Ahsanullah University of Science and Technology (AUST)

Department of Computer Science and Engineering

LABORATORY HANDBOOK

Course No. : CSE1200

Course Title: Software Engineering - I Lab

For the students of 1st Year, 2nd semester of
B.Sc. in Computer Science and Engineering program

Table of Contents

List of Codes

Chapter 1

Course Information

1.1 Course Objective

1. Learning the use of pointer, structure, file handling and custom header files in C.
2. Applying the knowledge gained from point 1 to solve complex problems.
3. Learning about different modules and functions of igraphics.h library in C.
4. Using igraphics.h to draw different graphical shapes and structures
5. Using knowledge and skills gained from point 1-4, develop a game or software project in group.
6. Having first hand experiences about development practices- art of debugging, GUI design,collaboration through cloud/version control etc.

1.2 Reference Books

1. Teach Yourself C- Herbert Schildt, 3rd(and later) editions
2. Complete Reference C - Herbert Schildt , 3rd(and later editions)
3. Documentation of igraphics.h (provided by instructors)

1.3 Preferred Languages & Tools

1. C/C++
2. `iGraphics.h`
3. Codeblocks
4. Visual Studio 2013+
5. Sourcetree

1.4 Administrative Policy of the Laboratory

1. Students must be performed class assessment tasks individually without help of others.
2. Viva for each task will be taken and considered as a performance.
3. Plagiarism is strictly forbidden and will be dealt with punishment.

Part I

Some Advanced Features of C & C++

Chapter 2

Pointers in C++

Objectives

After completing this tutorial, you will have learned-

- Necessity of Pointers
- Definition, Value Assignment and Retrieval of Pointers
- Use of Pointers in Array, String, Function and Structure
- Learning about Dynamic Memory Allocation

2.1 Pointer Definition, Value Assignment and Retrieval

```
#include<stdio.h>

int main()
{
    int val = 32;
    printf("Value of the variable = %d\n", val);
    /*Reference operator = "&" (unary)*/
    printf("Address of the variable = %d\n", &val);

    /*Pointer declaration*/
    int *ptr;
    ptr = &val;

    /*Dereference operator = "*" (unary)*/
    printf("Value of the pointer = %d\n", *ptr);
    printf("Address of the pointer = %d\n", ptr);

    val = 10;
    printf("Value of the pointer = %d\n", *ptr);
    printf("Address of the pointer = %d\n", ptr);

    /*Changing variable value by assigning through pointer*/
    *ptr = 100;
    printf("Value of the variable = %d\n", val);

    /*When the address of variable val is assigned to ptr, the following are true:
    1. ptr is the same as &val
    2. *ptr is treated the same as val*/
    return 0;
}
```

Listing 2.1: Pointer Definition, Value Assignment and Retrieval

The Listing 2.1 shows this output after running.

```
Value of the variable = 32
Address of the variable = 6356744
Value of the pointer = 32
Address of the pointer = 6356744
Value of the pointer = 10
Address of the pointer = 6356744
Value of the variable = 100
```

2.2 Array operations with Pointer

```c
#include<stdio.h>

void printArray(int *ptr, int len)
{
    for(int i = 0 ; i < len; i++ )
    {
        printf("%d ", *(ptr+i));
    }
    printf("\n");
}

int main()
{
    int arr[5] = {1, 2, 3, 4, 5};
    printf("Address of the first array element = %p\n", &arr[0]); /*Reference*/
    printf("Value of the first array element = %d\n", *(arr));     /*Dereference*/
    printf("Address of the second array element = %p\n", &arr[1]);/*Reference*/
    printf("Value of the second array element = %d\n", *(arr+1)); /*Dereference*/
    printf("Address of the array = %p\n", arr);
    /*Address of array is same as first array element address*/

    /*Pointer declaration to the array*/
    int *ptr;
    ptr = arr;  /* using ptr = &arr will not work*/

    /*Target is to get the sum of all the array elements using ptr*/
    int sum = 0;
    for(int i = 0; i < 5; i++ )
    {
        sum = sum + *(ptr+i); /*traverse the array using ptr*/
    }
    printf("Sum = %d\n", sum);

    /*Target is to get element wise address*/
    ptr = arr;
    for(int i = 0; i < 5; i++ )
    {
        printf("%d is situated at address %d\n",*(ptr+i),ptr+i);
    }

    /*We can use pointer to pass fixed arrays in a function instead of copying*/
    int brr[5] = {10, 20, 30, 40, 50};
    int len = sizeof(brr)/sizeof(brr[0]);
    printArray(brr, len);
    return 0;
}
```

Listing 2.2: Array operations with Pointer

The Listing 2.2 shows this output after running.

```
Address of the first array element = 0060FEE8
Value of the first array element = 1
Address of the second array element = 0060FEEC
Value of the second array element = 2
Address of the array = 0060FEE8
Sum = 15
1 is situated at address 6356712
2 is situated at address 6356716
3 is situated at address 6356720
4 is situated at address 6356724
5 is situated at address 6356728
10 20 30 40 50
```

2.3 String operations with Pointer

```c
#include<stdio.h>

int stringLength(char *ptr)
{   int len = 0;
    while( *ptr != '\0' )
    {
        *ptr++;
        len++;
    }
    return len;
}

int main()
{   char str[] = "Hello World";
    printf("Address of the first array element = %d\n", &str[0]); /*Reference*/
    printf("Value of the first array element = %c\n", *(str));    /*Dereference*/
    printf("Address of the array = %d\n", str);

    /*Target is find the length of the string by passing it in a function through pointer*/
    int length = stringLength(str);
    printf("%d\n", length);

    /*Pointer declaration for the string*/
    char *ptr;
    ptr = str; /* using ptr = &str will not work*/

    /*Target is to print the string using ptr*/
    for(int i = 0; i < length; i++ )
    {
        printf("%c", *(ptr+i));
    }
    printf("\n");
    /*Target is to see the element wise location*/
    ptr = str;
    for(int i = 0; i < length; i++ )
    {
        printf("%c is situated at address %d\n", *(ptr+i),ptr+i);
    }
    return 0;
}
```

Listing 2.3: String operations with Pointer

The Listing 2.3 shows this output after running.

```
Address of the first array element = 6356724
Value of the first array element = H
Address of the array = 6356724
11
Hello World
H is situated at address 6356724
e is situated at address 6356725
l is situated at address 6356726
l is situated at address 6356727
o is situated at address 6356728
 is situated at address 6356729
W is situated at address 6356730
o is situated at address 6356731
r is situated at address 6356732
l is situated at address 6356733
d is situated at address 6356734
```

2.4 Function call by Values, Reference and Address

```c
#include<stdio.h>

void passByValue(int x)
{
    printf("Value inside passByValue function = %d\n", x);
    x = 10;
    printf("Value inside passByValue function after value change = %d\n", x);
}

void passByReference(int &x)
{
    printf("Value inside passByReference function = %d\n", x);
    x = x + 10;
    printf("Value inside passByReference function after value change = %d\n", x);
}

void passByAddress(int *x)
{
    printf("Value inside passByAddress function = %d\n", *x);
    *x = 99;
    printf("Value inside passByAddress function after value change = %d\n", *x);

}

int main()
{
    /*passing arguments by value*/
    int val = 32;
    printf("Passing arguments by Value:\n");
    printf("Value of the variable before function call = %d\n", val);
    passByValue(val);
    printf("Value of the variable after function call = %d\n", val);

    /*passing arguments by reference*/
    int nextval = 100;
    printf("\nPassing arguments by Reference:\n");
    printf("Value of the variable before function call = %d\n", nextval);
    passByReference(nextval);
    printf("Value of the variable after function call = %d\n", nextval);

    /*passing arguments by address*/
    int preval = 1;
    printf("\nPassing arguments by Address:\n");
    printf("Value of the variable before function call = %d\n", preval);
    passByAddress(&preval);
    printf("Value of the variable after function call = %d\n", preval);

    return 0;
}
```

Listing 2.4: Function call by Values, Reference and Address

The Listing 2.4 shows this output after running.

```
Passing arguments by Value:
Value of the variable before function call = 32
Value inside passByValue function = 32
Value inside passByValue function after value change = 10
Value of the variable after function call = 32

Passing arguments by Reference:
Value of the variable before function call = 100
Value inside passByReference function = 100
Value inside passByReference function after value change = 110
Value of the variable after function call = 110

Passing arguments by Address:
Value of the variable before function call = 1
Value inside passByAddress function = 1
Value inside passByAddress function after value change = 99
Value of the variable after function call = 99
```

2.5 Dynamic Memory Allocation

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

void mallocAndFree()
{
    int n;
    printf("Enter size of the array you want: ");
    scanf("%d", &n);

    /*malloc() allocates the memory for n integers*/
    int *ptr;
    ptr = (int *) malloc(sizeof(int) * n); /* [n * 4 bytes] */
    /*pointer = (datatype *) malloc(sizeof(datatype)*numberofelements)*/

    if (ptr == NULL)
    {
        printf("Memory Allocation Failed!\n");
        return;
    }

    printf("Address of the array = %d\n", ptr);
    /*Deallocates memory previously allocated by malloc() function*/

    printf("Enter elements into array\n");
    for(int i=0;i<n;i++)
        scanf("%d",(ptr+i));
    printf("Printing elements of array\n");
    for(int i=0;i<n;i++)
        printf("%d\n",*(ptr+i));

    free(ptr);
}

void newAndDelete()
{
    int n;
    printf("Enter size of the array you want: ");
    scanf("%d", &n);

    /*Allocate memory using new for n integers*/
    int *ptr;
    ptr = new int[n];

    if (ptr == NULL)
    {
        printf("Memory Allocation Failed!\n");
        return;
    }

    printf("Address of the array = %d\n", ptr);
    /*Deallocate memory previously using delete*/

    printf("Enter elements into array\n");
    for(int i=0;i<n;i++)
        scanf("%d",(ptr+i));
    printf("Printing elements of array\n");
    for(int i=0;i<n;i++)
        printf("%d\n",*(ptr+i));

    if( ptr )
    {
        delete[] ptr;
    }
}

int main()
{
    /*Use of malloc and free*/
    mallocAndFree();
    /*Use of New and delete*/
    newAndDelete();

    return 0;
}
```

Listing 2.5: Dynamic Memory Allocation

The Listing 2.5's output is left as an exercise.

2.6 Using Structures with Pointer

```c
#include<stdio.h>

typedef struct Vector Vector;

struct Vector
{
    double x;
    double y;
};

void passByValue(Vector P)
{
    printf("Values inside passbyvalue function = %f %f\n", P.x, P.y);
    P.x = 10.0;
    P.y = 10.0;
    printf("Values inside passbyvalue function after value change = %f %f\n", P.x, P.y);
}

void passByReference(Vector &P)
{
    printf("Values inside passByReference function = %f %f\n", P.x, P.y);
    P.x = P.x + 10.0;
    P.y = P.y + 10.0;
    printf("Values inside passByReference function after value change = %f %f\n", P.x, P.y);
}

void passByAddress(Vector *P)
{
    printf("Values inside passByAddress function = %f %f\n", (*P).x, (*P).y);
    (*P).x = 99.0;
    (*P).y = 99.0;
    printf("Values inside passByAddress function after value change = %f %f\n", (*P).x, (*P).y);
}

int main()
{
    Vector V;
    V.x = 0.0;
    V.y = 0.0;

    printf("Address of the vector = %d\n", &V);    /*Reference*/

    /*Pointer declaration to the structure Vector*/
    Vector *ptr;
    ptr = &V; /*ptr = V will not work here*/

    ptr->x = 2.0;
    ptr->y = 3.0;

    printf("Address of the pointer = %d\n", ptr);    /*Reference*/
    printf("Values are = %f %f\n", ptr->x, ptr->y);
    /*You can also use (*ptr).x and (*ptr).y, see the passByAddress function*/

    /*Pass the structure by value*/
    passByValue(V);
    printf("Values after function call = %f %f\n", V.x, V.y);
    /*Pass the structure by reference*/
    passByReference(V);
    printf("Values after function call = %f %f\n", V.x, V.y);
    /*Pass the structure by address*/
    passByAddress(&V);
    printf("Values after function call = %f %f\n", V.x, V.y);

    return 0;
}
```

Listing 2.6: Using Structures with Pointer

The Listing 2.6 shows this output after running.

```
Address of the vector = 6356728
Address of the pointer = 6356728
Values are = 2.000000 3.000000
Values inside passbyvalue function = 2.000000 3.000000
Values inside passbyvalue function after value change = 10.000000 10.000000
Values after function call = 2.000000 3.000000
Values inside passByReference function = 2.000000 3.000000
Values inside passByReference function after value change = 12.000000 13.000000
Values after function call = 12.000000 13.000000
Values inside passByAddress function = 12.000000 13.000000
Values inside passByAddress function after value change = 99.000000 99.000000
Values after function call = 99.000000 99.000000
```

Chapter 3

Structures in `C & C++`

Objectives

After completing this tutorial, you will have learned-

- Necessity of structures
- Declaration and initialization of structures
- Learning about structure data and methods
- Learning about interaction between structure and functions

3.1 Introduction

As stated earlier, a challenge in the path of developing a software is the abstraction and implementation of different entities/ units/ structures/ characters. As a goalkeeper of a football game or the info of a particular book in a library management system cannot be implemented through using a single `int, char, float, double` type variables, we need to resort to a mechanism which can incorporate multiple information and functionality of an entity in a single data-type (i.e. the name, club, country, shot-stopping, reflexes and movement of a goalkeeper). `Structure` helps us to define custom-data types according to our requirements.

Caution

For this tutorial- in fact for the overall course, we will write codes in `.cpp` files rather than in `.c` files. It will help us in using vast number of features, which will be explained time to time.

3.2 Definition of the Problem used for Illustration

In our illustration, we will try to define a custom data type which will be helpful for handling 2-dimensional geometry. As in a 2D space, position of a point is denoted using position vector, we will define a `Vector` structure for designing a point entity, its properties and functionalities. Moreover, we will design a structure `Line` too, which will be extended using `Vector class` (because you need two points to define a line). The properties and functionalities have been discussed in Table 3.1.

Table 3.1: Design Requirements

Entity	Data	Functionality
Vector	x,y - coordinates	initialization, Scalar Multiple Self- Scaling Dot Product Information output Vector Addition
Line	Two endpoints (vectors)	Initialization Information Output

3.3 Structure Definition, Value Assignment and Retrieval

In this section, we will see how to declare a structure and assign values to its fields as well as retrieving them. Please go thorugh the Listing 3.1 and read the comments associated with code blocks. **Remember, structure data are by default public- which means you can access them from outside the scope.**

```c
#include<stdio.h>
/*Define your Vector structure here along with the data*/
/*Follow this format
struct <Structure name>
{
    datatype1 data1;
    datatype2 data2;
    ....
    datatypeN dataN;
} ;
*/
struct Vector
{
    double x; /*x co-ordinate*/
    double y; /*y co-ordinate*/
} ; /*Don't forget the semicolon*/

int main()
{
    /*Declare your structure variable here*/
    /*Follow this format
    struct <Structure name> <Structure Variable>;
    */
    struct Vector V;
    /*Assign values to your structure data here*/
    /*Follow This format
   <Structure variable> . <Structure data> = <Value to be assigned> ;
    */
   V.x = 2.0;
   V.y = 3.0;
    /*Get values from structure data */
    /*<Structure Variable> . <Structure Data> will return the data*/
    printf("Vector V has co-ordinates %lf %lf\n",V.x, V.y);
    return 0;
}
```

Listing 3.1: Structure definition, value assignment and retrieval

The Listing 3.1 shows this output after running.

```
Vector V has co-ordinates 2.000000 3.000000
```

3.4 Use of `typedef` and Defining Structure Functions

Defining each structure variable in previously stated manner is verbose. From now on, you can use a shorter format. Moreover, you will learn now how to define structure functions and call them in Listing 3.2.

```
#include<stdio.h>
/*Define your Vector structure here along with the data*/
/*From now on, you can use thisformat to shorten the declaration*/
/*Follow this format from now on
typedef struct <Structure name> <Your preferred shorter structure name>
struct <Structure name>
{
    datatype1 data1;
    datatype2 data2;
    ....
    datatypeN dataN;

    <Return type1> <Structure function1> ( <Parameters> )
    {
    <function body>
    }

    <Return type2> <Structure function2> ( <Parameters> )
    {
    <function body>
    }

    ........
    <Return typeN> <Structure functionN> ( <Parameters> )
    {
    <function body>
    }

} ;
*/
typedef struct Vector Vector; /*It will replace verbose "struct Vector" definitions*/
/*Never use "vector" as variable or structure name- as there is
already a vector library defined in C++*/
struct Vector
{
    double x; /*x co-ordinate*/
    double y; /*y co-ordinate*/
    void printVector()
    {
        printf("%lf %lf\n",x,y); /*Carefully see how dot(.) operator has not been used.
                              This is because now you do not need to denote which structure variable
    's data
                              is being referred- calling the function with (.) will do this job*/
    }
} ; /*Don't forget the semicolon*/

int main()
{
    /*Follow this shorter format from now on
    <Your preferred shorter structure name> <Structure Variable>;
    */
    Vector V;
    V.x = 2.0;
    V.y = 3.0;
    /*Call structure function in this as
    <Structure Variable> <.> <Structure Function with parameters>
    */
    printf("Vector V has co-ordinates ");
   V.printVector();    /*As you are already calling the printVector() */
    return 0;
}
```

Listing 3.2: Structure definition using `typedef` and structure function calling

The Listing 3.2 shows this output after running.

```
Vector V has co-ordinates 2.000000 3.000000
```

3.5 Constructor Functions, Structures as Function Arguments and Return Values

Declaring a structure variable and assigning values to their properties in different statements is a cumbersome process-bearing in mind that we can actually declare basic data type variables and initialize them simultaneously. Constructor functions can be used to declare structure variables with initial values.

To declare constructor functions, you need to declare functions with same name as the structure inside the declaration of structure. You need to pass the initial values as parameters to that function. But unlike typical structure functions, constructor functions are not called using structure variables with dot(.)- rather they are automatically called when you declare the structure variables.

Don't forget to include the empty constructor in structure definition. It will enable us to declare structure variables without initializing them.

Remember, constructor functions are always defined before all other functions.

Speaking of functions, you can pass a structure as an argument to a typical function and also get it back through `return` statement.

Listing 3.3 illustrates the above mentioned statements.

```
#include<stdio.h>
/*Follow this format from now on
typedef struct <Structure name> <Your preferred shorter structure name>
struct <Structure name>
{
    datatype1 data1;
    datatype2 data2;
    ....
    datatypeN dataN;

   <Structure Name> ()
    {
    ;
    }
   <Structure Name> ( <Parameters> )
    {
   <function body>
    }
   <Rest of the functions as shown before>
} ;
*/
typedef struct Vector Vector;
struct Vector
{
    double x; /*x co-ordinate*/
    double y; /*y co-ordinate*/
    /*Empty constructor functions so that we can declare variables like- "Vector V"*/
    Vector()
    {
        ;
    }
    /*Parametrized constructor functions so that we can declare variables like- "Vector V(2.0,3.0)"
    */
    Vector(double _x, double _y)
    {
        x = _x; /*V.x will now have value sent through _x*/
        y = _y; /*V.y will now have value sent through _y*/
    }
    void printVector()
    {
        printf("%lf %lf\n",x,y); /*Carefully see how dot(.) operator has not been used.
                              This is because now you do not need to denote which structure variable
    's data
                              is being referred- calling the function with (.) will do this job*/
    }
} ; /*Don't forget the semicolon*/
/*As Vector is a custom data-type, you can use it in place of return type when you return a Vector
    variable*/
/*You can send Vector variables as function parameters just as int or char variables*/
/*This function takes two Vectors as parameters and return their resultant Vector*/
Vector add(Vector V1, Vector V2)
{
    Vector VS;
    VS.x = V1.x + V2.x;
    VS.y = V1.y + V2.y;
    return VS;
}

int main()
{
    /*Use <Structure Name> <Variable Name) (<arguments to be sent>) to use parametrized constructor
    */
    Vector V1, V2(4.0,5.5);
    V1.x = 2.0;
    V1.y = 3.0;
    printf("Vector V1 has co-ordinates ");
    V1.printVector();
    printf("Vector V2 has co-ordinates ");
    V2.printVector();
    Vector vs = add(V1,V2); /*Sending Vector variables as arguments
    and assigning the returned Vector to a Vector variable*/
    printf("Resultant Vector vs has co-ordinates ");
    vs.printVector();
    return 0;
}
```

Listing 3.3: Constructor Functions, Structures as Function Arguments and Return Values

The Listing 3.3 shows this output after running.

```
Vector V1 has co-ordinates 2.000000 3.000000
Vector V2 has co-ordinates 4.000000 5.500000
Resultant Vector vs has co-ordinates 6.000000 8.500000
```

3.6 Using Structures as Arguments & Returned Values within Structure and Call by Reference.

You can also use structure variables as structure function arguments and return values. We will illustrate this through writing the function for getting the dot product of a *Vector* variable with another one, as well as getting the scalar multiple of it. Listing 6.5 shows this. Moreover, it also shows the concept of call by reference- how can we change the data value of a structure variable by passing it as reference argument.

```
#include<stdio.h>

typedef struct Vector Vector;
struct Vector
{
    double x;
    double y;
    Vector()
    {
        ;
    }
    Vector(double _x, double _y)
    {
        x = _x;
        y = _y;
    }
    void printVector()
    {
        printf("%lf %lf\n",x,y);
    }
    /*Look how the scalar multiple of a Vector is returned here*/
    /*The format is
    Vector <Structure Function Name> (<Parameters>)
    {

    function body;
    }

    */
    Vector getScalarMultiple(double k)
    {

        Vector V(x*k, y*k); /*Declaring a new Vector whose x and y co-ordinates are scaled by a
    factor of k*/
        return V;
    }
    /*Look how the dot product of a Vector with another one is returned here*/
    /*The format is
    <Return type> <Structure Function Name> (Vector V)
    {

    function body;
    }

    */
    double getDotProduct(Vector V)
    {
        double product;
        product = x*V.x + y*V.y;  /*Co-ordinatewise multiplication between the Vector whose function
     was called
                              and the Vector who was passed as parameter*/
        return product;
    }

} ;
Vector add(Vector V1,Vector V2)
{
    Vector VS;
    VS.x = V1.x + V2.x;
    VS.y = V1.y + V2.y;
    return VS;
}

void scaleVector(Vector &V, double k)
{
    V.x = V.x*k; /*Unlike the getScalarMultiple function, you need to write V.x instead of only x.*/
    V.y = V.y*k;
}
int main()
{
    Vector V1, V2(4.0,5.5);
    V1.x = 2.0;
    V1.y = 3.0;
    printf("Vector V1 has co-ordinates ");
    V1.printVector();
    printf("Vector V2 has co-ordinates ");
    V2.printVector();
    Vector vs = add(V1,V2);
    printf("Resultant Vector vs has co-ordinates ");
    vs.printVector();
    printf("The dot product between V1 and V2 is %lf\n",V1.getDotProduct(V2)); /*Calculate the real
    dot product*/
    double k = 5.00;
    printf("Getting the scalar multiple with factor %lf of Vector vs\n",k);
    Vector scaled_vs = vs.getScalarMultiple(k);
    printf("Scaled Vector scaled_vs has co-ordinates ");
    scaled_vs.printVector(); /*Co-ordinates of the scalar multiple vector*/
    printf("Vector vs has unchanged co-ordinates ");
    vs.printVector(); /*The Vector whose scalar multiple was calculated is unchanged*/
    scaleVector(vs,k);
    printf("Scaling Vector vs by a factor of %lf\n",k);
    printf("Vector vs has changed co-ordinates ");
    vs.printVector();
    return 0;
}
```

Listing 3.4: Using Structures as Arguments & Returned Values within Structure and Call by Reference.

The Listing 6.5 shows this output after running.

```
Vector V1 has co-ordinates 2.000000 3.000000
Vector V2 has co-ordinates 4.000000 5.500000
Resultant Vector vs has co-ordinates 6.000000 8.500000
The dot product between V1 and V2 is 24.500000
Getting the scalar multiple with factor 5.000000 of Vector vs
Scaled Vector scaled_vs has co-ordinates 30.000000 42.500000
Vector vs has unchanged co-ordinates 6.000000 8.500000
Scaling Vector vs by a factor of 5.000000
Vector vs has changed co-ordinates 30.000000 42.500000
```

3.7 Use of Array, Array of Structures and Structures inside a Structure

Things to learn:

- We can use array of built-in data types (`int, char etc.`) as structure data
- We can use array of structures (i.e- a collection of Vectors)
- We can also use structures as data of another structure

- Inferring the three above mentioned features- we can use array of structures as data of another structure; this will be shown in Listing 3.5

In the previous Listings 3.1 to 3.3 and 6.5, we solved our problem requirements of `Vector` structures as mentioned in Table 3.1. Now, we will solve the requirements regarding `Line` structure and as before, illustrate new things in comments of Listing 3.5.

```
#include<stdio.h>

typedef struct Vector Vector;
struct Vector
{
    double x;
    double y;
    Vector()
    {
        ;
    }
    Vector(double _x, double _y)
    {
        x = _x;
        y = _y;
    }
    void printVector()
    {
        printf("%lf %lf\n",x,y);
    }
    Vector getScalarMultiple(double k)
    {

        Vector V(x*k, y*k);
        return V;
    }
    double getDotProduct(Vector V)
    {
        double product;
        product = x*V.x + y*V.y;
        return product;
    }

} ;
Vector add(Vector V1,Vector V2)
{
    Vector VS;
    VS.x = V1.x + V2.x;
    VS.y = V1.y + V2.y;
    return VS;
}

void scaleVector(Vector &V, double k)
{
    V.x = V.x*k;
    V.y = V.y*k;
}
/*Line structure starts from here*/
typedef struct Line Line;
struct Line
{
    /*Start and end points*/
    Vector points[2]; /*Array of structures as another structure's data*/
    Line()
    {
        ;
    }
    Line(Vector Start, Vector End)
    {
        points[0] = Start; /*Assignment works same as before*/
        points[1] = End;

    }
    void printLine()
    {
        printf("The line consists of \n");
        for(int i=0; i<2; i++)
        {
            points[i].printVector(); /*Retrieving works same as before*/
        }

    }
};

int main()
{
    Vector V1, V2(4.0,5.5);
    V1.x = 2.0;
    V1.y = 3.0;
    printf("Vector V1 has co-ordinates ");
    V1.printVector();
    printf("Vector V2 has co-ordinates ");
    V2.printVector();
    Vector vs = add(V1,V2);
    printf("Resultant Vector vs has co-ordinates ");
    vs.printVector();
    printf("The dot product between V1 and V2 is %lf\n",V1.getDotProduct(V2)); /*Calculate the real
    dot product*/
    double k = 5.00;
    printf("Getting the scalar multiple with factor %lf of Vector vs\n",k);
    Vector scaled_vs = vs.getScalarMultiple(k);
    printf("Scaled Vector scaled_vs has co-ordinates ");
    scaled_vs.printVector(); /*Co-ordinates of the scalar multiple vector*/
    printf("Vector vs has unchanged co-ordinates ");
    vs.printVector(); /*The Vector whose scalar multiple was calculated is unchanged*/
    scaleVector(vs,k);
    printf("Scaling Vector vs by a factor of %lf\n",k);
    printf("Vector vs has changed co-ordinates ");
    vs.printVector();
    Line L(V1,V2);
    L.printLine();
    return 0;
}
```

Listing 3.5: Use of Array, Array of Structures and Structures inside a Structure.

The Listing 3.5 shows this output after running.

```
Vector V1 has co-ordinates 2.000000 3.000000
Vector V2 has co-ordinates 4.000000 5.500000
Resultant Vector vs has co-ordinates 6.000000 8.500000
The dot product between V1 and V2 is 24.500000
Getting the scalar multiple with factor 5.000000 of Vector vs
Scaled Vector scaled_vs has co-ordinates 30.000000 42.500000
Vector vs has unchanged co-ordinates 6.000000 8.500000
Scaling Vector vs by a factor of 5.000000
Vector vs has changed co-ordinates 30.000000 42.500000
The line consists of
2.000000 3.000000
4.000000 5.500000
```

Practice Tasks

1. Write separate structures for `Circle, Triangle, Rectangle, Pentagon`. Implement functionalities for calculating perimeter and area.
2. Extend Vector structures for 3-dimensional space. Implement cross-product, rotation around an axis etc. operations.
3. Write a square 3×3 matrix structures and implement determinant, addition, subtraction, rank-determination and inverse computation.

Resources

1. https://www.programiz.com/cpp-programming/structure
2. https://www.tutorialspoint.com/cplusplus/cpp_data_structures.htm
3. https://www.geeksforgeeks.org/difference-c-structures-c-structures/
4. www.cplusplus.com/doc/tutorial/structures/

Chapter 4

Files in C

Objectives

After completing this tutorial, you will have learned-

- Necessity of files
- Definitions and operations of file
- Different file modes
- Structure operations on file

4.1 Introduction

Often in a real life software or a game, we need to work with persistent data- data that will exist even if we shut down the program. This is not the same case with variables-once the run is over, the variables are lost. But in a student management database, we need to store students' CGPA, ID, name so that we can retrieve them anytime. Also, we often need to save a game or update the high-scorers list. All these works can be done using `C` File operations.

4.2 How is file represented?

File is actually a huge memory block as represented in C. Usually we use array pointers to direct to the first address of the contagious memory space allocated for the array. Similarly, a file pointer is used to denote the initial position of the file. Moreover- file itself **is a structure**. This structure holds all the necessary information regarding the file. As files, structures, array- all of these take contagious memory space, pointer is needed to direct to the initial position of the allocated space.

4.3 Definition of the Problem Used for Illustration

We will work with two separate problems- (i) storing some integers in a file (ii) storing student data in a file. For problem (i), each line in the file will store two integers separated by a space. While taking input for the integers, we will take 6 integers at a time. The `Student` structure will have three fields: `id, name, cgpa`. In the same directory as the codes, we will store a text file "data.txt" which contains integers $1, 2, \ldots, 6$ in three separate lines. We have pre-stored the data to illustrate file reading operations.

4.4 File Mode

We will work with two different file modes.

Text Mode In this mode, we use text files having "txt" as extension. They store data in ASCII format and are useful for simpler works and easy to read.

Binary Mode In this mode, we use binary files having "bin" as extension. They store data in binary format and are useful for storing structures and memory-efficiency.

4.5 File Operations

In general, there are three basic file operations.

- Reading- denoted with “r” /“r+”/ “rb”
- Writing- denoted with “w” /“w+”/ “wb”
- Appending- denoted with “a” /“a+”/ “ab”

4.6 File Path

For this discussion, we will assume that the .txt/ .bin files are in the same directory as the `.cpp` files. Details about canonical/absolute/relative path will be explained in the class.

4.7 Reading from Text File

This section will show a code (Listing 4.1) illustrating how to read from a text file. The text file “data.txt” initially has $1, 2, 3, 4, 5, 6$ in three lines, with two integers each line. For reading, in other words-taking input from a file, we use fscanf()- the scanf() equivalent for file. Notice the use of for loop in iterating lines of the file. We will further show two more styles of reading to the end of the file.

```
1  #include<stdio.h>
2  int main()
3  {
4      /*READ PART*/
5      /* Syntax of declaring file
6      FILE <file_pointer_variable> = fopen(<File Path>, <File mode = "r">)
7      */
8      FILE *fp = fopen("data.txt","r"); /*Read only mode*/
9      /*Comment the next line in and the previous line out to see what happens*/
10     //FILE *fp = fopen("data1.txt","r");
11     if(fp == NULL)
12     {
13         printf("File not found\n"); /*This will be printed if you comment out line no 8 instead of
       10*/
14     }
15     else
16     {
17         printf("The file contains:\n");
18         int x,y;
19         for(int i=0; i<3; i++)
20         {
21             /* File reading syntax
22             Almost same as scanf() as it also takes input, but from file instead of console.
23             Diiference is the preceeding "f" (meaning "File") and the use of file pointer variable
       as the first parameter.
24             */
25             fscanf(fp,"%d%d",&x,&y);
26             printf("%d %d\n",x,y);
27         }
28         fclose(fp);
29     }
30     return 0;
31 }
```

Listing 4.1: Reading from Text File

The Listing 4.1 shows this output after running.

```
The file contains:
1 2
3 4
5 6
```

Caution

If fopen() does not find the file, it will return the null pointer. It has been explained in the comments.

4.8 Writing to Text File

This section illustrates (Listing 4.2) writing to a text file. The text file "data.txt" initially has $1, 2, 3, 4, 5, 6$ in three lines, with two integers each line. After we open it in writing mode, all the previous data will be erased. Then, we will write the first six Fibonacci numbers. For writing, in other words-dumping output to a file, we use fprintf()- the printf() equivalent for file. Besides, notice the new style of reading from the files until the end. Usually, `scanf(), printf(), fscanf(), fprintf()` return the number of arguments they have pursed. So, if they purse two integers in a single line, they will return 2. This fact has been leveraged in reading the file until the end.

```c
#include<stdio.h>
int main()
{
    /*WRITE PART*/
    /*Syntax of declaring file
    FILE <file_pointer_variable> = fopen(<File Path>, <File mode= "w">)
    */
    FILE *fp = fopen("data.txt","w"); /*WRITE only mode*/
    /*If the file is non-existent, a new file will be created*/
    /*If the file is existent, it will be overwritten. Experiment with "data.txt"*/

    int x,y;
    for(int i=0; i<3; i++)
    {
        /*File writing syntax
        Almost same as printf() as it also writes output, but to file instead of console.
        Diiference is the preceeding "f" (meaning "File") and the use of file pointer variable as
    the first parameter.
        */
        scanf("%d %d",&x,&y);
        fprintf(fp,"%d %d\n",x,y);

    }
    fclose(fp);
    fp = fopen("data.txt","r"); /*Read only mode*/
    printf("The file contains:\n");
    /*scanf(), printf(), fscanf(), fprintf() always returns an integer equal to the number of
    arguments read.
    So, as long as it will read two arguments per line (thus returning 2), in other words until it
    reaches
    the end of file- the loop will run.
    */
    while(fscanf(fp,"%d%d",&x,&y)==2)
    {
        printf("%d %d\n",x,y);
    }
    fclose(fp);


    return 0;
}
```

Listing 4.2: Writing to Text File

The Listing 4.2 shows this output after running.

```
The file contains:
1 1
2 3
5 8
```

4.9 Appending to Text File

Listing 4.3 illustrates appending to a text file. The text file "data.txt" now has 1, 1, 2, 3, 5, 8 in three lines, with two integers each line. After we open it in append mode, no previous data will be erased- new data will be appended at the end of the file. Then, we will write the first six multiples of 10. If you try to open a file, which is not existent, in append mode, a new file will be created.

Also, have a look at the way the reading part is done. Every file has a marker named **End of File (EOF)**. It is essentially an integer. A file can be read unless the EOF is encountered- this fact has been leveraged in this program.

```c
#include<stdio.h>
int main()
{
    /*Append PART*/
    /*Syntax of declaring file
    FILE <file_pointer_variable> = fopen(<File Path>, <File mode ="a">)
    */
    FILE *fp = fopen("data.txt","a"); /*Append mode*/
    /*If the file is non-existent, a new file will be created*/
    /*If the file is existent, it will be appended with data."*/

    int x,y;
    for(int i=0; i<3; i++)
    {
        /*File appending syntax
        same syntax as writing.*/
        scanf("%d %d",&x,&y);
        fprintf(fp,"%d %d\n",x,y);

    }
    fclose(fp);
    fp = fopen("data.txt","r"); /*Read only mode*/
    printf("The file now contains:\n");
    while(fscanf(fp,"%d%d",&x,&y)!=EOF)
    {
        printf("%d %d\n",x,y);
    }
    fclose(fp);

    return 0;
}
```

Listing 4.3: Appending to Text File

The Listing 4.3 shows this output after running.

```
The file now contains:
1 1
2 3
5 8
10 20
30 40
50 60
```

4.10 Structure Operations on Text File

Saving structures in a file can be done in both text and binary mode. In text mode, we write down the structure elements in separate lines/blocks and while reading them we parse in the same pattern as we used for writing previously. But this is a complicated and non-generic method.

In Listing 4.4, you can see how structure variables are kept in as record and retrieved from files. Try putting different input values and see the output yourself.

```
#include<stdio.h>
#include<string.h>
typedef struct Student Student;

struct Student
{
    int id;
    char name[20];
    double cgpa;
    Student()
    {
        ;
    }
    Student(int _id, char* _name, double _cgpa)
    {
        id = _id;
        strcpy(name,_name);
        cgpa = _cgpa;
    }
};

int main()
{
    /*Create and write*/
    FILE *fp = fopen("student_database.txt","w");
    Student student;
    for(int i=0; i<3; i++)
    {
        scanf("%s %d %lf",student.name,&student.id,&student.cgpa);
        fprintf(fp,"%s %d %lf\n",student.name,student.id,student.cgpa);
    }
    fclose(fp);
    /*Read*/
    fp = fopen("student_database.txt","r");
    while(fscanf(fp,"%s %d %lf",student.name,&student.id,&student.cgpa)!=EOF)
    {
        printf("%s %d %lf\n",student.name,student.id,student.cgpa);
    }
    fclose(fp);
    /*Append*/
    fp = fopen("student_database.txt","a");
    for(int i=0; i<3; i++)
    {
        scanf("%s %d %lf",student.name,&student.id,&student.cgpa);
        fprintf(fp,"%s %d %lf\n",student.name,student.id,student.cgpa);
    }
    fclose(fp);
    /*Read*/
    fp = fopen("student_database.txt","r");
    while(fscanf(fp,"%s %d %lf",student.name,&student.id,&student.cgpa)!=EOF)
    {
        printf("%s %d %lf\n",student.name,student.id,student.cgpa);
    }
    fclose(fp);
    return 0;
}
```

Listing 4.4: Structure Operations on Text File

4.11 Structure Operations on Binary File

Writing structure in a binary file is way easier- so as the reading. For this, we need to open files in binary mode at first. Then , for reading, we use `fread()` and for writing we use `fwrite()`. Both of them has same set of parameters- reference to the structure variable, size of the memory chunk to be read at once (in this case, size of the structure), number of chunks to be read at once (usually 1), the file pointer. They will return the number of chunks written/read (in the illustrated case, it is 1). So use this fact to read until the end of file.

```
#include<stdio.h>
#include<string.h>
typedef struct Student Student;

struct Student
{
    int id;
    char name[20];
    double cgpa;

    Student()
    {
        ;
    }

    Student(int _id, char* _name, double _cgpa)
    {
        id = _id;
        strcpy(name,_name);
        cgpa = _cgpa;
    }

};

int main()
{
    Student S;
    /*
    Code for writing to a file in binary mode, "wb" instead of "w"
    */
    FILE *fp = fopen("student_database.bin","wb");
    for(int i=0; i<3; i++)
    {
        /*Taking input from console*/
        scanf("%d %s %lf",&S.id,S.name,&S.cgpa);
```

```c
        /*fwrite( reference to structure variable , size of memory chunk/structure , how many chunk , file pointer )*/
        fwrite(&S,sizeof(S),1,fp);
    }
    fclose(fp);
    /*
    Code for reading from a file in binary mode, "rb" instead of "r"
    */
    fp = fopen("student_database.bin","rb");
    if(fp==NULL)
    {
        printf("File not found\n");
    }
    else
    {
        /*fread( reference to structure variable , size of memory chunk/structure , how many chunk , file pointer )*/
        /*Both fread() and fwrite() returns the "how many chunk" read. Levarage this fact in running while loop for taking input*/
        while( fread(&S,sizeof(S),1,fp) == 1)
        {
            printf("%d %s %lf\n",S.id,S.name,S.cgpa);
        }
        fclose(fp);

    }
    /*
    Code for appending in a file in binary mode, "ab" instead of "a"
    */
    fp = fopen("student_database.bin","ab");
    for(int i=0; i<3; i++)
    {
        /*Taking input from console*/
        scanf("%d %s %lf",&S.id,S.name,&S.cgpa);
        /*fwrite( reference to structure variable , size of memory chunk/structure , how many chunk , file pointer )*/
        fwrite(&S,sizeof(S),1,fp);
    }
    fclose(fp);
    /*Reading code again*/
    fp = fopen("student_database.bin","rb");
    if(fp==NULL)
    {
        printf("File not found\n");
    }
    else
    {
        /*fread( reference to structure variable , size of memory chunk/structure , how many chunk , file pointer )*/
        /*Both fread() and fwrite() returns the "how many chunk" read. Levarage this fact in running while loop for taking input*/
        while( fread(&S,sizeof(S),1,fp) == 1)
        {
            printf("%d %s %lf\n",S.id,S.name,S.cgpa);
        }
        fclose(fp);

    }
    return 0;
}
```

Listing 4.5: Structure Operations on Binary File

Remember, you will not see the same kind of output in a binary file as you see directly in a text file- as they are kept as bytes.

Practice Tasks

1. Try to store different geometric shapes' info (as illustrated in Chapter 3) in text and binary files. Also, do the read and addition(of new data) operations.
2. Design a console based library management system with create, read, update, count, delete etc. functions.
3. Think of a simple console based game and implement the game saving functionality.

Resources

1. https://www.programiz.com/c-programming/c-file-input-output
2. https://www.tutorialspoint.com/cprogramming/c_file_io.htm

Chapter 5

Developing Custom Header Files in C

Objectives

After completing this tutorial, you will have learned-

- Necessity of header files
- Syntax of developing custom header file
- Using custom header file

5.1 Introduction

In a large software project, we have to deal with multiple structures, classes, functions, global variables. If we write them in a single source file, it becomes very difficult to update or work with them. So, it is a general convention to put separate classes, structures etc. in separate library/ header files.

5.2 How does a header work?

A header is nothing but a snippet of source code put in a different file rather than being put on a single main source file. So, if we cut out a structure declaration code from a main source file, write the structure in a header file and include that header file in main source file, it will still work. We will show that with the previously stated *Vector* structure.

5.3 Definition of the Problem Used for Illustration

We will revise the problem of *Vector* structure, but this time with header.

5.4 Header File

```
/*Guard words
#ifndef #define #endif
These are guard-words. They prevent a header fle from being included twice; otherwise it will show
    conflicts.
syntax:

#ifndef <HEADER FILE NAME IN UPPER CASE>_H_INCLUDED
#define <HEADER FILE NAME IN UPPER CASE>_H_INCLUDED
<Your code>
#ENDIF
*/
#ifndef MYVECTOR_H_INCLUDED
#define MYVECTOR_H_INCLUDED

/*Write your codes between #define and #endif*/
typedef struct Vector Vector;
```

```
struct Vector
{
    double x;
    double y;
    Vector()
    {
        ;
    }
    Vector(double _x, double _y)
    {
        x = _x;
        y = _y;
    }
    void printVector()
    {
        printf("%lf %lf\n",x,y);
    }
    /*Look how the scalar multiple of a Vector is returned here*/
    /*The format is
    Vector <Structure Function Name> (<Parameters>)
    {

    function body;
    }

    */
    Vector getScalarMultiple(double k)
    {

        Vector V(x*k, y*k); /*Declaring a new Vector whose x and y co-ordinates are scaled by a
    factor of k*/
        return V;
    }
    /*Look how the dot product of a Vector with another one is returned here*/
    /*The format is
    <Return type> <Structure Function Name> (Vector V)
    {

    function body;
    }

    */
    double getDotProduct(Vector V)
    {
        double product;
        product = x*V.x + y*V.y;  /*Co-ordinatewise multiplication between the Vector whose function
     was called
                                and the Vector who was passed as parameter*/
        return product;
    }

} ;
Vector add(Vector V1,Vector V2)
{
    Vector VS;
    VS.x = V1.x + V2.x;
    VS.y = V1.y + V2.y;
    return VS;
}


```

```
#endif // MYVECTOR_H_INCLUDED
```

Listing 5.1: *myvector.h* Header file containing *Vector* class

5.5 Main File

```
#include<stdio.h>
/*Syntax of declaring custom header file
#include "<header file name>".h
*/
#include"myvector.h"

void scaleVector(Vector &V, double k)
{
    V.x = V.x*k; /*Unlike the getScalarMultiple function, you need to write V.x instead of only x.*/
    V.y = V.y*k;
}
int main()
{
    Vector V1, V2(4.0,5.5);
    V1.x = 2.0;
    V1.y = 3.0;
    printf("Vector V1 has co-ordinates ");
    V1.printVector();
    printf("Vector V2 has co-ordinates ");
    V2.printVector();
    Vector vs = add(V1,V2);
    printf("Resultant Vector vs has co-ordinates ");
    vs.printVector();
    printf("The dot product between V1 and V2 is %lf\n",V1.getDotProduct(V2)); /*Calculate the real
    dot product*/
    double k = 5.00;
    printf("Getting the scalar multiple with factor %lf of Vector vs\n",k);
    Vector scaled_vs = vs.getScalarMultiple(k);
    printf("Scaled Vector scaled_vs has co-ordinates ");
    scaled_vs.printVector(); /*Co-ordinates of the scalar multiple vector*/
    printf("Vector vs has unchanged co-ordinates ");
    vs.printVector(); /*The Vector whose scalar multiple was calculated is unchanged*/
    scaleVector(vs,k);
    printf("Scaling Vector vs by a factor of %lf\n",k);
    printf("Vector vs has changed co-ordinates ");
    vs.printVector();
    return 0;
}
```

Listing 5.2: Listing 6.5 using *myheader.h* class

The Listing 5.2 shows this output after running.

```
Vector V1 has co-ordinates 2.000000 3.000000
Vector V2 has co-ordinates 4.000000 5.500000
Resultant Vector vs has co-ordinates 6.000000 8.500000
The dot product between V1 and V2 is 24.500000
Getting the scalar multiple with factor 5.000000 of Vector vs
Scaled Vector scaled_vs has co-ordinates 30.000000 42.500000
Vector vs has unchanged co-ordinates 6.000000 8.500000
Scaling Vector vs by a factor of 5.000000
Vector vs has changed co-ordinates 30.000000 42.500000
```

Practice Tasks

1. Try to do tasks for structures with header files
2. Try to do tasks for files with header files.

Resources

1. https://www.tutorialspoint.com CprogrammingC-Header Files

Part II

2D Graphics Rendering using `iGraphics.h`

Chapter 6

Introduction to `iGraphics.h`

Objectives

After completing this tutorial, you will have learned-

- Necessity of graphics in C
- Creation and management of different shapes using `iGraphics.h`
- Using igraphics.h to draw different graphical shapes and structures6.
- Using knowledge and skills gained, develop a game or software project in group.
- Having first hand experiences about development practices, art of debugging, GUI design etc.

6.1 Installation

The user has to copy the iGraphics folder in your PC. The folder mainly contains the following files- GLUT.H, GLUT32.LIB, GLUT32.DLL, iGraphics.h, iMain.cpp and iDoc.pdf and some demo programs.

6.2 Header File

The functions defined in the header file of `iGraphics.h`is defined below.

1. `void iInitialize(int width=500, int height=500,char* title="iGraphics")`
   Description: Creates a window of specified size and title.
   Parameters:
   width- Width of the window.
   height- Height of the window.
   title- Title of the window.
   Example: iInitialize(300, 300, ”demooo”);

2. `void iClear()`
   Description: Clears the screen.
   Parameters: none
   Example: iClear();

3. `void iSetColor(double r, double g, double b)`
   Description: Sets current drawing color.
   Parameters:
   r- Red component of color.
   g- Green component of color.
   b- Blue component of color.
   Example: iSetColor(255, 0, 0); //drawing color is now set to red.

4. `void iGetPixelColor (int x, int y, int rgb[])`
   Description: Gets pixel color at coordinate (x, y).
   Parameters:
   (x, y)- co-ordinate of the pixel to check
   rgb[]- a 1D array of size 3 that is passed by the caller. Red, green
   and blue components of color are stored in this array.
   Example: iGetPixelColor(100, 120, array);

5. `void iPoint(double x, double y, int size=0)`
   Description: Draws a ppoint(x, y) on screen with current color.
   Parameters:
   x, y- Coordinates of the point.
   size- (Optioal)Size of the point.
   Example: iPoint(10, 20);

6. `void iLine(double x1, double y1, double x2, double y2)`
   Description: Draws a line on the screen with current color.
   Parameters:
   x1, y1- Coordinates of one end point.
   x2, y2- Coordinates of other end point.
   Example: iLine(10, 20, 100, 120);

7. `void iCircle(double x, double y, double r, int slices=100)`
   Description: Draws a circle on the screen with current color.
   Parameters:
   x, y- Coordinates of center.
   r- Radius of circle.
   slices- Number of line segments used to draw the circle.
   Example: iCircle(10, 20, 10);

8. `void iFilledCircle(double x, double y, double r, int slices=100)`
   Description: Draws a filled-circle on the screen with current color.
   Parameters:
   x, y- Coordinates of center.
   r- Radius of circle.
   slices- Number of line segments used to draw the circle.
   Example: iFilledCircle(10, 20, 10);

9. `void iEllipse(double x, double y, double a, double b, int slices=100)`
   Description: Draws an ellipse on the screen with current color.
   Parameters:
   x, y- Coordinates of center.
   a, b- Sizes of major and minor axes.
   slices- Number of line segments used to draw the circle.
   Example: iEllipse(10, 20, 10, 5);

10. `void iFilledEllipse(double x, double y, double a, double b, int slices=100)`
    Description: Draws a filled-ellipse on the screen with current color.
    Parameters:
    x, y- Coordinates of center.
    a, b- Sizes of major and minor axes.

slices- Number of line segments used to draw the circle.
Example: iFilledEllipse(10, 20, 10, 5);

11. `void iRectangle(double left, double bottom, double dx, double dy)`
Description: Draws a rectangle on the screen with current color.
Parameters:
left- x-coordinate of lower-left corner of the screen.
bottom- y-coordinate of lower-left corner of the screen.
dx- width of rectangle.
dy- height of rectangle.
Example: iRectangle(10, 20, 10, 5);

12. `void iFilledRectangle(double left, double bottom, double dx, double dy)`
Description: Draws a filled-rectangle on the screen with current color.
Parameters:
left- x-coordinate of lower-left corner of the screen.
bottom- y-coordinate of lower-left corner of the screen.
dx- width of rectangle.
dy- height of rectangle.
Example: iFilledRectangle(10, 20, 10, 5);

13. `void iPolygon(double x[], double y[], int n)`
Description: Draws a polygon on the screen with current color.
Parameters:
x- x coordinates of vertices of a polygon.
y- y coordinates of vertices of a polygon.
n- Number of vertices.
Example:
double xa[]=0, 10, 5;
double ya[]=0, 0, 10;
iPolygon(xa, ya, 3);

14. `void iFilledPolygon(double x[], double y[], int n)`
Description: Draws a filled-polygon on the screen with current color.
Parameters:
x- x coordinates of vertices of a polygon.
y- y coordinates of vertices of a polygon.
n- Number of vertices.
Example:
double xa[]=0, 10, 5;
double ya[]=0, 0, 10;
iFilledPolygon(xa, ya, 3);

15. `void iText(GLdouble x, GLdouble y, char *str, void* font=GLUT_BITMAP_8_BY_13)`
Description: Displays a string on screen.
Parameters:
x, y- coordinates of the first character of the string.
str- The string to show.
font- (Optional)Specifies the font type. Values could be any one of the following- GLUT_BITMAP_8_BY_13, GLUT_BITMAP_9_BY_15, GLUT_BITMAP_TIMES_ROMAN_10, GLUT_BITMAP_TIMES_ROMAN_24, GLUT_BITMAP_HE
GLUT_BITMAP_HELVETICA_12, GLUT_BITMAP_HELVETICA_18
Example: iText(50, 60, This is a text, GLUT_BITMAP_TIMES_ROMAN_10);

16. `void iShowBMP(int x, int y, char filename[])`
    Description: Shows a 24-bit .bmp file on screen. The size must be an integral power of 2.
    Parameters:
    x, y- coordinates of lower-left corner of .bmp file.
    filename- The path of the .bmp file.

17. `void iShowBMP2(int x, int y, char filename[], int ignoreColor)`
    Description: Shows a 24-bit .bmp file on screen. The size must be an integral power of 2.
    Parameters:
    x, y- coordinates of lower-left corner of .bmp file.
    filename- The path of the .bmp file.
    ignoreColor - A specified color that should not be rendered. If you have an image strip that should be rendered on top of another back ground image, then the background of the image strip should not get rendered. Use the background color of the image strip in ignoreColor parameter. Then the strip's background does not get rendered. To disable this feature, put -1 in this parameter.

18. `void iShowBMPAlternative2(int x, int y, char fileName[], int ignoreColor)`
    Puts a BMP image on screen.
    parameters:
    x - x coordinate
    y - y coordinate
    fileName - name of the BMP file
    ignoreColor - A specified color that should not be rendered. If you have an image strip that should be rendered on top of another back ground image, then the background of the image strip should not get rendered. Use the background color of the image strip in ignoreColor parameter. Then the strip's background does not get rendered. To disable this feature, put -1 in this parameter.

19. `unsigned int iLoadImage(char filename[])`
    fileName - name of the PNG/JPG file

20. `void iShowImage(int x, int y, int width, int height, unsigned int texture)`
    Puts a BMP image on screen.
    parameters:
    x - x coordinate of lower left point of the image
    y - y coordinate of lower left point of the image
    width - width of the image
    height - height of the image
    texture - the returned value from the `iLoadImage(char filename[])` function.

21. `void iSetTimer(int msec, void (*f)(void))`
    Description: Schedules a task to perform after some pre-specified time interval. The specified function f() will be called again and again automatically after the specified time interval msec. There can be at most 10 timers in your program. Once started these timers cannot be stopped. But they can be paused and resumed.
    Parameters:
    msec- Time interval in mili-seconds.
    f- The function that will be called automatically by the system again and again after the specified time interval.
    Return value:
    An integer denoting the index of the created timer. It is used to pause or resume the timer afterwards. Indexing start from 0.
    Example:

```
void func(void)
{
//code of the task that will be repeated.
}
t = iSetTimer(100, func); //call it inside main() before iInitialize();
```

Listing 6.1: iSetTimer Example

22. `void iPauseTimer(int index)`
    Description: Pauses the timer. The timer is de-activated.
    Parameters:
    index- the index of the timer that is to be paused.
    Example: iPauseTimer(t);

23. `void iResumeTimer(int index)`
    Description: Resumes the timer. The timer is activated again.
    Parameters:
    index- the index of the timer that is to be resumed.
    Example: iResumeTimer(t);

24. `void iRotate(double x, double y, double degree)`
    Rotates the co-ordinate system.
    Parameters:
    (x, y) - The pivot point for rotation
    degree - degree of rotation
    After calling iRotate(), evrey subsequent rendering will happen in rotated fashion. To stop rotation of subsequent rendering, call iUnRotate(). Typical call pattern would be:
    iRotate();
    Render your objects, that you want rendered as rotated
    iUnRotate();

6.3 Main File

Users of iGraphics only have to edit, compile and run iMain.cpp using Visual C++ 10+.

6.3.1 Draw Functions

function `iDraw()` is called again and again by the system. Then we are going to present a sample code using those functions. In the sample code, several rectangles are drawn using that function. The sample output is given after the code.

```
#include "iGraphics.h"
#include "bitmap_loader.h"

int x = 300, y = 300, r = 20;
int win_b = 400, win_l = 400;
float delay = 0;
int speed = 0;
/*
 function iDraw() is called again and again by the system.

 */
void drawRoadLines(int offset)
{
  int len = 10;
  int width = 5;
  int gap = 10;
  int midx = 0, midy = win_l / 2;
  for (int i = 0; i < win_b / (len + gap); i++)
    iFilledRectangle((i * (len + gap) + offset) % win_b, midy, len, width);
}
int k = 0;
void iDraw()
{
  //place your drawing codes here
  iClear();
  /*
   iSetColor(20, 200, 200);
   iFilledCircle(x, y, r);
   //iFilledRectangle(10, 30, 20, 20);
   iSetColor(20, 200, 0);
   iText(40, 40, "Hi, I am iGraphics");*/

  iSetColor(0, 0, 200);
  iFilledRectangle(win_b / 2, win_l / 2, 30, 10);
  iSetColor(20, 200, 0);
  drawRoadLines(k++);
  iDelayMS(500);
  //iShowBMPAlternative(100, 100, "smurf.bmp");

}

/*
 function iMouseMove() is called when the user presses and drags the mouse.
 (mx, my) is the position where the mouse pointer is.
 */
void iMouseMove(int mx, int my)
{
  printf("x = %d, y= %d\n", mx, my);
  //place your codes here
}

/*
 function iMouse() is called when the user presses/releases the mouse.
 (mx, my) is the position where the mouse pointer is.
 */
void iMouse(int button, int state, int mx, int my)
{
  if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
    //   printf("x = %d, y= %d\n",mx,my);
    x += 10;
    y += 10;
  }
  if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
    x -= 10;
    y -= 10;
  }
}

/*
 function iKeyboard() is called whenever the user hits a key in keyboard.
 key- holds the ASCII value of the key pressed.
 */
void iKeyboard(unsigned char key)
```

```
{
  if (key == 'q')
  {
    exit(0);
  }
  else if (key == 'a' && speed < 5)
    speed += 1;
  else if (key == 's' && speed > 0)
    speed -= 1;
  //place your codes for other keys here
}

/*
 function iSpecialKeyboard() is called whenver user hits special keys like-
 function keys, home, end, pg up, pg down, arraows etc. you have to use
 appropriate constants to detect them. A list is:
 GLUT_KEY_F1, GLUT_KEY_F2, GLUT_KEY_F3, GLUT_KEY_F4, GLUT_KEY_F5, GLUT_KEY_F6,
 GLUT_KEY_F7, GLUT_KEY_F8, GLUT_KEY_F9, GLUT_KEY_F10, GLUT_KEY_F11, GLUT_KEY_F12,
 GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN, GLUT_KEY_PAGE UP,
 GLUT_KEY_PAGE DOWN, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT
 */
void iSpecialKeyboard(unsigned char key)
{

  if (key == GLUT_KEY_END)
  {
    exit(0);
  }
  //place your codes for other keys here
}

int main()
{
  //place your own initialization codes here.
  iInitialize(400, 400, "demo");
  return 0;
}
```

Listing 6.2: Draw Function

Output:

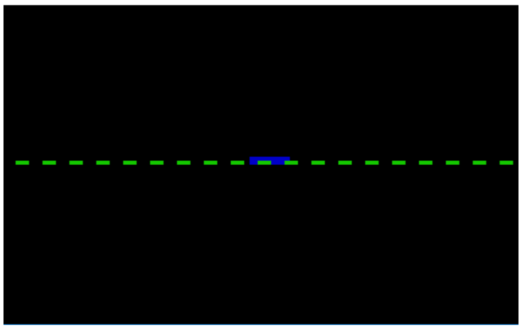


Figure 6.1: Output of sample code

6.3.2 Keyboard Functions

There are two mouse using functions in `iGraphics.h`. We are going to beriefly describe them in this section. Then we are going to present a sample code using those functions. In the sample code, the user has to click on the box to write something in that box. After that, he can write anything in that box.. The sample output is given after the code.

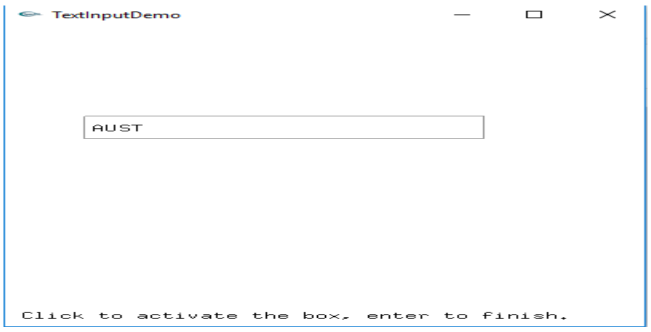
1. function `iKeyboard()` is called whenever the user hits a key in keyboard.
   key- holds the ASCII value of the key pressed.

2. function `iSpecialKeyboard()` is called whenver user hits special keys like- function keys, home, end, pg up, pg down, arraows etc. the user has to use appropriate constants to detect them.
   A list is: GLUT KEY F1, GLUT KEY F2, GLUT KEY F3, GLUT KEY F4, GLUT KEY F5, GLUT KEY F6, GLUT KEY F7, GLUT KEY F8, GLUT KEY F9, GLUT KEY F10, GLUT KEY F11, GLUT KEY F12, GLUT KEY LEFT, GLUT KEY UP, GLUT KEY RIGHT, GLUT KEY DOWN, GLUT KEY PAGE UP, GLUT KEY PAGE DOWN, GLUT KEY HOME, GLUT KEY END, GLUT KEY INSERT

```
# include "iGraphics.h"

char str[100], str2[100];
int len;
int mode;

void drawTextBox()
{
  iSetColor(150, 150, 150);
  iRectangle(50, 250, 250, 30);
}

/*
  function iDraw() is called again and again by the system.
*/
void iDraw()
{
  //place your drawing codes here

  iClear();
  iSetColor(255,255,255);
  iFilledRectangle(0,0,400,400);
  drawTextBox();
  if(mode == 1)
  {
    iSetColor(0, 0, 0);
    iText(55, 260, str);
  }

  iText(10, 10, "Click to activate the box, enter to finish.");
}

/*
  function iMouseMove() is called when the user presses and drags the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouseMove(int mx, int my)
{
  //place your codes here
}

/*
  function iMouse() is called when the user presses/releases the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouse(int button, int state, int mx, int my)
{
  if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
    if(mx >= 50 && mx <= 300 && my >= 250 && my <= 280 && mode == 0)
    {
      mode = 1;
    }
  }
  if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
  }
}

/*
  function iKeyboard() is called whenever the user hits a key in keyboard.
  key- holds the ASCII value of the key pressed.
*/
void iKeyboard(unsigned char key)
{
  int i;
  if(mode == 1)
  {
        if(key == '\r')
    {
      mode = 0;
      strcpy(str2, str);
      printf("%s\n", str2);
      for(i = 0; i < len; i++)
        str[i] = 0;
      len = 0;
    }
    else
    {
      str[len] = key;
      len++;
    }
  }

  if(key == 'x')
  {
    //do something with 'x'
    exit(0);
  }
  //place your codes for other keys here
}

/*
  function iSpecialKeyboard() is called whenver user hits special keys like-
  function keys, home, end, pg up, pg down, arraows etc. you have to use
  appropriate constants to detect them. A list is:
  GLUT_KEY_F1, GLUT_KEY_F2, GLUT_KEY_F3, GLUT_KEY_F4, GLUT_KEY_F5, GLUT_KEY_F6,
  GLUT_KEY_F7, GLUT_KEY_F8, GLUT_KEY_F9, GLUT_KEY_F10, GLUT_KEY_F11, GLUT_KEY_F12,
  GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN, GLUT_KEY_PAGE UP,
  GLUT_KEY_PAGE DOWN, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT
*/
void iSpecialKeyboard(unsigned char key)
{

  if(key == GLUT_KEY_END)
  {
    exit(0);
  }

  //place your codes for other keys here
}

int main()
{
  //place your own initialization codes here.
  len = 0;
  mode = 0;
  str[0]= 0;
  iInitialize(400, 400, "TextInputDemo");
  return 0;
}
```

Listing 6.3: Keyboard Functions in `iGraphics.h`

Output:



```
Click to activate the box, enter to finish.
```

Figure 6.2: Text box creation using keyboard function

6.3.3 Mouse Functions

There are two mouse using functions in `iGraphics.h`. We are going to briefly describe them in this section. Then we are going to present a sample code using those functions. In the sample code, each time, the user presses the GLUT BUTTON LEFT, a flower appears in that position and when the GLUT BUTTON RIGHT is pressed, the last flower disappears from the screen. The sample output is given after the code.

1. function `iMouse()` is called when the user presses/releases the mouse.
   (mx, my) is the position where the mouse pointer is.

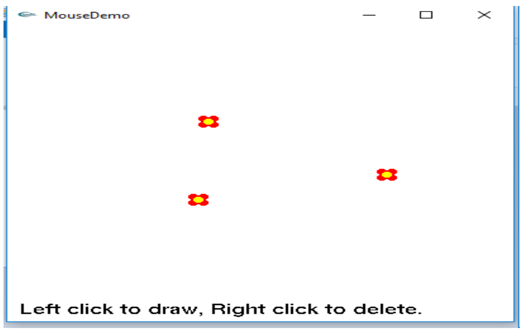
2. function `iMouseMove()` is called when the user presses and drags the mouse.
   (mx, my) is the position where the mouse pointer is.

```
# include "iGraphics.h"

int x[100];
int y[100];
int total;
int MX=-1,MY=-1;


void drawFlowerAt(int p, int q)
{
  iSetColor(255, 0, 0);
  iFilledCircle(p+4, q+4, 4);

  iFilledCircle(p+4, q-4, 4);

  iFilledCircle(p-4, q+4, 4);

  iFilledCircle(p-4, q-4, 4);


  iSetColor(255, 255, 0);
  iFilledCircle(p, q, 4);
}

/*
  function iDraw() is called again and again by the system.
*/
void drawThickLine(int mx,int my)
{
  if(MX!=-1 && MY !=-1)
  {
    iSetColor(255,0,0);
  iFilledRectangle(mx,my,20,20);
  }

}
void iDraw()
{
  //place your drawing codes here

  iClear();
  iSetColor(255,255,255);
  iFilledRectangle(0,0,400,400);
  drawThickLine(MX,MY);
  int i;
  for(i = 0; i < total; i++)
  {
    drawFlowerAt(x[i], y[i]);
  }
  iSetColor(0,0,0);
  iText(10, 10, "Left click to draw, Right click to delete.",GLUT_BITMAP_HELVETICA_18);
}

/*
  function iMouseMove() is called when the user presses and drags the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouseMove(int mx, int my)
{
  //place your codes here
    MX =mx;
 MY =my;
}

/*
  function iMouse() is called when the user presses/releases the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouse(int button, int state, int mx, int my)
{
  if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
    if(total < 100)
    {
      x[total] = mx;
      y[total] = my;
```

```
            total++;
        }
    }
    if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    {
        //place your codes here
        if(total>0)
        {
            total--;
        }
    }
}

/*
    function iKeyboard() is called whenever the user hits a key in keyboard.
    key- holds the ASCII value of the key pressed.
*/
void iKeyboard(unsigned char key)
{
    if(key == 'x')
    {
        //do something with 'x'
        exit(0);
    }
    //place your codes for other keys here
}

/*
    function iSpecialKeyboard() is called whenver user hits special keys like-
    function keys, home, end, pg up, pg down, arraows etc. you have to use
    appropriate constants to detect them. A list is:
    GLUT_KEY_F1, GLUT_KEY_F2, GLUT_KEY_F3, GLUT_KEY_F4, GLUT_KEY_F5, GLUT_KEY_F6,
    GLUT_KEY_F7, GLUT_KEY_F8, GLUT_KEY_F9, GLUT_KEY_F10, GLUT_KEY_F11, GLUT_KEY_F12,
    GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN, GLUT_KEY_PAGE UP,
    GLUT_KEY_PAGE DOWN, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT
*/
void iSpecialKeyboard(unsigned char key)
{

    if(key == GLUT_KEY_END)
    {
        exit(0);
    }
    //place your codes for other keys here
}

int main()
{
    //place your own initialization codes here.
    total = 0;
    iInitialize(400, 400, "MouseDemo");
    return 0;
}
```

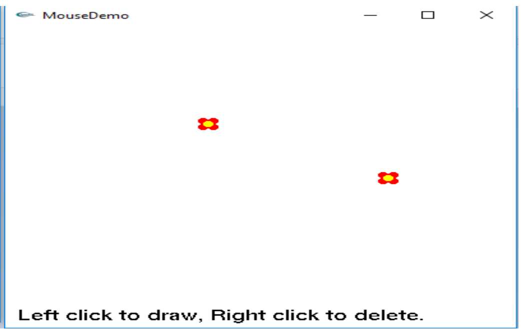
Listing 6.4: Mouse Functions in `iGraphics.h`

Output:



Left click to draw, Right click to delete.

(a) Pressed GLUT BUTTON LEFT



Left click to draw, Right click to delete.

(b) Pressed GLUT BUTTON RIGHT

Figure 6.3: Mouse Operations Example

6.3.4 Timer Functions

In `iGraphics.h`, there are functions to be used fo the task of animations, who are implemented as a definite task occuring in regular interval. The sample outputs is shown after the code.

```
# include "iGraphics.h"
# include "bitmap_loader.h"

int pic_x, pic_y;

/*
  function iDraw() is called again and again by the system.
*/
void iDraw()
{
  //place your drawing codes here

  iClear();

  iShowBMPAlternativeSkipWhite(-pic_x, -pic_y, "images\\smurf.bmp");

  iText(10, 10, "Use cursors to move the picture.");
}

/*
  function iMouseMove() is called when the user presses and drags the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouseMove(int mx, int my)
{
  //place your codes here
  pic_x = mx;
  pic_y = my;
}

/*
  function iMouse() is called when the user presses/releases the mouse.
  (mx, my) is the position where the mouse pointer is.
*/
void iMouse(int button, int state, int mx, int my)
{
  if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
  }
  if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
  {
    //place your codes here
  }
}

/*
  function iKeyboard() is called whenever the user hits a key in keyboard.
  key- holds the ASCII value of the key pressed.
*/
void iKeyboard(unsigned char key)
{
  if(key == 'x')
  {
    //do something with 'x'
    exit(0);
  }
  //place your codes for other keys here
}

/*
  function iSpecialKeyboard() is called whenver user hits special keys like-
  function keys, home, end, pg up, pg down, arraows etc. you have to use
  appropriate constants to detect them. A list is:
  GLUT_KEY_F1, GLUT_KEY_F2, GLUT_KEY_F3, GLUT_KEY_F4, GLUT_KEY_F5, GLUT_KEY_F6,
  GLUT_KEY_F7, GLUT_KEY_F8, GLUT_KEY_F9, GLUT_KEY_F10, GLUT_KEY_F11, GLUT_KEY_F12,
  GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN, GLUT_KEY_PAGE UP,
  GLUT_KEY_PAGE DOWN, GLUT_KEY_HOME, GLUT_KEY_END, GLUT_KEY_INSERT
*/
void iSpecialKeyboard(unsigned char key)
{

  if(key == GLUT_KEY_END)
  {
    exit(0);
  }
  if(key == GLUT_KEY_LEFT)
  {
    pic_x--;
  }
  if(key == GLUT_KEY_RIGHT)
  {
    pic_x++;
  }
  if(key == GLUT_KEY_UP)
  {
    pic_y++;
  }
  if(key == GLUT_KEY_DOWN)
  {
    pic_y--;
  }
  //place your codes for other keys here
}

int main()
{
  //place your own initialization codes here.
  pic_x = 30;
  pic_y = 20;
  iInitialize(500, 400, "PictureDemo");
  return 0;
```

```
}
```

Listing 6.5: Timer function example

Output:

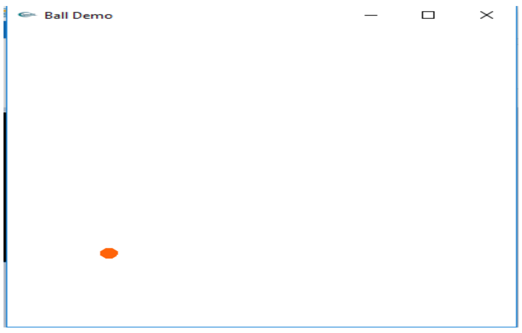


Figure 6.4: Timer function demo output

Practice Tasks

1. Try to draw different geometric shapes' using `iGraphics.h`.
2. Design a login page to take information from user using keyboard and mouse functions.
3. Store value in the file from GUI and show information in GUI after reading from file.

Part III

Some Important Development Issues

Chapter 7

Do’s and Don’ts of GUI Design

Objectives

After completing this tutorial, you will have learned-

- Main theme of Designing Graphical User Interface (GUI)
- Thinks to take care of while designing GUI
- Things to avoid while designing GUI

7.1 Introduction

Designing GUI has become a field of study due to its relation with behavioural psychology and Ergonomics. The first and foremost aim of designing GUI is to provide user comfort in such a way that user never feel the urge to quit the product. To do so, one has to maintain some standards of designing and avoid some bad notions- which will be discussed below in brief.

7.2 Tips

We will mention here the do’s and don’t of GUI in brief. They will be discussed thoroughly in lab class.

Eye-level Always try to put different elements corresponding with eye position.

- Codeblocks have their editor window at center, where the human eye generally can look effortlessly.
- In FIFA games, player health bars are usually at the bottom- because user is focused mostly on the ball which is moving along the ground level rather than center level.
- in King of Fighters, health bars are stationed in upper corners- because user focus is generally on the upper side of the screen while playing this game.

User command Try to avoid user command and instead put icons.

Color scheme Always try to put contrasting colors. Use color wheels or pallets from Google.

Uniformity Keep all the positioning, font sizes uniform.

Font and Shape Justification Remember to justify the texts and shapes in the screen

Intuitive Design Try to make the design intuitive to the point that usere.te s need to follow any documentation or manual.

Requirement Analysis and Survey Ask people what they expect about a good design and take feedback from themabout your designs.

Practice Tasks

1. Find some design issues from the softwares/websites/games you use regularly.
2. Implement the design principles in your own project.

Resources

1. https://en.wikibooks.org/wiki/GUI_Design_Principles
2. bokardo.com/principles-of-user-interface-design/

Chapter 8

Code Hygiene and Modularity

Objectives

After completing this tutorial, you will have learned-

- How to write a code in cleaner way according to convention
- How to make a code readable and reusable

8.1 Introduction

A code is written once, but used and read thousand times. So while writing the code, we need to write it in such way that whoever reads it, understands it at once (code hygiene), as well as reusing/extending/updating this code does not become a tedious task (code modularity).

8.2 Tips

We will mention here the tips about code hygiene and modularity . They will be discussed thoroughly in lab class.

1. Try to match the indentation level as much as you can. Indentation levels should match with the code scopes and blocks.
2. Follow Hungarian nomenclature for variables naming
3. While naming functions, start with action verbs (i.e.- get_ fibonacci_ number() instead of fibonaccinumber)
4. Class/structure names should start with capital letter, whereas objects/structure variable should start with small letter.
5. Try to write intuitive variable names. (i.e.- cnt instead of cnt).
6. Use header files/library files/packages to modularize the codes
7. Represent a single entity with a single class/structure
8. One function should do only one work.
9. Comment as much as you can. Comment the task, the parameters and the return values.

Practice Tasks

1. Find some code hygiene and modularity issues from open source projects.
2. Implement the coding principles in your own project.

Resources

1. https://www.nceclusters.no/globalassets/filer/nce/diverse/the-pragmatic-programmer.pdf

Chapter 9

Debugging Techniques

Objectives

After completing this tutorial, you will have learned-

- the strategies of debugging a program
- the use of debugging tools

9.1 Introduction

We will discuss three-fold ways of debugging: psychological, tactical and technical. These issues arise very often as it is largely improbable to write bug-free code at a single iteration.

9.2 Tips

We will mention here the tips about debugging.

9.2.1 Psychology

1. Convince yourself that the bug didn't appear for nothing or the program is behaving abruptly. It is because we make logical, syntactic and semantic mistakes for which the bug occurs. So be proactive and check those issues first. Convincing yourself that you can make a mistake is the first and foremost step of redeeming it.

9.2.2 Tactical Ways

1. Try to comment as much as possible, in a comprehensive way.
2. Do unit tests
3. Write one function at a time and test it immediately. Don't pile up the bugs.

9.2.3 Technical Ways

1. Learn the use of debuggers and breakpoints in IDE
2. Learn the use of stack traces

Practice Tasks

1. Experiment with Visual Studio/Codeblocks debugger tools
2. Implement the debugging principles in your own project.

Resources

1. https://docs.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour

2. jonskeet.uk/csharp/debugging.html

Chapter 10

Cloud Collaboration and Version Control Systems

Objectives

After completing this tutorial, you will have learned-

- Use of cloud services for collaborating (Drive, Dropbox)
- Use of Version Control Systems

10.1 Introduction

Collaborating through cloud and version control system enable a team to collaborate and merge their works efficiently. In this lab, we will inform them about it.

10.2 Tips

10.2.1 Version Control System

1. Learn to use Sourcetree/ Github.
2. Learn to use different git commands (push, pull, merge, add, stage etc.)

Practice Tasks

1. Open a repository in github/bitbucket with your project
2. Collaborate with your teammates using git

Resources

1. https://www.atlassian.com/git/tutorials