REFRESHING MEMORY: THE BASICS

1.1 Example of standard I/O in C:

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

int main(int argc, char** argv) {
    printf("Enter your roll number and name: ");
    int roll;
    char name[80];
    scanf("%d %s", &roll, name);
    printf("\nYour name: %s\nYour roll number: %d", name, roll);

    return (EXIT_SUCCESS);
}
```

1.2 String input:

- **`scanf`**`("%s")` – Discards the characters after first occurrence of a whitespace.
- **`gets()`** – Takes all the characters until Enter key is pressed.
- Mixing **`scanf`**`("%s")` with **`gets()`** is **dangerous**.

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

int main(int argc, char** argv) {
    char name[80];
    printf("Enter your full name: ");
    gets(name);
    printf("Your full name is: %s", name);

    return (EXIT_SUCCESS);
}
```

1.3 Control Structures (i.e. loops and conditional statements) & Arrays:

Just like in Java.

1.4 Constant Declaration Syntax:

```c
const int SOME_CONSTANT = 5;
```

If a `const` is placed inside a function its effect would be *localized* to that function, whereas, if its Is placed outside all functions then its effect would be *global*. We cannot exercise such finer control while using a `#define`.

1.5 Data Types in C (16-Bit Platform) & Format Specifiers for `printf()` & `scanf()`:

Data Type	Memory	Range	Declaration	printf()	scanf()
Single Character	1 Byte	–	char	%c	%c
String		–	char []	%s	%s
Signed Integer	2 Bytes	-2^{15} to $2^{15}-1$	int	%d	%d
Unsigned Integer	2 Bytes	$2^{16}-1$	unsigned	%u (integer) %x (hexadecimal) %o (octal)	%u
Long Integer	4 Bytes	-2^{16} to $2^{16}-1$ (signed) $2^{17}-1$ (unsigned)	long int	%ld, %lu, %lx, %lo	%ld, %lu, %lx, %lo
Floating Point	4 Bytes	10^{-38} to 10^{38} 6-digit precision	float	%f (decimal notation) %e (exponential notation) %g (%f or %e – the shorter)	%f or %e
Double	8 Bytes	10^{-308} to 10^{308} 15-digit precision	double	%le, %lf	%le, %lf
Long Double	10 Bytes	10^{-4932} to 10^{4932} 19-digit precision	long double	%Le, %Lf	%Le, %Lf

1.4 **Functions:**

Two situations:

- Function definition comes *before* the function call
- Function definition comes *after* the function call

Situation 1: Function definition comes *before* the function call – no problem at all!

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

void print() {    //Can also be used: void print(void) [C++ style, though…]
    printf("Hello World :P");
}

int main(int argc, char** argv) {
    print();
    return (EXIT_SUCCESS);
}
```

Situation 2: Function definition comes *after* the function call – must need function declaration (prototype).

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

void print();   //Prototype for the function print()

int main(int argc, char** argv) {
    print();
    return (EXIT_SUCCESS);
}

void print() {
    printf("Hello World :P");
}
```

Tip: *Both of the following prototypes are valid:*

```
int some_function(int param1, char *param2);
int some_function(int, char*);
```

POINTERS

2.1 Basic Concepts:

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

int main(int argc, char** argv) {
    int a = 5;
    printf("Value of a = %d\n", a);
    printf("Address of a = %d\n", &a);
    printf("Value of a = %d\n\n", *(&a));  // Or: *&a

    int *b;
    b = &a;
    printf("Address of a = %d\n", b);      // b = &a
    printf("Value of a = %d\n\n", *b);     // *b = *(&a) = a

    int **c;
    c = &b;
    printf("Address of b = %d\n", c);     // c = &b
    printf("Address of a = %d\n", *c);    // *c = *(&b) = b = &a
    printf("Value of a = %d\n\n", **c);   // **c = *(*c) = *(*(&b)) = *b = *(&a) = a

    return (EXIT_SUCCESS);
}

/*
 Sample Output:

 Value of a = 5
 Address of a = 2280676
 Value of a = 5

 Address of a = 2280676
 Value of a = 5

 Address of b = 2280672
 Address of a = 2280676
 Value of a = 5

 [PRESS ENTER TO CLOSE WINDOW]

 */
```

Note: The declaration `float *f` does not mean that `f` is going to contain a floating-point value. What it means is, `f` is going to contain the *address* (which is always an **integer**) of a **floating-point** value.

Similarly, `char *ch` means that `ch` is going to contain the *address* of a **char** value.

2.2 Passing addresses to functions:

Call-By-Value

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

void swap(int x, int y) {
    int temp = y;
    y = x;
    x = temp;

    printf("x = %d, y = %d\n", x, y);
}

int main(int argc, char** argv) {
    int a = 5, b = 10;
    swap(a, b);
    printf("a = %d, b = %d\n\n", a, b);

    return (EXIT_SUCCESS);
}
```

Call-By-Reference

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

void swap(int *x, int *y) {
    int temp = *y;
    *y = *x;
    *x = temp;  // Don't use "x = temp;"
}

int main(int argc, char** argv) {
    int a = 5, b = 10;
    swap(&a, &b);
    printf("a = %d, b = %d\n\n", a, b);

    return (EXIT_SUCCESS);
}
```

2.3 Returning pointer from functions:

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

int *sqr(int i) {
    static int result = i * i;
    return &result;
}

int main(int argc, char** argv) {
    int *p = sqr(5);
    printf("Address of p = %d\n", p);     // Output: 4206624
    printf("Value of p = %d\n\n", *p);    // Output: 25

    return (EXIT_SUCCESS);
}
```

- `int *sqr(int i)` means `sqr()` is a function which receives an **int** *value* and returns an **int** *pointer*.
- If the variable `result` weren't `static`, then the second `printf()` won't have printed 25. That's because, when the control comes back from `sqr()`, `result` dies. So, even if we have its address in p, we can't access `result` since it's already dead.
- Using 'call by reference' intelligently, we can make a function return more than one value at a time, which is not possible ordinarily.

2.4 Pointers & Arrays:

One Dimensional Array:

Passing Array Using Reference

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

void parse(int arr[], int no_of_elements) {
    for (int i = 0; i < no_of_elements; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n\n");
}

int main(int argc, char** argv) {
    int arr[] = {1, 2, 3, 4};
    int no_of_elements = sizeof(arr) / sizeof(arr[0]);
    parse(arr, no_of_elements);

    return (EXIT_SUCCESS);
}
```

Passing Array Using Pointer

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

void parse(int *arr, int no_of_elements) {
    for (int i = 0; i < no_of_elements; i++) {
        printf("%d ", *(arr + i));
    }
    printf("\n\n");
}

int main(int argc, char** argv) {
    int arr[] = {1, 2, 3, 4};
    int no_of_elements = sizeof(arr) / sizeof(arr[0]);
    parse(arr, no_of_elements);

    return (EXIT_SUCCESS);
}
```

- The name of an array represents its **base address**.
- The decalaration `int (*p)[5];` means that p is a pointer to a one-dimensional array of 5 elements.
- Every time a pointer is incremented, it points to the immediately next location of its ***type***.
- ***Only*** the following two operations can be performed on a pointer:
  (a) Addition of a number to a pointer.
  (b) Subtraction of a number from a pointer.
- It's wrong to use `arr++` in the above program, since `arr` is a constant. It's like using 5++. To increment the value of `arr`, it must be copied to a pointer variable. For example:

```
int *p = arr;
p++;          // Same as p = arr + 1;
```

Two Dimensional Array:

Passing Array Using Reference

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

void parse(int arr[][4], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, char** argv) {
    int arr[][4] = {{1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 0, 1, 2}};
    int rows = sizeof(arr) / sizeof(arr[0]);
    parse(arr, rows);

    return (EXIT_SUCCESS);
}
```

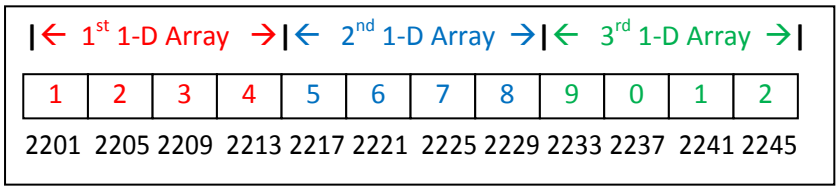
Passing Array Using Pointer

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

void parse(int *arr, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", *(arr + i * cols + j));
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, char** argv) {
    int arr[][4] = {{1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 0, 1, 2}};
    int rows = sizeof(arr) / sizeof(arr[0]);
    int cols = sizeof(arr[0]) / sizeof(arr[0][0]);
    parse(arr[0], rows, cols); //or, *arr

    return (EXIT_SUCCESS);
}
```



Arrays of Pointers:

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

int main(int argc, char** argv) {
    int *arr[3];
    int x = 5, y = 6, z = 7;

    arr[0] = &x;
    arr[1] = &y;
    arr[2] = &z;

    for (int i = 0; i < 3; i++) {
        printf("%d ", *(arr[i]));
    }

    return (EXIT_SUCCESS);
}
```

2.5 Dynamic Memory Allocation:

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

int main(int argc, char** argv) {
    printf("Enter the number of students: ");
    int n, *p;
    scanf("%d", &n);
    p = (int*) malloc(n * sizeof(int));    // Or, p = (int*) calloc(n, sizeof(int))
    if (p == NULL) {
        printf("\nMemory allocation failed.");
        exit(1);
    }
    printf("Enter the marks using Enter key:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", (p + i));
    }
    return (EXIT_SUCCESS);
}
```

2.6 Pointers and Strings:

Basic ideas:

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

int main(int argc, char** argv) {
    // Printing a string using pointers
    char name[] = "Somebody";
    char *x;
    x = name;
    while (*x != '\0') {
        printf("%c", *x);
        x++;
    }

    // A string (character array) cannot be assigned to another string,
    // but a pointer to a string can be assigned to another pointer.
    char str1[] = "Hi";
    char str2[3];
    char *si = "Hello";
    char *di;
    //str2 = str1;          //Error
    di = si;                //OK

    // Once a string has been defined, it cannot be initialized to another
    // set of characters. But such an operation is valid with char pointers.
    char str[] = "Hi";
    //str = "Bye"           //Error
    char *y = "Hi";
    y = "Bye";              //OK

    //However, a constant string (or pointer) cannot be altered...
    char *p = "Hi";                 //Pointer is variable, so is string
    *p = 'B';               //OK
    p = "Bye";              //OK
    const char *q = "Hi";           //String is constant, pointer is not
    //*q = 'B';             //Error
    q = "Bye";              //OK
    char *const r = "Hi";           //Pointer is constant, string is not
    *r = 'B';               //OK
    //r = "Bye";            //Error
    const char *const s = "Hi";     //String is constant, so is pointer
    //*s = 'B';             //Error
    //s = "Bye";            //Error

    return (EXIT_SUCCESS);
}
```

Array of Strings (or, array of pointers to arrays of characters):

```c
#include <stdio.h>
#include <stdlib.h>
void swap(char **str1, char **str2) {
    char *temp = *str2;
    *str2 = *str1;
    *str1 = temp;
}

int main(int argc, char** argv) {
    char *names[] = {"Somebody", "Nobody"};   // In array notation, char names[][9];
    printf("Original order: %s %s\n", names[0], names[1]); // Or, *(names),*(names+1)

    // Swap names
    char *temp = names[1];
    names[1] = names[0];
    names[0] = temp;
    printf("New order:      %s %s\n", names[0], names[1]);

    // Swap back through function
    swap(names, names + 1);         // Or, swap(&names[0], &names[1]);
    printf("After reorder:  %s %s\n\n", names[0], names[1]);
    return (EXIT_SUCCESS);
}
```

Printing the command-line arguments:

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

int main(int argc, char** argv) {
    int i;
    for (i = 0; i < argc; i++) {
        char *p = argv[i];          // Or, char *p = *(argv + i);
        while (*p != '\0') {
            printf("%c", *p);
            p++;
        }
        printf("\n");
    }

    return (EXIT_SUCCESS);
}
```

2.7 Pointers & Structures:

Pointers & Structures *without* using `typedef`:

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

int main(int argc, char** argv) {
    struct book {
        char name[255];
        char author[255];
        float price;
    };
    struct book b1 = {"C Revisited", "Sharafat", 0.0};
    printf("%s, %s, %f\n", b1.name, b1.author, b1.price);
    struct book *ptr = &b1;
    printf("%s, %s, %f\n", ptr->name, ptr->author, ptr->price);
    return (EXIT_SUCCESS);
}
```

Pointers & Structures using `typedef`:

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

int main(int argc, char** argv) {
    typedef struct {
        char name[255];
        char author[255];
        float price;
    } BOOK;
    BOOK b1 = {"C Revisited", "Sharafat", 0.0};
    printf("%s, %s, %f\n", b1.name, b1.author, b1.price);
    BOOK *ptr = &b1;
    printf("%s, %s, %f\n", ptr->name, ptr->author, ptr->price);
    return (EXIT_SUCCESS);
}
```

2.8 Pointers to Functions:

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

double sqr(float num) {
   return num * num;
}

int main(int argc, char** argv) {
    double (*func_ptr)(float);
    func_ptr = sqr;
    // Or, double (*func_ptr)(float) = sqr;
    double result = (func_ptr)(5.5);
    printf("%lf", result);

    return (EXIT_SUCCESS);
}
```

I/O

3.1 I/O Functions Reference:

File manipulation functions:

- `FILE *fopen(const char *filename, const char *mode)`

  Opens a file for reading or writing.

  `filename` – Name of the file (might include directory name).
  `mode` – One of the following modes:

`r`	Open for reading. The file must already exist.
`w`	Open for writing. If the file exists, its contents are overwritten. Otherwise, new file is created.
`a`	Open for appending. If the file exists, contents are appended. Otherwise, new file is created.
`r+`	Open for both reading and writing. [File existing conditions are the same as `r`.]
`w+`	Open for both reading and writing. [File existing conditions are the same as `w`.]
`a+`	Open for both reading and appending. [File existing conditions are the same as `a`.]

  Note: In case of `r+`, `w+` and `a+` mode, however, a program must not alternate immediately between reading and writing. After a write operation, you **must** call the `fflush()` function or a positioning function (`fseek()`, `fsetpos()`, or `rewind()`) before performing a read operation. After a read operation, you must call a positioning function before performing a write operation.

  Returns: A pointer to the `FILE` structure[1] on success. `NULL`[2] on failure.

- `int fclose(FILE *fp)`

  Flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the stream's input and output buffers.

  `fp` – The `FILE` pointer.

  Returns: 0 on success. `EOF`[3] on failure.

Sequential read/write functions:

Character I/O:

- ```
  int fgetc(FILE *fp)
  int getc(FILE *fp)
  int getchar(void)
  ```

  Reads a character from the input stream referenced by *fp*, or from keyboard in case of `getchar()`.

  `fp` – The `FILE` pointer.

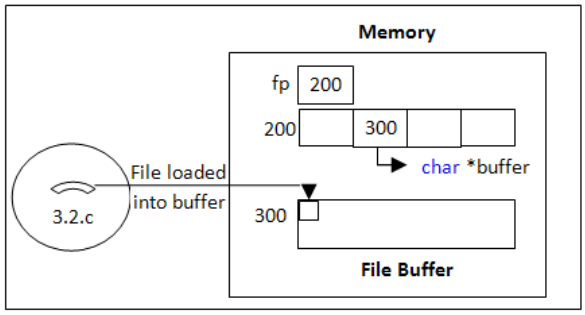
  Returns: `int` value of the character read on success. `EOF` on failure.

[1] The `FILE` structure as defined in stdio.h is as follows:

```
typedef struct {
    int             level;    /* Fill/empty level of buffer */
    unsigned        flags;    /* File status flags          */
    char            fd;       /* File descriptor            */
    unsigned char   hold;     /* Ungetc char if no buffer   */
    int             bsize;    /* Buffer size                */
    unsigned char   *buffer;  /* Data transfer buffer       */
    unsigned char   *curp;    /* Current active pointer     */
    unsigned        istemp;   /* Temporary file indicator   */
    short           token;    /* Used for validity checking */
} FILE;
```

[2] `NULL` is an integer constant whose value is 0.

[3] `EOF` is an integer constant whose value is -1.


```

- ```
  int fputc(int c, FILE *fp)
  int putc(int c, FILE *fp)
  int putchar(int c)
  ```

  Writes the char value of the argument *c* to the output stream referenced by *fp*, or to the monitor in case of putchar().

  c – int value of the character to be written.
  fp – The FILE pointer.

  Returns: int value of the character written on success. EOF on failure.

String I/O:

- ```
  char *fgets(char *buf, int n, FILE *fp)
  char *gets(char *buf)
  ```

  fgets() reads up to *n* - 1 characters from the input stream referenced by *fp* into the buffer addressed by *buf*, appending a null character to terminate the string. If the function encounters a newline character or the end of the file before it has read the maximum number of characters, then only the characters read up to that point are read into the buffer. The newline character '\n' is also stored in the buffer if read.

  gets() reads a line of text from standard input into the buffer addressed by *buf*. The newline character that ends the line is *replaced* by the null character that terminates the string in the buffer.

  buf – Pointer to the string where the characters read are to be stored.
  n – Number of characters to be read into the buffer.
  fp – The FILE pointer.

  Returns: The value of the argument *buf* on success. NULL on failure.

- ```
  int fputs(const char *s, FILE *fp)
  int puts(const char *s)
  ```

  fputs() writes the string *s* to the output stream referenced by *fp*. The null character that terminates the string is *not* written to the output stream.

  puts() writes the string *s* to the standard output stream, followed by a newline character.

  s – Pointer to the string which is to be written.
  fp – The FILE pointer.

  Returns: A non-negative value on success. EOF on failure.

Formatted I/O:

- ```
  int *fscanf(FILE *fp, const char *format, ...)
  int *scanf(const char *format, ...)
  ```

  Reads from the input stream specified by *fp*, or from the keyboard in case of scanf().

  fp – The FILE pointer.
  format – Format specifiers.

  Returns: Number of data items successfully converted and stored (on success). EOF on failure.

- ```
  int *fprintf(FILE *fp, const char *format, ...)
  int *printf(const char *format, ...)
  ```

  Writes to the output stream specified by *fp*, or to the monitor in case of printf().

  fp – The FILE pointer.
  format – Format specifiers.

  Returns: Number of characters written (on success). EOF on failure.
  ```

Block/Record I/O:

Note: In case of block/record IO, on systems that distinguish between text and binary file access modes, the file should be opened in binary mode (by appending b to the mode specified in the *mode* argument of `fopen()`).

- `int fread(const void *buffer, int size, int n, FILE *fp)`

  Reads up to *n* data objects of the specified *size* from file referenced by the `FILE` pointer *fp*, and stores them in the memory block pointed to by the *buffer* argument.

  buffer – Pointer to the object where the data read are to be stored.
  size – Size of ***one*** object.
  n – Number of objects to be read.
  fp – The `FILE` pointer.

  Returns: Number of data objects read (on success). If this number is less than *n*, then either the end of the file was reached or an error occurred.

- `int fwrite(const void *buffer, int size, int n, FILE *fp)`

  Writes up to *n* data objects of the specified *size* from the buffer addressed by the pointer argument *buffer* to the file referenced by the `FILE` pointer *fp*.

  buffer – Pointer to the object which is to be written.
  size – Size of ***one*** object.
  n – Number of objects to be written.
  fp – The `FILE` pointer.

  Returns: Number of data objects actually written to the file. (on success). 0 if either the object size *size* or the number of objects *n* was 0. If a write error occurs, then the return number would be less than *n*.

Random access file functions:

- `long ftell(FILE *fp)`

  Returns the file position of the stream specified by *fp*.

  fp – The `FILE` pointer.

  Returns: The offset (in bytes) of the current character from the beginning of the file (on success). -1 on error.

- `int fseek(FILE *fp, long offset, int origin)`

  Sets the file position indicator to a position specified by the value of *offset* and by a reference point indicated by the *origin* argument.

  fp – The `FILE` pointer.
  offset – The offset from the reference point indicated by the `origin` argument.
  origin – One of the following modes:

Macro name	Value	Offset is relative to
SEEK_SET	0	The beginning of the file.
SEEK_CUR	1	The current file position.
SEEK_END	2	The end of the file.

  Returns: 0 on success. Non-zero value on error.

- `void rewind(FILE *fp)`

  Sets the file position indicator to the beginning of the file. This function is equivalent to

  ```
  (void) fseek(fp, 0L, SEEK_SET)
  ```

  fp – The `FILE` pointer.

  Returns: Nothing.

Error detection functions:

- `int ferror(FILE *fp)`

  Used to determine if an error has occurred.

  `fp` – The `FILE` pointer.

  Returns: 0 if no error has occurred. Non-zero value if there is an error.

- `void perror(const char *string)`

  Prints a message to the standard error stream. The output includes first the string referenced by the pointer argument, if any; then a colon and a space, then the error message that corresponds to the current value of the `errno` variable, ending with a newline character.

  `string` – The custom message to be printed.

  Returns: Nothing.

Miscellaneous functions:

- `int fflush(FILE *fp)`

  Empties the I/O buffer of the open file specified by the `FILE` pointer argument. If the file was opened for writing, `fflush()` writes the contents of the file. If the file is only opened for reading, the function clears the buffer.

  `fp` – The `FILE` pointer.

  Returns: 0 on success. `EOF` if an error occurs in writing to the file.

3.2 **Complete Concepts Program:**

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

int main(int argc, char** argv) {
    FILE *in;
    FILE *out;


    // Copying a file using Character I/O
    printf("Character I/O\n");
    char ch;
    in = fopen("database.txt", "r");
    out = fopen("database_copy_char.txt", "w");
    while ((ch = fgetc(in)) != EOF) {
        printf("%c", ch);
        fputc(ch, out);
    }
    fclose(in);
    fclose(out);
    printf("\n\n");


    // Copying a file using String I/O
    printf("String I/O\n");
    char str[10];
    in = fopen("database.txt", "r");
    out = fopen("database_copy_string.txt", "w");
    while (fgets(str, sizeof(str), in) != NULL) {
        printf("%s", str);
        fputs(str, out);
    }
    fclose(in);
    fclose(out);
    printf("\n\n");
```

```c
    // Copying a file using Formatted I/O
    printf("Formatted I/O\n");
    char name[5];
    int roll;
    float marks;
    in = fopen("database.txt", "r");
    out = fopen("database_copy_formatted.txt", "w");
    while (fscanf(in, "%s\t%i\t%f\n", name, &roll, &marks) != EOF) {
        printf("%s\t%i\t%f\n", name, roll, marks);
        fprintf(out, "%s\t%i\t%f\n", name, roll, marks);
    }
    fclose(in);
    fclose(out);
    printf("\n\n");


    // Copying a file using Record/Block I/O
    printf("Record/Block I/O\n");
    struct database {
        char name[5];
        int roll;
        float marks;
    };
    struct database db[10];
    int no_of_records = 0;

    in = fopen("database.rec", "rb");
    out = fopen("database_copy.rec", "wb+");
    while (fread(&db[no_of_records], sizeof(db[0]), 1, in) == 1) {
        no_of_records++;
    }
    fwrite(db, sizeof(db[0]), no_of_records, out);
    fclose(in);

    // Verify copied file
    fflush(out);
    rewind(out);
    while (fread(&db[no_of_records], sizeof(db[0]), 1, out) == 1) {
        printf("%s\t%d\t%f\n", db[no_of_records].name, db[no_of_records].roll,
                db[no_of_records].marks);
        no_of_records++;
    }
    fclose(out);
    printf("\n\n");


    // Use of random access file and error detection functions
    in = fopen("database.rec", "rb");

    // Read the 2nd record
    printf("2nd record from database:\n");
    if (fseek(in, sizeof(db[0]), SEEK_SET) == 0) {
        if (fread(&db[1], sizeof(db[1]), 1, in) == 1) {
            printf("%s\t%d\t%f\n", db[1].name, db[1].roll, db[1].marks);
        } else {
            if (ferror(in)) {
                perror("Error while reading 2nd record: \n");
            } else {
                printf("No record left to be read.\n");
            }
        }
    } else {
        perror("Error while seeking: ");
    }
```

```
    // Read the 5th record
    printf("5th record from database:\n");
    if (fseek(in, sizeof(db[0]) * 2, SEEK_CUR) == 0) {
        if (fread(&db[1], sizeof(db[1]), 1, in) == 1) {
            printf("%s\t%d\t%f\n", db[1].name, db[1].roll, db[1].marks);
        } else {
            perror("Error while reading 5th record: \n");
        }
    } else {
        perror("Error while seeking: \n");
    }


    return (EXIT_SUCCESS);
}
```

THE END

Sharafat Ibn Mollah Mosharraf

12th Batch (2005-2006),
Dept. of Computer Science & Engineering,
University of Dhaka.
E-mail: sharafat_8271@yahoo.co.uk
Home Page: www.sharafat.info
Blog: http://blog.sharafat.info