Chp21.

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement. When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files. **The Standard Files** C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen. **Standard File File Pointer Device** Standard input stdin Keyboard Standard output stdout Screen Standard error stderr Your screen The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen. **getchar() and putchar() Functions** The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen. The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the

screen. Check the following example:

```
#include <stdio.h> int main( ) { int c; printf( "Enter a value :"); c = getchar( ); printf( "\nYou entered: "); putchar( c ); return 0; }
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows: `$./a.out Enter a value : this is test You entered: t` **The gets() and puts() Functions** The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File). The **int puts(const char *s)** function writes the string 's' and 'a' trailing newline to **stdout**.

```
#include <stdio.h> int main( ) { char str[100]; printf( "Enter a value :"); gets( str );
```

```
printf( "\nYou entered: "); puts( str ); return 0; }
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows: `$./a.out Enter a value : this is test You entered: This is test` **The scanf() and printf() Functions** The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided. The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided. The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character, or float, respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better: `#include <stdio.h> int main( ) { char str[100]; int i; printf( "Enter a value :"); scanf("%s %d", str, &i); printf( "\nYou entered: %s %d ", str, i);`

`return 0; }` When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows: `$./a.out Enter a value : seven 7 You entered: seven 7` Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

Chp22.

The last chapter explained the standard input and output devices handled by C programming language. This chapter covers how C programmers can create, open, close text or binary files for their data storage. A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high-level functions as well as low-level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management. **Opening Files** You can use the **fopen( )** function to create a new file or to open an existing file. This 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 as follows: `FILE *fopen( const char * filename, const char * mode );` 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. 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. 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** 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 );` 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. Try the following example. Make sure you have **/tmp** directory available. If it is not, then before proceeding, you must create this directory on your machine. `#include <stdio.h>`

`main() { FILE *fp; fp = fopen("/tmp/test.txt", "w+"); fprintf(fp, "This is testing for fprintf...\n"); fputs("This is testing for fputs...\n", fp); fclose(fp); }` When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section. **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. If this function encounters a newline character '\n' or the end of the file 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() { FILE *fp; char buff[255];`

`fp = fopen("/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); }` When the above code is compiled and executed, it reads the file created in the previous section and produces the following result: `1 : This 2: is testing for fprintf... 3: This is testing for fputs...` Let's see a little more in detail about what happened here. First, **fscanf()** reads just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely. **Binary I/O Functions** There are two functions that can be used for binary input and output: `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);` Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

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 Returns true if this macro is defined. #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

Chp23.

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 Returns true if this macro is defined. #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 Examples** Analyze the following examples to understand various directives. `#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 *-DDEBUG* flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on-the-fly during compilation. **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. Let's try the following example: `#include <stdio.h> 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__ ); }` When the above code in a file **test.c** is compiled and executed, it produces the following result: `File :test.c Date :Jun 2 2012 Time :03:36:24 Line :8 ANSI :1`

Preprocessor Operators The C preprocessor offers the following operators to help create macros: **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")` **Stringize (#)** 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 ": We love you!\n") int main(void) { message_for(Carole, Debra); return 0; }` When the above code is compiled and executed, it produces the following result: `Carole and Debra: We love you!` **Token Pasting (##)** 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; }` When the above code is compiled and executed, it produces the following result: `token34 = 40` It happened so because this example results in the following actual output from the preprocessor: `printf ("token34 = %d", token34);` This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**. **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; }`

When the above code is compiled and executed, it produces the following result: `Here is the message: You wish!` **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 follows: `int square(int x) { return x * x; }` We can rewrite the above code using a macro as follows: `#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. 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; }` When the above code is compiled and executed, it produces the following result: `Max between 20 and 10 is 20`