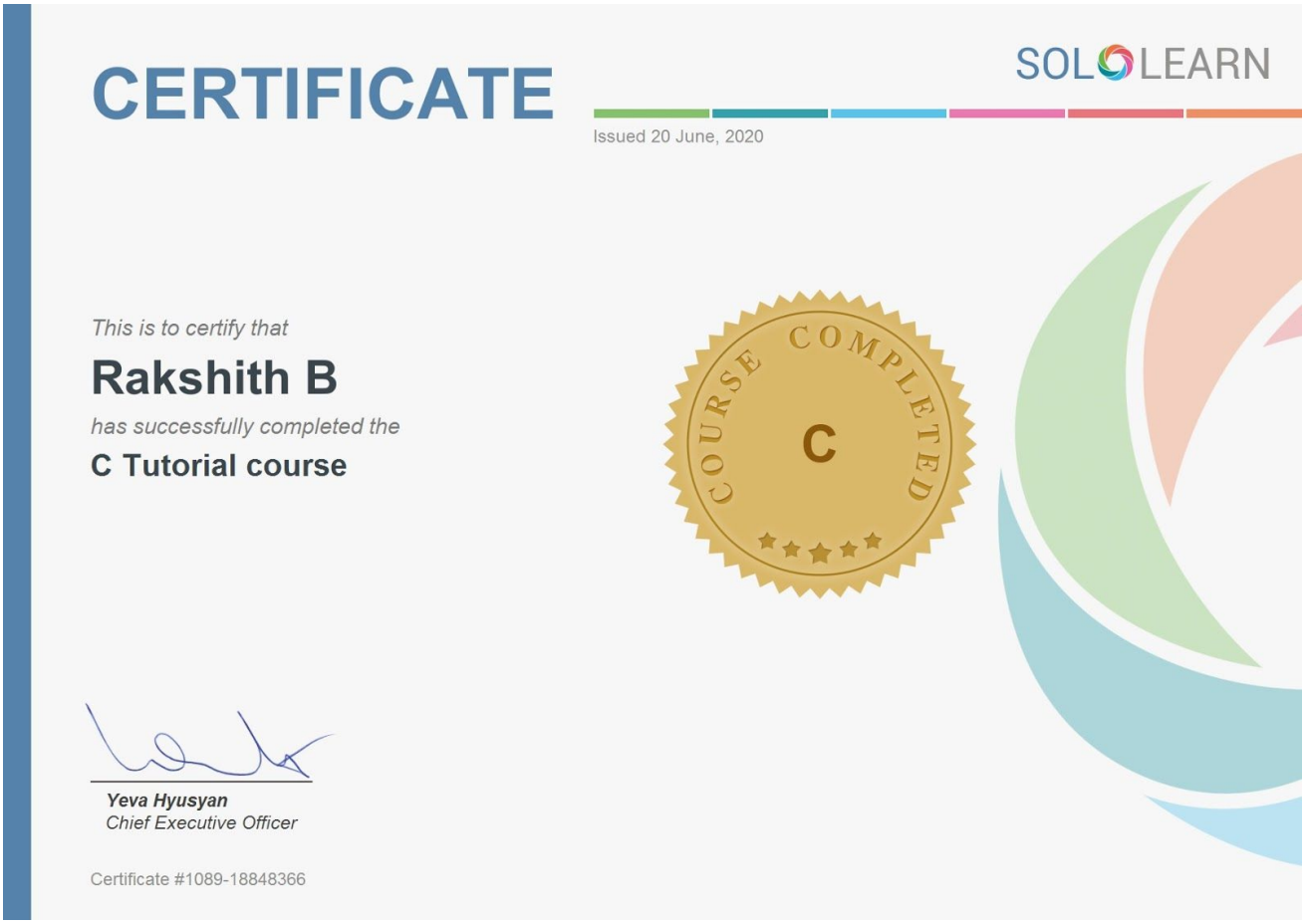
# REPORT JUNE 20

| Date: | 20 JUNE 2020 | Name: | Rakshith B |
|---|---|---|---|
| Course: | Solo Learning | USN: | 4AL16EC409 |
| Topic: | Files and Error Handling,Pre-Processor | Semester & Section: | 6th SEM B |
| Github Repository: | Rakshith-B | | |

**FORENOON SESSION DETAILS**

**Image of session**




**Report –**

Accessing Files

An external file can be opened, read from, and written to in a C program. For these operations, C includes the **FILE** type for defining a file stream. The **file stream** keeps track of where reading and writing last occurred.

The **stdio.h** library includes file handling functions:
**FILE** Typedef for defining a file pointer.

**fopen(filename, mode)** Returns a FILE pointer to file *filename* which is opened using *mode*. If a file cannot be opened, NULL is returned.

Mode options are:
- **r** open for reading (file must exist)
- **w** open for writing (file need not exist)
- **a** open for append (file need not exist)
- **r+** open for reading and writing from beginning
- **w+** open for reading and writing, overwriting file
- **a+** open for reading and writing, appending to file

**fclose(fp)** Closes file opened with FILE fp, returning 0 if close was successful. **EOF** (end of file) is returned if there is an error in closing.

```
#include <stdio.h>

int main() {
 FILE *fptr;

 fptr = fopen("myfile.txt", "w");
 if (fptr == NULL) {
   printf("Error opening file.");
   return -1;
 }
 fclose(fptr);
 return 0;
}
```

## Reading from a File

The **stdio.h** library also includes functions for reading from an open file. A file can be read one character at a time or an entire string can be read into a character **buffer**, which is typically a char array used for temporary storage.

**fgetc(fp)** Returns the next character from the file pointed to by *fp*. If the end of the file has been reached, then **EOF** is returned.

**fgets(buff, n, fp)** Reads n-1 characters from the file pointed to by *fp* and stores the string in buff. A NULL character '\0' is appended as the last character in *buff*. If fgets encounters a newline character or the end of file before n-1 characters is reached, then only the characters up to that point are stored in buff.

**fscanf(fp, conversion_specifiers, vars)** Reads characters from the file pointed to by *fp* and assigns input to a list of variable pointers *vars* using *conversion_specifiers*. As with scanf, fscanf stops reading a string when a space or newline is encountered.

The following program demonstrates reading from a file:

```
#include <stdio.h>

int main() {
 FILE *fptr;
```

```
 int c, stock;
 char buffer[200], item[10];
 float price;

 /* myfile.txt: Inventory\n100 Widget 0.29\nEnd of List */

 fptr = fopen("myfile.txt", "r");

 fgets(buffer, 20, fptr);    /* read a line */
 printf("%s\n", buffer);

 fscanf(fptr, "%d%s%f", &stock, item, &price); /* read data */
 printf("%d  %s  %4.2f\n", stock, item, price);

 while ((c = getc(fptr)) != EOF) /* read the rest of the file */
   printf("%c", c);

 fclose(fptr);
 return 0;
}
```

## Writing to a File

The stdio.h library also includes functions for writing to a file. When writing to a file, newline characters '\n' must be explicitly added.

fputc(char, fp) Writes character *char* to the file pointed to by *fp*.

fputs(str, fp) Writes string *str* to the file pointed to by *fp*.

fprintf(fp, str, vars) Prints string *str* to the file pointed to by *fp*. *str* can optionally include format specifiers and a list of variables vars.

The following program demonstrates writing to a file:

```
FILE *fptr;
char filename[50];
printf("Enter the filename of the file to create: ");
gets(filename);
fptr = fopen(filename, "w");

/* write to file */
fprintf(fptr, "Inventory\n");
fprintf(fptr, "%d %s %f\n", 100, "Widget", 0.29);
fputs("End of List", fptr);
```

## Binary File I/O

Writing only characters and strings to a file can become tedious when you have an array or structure. To write entire blocks of memory to a file, there are the following binary functions:

Binary file mode options for the fopen() function are:
- rb open for reading (file must exist)
- wb open for writing (file need not exist)
- ab open for append (file need not exist)
- rb+ open for reading and writing from beginning
- wb+ open for reading and writing, overwriting file
- ab+ open for reading and writing, appending to file

fwrite(ptr, item_size, num_items, fp) Writes *num_items* items of *item_size* size from pointer *ptr* to the file pointed to by file pointer *fp*.

fread(ptr, item_size, num_items, fp) Reads *num_items* items of *item_size* size from the file pointed to by file pointer *fp* into memory pointed to by *ptr*.

fclose(fp) Closes file opened with file *fp*, returning 0 if close was successful. EOF is returned if there is an error in closing.

The following program demonstrates writing to and reading from binary files:

```
FILE *fptr;
int arr[10];
int x[10];
int k;

/* generate array of numbers */
for (k = 0; k < 10; k++)
 arr[k] = k;

/* write array to file */
fptr = fopen("datafile.bin", "wb");
fwrite(arr, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* read array from file */
fptr = fopen("datafile.bin", "rb");
fread(x, sizeof(arr[0]), sizeof(arr)/sizeof(arr[0]), fptr);
fclose(fptr);

/* print array */
for (k = 0; k < 10; k++)
 printf("%d", x[k]);
```

## Controlling the File Pointer

There are functions in stdio.h for controlling the location of the file pointer in a binary file:
ftell(fp) Returns a long int value corresponding to the *fp* file pointer position in number of bytes from the start of the file.

fseek(fp, num_bytes, from_pos) Moves the *fp* file pointer position by *num_bytes* bytes relative to position *from_pos*, which can be one of the following constants:
- SEEK_SET start of file
- SEEK_CUR current position
- SEEK_END end of file

The following program reads a record from a file of structures:

```
typedef struct {
 int id;
 char name[20];
} item;

int main() {
 FILE *fptr;
 item first, second, secondf;

 /* create records */
 first.id = 10276;
 strcpy(first.name, "Widget");
 second.id = 11786;
 strcpy(second.name, "Gadget");

 /* write records to a file */
 fptr = fopen("info.dat", "wb");
 fwrite(&first, 1, sizeof(first), fptr);
 fwrite(&second, 1, sizeof(second), fptr);
 fclose(fptr);

 /* file contains 2 records of type item */
 fptr = fopen("info.dat", "rb");

 /* seek second record */
 fseek(fptr, 1*sizeof(item), SEEK_SET);
 fread(&secondf, 1, sizeof(item), fptr);
 printf("%d  %s\n", secondf.id, secondf.name);
 fclose(fptr);
 return 0;
}
```

## Exception Handling

Central to good programming practices is using error handling techniques. Even the most solid coding skills may not keep a program from crashing should you forget to include exception handling.

An **exception** is any situation that causes your program to stop normal execution. **Exception handling**, also called **error handling**, is an approach to processing runtime errors.

C does not explicitly support exception handling, but there are ways to manage errors:

- Write code to prevent the errors in the first place. You can't control user input, but you can check to be sure that the user entered valid input. When performing division, take the extra step to ensure that **division by 0** won't occur.
- Use the **exit** statement to gracefully end program execution. You may not be able to control if a file is available for reading, but you don't need to allow the problem to crash your program.

### The exit Command

The exit command immediately stops the execution of a program and sends an exit code back to the calling process. For example, if a program is called by another program, then the calling program may need to know the exit status.

Using exit to avoid a program crash is a good practice because it closes any open file connections and processes.

You can return any value through an exit statement, but 0 for success and -1 for failure are typical. The predefined stdlib.h macros EXIT_SUCCESS and EXIT_FAILURE are also commonly used.
For example:

```
int x = 10;
int y = 0;

if (y != 0)
 printf("x / y = %d", x/y);
else {
 printf("Divisor is 0. Program exiting.");
 exit(EXIT_FAILURE);
}
```

## Using errno

Some library functions, such as fopen(), set an error code when they do not execute as expected. The error code is set in a global variable named errno, which is defined in the errno.h header file. When using errno you should set it to 0 before calling a library function.

To output the error code stored in errno, you use fprintf to print to the stderr file stream, the standard error output to the screen. Using stderr is a matter of convention and a good programming practice.

You can output the errno through other means, but it will be easier to keep track of your exception handling if you only use stderr for error messages.

To use errno, you need to declare it with the statement extern int errno; at the top of your program (or you can include the errno.h header file).
For example:

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

```
extern int errno;

int main() {
 FILE *fptr;

 errno = 0;
 fptr = fopen("c:\\nonexistantfile.txt", "r");
 if (fptr == NULL) {
   fprintf(stderr, "Error opening file. Error code: %d\n", errno);
  exit(EXIT_FAILURE);
 }

 fclose(fptr);
 return 0;
}
```

## The perror and strerror Functions

When a library function sets errno, a cryptic error number is assigned. For a more descriptive message about the error, you can use perror(). You can also obtain the message using strerror() in the string.h header file, which returns a pointer to the message text.

perror() must include a string that will precede the actual error message. Typically, there is no need for both perror() and strerror() for the same error, but both are used in the program below for demonstration purposes:

```
FILE *fptr;
errno = 0;

fptr = fopen("c:\\nonexistantfile.txt", "r");
if (fptr == NULL) {
 perror("Error");
 fprintf(stderr, "%s\n", strerror(errno));
 exit(EXIT_FAILURE);
}
```

## EDOM and ERANGE Error Codes

**Some of the mathematical functions in the math.h library set errno to the defined macro value EDOM when a domain is out of range.**
**Similarly, the ERANGE macro value is used when there is a range error.**
**For example:**

```
float k = -5;
float num = 1000;
float result;

errno = 0;
result = sqrt(k);
if (errno == 0)
```

```
 printf("%f ", result);
else if (errno == EDOM)
 fprintf(stderr, "%s\n", strerror(errno));

errno = 0;
result = exp(num);
if (errno == 0)
 printf("%f ", result);
else if (errno == ERANGE)
 fprintf(stderr, "%s\n", strerror(errno));
```

## Pre-Processor

## Preprocessor Directives

The C preprocessor uses the # directives to make substitutions in program source code before compilation.
For example, the line #include <stdio.h> is replaced by the contents of the stdio.h header file before a program is compiled.

Preprocessor directives and their uses:

#include Including header files.

#define, #undef Defining and undefining macros.

#ifdef, #ifndef, #if, #else, #elif, #endif Conditional compilation.

#pragma Implementation and compiler specific.

#error, #warning Output an error or warning message An error halts compilation.

## The #include Directive

The #include directive is for including header files in a program. A header file declares a collection of functions and macros for a library, a term that comes from the way the collection of code can be reused.

Some useful C libraries are:
stdio input/output functions, including printf and file operations.
stdlib memory management and other utilities
string functions for handling strings
errno errno global variable and error code macros
math common mathematical functions
time time/date utilities

Corresponding header files for the libraries end with .h by convention. The #include directive expects brackets <> around the header filename if the file should be searched for in the compiler include paths.

A user-defined header file is also given the .h extension, but is referred to with quotation marks, as in "myutil.h". When quotation marks are used, the file is searched for in the source code directory. For example:

```
#include <stdio.h>

#include “myutil.h”
```

## The #define Directive

The #define directive is used to create object-like macros for constants based on values or expressions.

#define can also be used to create function-like macros with arguments that will be replaced by the preprocessor.

Be cautious with function-like definitions. Keep in mind that the preprocessor does a direct replacement without any calculations, which can lead to unexpected results, as demonstrated with the following program:

```
#include <stdio.h>
#define PI 3.14
#define AREA(r) (PI*r*r)

int main() {
 float radius = 2;
 printf("%3.2f\n", PI);
 printf("Area is %5.2f\n", AREA(radius));
 printf("Area with radius + 1: %5.2f\n", AREA(radius+1));
 return 0;
}
```

Formatting Preprocessor Directives
When using preprocessor directives, the # must be the first character on a line. But there can be any amount of white space before # and between the # and the directive.

If a # directive is lengthy, you can use the \ continuation character to extend the definition over more than one line.

**For example:**

```
#define VERY_LONG_CONSTANT \
23.678901

#define MAX 100
#define MIN 0
#    define SQUARE(x) \
  x*x
```

## Predefined Macro Definitions

In addition to defining your own macros, there are several standard predefined macros that are always available in a C program without requiring the #define directive:

__DATE__ The current date as a string in the format Mm dd yyyy
__TIME__ The current time as a string in the format hh:mm:ss
__FILE__ The current filename as a string
__LINE__ The current line number as an int value
__STDC__ 1

**For example:**

```
char curr_time[10];

char curr_date[12];

int std_c;



strcpy(curr_time, __TIME__);

strcpy(curr_date, __DATE__);

printf("%s %s\n", curr_time, curr_date);

printf("This is line %d\n", __LINE__);

std_c = __STDC__;

printf("STDC is %d", std_c);
```

## The #ifdef, #ifndef, and #undef Directives

The #ifdef, #ifndef, and #undef directives operate on macros created with #define.

For example, there will be compilation problems if the same macro is defined twice, so you can check for this with an #ifdef directive. Or if you may want to redefine a macro, you first use #undef.

The program below demonstrates these directives:

```
#include <stdio.h>

#define RATE 0.08
#ifndef TERM
 #define TERM 24
#endif

int main() {
```

```
 #ifdef RATE  /* this branch will be compiled */
  #undef RATE
  printf("Redefining RATE\n");
  #define RATE 0.068
 #else  /* this branch will not be compiled */
  #define RATE 0.068
 #endif
 printf("%f  %d\n", RATE, TERM);

 return 0;
}
```

## Preprocessor Operators

The C preprocessor provides the following operators.

The # Operator

The # macro operator is called the stringification or stringizing operator and tells the preprocessor to convert a parameter to a string constant.

White space on either side of the argument are ignored and escape sequences are recognized.

For example:

```
#define TO_STR(x) #x

printf("%s\n", TO_STR( 123\\12 ));
```