p(int, char);`
- `int (*p)();`
- `int (*p)(int);`
- `int (*p)(int, int);`
- `int (*p)(int, char);`
- `int (*p)(float, char, int);`
- `float (*p)(int, char);`
- `int *a[3];`
- `char *a[10];`
- `float *a[15];`
- `double *a[];`
- `int (*a)[3];`
- `char (*a)[10];`
- `float (*a)[15];`



# Test yourself

- `int *p;`
- `int **p;`
- `int ***p;`
- `float *p;`
- `char *p;`
- `float **p;`
- `double *a;`
- `long *a;`
- `short *a;`
- `void *a;`
- `long double *a;`
- `long long *i;`
- `long long long *i;`
- `struct book *b;`
- `struct a *b;`
- `union book *i;`
- `union i *j;`

All are correct.

# Test yourself

Apply following in programs and know the differences.

- static variable vs constant variable
- #define vs global variable

# Assignment

Define and differentiate **clearly** following functions.

## Input functions

scanf();

gets();

getc();

getchar();

getch();

getche();

## Output functions

printf();

puts();

putc();

putchar();

putch();

# File handling

**File:** It is a named collection of data.

## **Need of file handling:**

- So far, the operations using C program are done on a prompt / terminal which is not stored anywhere
- You can easily move your data from one computer to another without any changes

## **Types of Files**

When dealing with C, there are two types of files you should know about:

- Text files
- Binary files

# File Operations

- Creating a new file
- Opening an existing file
- Reading from a file
- Writing to a file
- Moving to a specific location in a file
- Closing the file

# File Opening Modes

**“r”**: It is a read only mode, which means if the file is opened in r mode, it won't allow you to write and modify content of it. When fopen() opens a file successfully then it returns the address of first character of the file, otherwise it returns NULL.

**“w”**: It is a write only mode. The fopen() function creates a new file when the specified file doesn't exist and if it fails to open file then it returns NULL.

**“a”**: Using this mode Content can be appended at the end of an existing file. Like Mode “w”, fopen() creates a new file if it file doesn't exist. On unsuccessful open it returns NULL.  
File Pointer points to: last character of the file.

**“r+”**: This mode is same as mode “r”; however you can perform various operations on the file opened in this mode. You are allowed to read, write and modify the content of file opened in “r+” mode.  
File Pointer points to: First character of the file.

**“w+”**: Same as mode “w” apart from operations, which can be performed; the file can be read, write and modified in this mode.

# File Opening Modes

“**a+**”: Same as mode “a”; you can read and append the data in the file, however content modification is not allowed in this mode.

As given above, if you want to perform operations on a **binary file**, then you have to **append ‘b’** at the last.

Example

instead of “w”, you have to use “wb”, instead of “a+” you have to use “a+b”.

Modes for binary file

rb

wb

ab

r+b

w+b

a+b

# File handling functions

File handling functions	Description
<a href="#">fopen ()</a>	fopen () function creates a new file or opens an existing file.
<a href="#">fclose ()</a>	fclose () function closes an opened file.
<a href="#">getw ()</a>	getw () function reads an integer from file.
<a href="#">putw ()</a>	putw () functions writes an integer to file.
<a href="#">fgetc ()</a>	fgetc () function reads a character from file.
<a href="#">fputc ()</a>	fputc () functions write a character to file.
<a href="#">gets ()</a>	gets () function reads line from keyboard.
<a href="#">puts ()</a>	puts () function writes line to o/p screen.
<a href="#">fgets ()</a>	fgets () function reads string from a file, one line at a time.
<a href="#">fputs ()</a>	fputs () function writes string to a file.
<a href="#">feof ()</a>	feof () function finds end of file.
<a href="#">fgetchar ()</a>	fgetchar () function reads a character from keyboard.
<a href="#">fprintf ()</a>	fprintf () function writes formatted data to a file.
<a href="#">fscanf ()</a>	fscanf () function reads formatted data from a file.
<a href="#">fputchar ()</a>	fputchar () function writes a character onto the output screen from keyboard input.
<a href="#">fseek ()</a>	fseek () function moves file pointer position to given location.



# File handling functions

SEEK_SET	SEEK_SET moves file pointer position to the beginning of the file.
SEEK_CUR	SEEK_CUR moves file pointer position to given location.
SEEK_END	SEEK_END moves file pointer position to the end of file.
ftell ()	ftell () function gives current position of file pointer.
rewind ()	rewind () function moves file pointer position to the beginning of the file.
getc ()	getc () function reads character from file.
getch ()	getch () function reads character from keyboard.
getche ()	getche () function reads character from keyboard and echoes to o/p screen.
getchar ()	getchar () function reads character from keyboard.
putc ()	putc () function writes a character to file.
putchar ()	putchar () function writes a character to screen.
printf ()	printf () function writes formatted data to screen.
sprintf ()	sprintf () function writes formatted output to string.
scanf ()	scanf () function reads formatted data from keyboard.
sscanf ()	sscanf () function Reads formatted input from a string.
remove ()	remove () function deletes a file.
fflush ()	fflush () function flushes a file.

# File handling

## Opening or Creating file

For opening a file, `fopen( )` function is used with the required access mode

For performing operations on the file, a special pointer called 'file pointer' is used

The code snippet for opening or creating a file is as:

```
FILE * fp;  
fp = fopen("C:\\user\\pro1.txt", "r");
```

# File handling

## Reading from a file

File read operations can be performed using functions `fscanf( )` or `fgets ( )` or `fgetc ( )` with an additional parameter, the file pointer

So, it depends on you if you want to read the file line by line or character by character.

The code snippet for reading a file is as:

```
FILE * fp;  
fp = fopen("C:\\user\\pro2.txt", "r");  
fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
```

# File handling

## Writing to a file

File write operations can be performed by the functions `fprintf( )` or `fputs( )` or `fputc( )` etc.

The snippet for writing to a file is as :

```
FILE *fp;  
fp= fopen("C:\\user\\pro3.txt", "w");  
fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
```

# File handling

## Closing a file

After every successful file operations, you must always close a file. For closing a file, you have to use `fclose( )` function.

The snippet for closing a file is given as :

```
FILE *fp;  
fp=fopen("C:\\user\\pro4.txt", "w");  
- - - -  
- - - -  
- - - -  
fclose(fp);
```

# Programs

A Simple C Program to open, read and close the file

---

```
#include <stdio.h>
void main()
{
    FILE *fp1;
    char c;
    fp1=fopen("C:\\user\\pro2.txt", "r");
    while(1)
    {
        c = fgetc(fp1);

        if(c==EOF)
            break;

        else
            printf("%c", c);
    }

    fclose(fp1);
}
```

# Programs

## C Program to write the file

```
#include <stdio.h>
int main()
{
    char  ch;
    FILE  *f;
    f = fopen("C:\\user\\prim.txt",  "w");

    if(f == NULL)
    {
        printf("Error");
        exit(1);
    }

    printf("Enter any character");
    scanf("%c",  &ch);
    fprintf(f,  "%c",  ch);    -->//  You can also use fputc(ch, f);
    fclose(f);
    return 0;
}
```

# Programs

Example to read the strings from a file in C programming

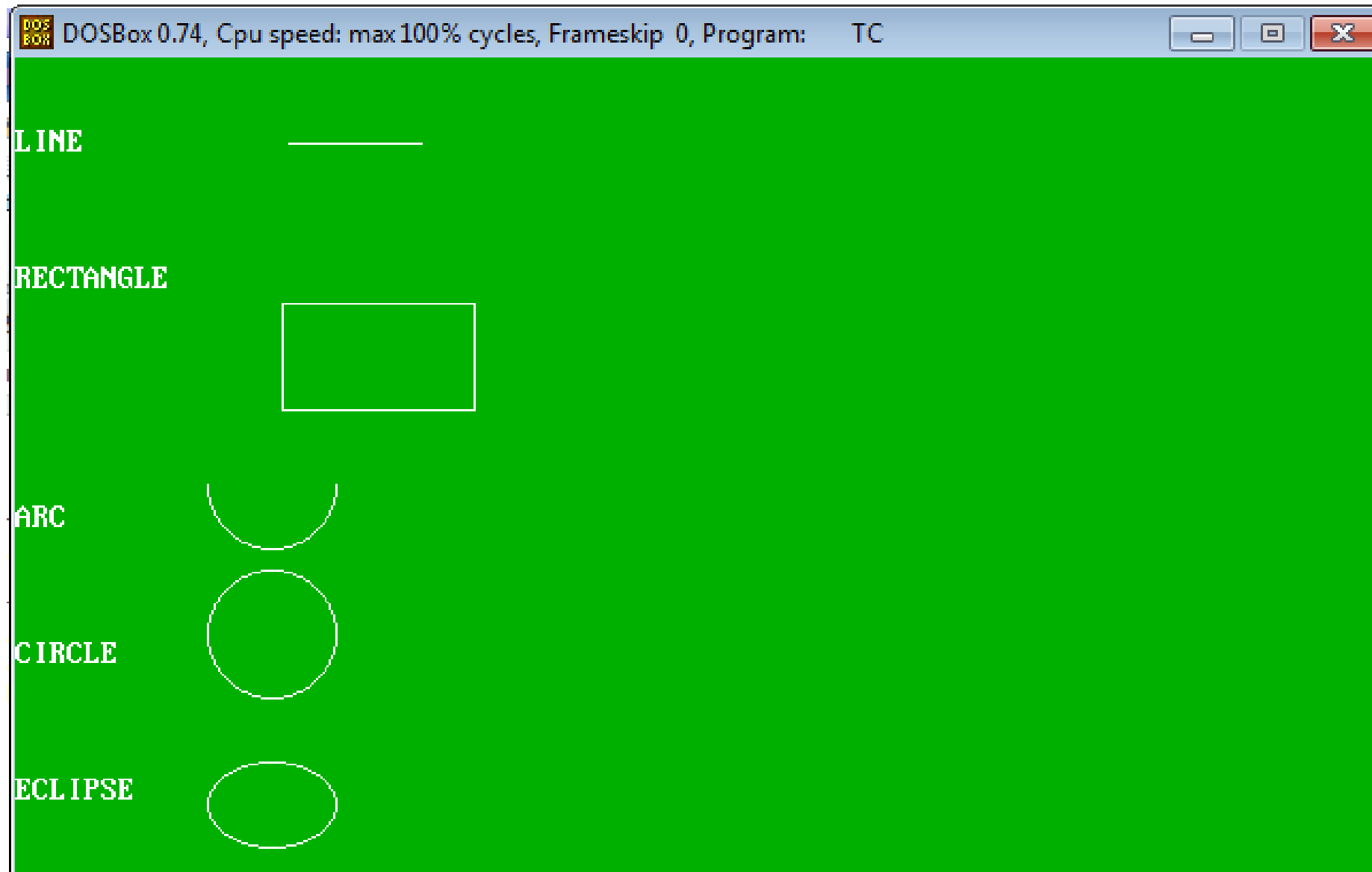
```
#include <stdio.h>
int main()
{
    FILE    *a;
    char    str[100];
    a = fopen("C:\\user\\sumit.txt",    "r");
    if (a == NULL)
    {
        puts("Issue in opening the input file");
    }
    while(1)
    {
        if(fgets(str,    10,    fpr) ==NULL)
            break;
        else
            printf("%s",    str);
    }
    fclose(a);
    return 0;
}
```



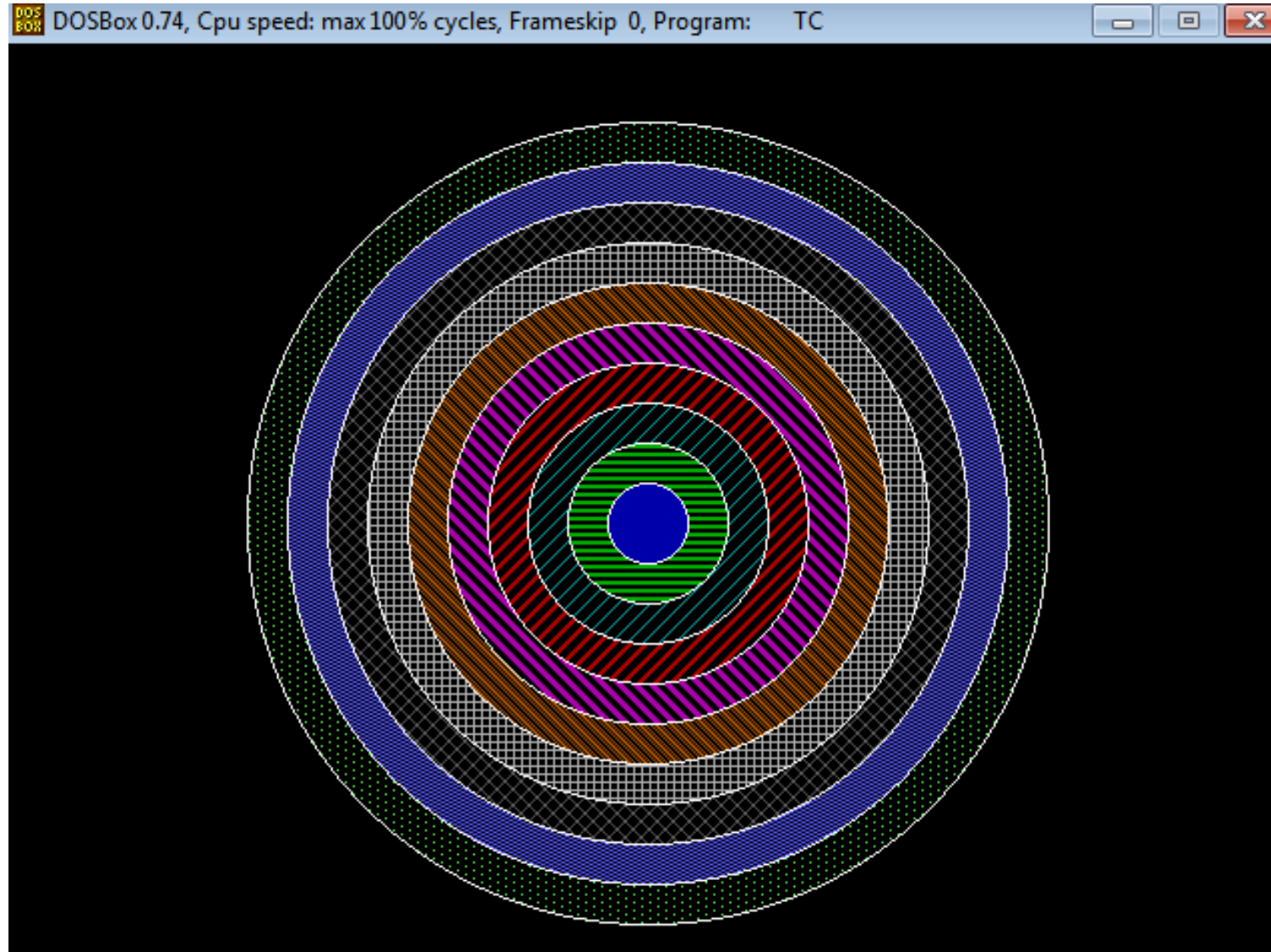
# Introduction to Graphics Programming

In Turbo C graphics, we use graphics.h functions to draw different shapes(like circle, rectangle etc), display text(any message) in different format(different fonts and colors). By using graphics.h we can make programs, animations and also games. These can be

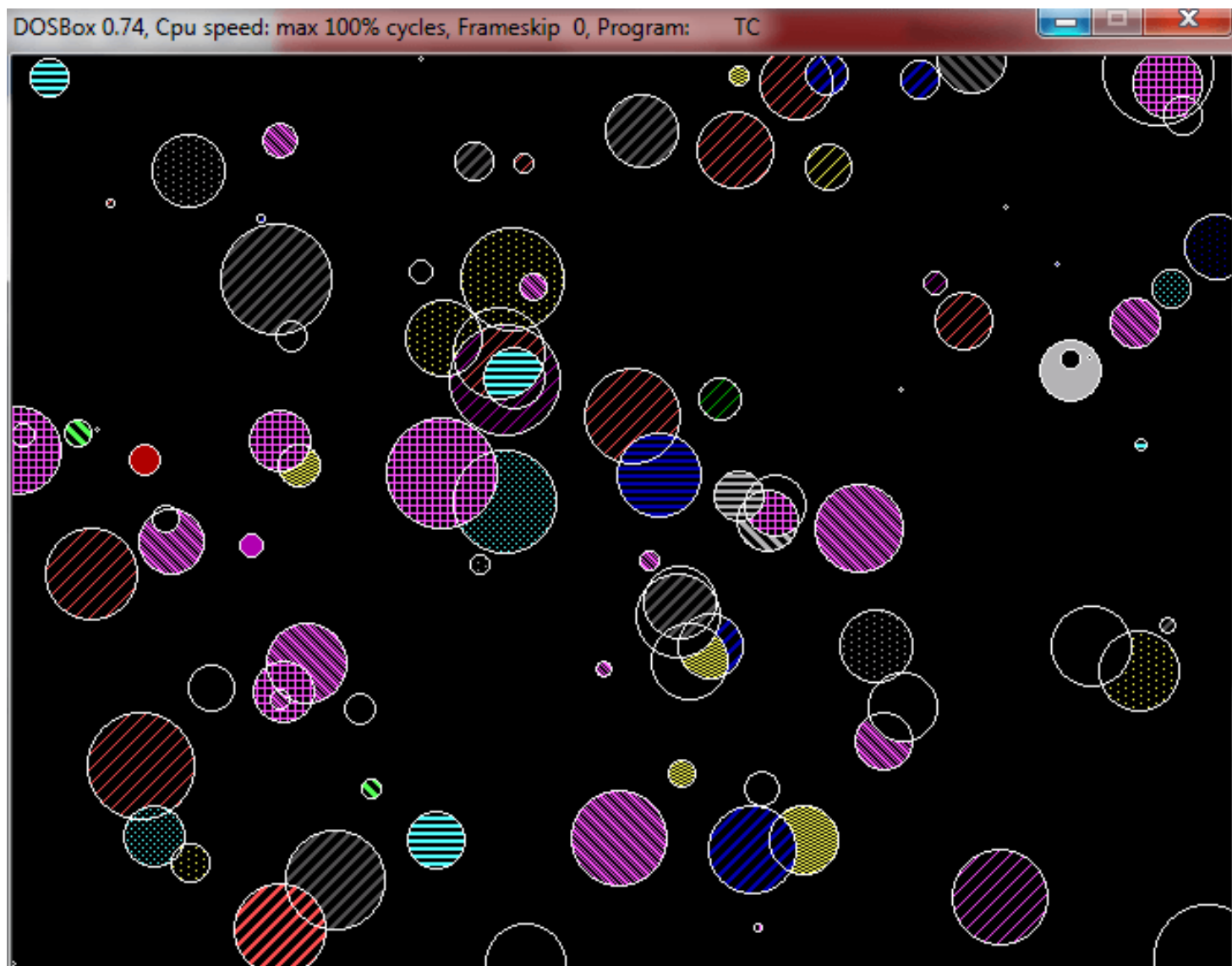
# Introduction to Graphics Programming



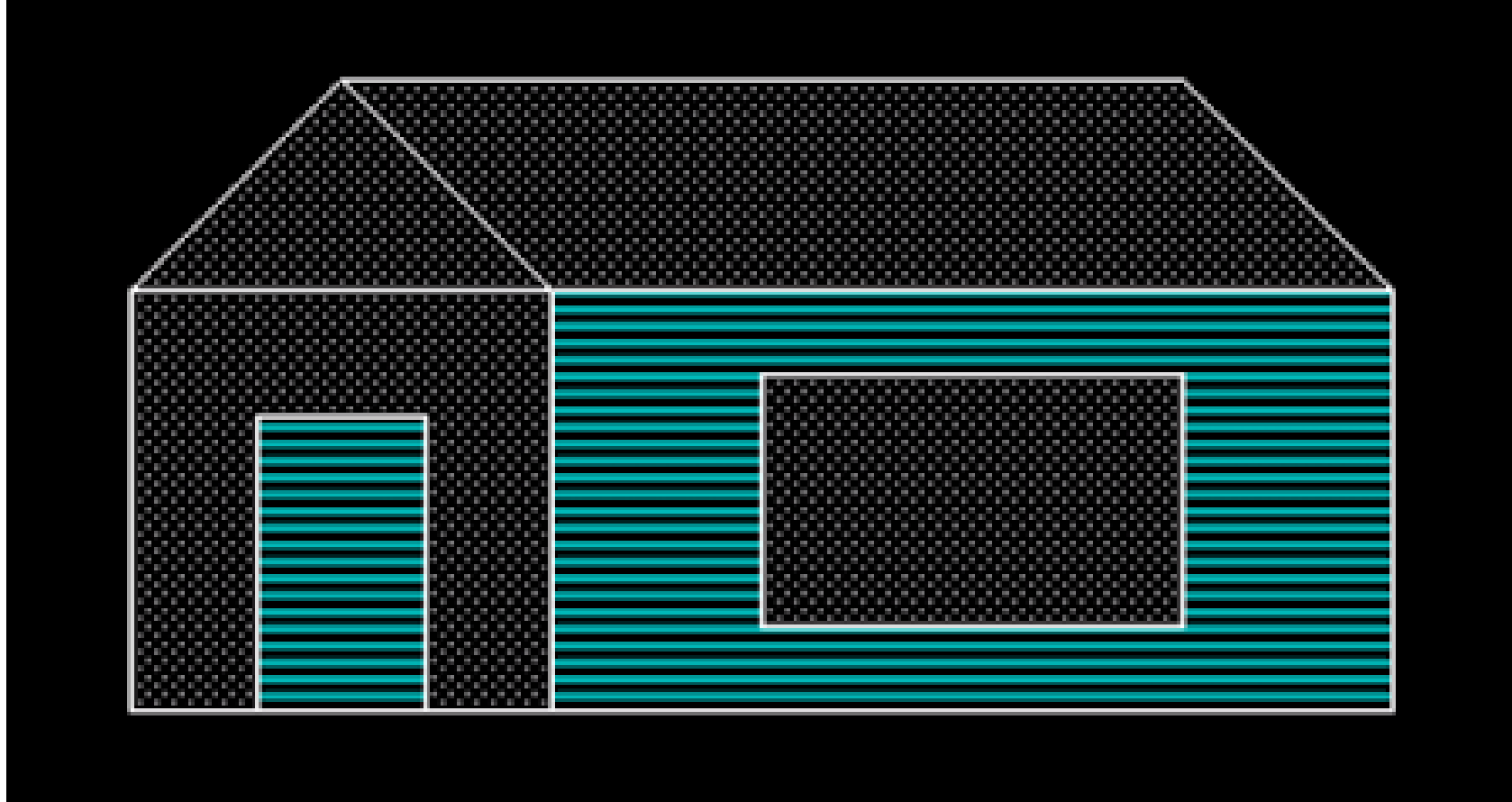
# Introduction to Graphics Programming



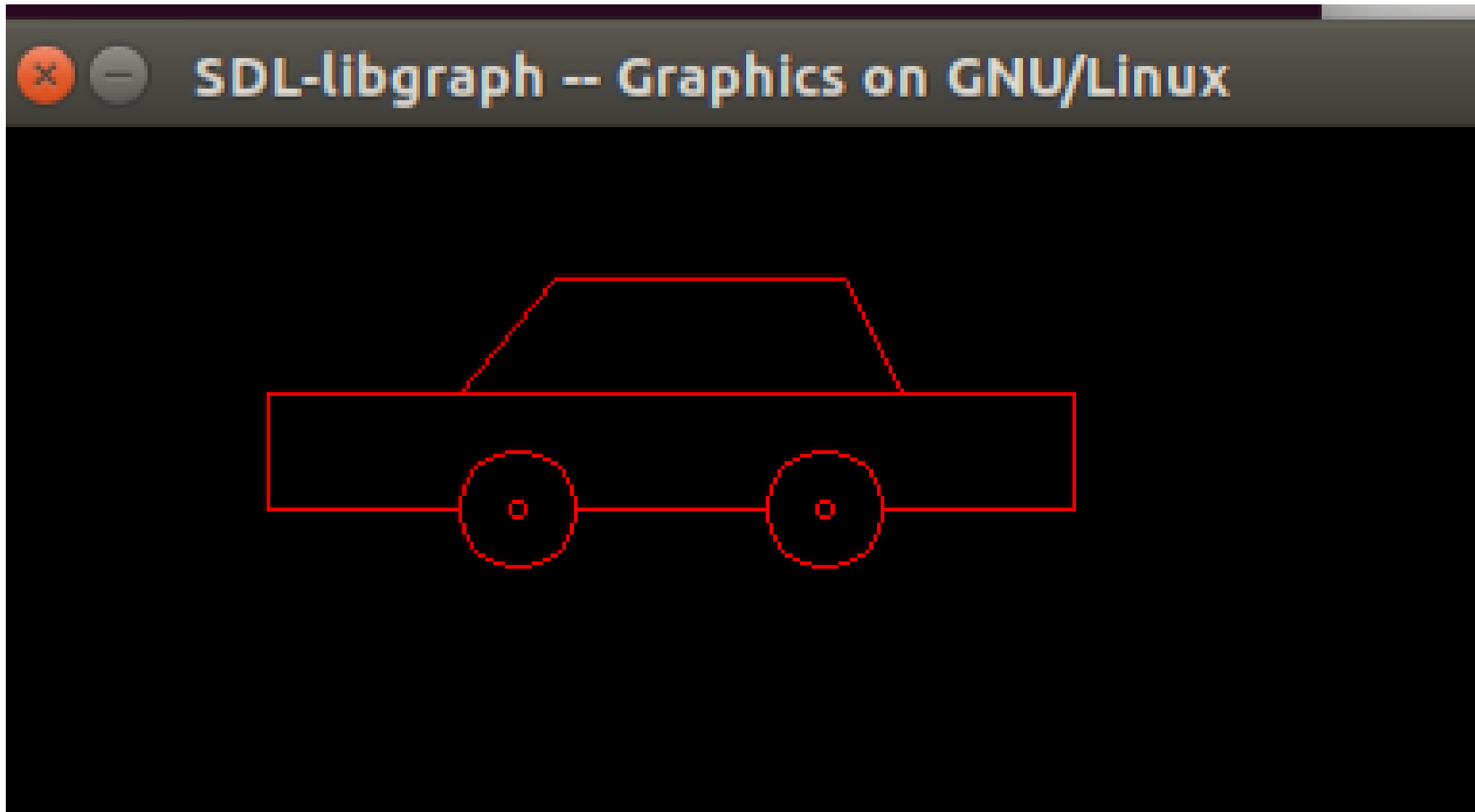
# Introduction to Graphics Programming



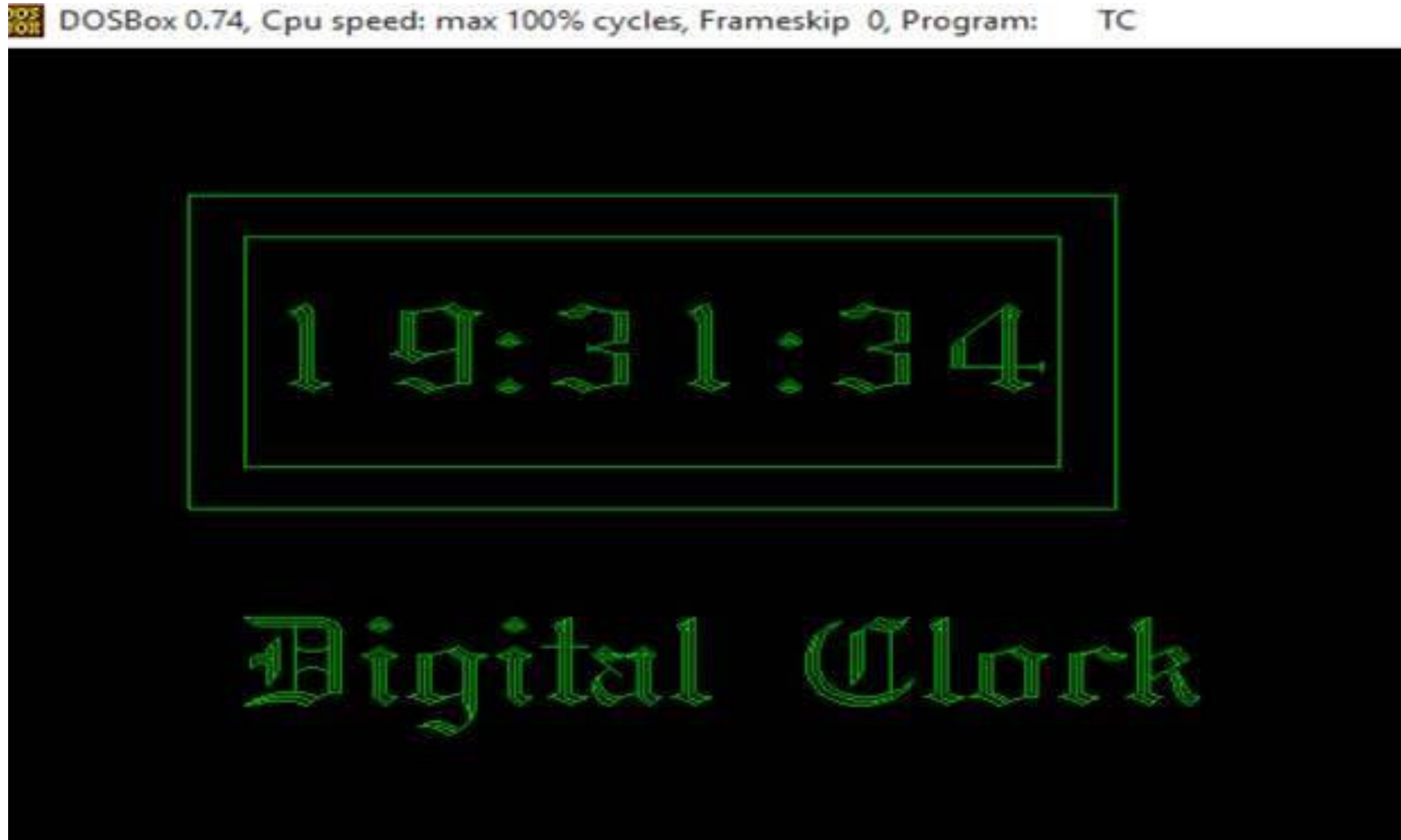
# Introduction to Graphics Programming



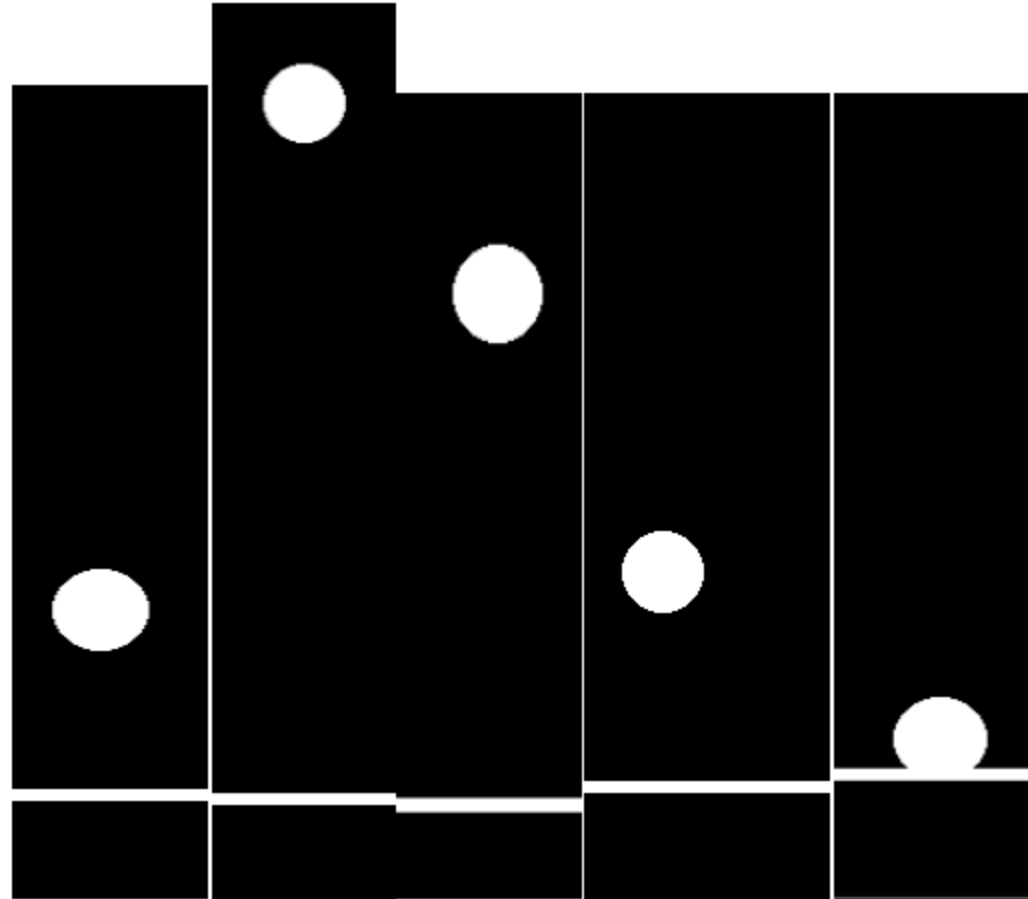
# Introduction to Graphics Programming



# Introduction to Graphics Programming



# Introduction to Graphics Programming





# Introduction to Graphics Programming

Various functions for these shapes, colors and animations

- `initgraph()`
- `closegraph()`
- `line()`
- `circle()`
- `rectangle()`
- `setcolor()`
- `getcolor()`
- `setbkcolor()`
- `getbkcolor()`
- `textheight()`
- `textwidth()`
- `delay()`
- `arc()`

etc.

# Introduction to Graphics Programming

Function	Description
initgraph	It initializes the graphics system by loading the passed graphics driver then changing the system into graphics mode.
getmaxx	It returns the maximum X coordinate in current graphics mode and driver.
getmaxy	It returns the maximum X coordinate in current graphics mode and driver.
setcolor	It changes the current drawing colour. Default colour is white. Each color is assigned a number, like BLACK is 0 and RED is 4. Here we are using colour constants defined inside graphics.h header file.
setfillstyle	It sets the current fill pattern and fill color.
circle	It draws a circle with radius r and centre at (x, y).
line	It draws a straight line between two points on screen.
arc	It draws a circular arc from start angle till end angle.
floodfill	It is used to fill a closed area with current fill pattern and fill color. It takes any point inside closed area and color of the boundary as input.
cleardevice	It clears the screen, and sets current position to (0, 0).
delay	It is used to suspend execution of a program for a M milliseconds.
closegraph	It unloads the graphics drivers and sets the screen back to text mode.

# Introduction to Graphics Programming

Sample

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

int main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\\\TC\\\\BGI");

    getch();
    closegraph();
    return 0;
}
```

# Introduction to Graphics Programming

The first step in any graphics program is to include graphics.h header file.

**graphics.h** header file provides access to a simple graphics library that makes it possible to draw lines, rectangles, ovals, arcs, polygons, images, and strings on a graphical window.

The second step is initializing the graphics drivers on the computer using `initgraph( )` function which is defined in `graphics.h` library.

**initgraph( )** initializes the graphics system by loading the passed graphics driver then changing the system into graphics mode. It also resets or initializes all graphics settings like color, palette, current position etc, to their default values. Below is the description of passed arguments of `initgraph( )` function.

**graphicsDriver** : It is the integer that specifies which graphics driver is to be used. It tells the compiler that what graphics driver to use or to automatically detect the drive. In all our programs we will use `DETECT` macro of `graphics.h` library that instruct compiler for auto detection of graphics driver. We can also give it a value using a constant of the `graphics_drivers` enum type, which is defined in `graphics.h` and listed below.

# Introduction to Graphics Programming

<code>graphics_drivers</code> constant	Numeric value
<code>DETECT</code>	0 (requests autodetect)
<code>CGA</code>	1
<code>MCGA</code>	2
<code>EGA</code>	3
<code>EGA64</code>	4
<code>EGAMONO</code>	5
<code>IBM8514</code>	6
<code>HERCMONO</code>	7
<code>ATT400</code>	8
<code>VGA</code>	9
<code>PC3270</code>	10

# Introduction to Graphics Programming

**graphicsMode** : It is an integer that specifies the graphics mode to be used. If gdriver is set to DETECT, then initgraph( ) sets gmode to the highest resolution available for the detected driver.

Graphics		
Driver	graphics_mode	Value
CGA	CGAC0	0
	CGAC1	1
	CGAC2	2
	CGAC3	3
	CGAHI	4
MCGA	MCGAC0	0
	MCGAC1	1
	MCGAC2	2
	MCGAC3	3
	MCGAMED	4
	MCGAHI	5
EGA	EGALO	0
	EGAHI	1
EGA64	EGA64LO	0

# Introduction to Graphics Programming

**driverDirectoryPath** : It specifies the directory path where graphics driver files (BGI files) are located. If directory path is not provided, then it will search for driver files in current working directory. In all our sample graphics programs, you have to change path of BGI directory accordingly where you Turbo C compiler is installed.

**Closegraph:** It unloads the graphics drivers and sets the screen back to text mode.

**BGI:** (Borland Graphics Interface) is a graphics library. The library loads graphic drivers (\*.BGI) and vector fonts (\*.CHR) from disk so to provide device independent graphics support to the programmers.

## Colors in C Graphics Programming:

There are 16 colors declared in C Graphics. We use colors to set the current drawing color, change the color of background, change the color of text, to color a closed shape etc. To specify a color, we can either use color constants like setcolor(RED), or their corresponding integer codes like setcolor(4). Below is the color code in increasing order.

# Introduction to Graphics Programming

COLOR MACRO	INTEGER VALUE
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15



# Introduction to Graphics Programming

Sample

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

int main()
{
    int gd = DETECT, gm;

    initgraph(&gd, &gm, "C:\\\\TC\\\\BGI");

    getch();
    closegraph();
    return 0;
}
```

# Introduction to Graphics Programming

Program to Draw line

```
#include<graphics.h>
#include<conio.h>
int main()
{
    int gd = DETECT, gm;
    initgraph(&gd, &gm, "C:\\TC\\BGI");
    line(100,100,200, 200);
    getch( );
    closegraph( );
    return 0;
}
```

# Introduction to Graphics Programming

Program to Draw line

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

void main(void) {
    int gdriver = DETECT, gmode;
    int x1 = 200, y1 = 200;
    int x2 = 300, y2 = 300;
    clrscr();

    initgraph(&gdriver, &gmode, "c:\\turbo3\\bgi");
    line(x1, y1, x2, y2);
    getch();
    closegraph();
}
```

# Introduction to Graphics Programming

Program to Draw rectangle of color blue.

```
//Include the graphics header file
#include<graphics.h>
#include<stdio.h>
#include<conio.h>

void main()
{
    //Initialize the variables for the graphics driver and mode
    int gd = DETECT, gm;
    clrscr();
    initgraph(&gd, &gm, "C:\\\\TURBOC3\\\\BGI");

    //Set the color of the object you want to draw.
    setcolor(BLUE);

    //Draw an object. For this example, drawing a rectangle using the rectangle function
    rectangle(50,50,100,100);

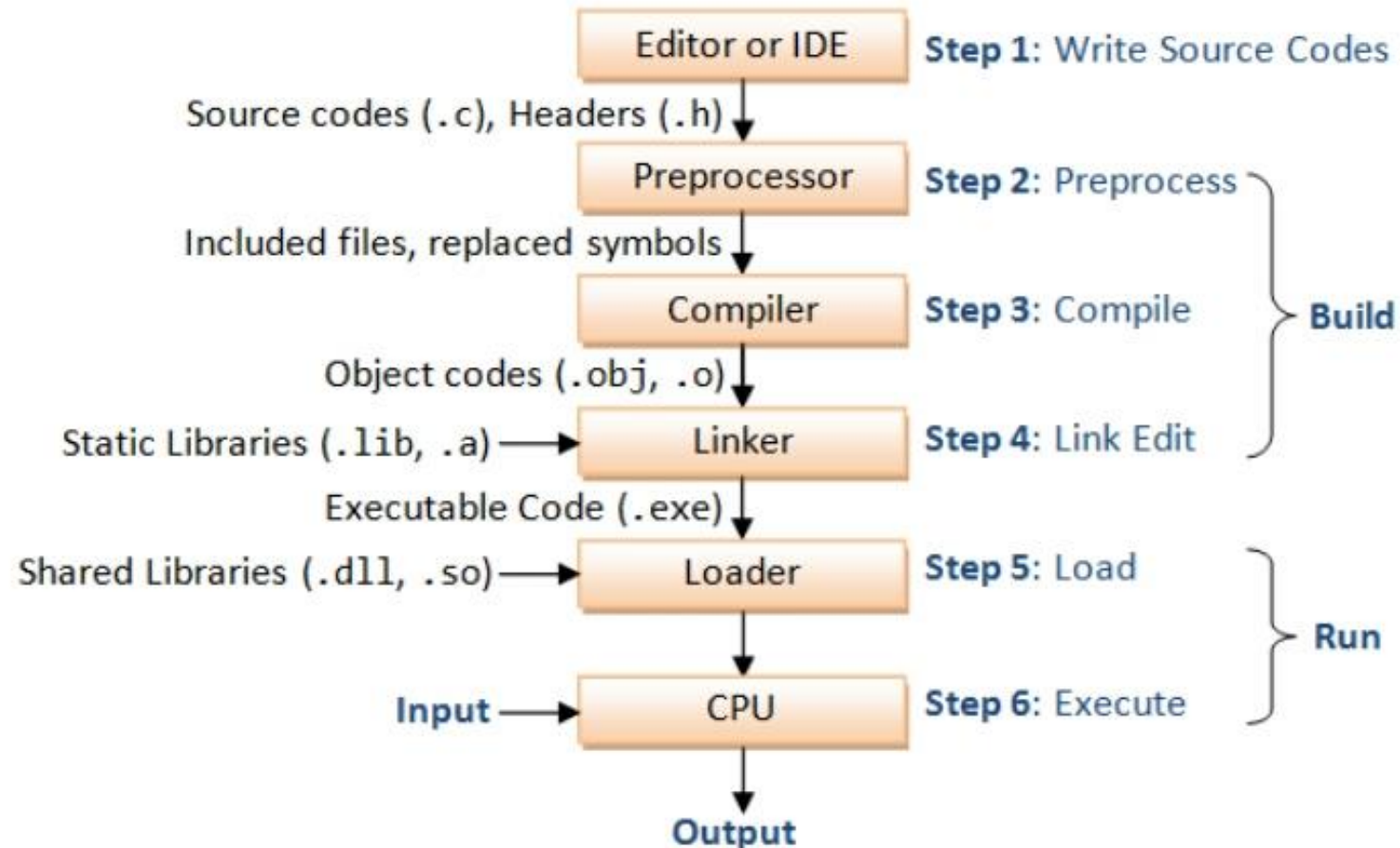
    getch();

    //unloads the graphics drivers
    closegraph();
}
```

# Preprocessor

**Preprocessor** is a system software. It is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

## Program execution process



# Preprocessor

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character.

## **A Preprocessor mainly performs three tasks**

- 1. Removing comments:** It removes all the comments. A comment is written only for the humans to understand the code.
- 2. File inclusion:** Including all the files from library that our program needs. In C, we write `#include` which is a preprocessor directive that tells preprocessor to include the contents of the library file specified.  
For example, `#include` will tell the preprocessor to include all the contents in the library file `stdio.h`
- 3. Macro expansion:** It uses the MACRO.

# Preprocessor

**Preprocessors Directives:** Preprocessor programs provide preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that, whatever statement starts with #, is going to the preprocessor program, and preprocessor program will execute this statement.

## Examples of Preprocessors Directives:

Directive	Function
<i>#define</i>	Defines a <b>Macro</b> Substitution
<i>#undef</i>	Undefines a <b>Macro</b>
<i>#include</i>	Includes a File in the Source Program
<i>#ifdef</i>	Tests for a <b>Macro</b> Definition
<i>#endif</i>	Specifies the end of #if
<i>#ifndef</i>	Checks whether a <b>Macro</b> is defined or not
<i>#if</i>	Checks a Compile Time Condition
<i>#else</i>	Specifies alternatives when #if Test Fails

# Preprocessor

**There are 4 main types of preprocessor directives:**

1. Macro
2. File Inclusion
3. Conditional Compilation
4. Other directives



# Preprocessor

**Macro:** Macro is a piece of code in a program which is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The ‘#define’ directive is used to define a macro.

```
#include <stdio.h>
#define LIMIT 5           // macro definition
int main()
{
    for (int i = 0; i < LIMIT; i++)
        printf("%d", i);
    return 0;
}
```

Output:0 1 2 3 4

In the above program, when the compiler executes the word LIMIT it replaces it with 5. The word ‘LIMIT’ in the macro definition is called a macro template and ‘5’ is macro expansion. It is object-like Macro (do not take parameters).

**Note:** There is no semi-colon(‘;’) at the end of macro definition.

# Preprocessor

**Macros with arguments (function-like Macro):** We can also pass arguments to macros. Macros defined with arguments works similarly as functions. Let us understand this with a program:

```
#include <stdio.h>
#define AREA(L, B) (L * B)           // macro with parameter
int main()
{
    int a = 10, b = 5, A;
    A = AREA(a, b);
    printf("Area:%d", A);
    return 0;
}
```

Output: Area:50

We can see from the above program that, whenever the compiler finds `AREA(L, B)` in the program it replaces it with the statement `(L * B)`.

Not only this, the values passed to the macro template `AREA(L, B)` will also be replaced in the statement `(L * B)`. Therefore `AREA(10, 5)` will be equal to `10*5`.

# Preprocessor

**File Inclusion:** This type of preprocessor directive tells the compiler to include a file in the source code program.

There are two types of files which can be included by the user in the program:

**1. Header File or Standard files:** These files contains definition of pre-defined functions like printf(), scanf(), clrscr(), getch() etc. These files must be included for working with these functions.

Syntax:

```
#include< file_name >
```

where file\_name is the name of file to be included. The ‘<’ and ‘>’ brackets tells the compiler to look for the file in standard directory.

**2. User defined files:** When a program becomes very large, it is good practice to divide it into smaller files and include whenever needed. These types of files are user defined files. These files can be included as:

```
#include"filename"
```

# Preprocessor

**Conditional Compilation:** Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.

This can be done with the help of two preprocessing commands ‘ifdef’ and ‘endif’.

```
#ifdef macro_name
    statement1;
    statement2;
    statement3;
    .
    .
    .
    statementN;
#endif
```

If the macro with name as ‘macroname’ is defined then the block of statements will execute normally but if it is not defined, the compiler will simply skip this block of statements.

# Preprocessor

**Other directives:** Apart from the above directives there are two more directives which are not commonly used. These are:

- 1. #undef Directive:** The #undef directive is used to undefine an existing macro. This directive works as:

```
#undef LIMIT
```

Using this statement will undefine the existing macro LIMIT. After this statement every “#ifdef LIMIT” statement will evaluate to false.

- 2. #pragma Directive:** This directive is a special purpose directive and is used to turn on or off some features. This type of directives vary from compiler to compiler.  
Some of the #pragma directives are: #pragma startup and #pragma exit, #pragma warn Directive.

# Preprocessor

We can write multi-line macro same like function, but each statement ends with “\”.

```
#include <stdio.h>

#define MACRO(num, str) {\
    printf("%d", num);\
    printf(" is");\
    printf(" %s number", str);\
    printf("\n");\
}
```

# Command Line Arguments

So far, we have seen that no arguments were passed in the main function. But the C programming language gives the programmer the provision to add parameters or arguments inside the main function to reduce the length of the code. These arguments are called Command Line Arguments in C.

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

To pass command line arguments, we typically define main( ) with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

# Command Line Arguments

- **argc (ARGument Count):** It 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):** It 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.



# Command Line Arguments

```

[■] \TURBOC3\BIN\CLA.C 8=
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[])
{
    int c;
    clrscr();

    printf("Program Name is: %s\n ", argv[0]);

    if(argc==1)
        printf("No extra argument passed only program name\n");

    if(argc>=2)
    {
        printf("Number of arguments passed: %d\n Arguments are:- \t",  argc);
        for(c=0; c<argc; c++)
            printf("argv[%d]: %s\t",  c,  argv[c]);
    }
    getch();
    return 0;
}

```

# Command Line Arguments

```
[■]  
#include<conio.h>  
void main()  
{  
    char i='A';  
    char j='5';  
    char k=65;  
    char l=5;  
    clrscr();  
    printf("%c %d\n", i, i);  
    printf("%c %d\n", j, j);  
    printf("%c %d\n", k, k);  
    printf("%c %d", l, l);  
    getch();  
}
```

```
A 65  
5 53  
A 65  
5 5_
```

# Command Line Arguments

**atoi( )** function in C takes a string (which represents an integer) as an argument and returns its value of type int. So, basically the function is used to convert a string argument to an integer.

atoi stands for ASCII to integer. It is included in the C standard library header file `stdlib.h`

Syntax:

```
int atoi(const char *strn);
```

Parameters: The function accepts one parameter `strn` which refers to the string argument that is needed to be converted into its integer equivalent.

Return Value: If `strn` is a valid input, then the function returns the equivalent integer number for the passed string number. If no valid conversion takes place, then the function returns zero.

# Command Line Arguments

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

int main()
{
    int val;
    char strn1[] = "12546";

    val = atoi(strn1);
    printf("String value = %s\n", strn1);
    printf("Integer value = %d\n", val);

    char strn2[] = "GeeksforGeeks";
    val = atoi(strn2);
    printf("String value = %s\n", strn2);
    printf("Integer value = %d\n", val);

    return (0);
}
```

String value = 12546

Integer value = 12546

String value = GeeksforGeeks

Integer value = 0

# Command Line Arguments

```

[ ] \TURBOC3\BIN\CLAFAC.T.C
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[])
{
    int n, fact=1;
    clrscr();
    n=atoi(argv[1]);
    while(n>0)
    {
        fact=fact*n;
        n--;
    }
    printf("%d", fact);
    getch();
    return 0;
}

```

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: ?6U?6R

```
120
C:\TURBOC3\SOURCE>clafact 5_
```

# Command Line Arguments

```
[■]===== CLAATOB.C =====
#include<stdio.h>
#include<conio.h>
int main(int argc, char *argv[])
{
    int m, n, pow=1;
    clrscr();
    m=atoi(argv[1]);
    n=atoi(argv[2]);
    while(n>0)
    {
        pow=pow*m;
        n--;
    }
    printf("%d", pow);
    getch();
    return 0;
}
```

\* 19:1

```
64
C:\TURBOC3\SOURCE>claatob 4 3
```

# Command Line Arguments

```
char a[]="amit";
```

```
char a[]="1234";
```

```
char a[]={ 'h','e','l','l','o', '\0'};
```

```
"5"
```