SDN150S – SOFTWARE DESIGN 1

File Handling, Preprocessors, And Memory Management

DEPARTMENT OF ELECTRICAL ELECTRONIC AND COMPUTER ENGINEERING
CAPE PENINSULA UNIVERSITY OF TECHNOLOGY

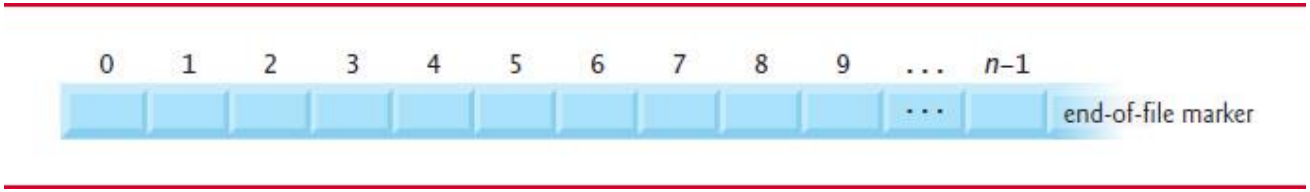
Cape Peninsula University of Technology

OUTLINE

- **File Handling**
- Open Files
- Closing a File
- Writing a File
- Reading a File
- Binary Input/Output Functions
- **Preprocessors**
- Predefined Macro
- Marco Continuation Operator
- Stringize Operator
- Token Pasting Operator
- Defined Operator
- Parameterized Macro
- **Dynamic Memory Management**
- Dynamically Allocate Memory
- Resize and Release Memory
- Recommended Read

FILE HANDLING (1 OF 9)

- Computers store files on secondary storage devices, such as solid-state drives, flash drives and hard drives. These files enable long-term data retention.
- C programming language provides access on high-level and low-level function calls to handle files stored on storage devices.
- C views each file as a sequential stream of bytes, regardless of it being a text file or a binary file.
- Each file ends with an end-of-file (EOF) marker or at a specific byte number recorded in a system-maintained administrative data structure.



FILE HANDLING (2 OF 9)

- When you open a file or execute a program, C associates the file or program with one of three streams automatically:
  - The standard input (stdin) stream that receives input from the keyboard.
  - The standard output (stdout) stream that displays output on the screen.
  - The standard error (stderr) stream that displays error messages on the screen.

Opening Files:

- You can use the **fopen( )** function to create a new file or to open an existing file.
- This function call will initialize an object of the type FILE, which contains all the information necessary to control the stream. The prototype of this function call is given as:

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

FILE HANDLING (3 OF 9)

- Here, filename is a string literal, which you will use to name your file, and access mode can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

FILE HANDLING (4 OF 9)

- If you are going to handle binary files, then you will use the following access modes instead of the above-mentioned ones:

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

Closing a File:

- To close a file, use the **fclose( )** function. The prototype of this function is: `int fclose( FILE *fp );`
- The fclose() function returns zero on success, or EOF if there is an error in closing the file.
- This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file.

FILE HANDLING (5 OF 9)

- The EOF is a constant defined in the header file stdio.h. There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Writing a File:

- Following is the simplest function to write individual characters to a stream: `int fputc( int c, FILE *fp );`
- The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error.
- You can use the following functions to write a null-terminated string to a stream: `int fputs( const char *s, FILE *fp );`

FILE HANDLING (6 OF 9)

- The function fputs() writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise EOF is returned in case of any error.
- You can use `int fprintf(FILE *fp,const char *format, ...)` function as well to write a string into a file.
- Lets try the following example using codeblocks. Before proceeding, make sure you create a directory called **"tmp"** on your machine. Copy the path and replace "XXXXX...".

```
#include <stdio.h>
main(void)
    {FILE *fp;
        fp = fopen("/XXXXX.../tmp/test.txt", "w+");
        fprintf(fp, "This is testing for fprintf...\n");
        fputs("This is testing for fputs...\n", fp);
        fclose(fp);}
```

FILE HANDLING (7 OF 9)

Reading a File:

- Given below is the simplest function to read a single character from a file:

```
int fgetc( FILE * fp );
```

- The fgetc() function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns EOF.
- The following function allows to read a string from a stream:

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

- The functions fgets() reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer buf, appending a null character to terminate the string.

FILE HANDLING (8 OF 9)

- If this function encounters a newline character '\n' or EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character.
- You can also use `int fscanf(FILE *fp, const char *format, ...)` function to read strings from a file, but it stops reading after encountering the first space character.

```
#include <stdio.h>
main(void)
    {FILE *fp; char buff[255];
    fp = fopen("/Users/sid/Documents/CPUT/tmp/test.txt", "r");
    fscanf(fp, "%s", buff);
    printf("1: %s\n", buff);
    fgets(buff, 255, (FILE*)fp);
    printf("2: %s\n", buff);
    fgets(buff, 255, (FILE*)fp);
    printf("3: %s\n", buff);
    fclose(fp);}
```

OUTPUT

```
1: This
2: is testing for fprintf...

3: This is testing for fputs...
```

FILE HANDLING (9 OF 9)

- Analyzing the example, we see that fscanf() only reads **This** because it encountered a space. While fgets() reads the remaining line till it encountered end of line. The same is seen in the last call fgets(), which reads the second line completely.

Binary I/O Functions:

- The two functions shown below are binary input and output functions. They can be used to read or write blocks of memories (usually used on arrays or structures).

```
size_t fread(void *ptr, size_t size_of_elements,
                size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
                size_t number_of_elements, FILE *a_file);
```

PREPROCESSORS (1 OF 10)

- The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required preprocessing before the actual compilation.
- We'll refer to the C Preprocessor as CPP. All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.
- The following section lists down all the important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Opens a text file for both reading and writing.

PREPROCESSORS (2 OF 10)

#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

PREPROCESSORS (3 OF 10)

- Lets analyze the following examples to understand various directives.

Directive Example	Description
`#define MAX_ARRAY_LENGTH 20`	This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use *#define* for constants to increase readability.
`#include <stdio.h>` `#include "myheader.h"`	These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.
`#undef FILE_SIZE` `#define FILE_SIZE 42`	It tells the CPP to undefine existing FILE_SIZE and define it as 42.

`#ifndef MESSAGE`
`  #define MESSAGE "You wish!"`
`#endif`
	It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

`#ifdef DEBUG`
`/* Your debugging statements here */`
`#endif`
	It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the a DEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off as desired during compilation.

PREPROCESSORS (4 OF 10)

Predefined Macros:

- ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format.
__TIME__	The current time as a character literal in "HH:MM:SS" format.
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

PREPROCESSORS (5 OF 10)

- The example below is a good illustration of the use of predefined macros

```
#include <stdio.h>
int main()
    {printf("File :%s\n", __FILE__ );
        printf("Date :%s\n", __DATE__ );
        printf("Time :%s\n", __TIME__ );
        printf("Line :%d\n", __LINE__ );
        printf("ANSI :%d\n", __STDC__ );
    }
```

OUTPUT

```
File :main.c
Date :Jun 20 2022
Time :20:38:48
Line :7
ANSI :1
```

The Macro Continuation (\) Operator:

- A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example:

```
#define message_for(a, b) \
    printf(#a " and " #b ": We love you!\n")
```

PREPROCESSORS (6 OF 10)

The Stringize (#) Operator:

- The stringize or number-sign operator (#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example:

```
#include <stdio.h>
#define message_for(a, b) \
        printf(#a " and " #b ": Congratulations!\n")
int main(void)
    { message_for(Carole, Tatenda);
        return 0;}
```

OUTPUT

```
Carole and Tatenda:
Congratulations!
```

PREPROCESSORS (7 OF 10)

The Token Pasting (##) Operator:

- The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void)
    {int token34 = 40;
        tokenpaster(34);
        return 0;}
```

OUTPUT
token34 = 40

- This example shows the concatenation of token##n into token34 and here we have used both stringize and token-pasting. So that the actual output from the preprocessor is:

```
printf ("token34 = %d", token34);
```

PREPROCESSORS (8 OF 10)

The Defined () Operator:

- The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define.
- If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero).
- The defined operator is specified as follows:

```
#include <stdio.h>
#if !defined (MESSAGE)
    #define MESSAGE "You wish!"
#endif
int main(void)
    {printf("Here is the message: %s\n", MESSAGE);
        return 0;}
```

OUTPUT

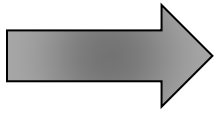
```
Here is the message:
You wish!
```

PREPROCESSORS (9 OF 10)

Parameterized Macros:

- One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros.
- For example, we might have some code to square a number as shown below in red, which can be rewrite as shown in blue using a macro:

```
int square(int x)
{return x * x; }
```



```
#define square(x) ((x) * (x))
```

- Macros with arguments must be defined using the #define directive before they can be used.
- The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis

PREPROCESSORS (10 OF 10)

- For example:

```
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(void)
    {printf("Max between 20 and 10 is %d\n", MAX(10, 20));
        return 0;}
```

OUTPUT

```
Max between 20 and 10 is 20
```

DYNAMIC MEMORY MANAGEMENT (1 OF 5)

- The C programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

S.N	Description
1	**void *calloc(int num, int size);** This function allocates an array of **num** elements each of which size in bytes will be **size.**
2	**void free(void *address);** This function releases a block of memory block specified by address.
3	**void *malloc(int num);** This function allocates an array of **num** bytes and leave them initialized.
4	**void *realloc(void *address, int newsize);** This function re-allocates memory extending it upto **newsize.**

DYNAMIC MEMORY MANAGEMENT (2 OF 5)

Dynamically Allocate Memory:

- While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, a name of any person with a maximum of 100 characters can be define as follows:

```
char name[100];
```

- There are situations where you have no idea about the length of the text you need to store.
- For example, lets assume you want to store a detailed description about a topic and need to define a pointer to character without defining how much memory is required, however, based on requirement, you want to allocate memory dynamically.

DYNAMIC MEMORY MANAGEMENT (3 OF 5)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
    {char name[100]; char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 200 * sizeof(char) );
if( description == NULL )
    {fprintf(stderr, "Error - unable to allocate required memory\n");}
else{strcpy( description, "Zara ali a DPS student in class 10th");}
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );}
```

OUTPUT

```
Name = Zara Ali
Description: Zara ali a
DPS student in class 10th
```

- Same program can be written using calloc(); only thing is you need to replace malloc with calloc as follows: `calloc(200, sizeof(char));`
- So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size is defined, you cannot change it.

DYNAMIC MEMORY MANAGEMENT (4 OF 5)

Resizing and Releasing Memory:

- When your write code/program run comes to an end, operating system automatically release all the memory allocated by the program.
- However, as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().
- Alternatively, you can increase or decrease the size of an allocated memory block by calling the function realloc().
- Using the previous example, we illustrate how to resize and release a memory block using the realloc() and free() functions below:

DYNAMIC MEMORY MANAGEMENT (5 OF 5)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
    {char name[100]; char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 30 * sizeof(char) );
if( description == NULL )
    {fprintf(stderr, "Error - unable to allocate required memory\n");}
else{strcpy( description, "Zara ali a DPS student.");}
    /* suppose you want to store bigger description */
    description = realloc( description, 100 * sizeof(char) );
if( description == NULL )
    {fprintf(stderr, "Error - unable to allocate required memory\n");}
else{strcat( description, "She is in class 10th");}
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
    /* release memory using free() function */
    free(description);}
```

OUTPUT

```
Name = Zara Ali
Description: Zara ali a DPS
student.She is in class 10th
```

RECOMMENDED READ

- Read Up on Data Structures from Chapter 12 of the recommended textbook (pg. 651 – 711).