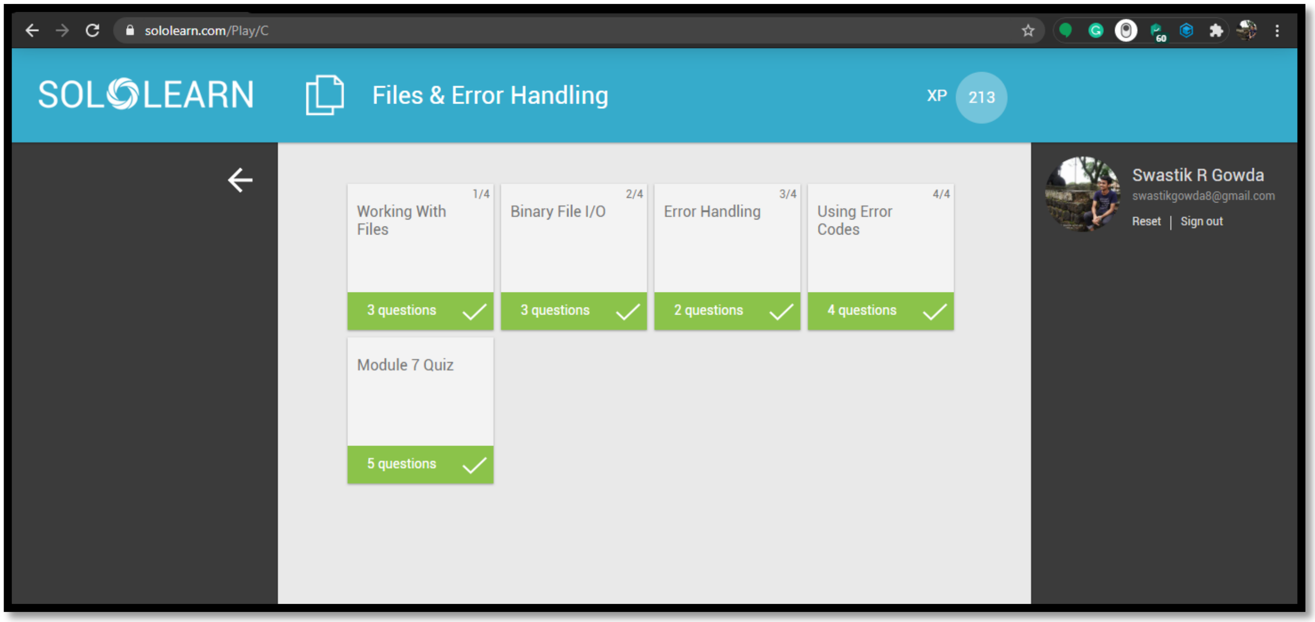
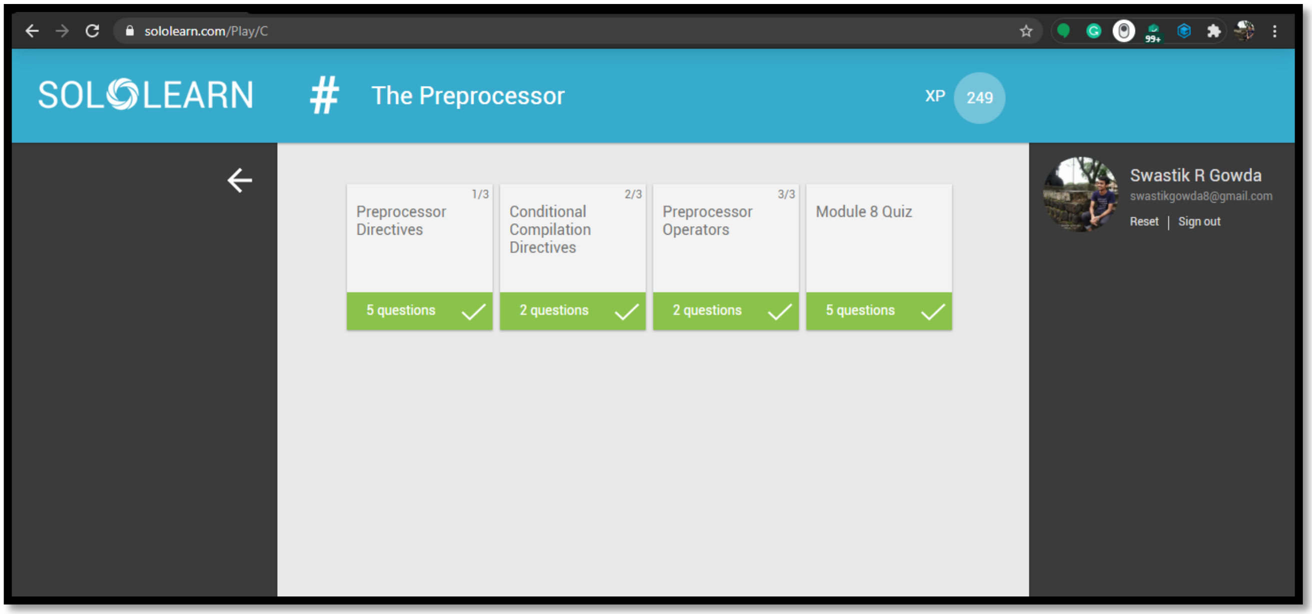
DAILY ASSESSMENT

Date:	20-June-2020	Name:	Swastik R Gowda
Course:	Solo-Learn	USN:	4AL17EC091
Topic:	❖ Files & Error Handling ❖ The Preprocessor	Semester & Section:	6th Sem 'B' Sec
Github Repository:	swastik-gowda		

FORENOON SESSION DETAILS

Image of session


sololearn.com/Play/C
SOLOLEARN
Files & Error Handling
XP 213
Working With Files 1/4 3 questions ✓
Binary File I/O 2/4 3 questions ✓
Error Handling 3/4 2 questions ✓
Using Error Codes 4/4 4 questions ✓
Module 7 Quiz 5 questions ✓
Swastik R Gowda
swastikgowda8@gmail.com
Reset | Sign out


sololearn.com/Play/C
SOLOLEARN
#
The Preprocessor
XP 249
Preprocessor Directives 1/3 5 questions ✓
Conditional Compilation Directives 2/3 2 questions ✓
Preprocessor Operators 3/3 2 questions ✓
Module 8 Quiz 5 questions ✓
Swastik R Gowda
swastikgowda8@gmail.com
Reset | Sign out

Report – Report can be typed or hand written for up to two pages.

Files & Error Handling

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.
The following program opens a file for writing and then closes it:

```
#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;
}
```

When a string literal is used to specify a filename, the escape sequence \\ indicates a single backslash. In this program, if there is an error when opening the file, a -1 error code is returned to the system.

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;
}
```

The ***gets()*** function reads up until the newline. ***fscanf()*** reads data according to conversion specifiers. And then the while loop reads one character at a time until the end of file. Checking for a problem when opening the file (a NULL pointer) was left out to shorten the example.

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:

```
#include <stdio.h>
int main()
{
        FILE *fptr;
        char filename[50];
        char c;

        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);

        fclose(fptr);

        /* read the file contents */
        fptr = fopen(filename, "r");
        while ((c = getc(fptr)) != EOF)
         printf("%c", c);
        fclose(fptr);
        return 0;
}
```

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.

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.

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).

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.

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.

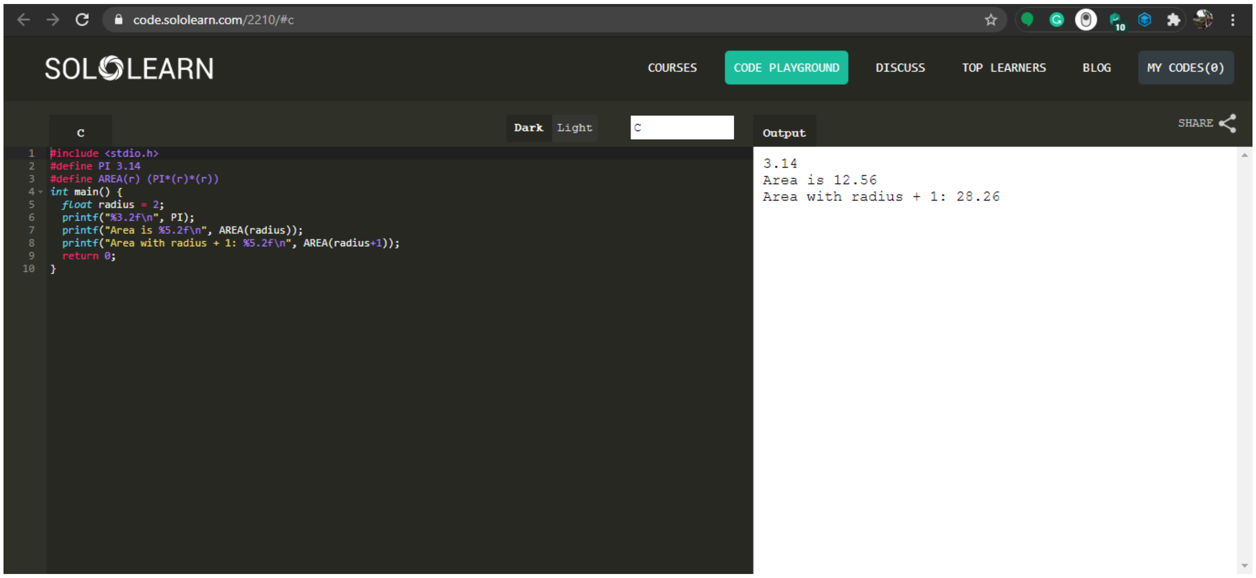
```
#include <stdio.h>
#define PI 3.14
#define AREA(r) (PI*r*r)

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;
}
```

Before compilation, the preprocessor expands every macro identifier. In this case, every occurrence of PI is replaced with 3.14 and AREA(arg) is replaced with the expression PI*arg*arg. The final code sent to the compiler will already have the constant values in place.

Not what we may expect! However, if you consider that #define works strictly by replacing text, you will see that AREA(radius+1) becomes PI*radius+1*radius+1, which is 3.14*2+1*2+1.

The solution to this is to enclose each parameter in parentheses to obtain the correct order of operations.



```c
#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;
}
```

```
3.14
Area is 12.56
Area with radius + 1: 28.26
```

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:

```c
#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;
}
```

Because RATE is defined at the top, only the ***#ifdef*** clause will be compiled. The optional #else branch compiles when ***#ifdef*** RATE is false during preprocessing. An ***#endif*** is required to close the block of code.

Preprocessor Operators

The C preprocessor provides the following operators.

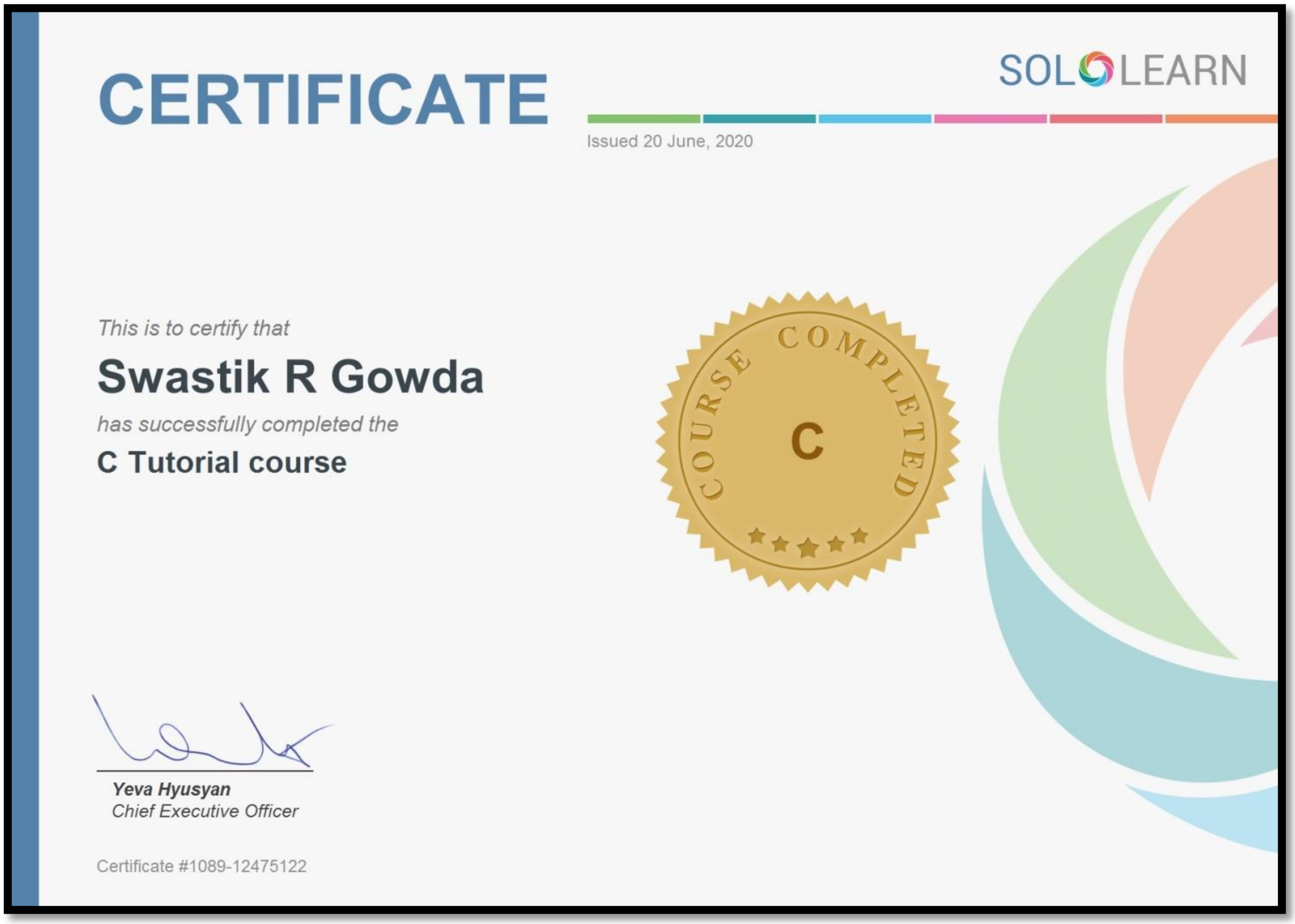
The # Operator

- 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 is ignored and escape sequences are recognized.

The ## Operator

- The ## operator is also called the token pasting operator because it appends, or "pastes", tokens together.

CERTIFICATE



CERTIFICATE

SOLOLEARN

Issued 20 June, 2020

This is to certify that

Swastik R Gowda

has successfully completed the

C Tutorial course

COURSE COMPLETED

C

Yeva Hyusyan
Chief Executive Officer

Certificate #1089-12475122