Practical Workbook
CS-219
Computer Engineering Workshop

Name	: ____________________
Year	: ____________________
Batch	: ____________________
Roll No	: ____________________
Department:	____________________

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

Practical Workbook

CS-219
Computer Engineering Workshop

INTRODUCTION

This workbook has been compiled to assist the conduct of practical classes for CS-219 Computer Engineering Workshop. Practical work relevant to this course aims to guide you through various aspects of C programming and Linux system administration. This workbook is tailored to provide hands-on experience and practical knowledge in mastering C programming fundamentals, exploring advanced topics, and delving into essential Linux system tasks.

The Course Profile of CS-219 Computer Engineering Workshop lays down the following Course Learning Outcome:

Attain hands-on experience with contemporary technologies of Computer Engineering (C3, PLO-5).
All lab sessions of this workbook have been designed to assist the achievement of the above CLO. A rubric to evaluate student performance has been provided at the end of the workbook.

These labs cover a comprehensive range of topics in C programming and Linux system administration. Students will begin with the fundamentals of C, including syntax and data structures, progressing to advanced concepts like pointers, dynamic memory allocation, and file handling. The Linux-focused labs teach essential commands, file and directory management, and shell scripting. Additionally, participants will explore debugging tools, automation with Makefiles, and interactions with the Linux file system. The final labs concentrate on managing Linux users and groups, emphasizing security and access control. Overall, these hands-on exercises provide a well-rounded learning experience for both C programming and Linux system tasks.

CONTENTS

Lab Session 01

Exploring C Programming Fundamentals

C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972. It is a very popular language, despite being old. If you know C, you will have no problem learning other popular programming languages such as Java, Python, C++, C#, etc, as the syntax is similar. It is very fast, compared to other programming languages, like Java and Python.

In this lab, we'll walk through the following Programming Fundamentals using C language:

- Structure of a basic C program
- Constants
- Variables
- Operators
- Methods to display output (Format Specifier, Escape Sequences etc.) and so on.
- Methods to get input from user (scanf( ), getch( ) etc.)
- Conditional statements
- Functions

Dissecting a C program

Let's walk through the syntax of a simple C program:

```
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

- **#include <stdio.h>** is a header file library that lets us work with input and output functions, such as printf(). Header files add functionality to C programs(will be discussed later).
- The main function is the entry point of a program where the execution of a program starts. Any code inside its curly brackets {} will be executed.
- printf() is a function used to output/print text to the screen. In our example it will output "Hello World!".
- Note that every C statement ends with a semicolon ;.
- Since the function's return type is integer, the function is returning an integer value. If the function has a void return type, there is no necessity to provide a return value.

Variables

If the value of an item can be changed in the program then it is a variable. The various variable types (also called *data type*) in C are: *int*, *float*, *char*, *long* etc.

Example 1

```
int myNum = 15;
float myFloatNum = 5.99;
char myLetter = 'D';
```

Data Types

The data type specifies the size and type of information the variable will store. The most basic data types are as follows:

Data Type	**Size**	**Description**
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
char	1 byte	Stores a single character/letter/number, or ASCII values

Constants

If you don't want to change existing variable values, you can use the **const** keyword. This will declare the variable as **constant**, which means unchangeable and read-only:

Example 2

```
const float PI = 3.14;  // PI will always be 3.14
PI = 10;
```

The second line will generate an error due to assignment of read only variable because you have declared PI as a constant. Note that you should always declare a constant when you have values that are unlikely to change.

Operators

There are various types of operators that may be placed in the following categories:

Basic:	+ - * / %
Assignment:	= += -= *= /= %=
	(++, -- may also be considered as assignment operators)
Relational:	< > <= >= == !=
Logical:	&& || !

Example 3

```
 int x = 5;  //Basic
 int y = 3;
 printf("%d", x + y); //Assignment
 printf("%d", x > 3 && y < 10);  //Relational and Logical
```

Format Specifiers

Format specifiers tell the *printf* statement where to put the text and how to display the text.
The various format specifiers are:

```
          %d   =>   integer
          %c   =>   character
          %f   =>   floating point
          %lf  =>       double etc.
```

Example 4

```
int myNum = 15;             // Integer (whole number)
float myFloatNum = 5.99;    // Floating point number
char myLetter = 'D';        // Character

printf("%d\n", myNum);
printf("%f\n", myFloatNum);
printf("%c\n", myLetter);
```

Field Width Specifiers

They are used with **%** to limit precision in floating point number. The number showing limit follows the radix point.

Example 5

```
float myFloatNum = 3.5;

printf("%f\n", myFloatNum); // Default will show 6 digits after the
decimal point
printf("%.1f\n", myFloatNum); // Only show 1 digit
printf("%.2f\n", myFloatNum); // Only show 2 digits
printf("%.4f", myFloatNum);   // Only show 4 digits
```

Escape Sequences

Escape sequence causes the program to **escape** from the normal interpretation of a string, so that the next character is recognized as having a special meaning. The back slash “\” character is called the **Escape Character**. The escape sequence includes the following:

```
          \n   =>   new line
          \b   =>   back space
          \r   =>   carriage return
          \"   =>   double quotations
          \\   =>   back slash     etc.
```

Getting Input From the User

The input from the user can be taken by the following techniques: scanf( ), getch( ), getche( ), getchar( ) etc.

Example 6

```
int myNum;
printf("Type a number: \n");
scanf("%d", &myNum);
printf("Your number is: %d", myNum);
```

Conditional Statements

Normally, your program flows along line by line in the order in which it appears in your source code. But, it is sometimes required to execute a particular portion of code only if certain condition is true; i.e. you have to make decision in your program. The decision making structures that are used in C are as follows:

- If statement
- If-else statement
- Switch case
- Short hand If-else (Ternary operator)

Example 7

```
int time = 22;
if (time < 10) {
  printf("Good morning.");
} else if (time < 20) {
  printf("Good day.");
} else {
  printf("Good evening.");
}
```

Example

```
int num = 8;
        switch (num) {
            case 7:
                printf("Value is 7");
                break;
            case 8:
                printf("Value is 8");
                break;
            case 9:
                printf("Value is 9");
                break;
            default:
                printf("Out of range");
                break;
        }
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- The break statement breaks out of the switch block and stops the execution.
- The default statement is optional, and specifies some code to run if there is no case match.

Example 8

```
int time = 20;
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

In the example above, when the condition is met, the first print statement will be carried out. Alternatively, if the condition is not true, the second print statement will be executed.

The above example is implemented using short-hand if else, which is known as the ternary operator because it consists of three operands. It can be used to replace multiple lines of code with a single line.

Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Example 9

```
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}
```

- myFunction() is the name of the function.
- void means that the function does not have a return value. You can also return value of any required data type.

- Inside the function (the body), add code that defines what the function should do.
- Declared functions are not executed immediately. They are saved for later use, and will be executed when they are called.
- To call a function, write the function's name followed by two parentheses () and a semicolon .

Example 10

```
int myFunction(int x, int y) {
  return x + y;
}

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);
  return 0;
}
```

For code optimization, it is recommended to separate the declaration and the definition of the function. You will often see C programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read.

Example 11

```
//Declaration
int myFunction(int, int);

// The main method
int main() {
  int result = myFunction(5, 3); // call the function
  printf("Result is = %d", result);
  return 0;
}

// Function definition
int myFunction(int x, int y) {
  return x + y;
}
```

Exercises

1. Write a C program that accepts an employee's ID, total worked hours in a month and the amount received per hour. Print the ID and salary (with two decimal places) of the employee for a particular month.
2. Write a C program that takes the height and width of a rectangle as an input from user and compute the perimeter and area of a rectangle.
3. Write a C program to accept the height of a person in centimeters and categorize the person according to his height. (Height < 150cm – Dwarf, Height=150cm – Average, Height>=165cm – Tall).
4. Write a program in C to convert a decimal number to a binary number using functions.
5. Write a function to calculate the nth Fibonacci number and call it recursively to print the Fibonacci series.

Lab Session 02

Exploring Loop, Arrays, and Structures in C Programming

In this lab, we'll walk through the following concepts in C:

- Loop
- Loop Control Statements
- Arrays
- Strings
- String Functions
- Structure

Loop

A loop statement allows us to execute a statement or group of statements multiple times. C programming language provides the following types of loops to handle looping requirements.

- For loop
- While loop
- Do…While loop

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. C supports the following control statements:

- **break** - Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

- **continue** - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Example 1

```
for(int  a = 10; a < 20; a++ ){
      printf("value of a: %d\n", a);
   }
```

In the above example, the initialization statement is executed only once. Then, the test expression is evaluated. If the test expression is evaluated to false, the loop is terminated.
However, if the test expression is evaluated to true, statement inside the body of the loop is executed. After that the increment statement is executed, and then again the test expression is evaluated.
This process goes on until the test expression is false. When the test expression is false, the loop terminates.

Example 2

```
int a = 10;

   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }
```

In the above example, we can see that a conditional expression is used. The statements defined inside the while loop will repeatedly execute until the given condition fails.

Example 3

```
int a = 10;

   do {
      printf("value of a: %d\n", a);
      a = a + 1;
   }while( a < 20 );
```

In the above example, we can see that the code inside the do block will execute atleast once even if the condition is false.

Example 4

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  printf("%d\n", i);
}
```

In the above example, when the value of i is set to 4, break statement terminates the loop.

Example 5

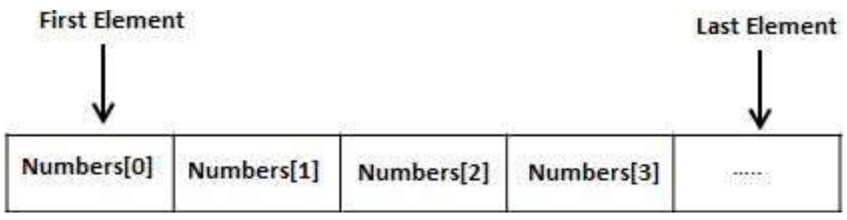
```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  printf("%d\n", i);
}
```

In the above example, when i is set to 4, continue causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Arrays

Arrays can store a fixed-size sequential collection of elements of the same type. Instead of declaring individual variables, such as num0, num1, ..., and num99, one array variable can be declared such as num, and num[0], num[1], and ..., num[99] can be used to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows :

```
type arrayName [ arraySize ];
```

This is called a uni dimensional array. The **arraySize** must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 5-element array called **marks** of type **double**, the following statement will be used:

```
double marks[5];
```

Here **marks** is a variable array which is sufficient to hold up to 5 double numbers.

Initializing Arrays

An array in C can be initialized either one by one or using a single statement as follows:

```
double marks[5] = {2.5, 5.3, 6.5, 7.5, 8.5};
```

Note that the number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If the size of the array is not mentioned, an array just big enough to hold the initialization is created. Therefor

```
double marks[] = {2.5, 5.3, 6.5, 7.5, 8.5};
```

Change an array element

Following is an example to modify a single element of the array:

```
marks [4] = 9.5;
```

The above statement assigns the 5th element in the array with a value of 9.5. Note that all arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1.

Accessing an array element

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array:

```
double result = marks[3];
```

The above statement will take the 4th element from the array and assign the value to result variable.

Example

Let's loop through an array:

```
int myNumbers[] = {25, 50, 75, 100};

for (int i = 0; i < 4; i++) {
  printf("%d\n", myNumbers[i]);
}
```

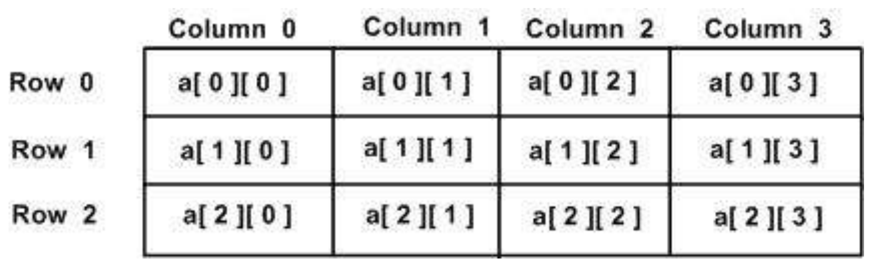
Multi-Dimensional Arrays

Previously, we looked at one dimensional arrays. However, if you want to store data in a tabular form, like a table with rows and columns, multidimensional arrays are needed. A multidimensional array is basically an array of arrays.

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Following is a 2D- array with three rows and four columns:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Initializing Two-Dimensional Arrays

Let’s create a 2D-array with two rows and three columns:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

The first dimension represents the number of rows [2], while the second dimension represents the number of columns [3]

Change Elements in a 2D Array

The following statement will change the value of the element in the first row and first column:

```
matrix[0][0] = 9;
```

Access the Elements of a 2D Array

The following statement accesses the value of the element in the first row and third column of the matrix array.

printf("%d", matrix[0][2]);

Example 6

Let’s loop through an array:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 3; j++) {
    printf("%d\n", matrix[i][j]);
  }
}
```

Strings

Strings are used for storing text/characters. Unlike many other programming languages, C does not have a string type to easily create string variables. Instead, you must use the char type and create an array of characters to make a string in C:

```
char greetings[] = "hello world";
```

Note that you can create a string with a set of characters. This example will produce the same result as the example above:

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l',
'd', '!', '\0'};
printf("%s", greetings);
```

\0 is known as the null terminating character, and must be included when creating strings using this method. It tells C that this is the end of the string.

Access Strings

A string can be accessed by referring to its index number inside square brackets []:

```
char greetings[] = "Hello World!";
printf("%c", greetings[0]);
```

The above example prints the first character in greetings.

Modify Strings

To change the value of a specific character in a string, refer to the index number, and use single quotes:

```
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
```

Example 7

Let's loop through a string:

```
char carName[] = "Volvo";
int i;

for (i = 0; i < 5; ++i) {
  printf("%c\n", carName[i]);
}
```

String Functions

C also has many useful string functions, which can be used to perform certain operations on strings. To use them, you must include the <string.h> header file in your program:

```
#include <string.h>
```

Let's look at the functions briefly:

- **strlen(str1)** – Returns length of the string str1.
- **strcat(str1, str2)** – Concatenates str2 to str1 (result is stored in str1).
- **strcpy(str2, str1)** – Copies str1 to str2.
- **strcmp(str1, str2)** – Compares str1 and str2, and prints the result.

C Structures

Arrays allow to define type of variables that can hold several data items of the same kind. On the contrary, structure is another user defined data type available in C that allows to combine data items of **different** kinds.To define a structure, you must use the **struct** keyword. The C structures define a new data type, with more than one member. The format of the structure is defined in the following example:

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
```

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type. The following example shows how to use a structure in a program:

Example 8

```
#include <stdio.h>
#include <string.h>
 struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
 int main( ) {
   struct Books myBook;        /* Declare myBook of type Book */
      /* myBook specification */
   strcpy(myBook.title, "Operating Systems: Internals and Design 
Principles");
   strcpy(myBook.author, "William Stallings");
   strcpy(myBook.subject, " Operating Systems");
   myBook.book_id =  1
   /* print myBook  info */
   printf( "Title : %s\n", myBook.title);
   printf( "Author : %s\n", myBook.author);
   printf( "Subject : %s\n", myBook.subject);
   printf( "Book ID : %d\n", myBook.book_id);
   return 0; }
```

Exercises

1. Write a C program to display the first n odd natural numbers and their sum using for, while and do-while loop.
2. Write a C program to make the following pattern as a pyramid with an asterisk.

```
   *
  * *
 * * *
* * * *
```

3. Write a C program to compare two strings without using string library functions.
4. Write a C program to read a sentence and replace lowercase characters with uppercase and vice versa.
5. Write a C program to print all unique elements in an array.
6. Write a C program to add two distances in inch-feet system using structures.

Lab Session 03

Exploring Pointers in C

When a variable is created in C, a memory address is assigned to the variable. The memory address is the location of where the variable is stored on the computer. When we assign a value to the variable, it is stored in this memory address. To access it, we can use the reference operator (&), and the result represents where the variable is stored:

```
int myNum = 43;
printf("%p", &myNum);
```

Note that the memory address is in hexadecimal form. This takes us to the concept of pointers.

Pointers

A pointer is a variable that stores the memory address of another variable as its value. Note that pointers are one of the things that make C stand out from other programming languages, like Python and Java.

A pointer variable points to a data type (like int) of the same type, and is created with the ***** operator. The address of the variable you are working with is assigned to the pointer. Let's have a look at an example:

Example 1

```
int myNum = 43;
int* ptr = &myNum;
printf("%d\n", myNum);
printf("%p\n", &myNum);
printf("%p\n", ptr);
```

In the above example, a pointer variable with the name ptr is created, that points to an int variable. Note that the type of the pointer has to match the type of the variable you're working with (int in our example). Use the **&** operator to store the memory address of the myNum variable, and assign it to the pointer. Now, ptr holds the value of myNum's memory address. Note that the memory location is going to be different for everyone. The output for the above example will be as shown below:

```
43
000000000022FE44
000000000022FE44
```

You can also get the value of the variable the pointer points to, by using the * operator (the dereference operator):

```
printf("%d\n", *ptr); //outputs the value of myNum
```

Null pointer

It is always good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Let's have a look at an example:

Example 2

```
#include <stdio.h>

int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %p\n", ptr  );
   return 0;
}
```

When the above code is compiled and executed, it says that the value of ptr is 0. In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, do the following:

```
if(ptr)      // succeeds if p is not null
if(!ptr)     //succeeds if p is null
```

Pointers and Arrays

The name of an array is actually a pointer to the first element of the array. Let's understand this through an example:

Example 3

```
int myNum[4] = {25, 50, 75, 100};
printf("%p\n", myNum);
printf("%p\n", & myNum [0]);
```

Both the print statements will give the same output. This basically means that we can work with arrays through pointers. Since myNum is a pointer to the first element in myNum, you can use the * operator to access it:

```
printf("%d", *myNum);
```

To access the rest of the elements in myNum, you can increment the pointer/array (+1, +2, etc):

```
printf("%d", *( myNum + 1)); //access first element
printf("%d", *( myNum + 2));  //access third element
```

We can also loop through the array using pointers:

```
int *ptr = myNum;
for (int i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

The elements of an array can also be modified:

```
*myNum = 13; //changes the first value
*(myNum +1) = 17; //changes the second value
```

Pointer arithmetic

There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer :

```
ptr++
```

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. Similarly, if ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Pointer Comparisons
Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

Example 4
```
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have address of the first element in pointer */
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] ) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      /* point to the next location */
      ptr++;
      i++;
   }
   return 0;
}
```

Array of pointers

There may be a situation when we want to maintain an array, which can store pointers to variables. Following is the declaration of an array of pointers of type integer.

```
int *ptr[MAX];
```

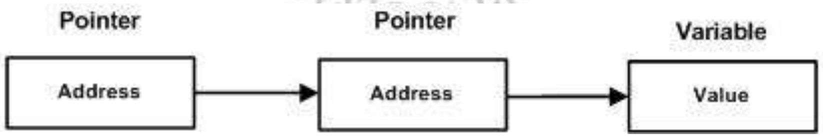
This declares an array of pointers containing MAX elements. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers.

Example 5

```
const int MAX = 3;
 int main () {
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
   for ( i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
   for ( i = 0; i < MAX; i++) {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
   return 0;
}
```

Pointer to pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int :

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

Example 6

```
int main () {
   int  var;
   int  *ptr;
   int  **pptr;
   var = 3000;
   /* take the address of var */
   ptr = &var;
   /* take the address of ptr using address of operator & */
   pptr = &ptr;
   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

Passing pointers to functions in C

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.
Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

Example 7

```
void getSeconds(unsigned long *par);
int main () {
   unsigned long sec;
   getSeconds( &sec );
   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );

   return 0;
}
void getSeconds(unsigned long *par) {
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

A function can also accept an array as a pointer as shown in the following example:

Example 8

```
/* function declaration */
double getAverage(int *arr, int size);
int main () {
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;
   /* output the returned value  */
   printf("Average value is: %f\n", avg );
   return 0;
}
double getAverage(int *arr, int size) {
   int  i, sum = 0;
   double avg;
   for (i = 0; i < size; i++) {
      sum += arr[i];
   }
   avg = (double)sum / size;
   return avg;
}
```

Return pointer from functions in C

C also allows returning a pointer from a function as shown below:

Example 9

```
/* function to generate and return random numbers. */
int * getRandom( ) {
   static int  r[10];
   int i;
   /* set the seed */
   srand( (unsigned)time( NULL ) );
   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf("%d\n", r[i] );
   }
   return r;
}
int main () {
   /* a pointer to an int */
   int *p;
   int i;
   p = getRandom();
   for ( i = 0; i < 10; i++ ) {
      printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }
   return 0;
}
```

In the above example, it can be seen that the getRandom() function generates 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element. Note that it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as static variable.

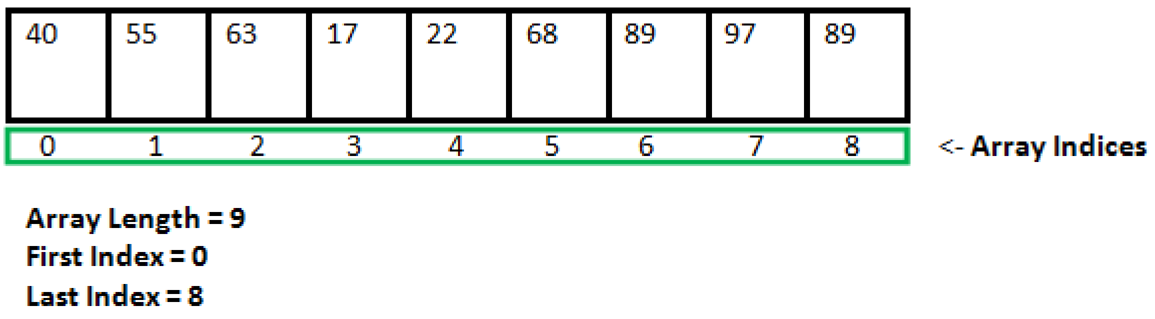
Exercises

1. Write a program in C to swap elements using call by reference.
2. Write a program in C to print a string in reverse using pointers.
3. Write a C program to input and print array elements using pointers.
4. Write a C program to search for an element in an array using pointers.
5. Write a C program to add two matrices using pointers.

Lab Session 04

Exploring Dynamic Memory Allocation and Linked list in C

In the following figure, the size of the array is 9. But what if there is a requirement to change this size? We know that an array is a collection of items stored at contiguous memory locations. Now how this requirement can be met?



This requirement can be fulfilled through Dynamic Memory Allocation in C. It can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming:

- malloc()
- calloc()
- free()
- realloc()

C malloc() method

The **malloc** or **memory allocatio**n method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. Note that it doesn't initialize memory at compile time. The syntax of malloc is as follows:

```
ptr = (cast-type*) malloc(byte-size)
```

Let's understand this through an example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer will ptr hold the address of the first byte in the allocated memory. Note that if the space is insufficient, allocation fails and returns a NULL pointer.

Example 1

```
int main()
```

```
{
    int* ptr;
    int n, i;

    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        return 0;
    }
    else {
         printf("Memory successfully allocated using malloc.\n");
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

C calloc() method

Calloc or **contiguous allocation** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It is very much similar to malloc but has two differences that include initializing each block with a default value 0, and it has two arguments as compared to malloc. The syntax of calloc is as follows:

```
ptr = (cast-type*)calloc(n, element-size);
```

Let's understand this through an example:

```
ptr = (int*) calloc(5, sizeof(int));
```

This statement allocates contiguous space in memory for 5 elements each with the size of the int. Note that if space is insufficient, allocation fails and returns a NULL pointer.

The malloc example can be repeated with calloc. The only difference is that initially the values will be set to 0 and n is provided as the first argument.

Resizing and Releasing Memory

free method is used to dynamically de-allocate the memory. Memory allocated using the malloc and calloc functions is not automatically deallocated. This method helps to reduce wastage of memory by freeing it.

```
free(ptr);
```

realloc or **re-allocation** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. Re-allocation of memory maintains the already present value and new blocks are initialized with the default garbage value.

```
ptr = realloc(ptr, newSize);
```

Let’s have a look at an example to understand both the concepts:

Example 2

```
int main() {
   char *description;
   description = malloc( 30 * sizeof(char) );

   if( description == NULL ) {
     printf("Error - unable to allocate required memory\n");
   } else {
      strcpy( description, "Zara is a CIS student ");
   }

   /* suppose you want to store bigger description */
   description = realloc( description, 100 * sizeof(char) );

   if( description == NULL ) {
        printf("Error - unable to allocate required memory\n");
        } else {
      strcat( description, "She is in second semester");
   }
   printf("Description: %s\n", description );
   /* release memory using free() function */
   free(description);}
```

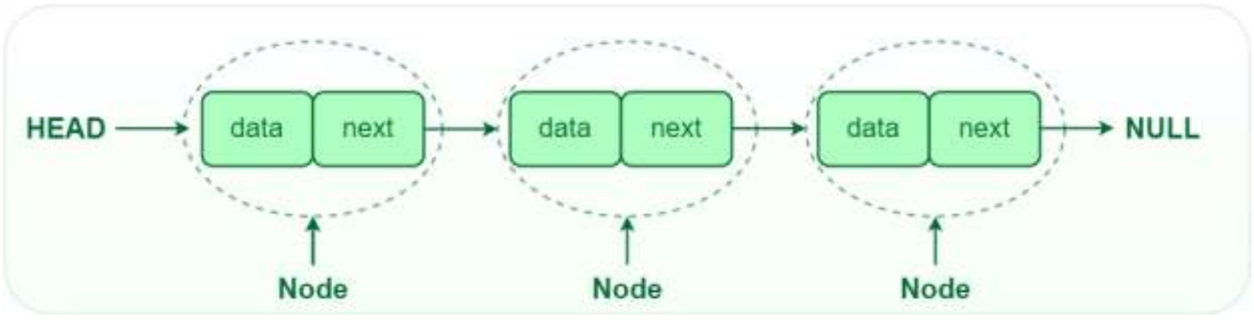
Linked List

Linked lists are the best and simplest example of a dynamic data structure that uses pointers for its implementation. Essentially, linked lists function as an array that can grow and shrink as needed, from any point in the array. Linked lists have a few advantages over arrays:

- Items can be added or removed from the middle of the list
- There is no need to define an initial size

However there are disadvantages as well. One of the disadvantages of linked list is that there is no **random** access - it is impossible to reach the nth item in the array without first iterating over all items up until that item.

Linked List is a linear data structure, in which elements are not stored at a contiguous location; rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.



- **Node** - A node in a linked list typically consists of two components: Data and next pointer.
- **Data** - It holds the actual value or data associated with the node.
- **Next Pointer** - It stores the memory address (reference) of the next node in the sequence.
- **Head and Tail** - The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Operations on Linked Lists

- **Insertion**: Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list
- **Deletion**: Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
- **Searching**: Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

Linked List implementation

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

// Define a structure for a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

```c
// Function to insert a node at the beginning of the linked list
struct Node* insertAtBeginning(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = head;
    return newNode;
}

// Function to insert a node at the end of the linked list
struct Node* insertAtEnd(struct Node* head, int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        return newNode;
    }
    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
    return head;
}

// Function to insert a node after a specific node
struct Node* insertAfter(struct Node* head, int data, int searchValue)
{
    struct Node* newNode = createNode(data);
    struct Node* current = head;
    while (current != NULL && current->data != searchValue) {
        current = current->next;
    }
    if (current == NULL) {
        printf("Node with search value not found\n");
        free(newNode); // Free the allocated node
        return head;
    }
    newNode->next = current->next;
    current->next = newNode;
    return head;
}

// Function to delete a node with a specific value
struct Node* deleteNode(struct Node* head, int data) {
    struct Node* current = head;
    struct Node* prev = NULL;
    while (current != NULL && current->data != data) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Node with value not found\n");
        return head;
    }
    if (prev == NULL) {
        head = current->next;
    } else {
        prev->next = current->next;
    }
    free(current);
    return head;
}

// Function to search for a node with a specific value
struct Node* searchNode(struct Node* head, int data) {
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == data) {
            return current;
        }
        current = current->next;
    }
    return NULL; // Node not found
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

// Function to free the memory used by the linked list
void freeList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning
    head = insertAtBeginning(head, 3);
    head = insertAtBeginning(head, 2);
    head = insertAtBeginning(head, 1);

    // Insert nodes at the end
    head = insertAtEnd(head, 4);
    head = insertAtEnd(head, 5);

    // Insert a node after a specific value
    head = insertAfter(head, 6, 3);

    // Print the linked list
    printf("Linked List: ");
    printList(head);

    // Search for a node
    int searchValue = 4;
    struct Node* foundNode = searchNode(head, searchValue);
    if (foundNode != NULL) {
        printf("Node with value %d found\n", searchValue);
    } else {
        printf("Node with value %d not found\n", searchValue);
    }

    // Delete a node
    int deleteValue = 2;
    head = deleteNode(head, deleteValue);

    // Print the linked list after deletion
    printf("Linked List after deletion: ");
    printList(head);

    // Free the memory
    freeList(head);

    return 0;
}
```

The above implementation of linked list includes functions to insert nodes at the beginning, end, and after a specific value, search for nodes, and delete nodes with a specific value. It also provides functions to print the linked list and free the allocated memory when the program is done with the list.

Exercises

1. Write a program that simulates a simple address book. Define a structure to store contact information (name, email, phone number). Allow the user to add new contacts to the address book dynamically. Use dynamic memory allocation for storing the contacts using malloc and update the memory allocation using realloc when adding new contacts. Implement an option to delete a contact and free the memory. Ensure that memory is properly managed throughout the program's execution.
2. Write a C program to merge two sorted singly linked lists into a single sorted linked list.
3. Write a C program that converts a singly linked list into an array and returns it.
4. Write a C program that removes elements with odd indices from a singly linked list.

Lab Session 05

Exploring File Handling in C

A File is a collection of data stored in the secondary memory. In order to use files, we have to learn file input and output operations. For file handling in C, the following four essential actions should be carried out.

- Declaring a file pointer variable.
- Creating or opening a file.
- Processing the file using suitable functions.
- Closing the file.

Create a file

In C, FILE is basically a data type, and we need to create a pointer variable to work with it as shown below:

```
FILE *fptr;
fptr = fopen(filename, mode);
```

The fopen() function takes the following two parameters:

- **Filename** - The name of the file you want to open or create.
- **Mode** - A single character, which represents what you want to do with the file (read, write or append):
    - w - Writes to a file.
    - a - Appends new data to a file.
    - r - Reads from a file.

To create a file, you can use the **w** mode inside the fopen() function. The **w** mode is used to write to a file. However, if the file does not exist, it will create one for you. Note that the file is created in the same directory as your other C files, if nothing else is specified. If you want to create the file in a specific folder, just provide an absolute path. Let's look an example for creating a file:

Example 1

```
FILE *fptr;
fptr = fopen("lab6.txt", "w");
fclose(fptr);
```

In the above example, fclose() is used to close the file when you are done with it. After successful file operations, you must always close a file to remove it from the memory. Note that fclose() returns 0 on success and -1on failure.

Handling Null Pointer Exception

If you try to open a file for reading that does not exist, the fopen() function will return NULL. It's a good practice to handle the null pointer exception in the code as shown below:

```
if(fptr == NULL) {
  printf("Unable to open the file");
}
```

Working with text files in C

C provides various functions for working with text files. The functions that can be used to read from the text files are following:

1- **`int fscanf(FILE *stream, const char *format,list);`**

The fscanf() function reads data from the current position of the specified stream into the locations that are given by the entries in argument-list, if any. Each entry in argument-list must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format-string controls the interpretation of the input fields and has the same form and function as the format-string argument for the scanf() function. The fscanf() function returns the number of fields that it successfully converted and assigned. The return value is EOF(End Of File) if an input failure occurs before any conversion, or the number of input items assigned if successful. The following snippet explains the function:

```
fscanf(stream, "%ld", &l);
```

In the above code, fscanf() reads data from the stream into the long variable provided in the argument list with the format specified.

2- **`int getc(FILE *stream);`**

The getc() function reads a single character from the current stream position and advances the stream position to the next character. The getc() function returns the character read. A return value of EOF indicates an error or end-of-file condition. The following snippet explains the function:

```
while( (c=getc(stream))!=EOF )
{
      printf("%c", ch );
}
```

In the above code, getc() reads each character from the stream into the character variable until it reaches end of file.

3- **`char* fgets(char *str,int n,FILE *stream);`**

The fgets() function reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the EOF is reached, whichever comes first. On success, the function returns the same str parameter. If the EOF is encountered and no characters have been read, the contents of str remains unchanged and a null pointer is returned. Similarly, if an error occurs, a null pointer is returned. The following block of code explains the function:

```
if( fgets (str, sizeOfStr, stream)!=NULL ) {
      printf("%s", str);
   }
```

In the above example, fgets() function reads a complete string from the stream into str.

4- **`size_t fread(void *buffer, size_t size, size_t count, FILE *stream);`**

The fread() function reads up to count items of size length from the input stream and stores them in the given buffer. The position in the file increases by the number of bytes read. The fread() function returns the number of items successfully read, which can be less than count if an error occurs, or if the end-of-file is met before reaching count. If size or count is 0, the fread() function returns zero, and the contents of the array and the state of the stream remains unchanged. The following code explains the function:

```
num = fread( buffer, sizeof( char ), count_items, stream );
 if ( num ) {
      printf( "buffer = %s\n", buffer );
          }
```

In the above example, fread() function reads number of characters equal to the number stored in count_items from the stream into the buffer.

The functions that can be used to write to the text files are following:

1- **`int fprintf(FILE *stream, const char *format,list);`**

The fprintf() function formats and writes output to a stream. It converts each entry in the argument list, if any, and writes to the stream according to the corresponding format specification in the format-string. If successful, fprintf() returns the number of characters output. Note that the ending NULL character is not counted. If unsuccessful, it returns a negative value. The following code explains the function:

```
   fprintf(stream, "%s %s %s %d", "We", "are", "in", 2023);
```

In the above code, we can see that fprintf() formats and writes the data to the stream.

2- **`int fputc(int c, FILE *stream);`**

The fputc() function converts c to an unsigned char and then writes c to the output stream at the current position and advances the file position appropriately. The fputc() function returns the character that is written. A return value of EOF indicates an error. The following code explains the function:

```
for ( i = 0; ( i < sizeof(buffer) ) &&
           ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
```

In the above code, we can see that fputc() writes the character from the buffer to the output stream.

3- **`int fputs(const char *str,FILE *stream);`**

The fputs() function copies string to the output stream at the current position. It does not copy the null character (\0) at the end of the string. The fputs() function returns EOF if an error occurs; otherwise, it returns a non-negative value.

```
if ( (num = fputs( buffer, stream )) != EOF ){
     }
```

In the above code, we can see that fputs() function copies the data from the buffer to the output stream.

4- **`size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);`**

The fwrite() function writes up to count items, each of size bytes in length, from buffer to the output stream. The fwrite() function returns the number of items successfully written, which can be fewer than count if an error occurs. The following code explains the function.

```
fwrite(list, sizeof(long), count_items, stream);
```

In the above code, fwrite() writes from the list to the stream with the total number of elements equal to count_items each of size long.

Next, we'll implement some examples to write, read or append to afile.

Write to a file

Here we'll use the **w** mode to write something to the file that we just created. Let's have a look at the example:

Example 2

```
FILE *fptr;
fptr = fopen("lab6.txt", "w");
if(fptr!=NULL){
fprintf(fptr, "Greetings = %s\n","Hello World");}
else{
printf("Unable to open the file");
}
fclose(fptr);
```

In the above example, the **w** mode means that the file is opened for writing. To insert content to it, fprintf() function is used in which the file pointer and a sample text is provided as an input. fprintf() allows for formatting of the output using a format string, which can include placeholders for variables of different types. Note that if you write to a file that already exists, the old contents are deleted, and the new contents are inserted. This is important to know, as you might accidentally erase existing content.
Now let's have a look at an example that writes a string into a text file using fputs() function.

Example 3

```
FILE *fptr;
char myString[80];
int i;
fptr=fopen("lab6.txt","w");
printf("Enter the text : ");
gets(myString);
if(fptr!=NULL){
   fputs(myString,fptr);
    printf("The string has been copied to the file");}
else{
    printf("Unable to open the file");}
fclose(fp);
```

In the above example, we can see that fputs() simply write the string as it is whereas fprintf() writes the formatted data to the file.

Append to a File

If you want to add content to a file without erasing the old content, you can use the append mode as shown below:

Example 4

```
FILE *fptr;
fptr = fopen("lab6.txt", "a");
if(fptr!=NULL){
 fprintf(fptr, "\nGood Morning!");}
else{
printf("Unable to open the file");}
fclose(fptr);
```

In the above example, the text mentioned is appended to the previously created file.

Read a file

To read a file in C, you'll use the **r** mode. Reading a file is a bit tricky in C. First, let's have a look at the example below and then we'll dive deep into the implementation.

Example 5

```
FILE *fptr;
fptr = fopen("lab6.txt", "r");
char myString[100];
fgets(myString, 100, fptr);
printf("%s", myString);
fclose(fptr);
```

In the above example, first we have declared a file pointer. The mode **r** means that the file is opened for reading. Next, a character array is declared in order to store the contents of the file so that they can be read later. Here you can see that fgets() function is used to store the file contents inside myString. The first parameter of the fgets() function is the character array, the second parameter specifies the maximum size of data to read whereas the third parameter is the file pointer. Note that the fgets() function only reads the first line of the file. If there are multiple lines in the file, we can use loop as shown below:

```
while(fgets(myString, 100, fptr)) {
  printf("%s", myString);
}
```

Now let's have a look at another example in which we'll read the contents of a file using getc() function.

Example 6

```
FILE *fptr;
```

```
int ch;
fptr=fopen("lab6.txt","r");
if(fptr!=NULL){
while( (ch=getc(fptr))!=EOF )
{
      printf("%c", ch );}
}else{
    printf("Unable to open the file");}
fclose(fptr);
```

In the above example, getc() function is used to read a single character from the given file stream. If some error occurs or the End-Of-File is reached, the function returns EOF (End of File).
Let's have a look at another example in which we'll copy the contents from one file to another.

Example 7

```
FILE *fptr,*fptrCopy;
int ch;
fptr=fopen("lab6.txt","r");
fptrCopy=fopen("lab6copy.txt","w");
if(fptr!=NULL){
while( (ch=getc(fptr))!=EOF )
{
  putc(ch,fptrCopy);}
}else{
    printf("Unable to open the file");}
fclose(fptr);
fclose(fptrCopy);
```

In the above example, getc() along with putc() function is used to first read the contents of lab6.txt and then copy it to lab6copy.txt.

Change File Position

The following functions can be used to change the current file position associated with a stream:

1- **`int fseek(FILE *stream, long int offset, int origin);`**

The fseek() function sets the file position of the stream to the given offset. The origin must be one of the following constants defined in stdio.h:
SEEK_SET – Indicates beginning of file
SEEK_CUR – Indicates the current position of file pointer
SEEK_END – Indicates the end of file
This function returns zero if successful, or else it returns a non-zero value. Now let's have a look at an example that explains the concept:

Example 8

```
FILE *stream;
stream = fopen("lab6.txt","w");
```

```
fputs("This is a test", stream);
fseek(stream, 7, SEEK_SET );
fputs(" test for lab6 fseek function", stream);
fclose(stream);
```

In the above example, we can see that first we write to the stream using fputs(), then with the help of fseek() we changed the position of the file so that we can update the text that we wrote earlier. The output of the above example will be as shown below:

```
This is a test for lab6 fseek function
```

2- void rewind(FILE *stream);

The rewind() function repositions the file pointer associated with stream to the beginning of the file. Let’s look at an example to understand rewind() function.

Example 9

```
FILE *stream;
int writeData = 1;
int readData;
stream = fopen(“lab6.txt”, "w+");
fprintf(stream, “%d", writeData);
rewind(stream);
fscanf(stream, "%d", & readData);
printf("The value from the file is : %d\n", readData);
```

In the above example, fprintf() writes data then rewind() repositions the file pointer so that the data can be read from the file.

Exercises

1. Write a program to create a new text file and write some text into it. Open the file in append mode and add more text to it. Read the contents of the file and display them on the console.

2. Create a program that reads a text file and counts the number of words in it. Display the total number of words at the end.

3. Implement a student database system using a file. Include features to add, delete, modify, and display student records.

Lab Session 06

Exploring Header Files in C

A header file is a file containing C declarations and macro definitions to be shared between several source files. You request the use of a header file in your program by including it, with the C preprocessing directive '#include'. Header files serve two purposes.

- System header files in C are files that contain declarations, constants, and macros that define interfaces to various parts of the operating system and system libraries
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program. In C, the usual convention is to give header files names that end with .h.

Include Syntax

Both user and system header files are included using the preprocessing directive #include. It has two variants:

#include <file>

This variant is used for system header files. It searches for a file named file in a standard list of system directories.

#include "file"

This variant is used for header files of your own program. It searches for a file named file first in the directory containing the current file, then in the quote directories and then the same directories used for <file>. You can prepend directories to the list of quote directories with the -iquote option.

Include Operation

The #include directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the #include directive. For example, if you have a header file header.h as follows:

```
char* greetings(void){
return "hello";
}
```

And a main program called main.c that uses the header file, like this:

```
#include "header.h"
int main ()
{
printf("%s",greetings());
}
```

The compiler will see the same token stream as it would if main.c read as follows:

```
char* greetings(void){
return "hello";
}
int main (){
printf("%s",greetings());
}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error, e.g. when the compiler sees the same structure definition twice. Even if it does not, it will certainly waste time. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef MYHEADER_H
#define MYHEADER_H
the entire file
#endif
```

This construct is commonly known as a wrapper #ifndef. When the header is included again, the conditional will be false, because MYHEADER_H is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.
CPP optimizes even further. It remembers when a header file has a wrapper #ifndef. If a subsequent #include specifies that header, and the macro in the #ifndef is still defined, it does not bother to rescan the file at all. You can put comments outside the wrapper. They will not interfere with this optimization. The macro MYHEADER_H is called the controlling macro or guard macro.
In a user header file, the macro name should not begin with '_'. In a system header file, it should begin with '-' to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

Computed Includes

Sometimes it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could do this with a series of conditionals,

```
#if SYSTEM_1
# include "system_1.h"
```

```
#elif SYSTEM_2
# include "system_2.h"
#elif SYSTEM_3
…
#endif
```

That rapidly becomes tedious. Instead, the preprocessor offers the ability to use a macro for the header name. This is called a computed include. Instead of writing a header name as the direct argument of '#include', you simply put a macro name there instead:

```
#define SYSTEM_H "system_1.h"
…
#include SYSTEM_H
```

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the '#include' had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.

Example 1

Here's an example of creating a header file for a simple calculator with functions for addition, subtraction, multiplication, and division. We'll also provide a source file that implements these functions and a main program that allows the user to perform arithmetic operations.

calculator.h (Header file):

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
// Function declarations
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
double divide(int a, int b);
#endif
```

calculator.c (Source File):

```
#include "calculator.h"

int add(int a, int b) {
    return a + b;
}

int subtract(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}
```

```
double divide(int a, int b) {
    if (b == 0) {
        // Handle division by zero
        return 0.0;
    }
    return (double)a / b;
}
```

main.c (Main Program):

```
#include <stdio.h>
#include "calculator.h"

int main() {
    int num1, num2;
    char operator;

    printf("Enter an arithmetic expression (e.g., 5 + 3): ");
    scanf("%d %c %d", &num1, &operator, &num2);

    switch (operator) {
        case '+':
            printf("Result: %d\n", add(num1, num2));
            break;
        case '-':
            printf("Result: %d\n", subtract(num1, num2));
            break;
        case '*':
            printf("Result: %d\n", multiply(num1, num2));
            break;
        case '/':
            if (num2 != 0) {
                printf("Result: %lf\n", divide(num1, num2));
            } else {
                printf("Error: Division by zero is not allowed.\n");
            }
            break;
        default:
            printf("Error: Invalid operator\n");
    }

    return 0;
}
```

Compile the program using the following command:

```
gcc calculator.c main.c -o calculator_program
```

Then run the program:

```
./calculator_program
```

This program will prompt the user to enter an arithmetic expression, perform the corresponding operation using the functions declared in the calculator.h header file, and display the result. It includes error handling for division by zero and invalid operators.

Example 2

Here's an example of creating a small string manipulation utility library, along with a main program that demonstrates how to use it. This library will provide functions for common string operations, such as string concatenation, string length, and string comparison.

stringutils.h (Header File):

```
#ifndef STRINGUTILS_H
#define STRINGUTILS_H

// Function declarations
int stringLength(const char* str);
void stringConcat(char* destination, const char* source);
int stringCompare(const char* str1, const char* str2);

#endif
```

stringutils.c (Source File):

```
#include "stringutils.h"
#include <string.h>

int stringLength(const char* str) {
    return strlen(str);
}

void stringConcat(char* destination, const char* source) {
    strcat(destination, source);
}

int stringCompare(const char* str1, const char* str2) {
    return strcmp(str1, str2);
}
```

main.c (Main Program):

```
#include <stdio.h>
#include "stringutils.h"

int main() {
    char str1[50] = "Hello, ";
    char str2[] = "world!";

    printf("String 1: %s\n", str1);
    printf("String 2: %s\n", str2);

    printf("Length of String 1: %d\n", stringLength(str1));
    printf("Length of String 2: %d\n", stringLength(str2));

    stringConcat(str1, str2);
    printf("Concatenated String: %s\n", str1);

    int result = stringCompare(str1, "Hello, world!");
    if (result == 0) {
        printf("The strings are equal.\n");
    } else {
        printf("The strings are not equal.\n");
    }

    return 0;
}
```

Compile the program using the following command:

```
gcc stringutils.c main.c -o stringutils_program
```

Then run the program:

```
./stringutils_program
```

In this example, we created a small utility library for string manipulation. The stringutils.h header file defines functions for string length, concatenation, and comparison. The stringutils.c source file implements these functions. The main.c program demonstrates the usage of the library by performing various string operations on two strings.

Exercises

1. Create a header file called that provides functions for reading and writing files. Define functions for reading text from a file, writing text to a file, and checking file existence. Implement these functions in a source file and use them in a program.

2. Develop a header file for a linked list. Define the necessary functions for this data structure, and implement them in a source file. Then, write a program to demonstrate the data structure's usage.

3. Develop a header file that defines functions for matrix operations, such as matrix addition, multiplication, transposition, and determinant calculation. Implement these functions and use them to work with matrices in a program.

Lab Session 07

Use some of the most frequently executed Linux operating system commands

A Linux command is any executable file. This means that any executable file added to the system becomes a new command on the system. A Linux command is a type of file that is designed to be run, as opposed to files containing data or configuration information.

Input and Output Redirection

Linux allows to "pipe" the output from a command so that it becomes another command's input. This is done by typing two or more commands separated by the | character. The | character means "Use the output from the previous command as the input for the next command." Therefore, typing command_1|command_2 does both commands, one after the other, before giving you the results.
Another thing you can do in Linux is to send output to a file instead of the screen. To send output to a file, use the “>” symbol. There are many different reasons why you might want to do this. You might want to save a "snapshot" of a command's output as it was at a certain time, or you might want to save a command's output for further examination. You might also want to save the output from a command that takes a very long time to run, and so on.

Command Options & other parameters

You can use command options to fine-tune the actions of a Linux command. Instead of making you learn a second command, Linux lets you modify the basic, or default, actions of the command by using options. Linux commands often use parameters that are not actual command options. These parameters, such as filenames or directories, are not preceded by a dash.

Executing a Linux Command

From the command prompt simply type the name of the command:

```
	$ command
```

Where $ is the prompt character for the Bourne shell.

Or if the command is not on your path type the complete path and name of the command such as:

```
	$ /usr/bin/command.
```

Some of the frequently used Linux commands are discussed below:

1) **ls (list)**

The ls command lists the contents of a directory. It is a program in the /bin directory.
Examples:

ls	list the current working directory
ls -l	list the current working directory in a long format (with details)
ls /home	list the /home directory
ls -d *.dat	list the current working directory for the .dat files/directories, for directories print only the names not the contents
ls -a	list all files in the current working directory including those whose names begin with a dot
ls ~	list the user’s home directory

2) `pwd (print working directory)`

The pwd command displays the name of (absolute path to) the current working directory. It is a program in the /bin directory.
Example:

`pwd`	print the absolute path to the working directory

3) `cd (change directory)`

The cd command changes the current working directory. It is a part of the bash.
Examples:

`cd mydir`	change to the mydir directory
`cd ..`	change to the parent directory
`cd`	change to the home directory
`cd ../somedir`	change to the somedir directory residing in the parent directory

4) `less`

The less command displays the contents of a file on the screen in its own environment. Listing with the cursor up and down, page up and page down keys is enabled. It is a program in the /usr/bin directory. To terminate the 'less' environment, press q.
Examples:

`less data`	display the contents of the data file
`less -N data`	display the contents of the data file with the line numbers

Searching for a particular phrase or sequence of characters is possible. To find all the occurrences of a phrase consisting for instance of two words, i.e. word1 word2, type /word1 word2 in the less environment. No quotes are needed. The space character is treated as any other character. To display the next occurrence of the searched phrase, type n.

5) `find`

The find command finds the files, directories, etc., matching a given name pattern. It searches the entire directory structure beneath one or more given directories. All subdirectories are included recursively. The find command is a program in the /usr/bin directory.
Examples:

`find /usr . -name '*.txt' -type f`	search /usr and current directory for the files ending with .txt
`find . -name 'data' -type d`	search the current directory for the data directories
`find . -not -name '*.txt' -type f`	search the current directory for the non .txt files

6) `grep (Global Regular Expression Print)`

The grep command searches for a particular phrase or a sequence of characters in a file. It is a program in the /bin directory. The following characters have a special meaning in the search description:
. any single character
***** arbitrary characters
[] any single character listed within brackets

^ beginning of a line
$ end of a line
\\ escape sequence for special characters

`grep 'pattern' *.txt`	search all the .txt files in the current directory and print all lines containing the pattern word
`grep 'pattern' /home/user/data/*`	search all files in the /home/user/data directory and print all lines containing a pattern
`grep -l 'pattern' /home/user/data/*`	print the filenames of the files in /home/user/data containing the pattern
`grep -i 'pattern' data`	word print all lines in the data file containing the case-insensitive pattern word
`grep -v 'pattern' data`	print all lines in the data file not containing the word pattern
`grep -n 'pattern' data`	print all lines in the data file containing the word pattern with the line numbers
`grep -c 'pattern' data`	print the number of lines in the data file containing the pattern word
`grep 'j..k' data`	print all lines in the data file containing a four-character word beginning with j and ending with k (e.g. jack)
`grep 'j*k' data`	print all lines in the data file containing a word beginning with j and ending with k (e.g. jack, jailbreak)
`grep 'j[ao]ck' data`	print all lines in the data file containing the word jack or jock
`grep '^jack' data`	print all lines in the data file starting with the word jack
`grep 'jack$' data`	print all lines in the data file ending with the word jack

7) wc (word count)

The wc command counts the lines, words and characters (bytes) in a file. It is a program in the /usr/bin directory.
Examples:

`wc data`	count the lines, words and bytes of the data file
`wc –l data`	count the lines in the data file
`wc –w data`	count the words in the data file
`wc –c data`	count the bytes in the data file

8) su (substitute user)

The su command changes the user identity. It is a program in the /bin directory.
Examples:

`su`	become the root user
`su jekyll`	become the jekyll user

9) man (manual)

The man command provides in-depth information about a requested command or allows users to search for commands related to a particular keyword.
To terminate, press q. It is a program in the /usr/bin directory.
Examples:

`man ls`	Print the manual pages of ls command

10) history

The history command displays a list of the executed commands. It is a part of the bash.
Examples:

`history`	print the history list
`history -c`	clear the history list

Exercises

1. Write a command to:
   - List all files (and subdirectories) in the home directory.
   - List all files named *chapter1* in the /work directory.
   - List all files beginning with *memo* owned by *ann*.
   - Display the content of /etc/passwd file with as many lines at a time as the last digit of your roll number.
   - Search the current directory, look for filenames that don't begin with a capital letter.
   - Search the system for files that were modified within the last two days.
   - Recursively grep for *your-name* down a directory tree.
   - List all file names containing your roll number in the end.
   - List files in your home folder in human readable format.
   - List the contents of directories /bin and /etc.
   - List C source files in the current directory, showing larger file first.
   - Count all files in the current directory.
   - Create a group called *directors* and assign it a password. Add user *Ann* and user *James* to the group and assign passwords to each of them.
   - Use the pipe (|) operator to combine the output of the ls command with the grep command to filter and display only the files that contain the pattern "hello" in the current directory.
   - Create a file named sample.txt and write the output of the echo command, containing the text "Hello, Linux!" into this file.

Lab Session 08

Handle files & directories in Linux Operating System

Files & Filenames

The most basic concept of a file defines it as a distinct chunk of information that is found on the hard drive. Distinct means that there can be many different files, each with its own particular contents. To keep files from getting confused with each other, every file must have a unique identity. In Linux, you identify each file by its name and location. In each location or directory, there can be only one file by a particular name. Linux allows filenames to be up to 256 characters long. These characters can be lower- and uppercase letters, numbers, and other characters, usually the dash (-), the underscore (_), and the dot (.). They can't include reserved meta characters such as the asterisk, question mark, backslash, and space, because these all have meaning to the shell.

Directories

Linux, like many other computer systems, organizes files in directories. You can think of directories as file folders and their contents as the files. However, there is one absolutely crucial difference between the Linux file system and an office filing system. In the office, file folders usually don't contain other file folders. In Linux, file folders can contain other file folders. In fact, there is no Linux "filing cabinet"—just a huge file folder that holds some files and other folders. These folders contain files and possibly other folders in turn, and so on.

Parent Directories and Subdirectories

Imagine a scenario in which you have a directory, A, that contains another directory, B. Directory B is then a subdirectory of directory A, and directory A is the parent directory of directory B.

The Root Directory

In Linux, the directory that holds all the other directories is called the root directory. This is the ultimate parent directory; every other directory is some level of subdirectory. From the root directory, the whole structure of directory upon directory springs and grows like some electronic elm. This is called a tree structure because, from the single root directory, directories and subdirectories branch off like tree limbs.

Naming Directories

Directories are named just like files, and they can contain upper- and lowercase letters, numbers, and characters such as -, ., and _. The slash (/) character is used to show files or directories within other directories. For instance, usr/bin means that bin is found in the usr directory. Note that you can't tell, from this example, whether bin is a file or a directory, although you know that usr must be a directory because it holds another item—namely, bin. When you see usr/bin/grep, you know that both usr and bin must be directories, but again, you can't be sure about grep. The ls program shows directories with a following /—for example, fido/. This notation implies that you could have, for instance, fido/file; therefore, fido must be a directory. The root directory is shown simply by the symbol / rather than mentioned by name. It's very easy to tell when / is used to separate directories and when it's used to signify the root directory. If / has no name before it, it stands for the root directory. For example, /usr means that the usr subdirectory is found in the root directory, and /usr/bin means that bin is found in the usr directory and that usr is a subdirectory of the root directory. Remember, by definition the root directory can't be a subdirectory.

The Home Directory

Linux provides each user with his or her own directory, called the home directory. Within this home directory, users can store their own files and create subdirectories. Users generally have complete control over what's found in their home directories. Because there are usually no Linux system files or files belonging to other users in your home directory, you can create, name, move, and delete files and directories as you see fit. The location of a user's home directory is specified by Linux and can't be changed by the user. This is both to keep things tidy and to preserve system security.

Important Directories in the Linux File System

Most of the directories that hold Linux system files are "standard." Other UNIX systems will have identical directories with similar contents. This section summarizes some of the more important directories on a Linux system.

/
This is the root directory. It holds the actual Linux program, as well as subdirectories. Do not clutter this directory with your files!

/home
This directory holds users' home directories. In other UNIX systems, this can be the /usr or /u directory.

/bin
This directory holds many of the basic Linux programs. bin stands for binaries, files that are executable and that hold text only computers could understand.

/usr
This directory holds many other user-oriented directories. Some of the most important are described in the following sections. Other directories found in /usr include

docs	Various documents, including useful Linux information
man	The man pages accessed by typing man <command>
games	The fun stuff!

/usr/bin
This directory holds user-oriented Linux programs.

/var/spool
This directory has several subdirectories. Mail holds mail files, spool holds files to be printed, and uucp holds files copied between Linux machines.

/dev
Linux treats everything as a file! The /dev directory holds devices. These are special files that serve as gateways to physical computer components. For instance, if you copy to /dev/fd0, you're actually sending

data to the system's floppy disk. Your terminal is one of the /dev/tty files. Partitions on the hard drive are of the form /dev/hd0. Even the system's memory is a device!
A famous device is /dev/null. This is sometimes called the bit bucket. All information sent to /dev/null vanishes—it's thrown into the trash.

/usr/sbin
This directory holds system administration files. If you do an ls -l, you see that you must be the owner, root, to run these commands.

/sbin
This directory holds system files that are usually run automatically by the Linux system.

/etc
This directory and its subdirectories hold many of the Linux configuration files. These files are usually text, and they can be edited to change the system's configuration (if you know what you're doing!).

Creating Files

Linux has many ways to create and delete files. In fact, some of the ways are so easy to perform that you have to be careful not to accidentally overwrite or erase files!
Return to your home directory by typing cd. Make sure you're in your /home/<user> directory by running pwd. A file can be created by typing ls -l /bin > test. Remember, the > symbol means "redirect all output to the following filename." Note that the file test didn't exist before you typed this command. When you redirect to a file, Linux automatically creates the file if it doesn't already exist.
What if you want to type text into a file, rather than some command's output? The quick way is to use the command cat.

The cat Command

The cat command is one of the simplest, yet most useful, commands in Linux. The cat command basically takes all its input and outputs it. By default, cat takes its input from the keyboard and outputs it to the screen. Type cat at the command line:

```
$ cat
```

The cursor moves down to the next line, but nothing else seems to happen. Now cat is waiting for some input:

```
Hello
Hello
What
```

Everything you type is repeated on-screen as soon as you press Enter! How do you get out of this? At the start of a line, type ^D (Ctrl-D). (In other words, hold down the Ctrl key and press D.) If you're not at the beginning of a line, you have to type ^D twice. ^D is the Linux "end of file" character. When a program such as cat encounters a ^D, it assumes that it has finished with the current file, and it goes on to the next one. In this case, if you type ^D by itself on an empty line, there is no next file to go on to, and cat exits.
So how do you use cat to create a file? Simple! You redirect the output from cat to the desired filename:

```
      $ cat > newfile
      Hello world
      Here's some text
```

Type as much as you want. When you are finished, press ^D by itself on a line; you will be back at the Linux prompt. Now you want to look at the contents of newfile. You could use the more or less commands, but instead, let's use cat. Yes, you can use cat to look at files simply by providing it with a filename:

```
      $ cat newfile
      Hello world
      Here's some text
```

You can also add to the end of the file by using >>. Whenever you use >>, whether with cat or any other command, the output is always appended to the specified file. (Note that the ^D character does not appear on-screen)

```
      $ cat >> newfile
      Some more lines
      ^D
      $ cat newfile
      Hello world
      Here's some text
      Some more lines
```

To discover what cat actually stands for, let's first create another file.

```
$ cat > anotherfile
Different text
^D
$
```

Now, try this:

```
      $ cat newfile anotherfile> thirdfile
      $ cat thirdfile
      Hello world
      Here's some text
      Some more lines
      Different text
```

cat stands for concatenate; cat takes all the specified inputs and regurgitates them in a single lump. This by itself would not be very interesting, but combine it with the forms of input and output redirection available in Linux and you have a powerful and useful tool.

Moving and Copying Files

You often need to move or copy files. The mv command moves files, and the cp command copies files. The mv command is much more efficient than the cp command. When you use mv, the file's contents are not moved at all; rather, Linux makes a note that the file is to be found elsewhere within the file system's structure of directories.
When you use cp, you are actually making a second physical copy of your file and placing it on your disk. This can be slower (although for small files, you won't notice any difference), and it causes a bit more wear and tear on your computer. Don't make copies of files when all you really want to do is move them!

The syntax for the two commands is similar:

```
mv <source> <destination>
cp <source> <destination>
```

In the Linux environment renaming a file is just a special case of moving a file. To move a file to /tmp use this:

```
$ mv fileone /tmp
```

To move the file to /tmp and change the name of the file use this:

```
$ mv fileone /tmp/newfilename
```

By using the above a file can be renamed. Simply move a file from its existing name to a new name in the same directory:

```
$ mv fileone newfilename
```

Because the mv command can accept more than two arguments so more than one file can be moved. To move all files in the current directory with the extension .bak, .tmp, .old to /tmp use this:

```
$ mv *.bak *.tmp *.old /tmp
```

Here are some more examples:

```
$ ls

anotherfile newdir/ newfile thirdfile

$ mv anotherfile movedfile

$ ls

movedfile newdir/ newfile thirdfile

$ cp thirdfile xyz

$ ls
```

```
      anotherfile newdir/ newfile thirdfile xyz
```

You can use cat (or more or less) at any time to verify that another file became moved file, and that the contents of file xyz are identical to the contents of third file.
The cp command found at /bin/cp is used for copying and provides a powerful tool for copy operations. The most basic uses for cp command are to copy a file from one place to another or to make a duplicate file in the same directory. For instance to copy a file this file in the current directory to a second file to be called this file-copy in the same directory enter the following command:

```
      $ cp thisfile thisfile-copy
```

Using ls –1 to look at the directory listing of the files, you would find two files with identical sizes but different date stamps. The new file has a date stamp indicating when the copy operation took place:it is a new, separate file. Changes to thisfile-copy do not affect the original thisfile file.
Similarly to make a copy of thisfile in the /tmp directory use the following command:

```
      $ cp thisfile /tmp
```

and if you want to copy thisfile to /tmp but give the new file a different name, enter

```
      $ cp thisfile /tmp/newfilename
```

Also to avoid overwriting a file accidentally use the –i flag of the cp command which forces the system to confirm any file it will overwrite when copying. Then a prompt like the following appears:

```
      $ cp –i  thisfile newfile
cp: overwrite thisfile?
```

An alias for the cp command can be created by using the following:

```
      $ alias cp='cp –i'
```

Here an alias has been defined so that when the cp command is issued, then actually cp –i
Is issued. In this way the user is always prompted before overwriting a file while copying. If the user logs in as the super user or root user this alias is set by default in most Linux distributions. This is especially important because making a small mistake as the root user can have drastic consequences for the whole system.

Copying Multiple Files in One Command

In DOS only one file or file expression can be copied at a time. To copy three separate files then three commands must be issued. The Linux cp command makes this a bit easier. The cp command can take more than two arguments. If more than two arguments are passed to the command then the last one is treated as the destination and all preceding files are copied to this destination.
For example to copy fileone ,filetwo and filethree in the current directory to /tmp then the following commands can be issued:

```
      $ cp fileone /tmp
      $ cp filetwo /tmp
      $ cp filethree /tmp
```

All this can be bundled into one command like this:

```
$ cp fileone filetwo filethree /tmp
```

Similarly wildcards can be used to mix and copy a large number of files in one command. For instance, this command copies all files with any one of the three extensions in one command.

```
$ cp *.txt *.doc *.bak /tmp
```

When copying multiple files in this way it is important to remember that the last argument must be a directory, since it is impossible to copy two or more files into a single file.
To copy an entire directory and all its subdirectories the –R flag of the cp command is used. This command indicates that a directory is to be recursively copied. For example if a subdirectory called “somedir” exists in the current directory and is to be copied to /tmp then the following command can be used:

```
$ cp –R somedir /tmp
```

Creating a Directory

To create a new directory, use the mkdir command. The syntax is mkdir <name>, where <name> is replaced by whatever you want the directory to be called. This creates a subdirectory with the specified name in your current directory:

```
$ ls
anotherfile newfile thirdfile
$ mkdir newdir
$ ls
anotherfile newdir/ newfile thirdfile
```

Note: The mkdir command is already familiar to you if you have used MS-DOS systems. In MS-DOS, you can abbreviate mkdir as md. You might think that md would work in Linux, because, after all, most of the commands we've seen have extremely concise names. However, Linux doesn't recognize md; it insists on the full mkdir. If you frequently switch between Linux and MS-DOS, you might want to use mkdir for both systems. However, be warned that you might start typing other Linux commands in MS-DOS—for example, typing ls instead of dir!

Moving Directories

To move a directory, use the mv command. The syntax is mv <directory> <destination>. In the following example, you would move the newdir subdirectory found in your current directory to the /tmp directory:

```
$ mv newdir /tmp
$ cd /tmp
$ ls
/newdir
```

The directory newdir is now a subdirectory of /tmp.

Note: When you move a directory, all its files and subdirectories go with it.

Removing Files and Directories

Now that you know how to create files and directories, it's time to learn how to undo your handiwork.

Removing Files

To remove (or delete) a file, use the rm command found at /bin/rm. (rm is a very terse spelling of remove). The syntax is rm <filename>. For instance:

`$ rm myfile` removes the file myfile from your current directory.

`$ rm /tmp/myfile` removes the file myfile from the /tmp directory.

`$ rm *` removes all files from your current directory. (Be careful when using wildcards!)

`$ rm /tmp/*files` removes all files ending in “files” from the /tmp directory.

Note: As soon as a file is removed, it is gone! Always think about what you're doing before you remove a file. You can use one of the following techniques to keep out of trouble when using wildcards.
1. Run ls using the same file specification you use with the rm command. For instance:

```
$ ls *files
```

myfiles newfiles samefiles

```
$ rm *files
```

In this case, you thought you wanted to remove all files that matched *files. To verify that this indeed was the case, you listed all the *files (wildcards work the same way with all commands). The listing looked okay, so you went ahead and removed the files.
2. Use the i (interactive) option with rm:

```
$ rm -i *files

rm: remove 'myfiles'? y

rm: remove 'newfiles'? n

rm: remove 'samefiles'? y
```

Note: When you use rm -i, the command goes through the list of files to be deleted one by one, prompting you for the OK to remove the file. If you type y or Y, rm removes the file. If you type any other character, rm does not remove it. The only disadvantage of using this interactive mode is that it can be very tedious when the list of files to be removed is long.

Removing Directories

The command normally used to remove (delete) directories is rmdir. The syntax is rmdir <directory>. Before you can remove a directory, it must be empty (the directory can't hold any files or subdirectories). Otherwise, you see

```
rmdir: <directory>: Directory not empty
```

This is as close to a safety feature as you will see in Linux!
This one might mystify you (in your home directory)

```
     $ ls
     fido/ root/ zippy/
     $ ls zippy
     core kazoo stuff
     $ rm zippy/*
     $ ls zippy
     $ rmdir zippy
     rmdir: zippy: Directory not empty
```

The reason for the Directory not empty message is that files starting with . usually are special system files and are usually hidden from the user. To list files whose names start with ., you have to use ls -a. To delete these files, use rm .*:

```
     $ ls -a zippy
     ./ ../ .bashrc .profile
     $ rm zippy/.*
     rm: cannot remove '.' or '..'
     $ ls -a zippy
     ./ ../
     $ rmdir zippy
     $ ls
     fido/ root/
     $
```

You will most often come across this situation in a system administrator role.
Sometimes you want to remove a directory with many layers of subdirectories. Emptying and then deleting all the subdirectories one by one would be very tedious. Linux offers a way to remove a directory and all the files and subdirectories it contains in one easy step. This is the r (recursive) option of the rm command. The syntax is rm -r <directory>. The directory and all its contents are removed.

Note: You should use rm -r only when you really have to. To paraphrase an old saying, "It's only a shortcut until you make a mistake." For instance, if you're logged in as root, the following command removes all files from your hard disk, and then it's "Hello, installation procedure" time (do not type the following command!):

rm -rf /

Believe it or not, people do this all too often. Don't join the club!

File Permissions and Ownership

All Linux files and directories have ownership and permissions. You can change permissions, and sometimes ownership, to provide greater or lesser access to your files and directories. File permissions also determine whether a file can be executed as a command.

If you type ls -l or dir, you see entries that look like this:

-rw-r—r— 1 fido users 163 Dec 7 14:31 myfile

The -rw-r—r— represents the permissions for the file myfile. The file's ownership includes fido as the owner and users as the group.

File and Directory Ownership

When you create a file, you are that file's owner. Being the file's owner gives you the privilege of changing the file's permissions or ownership. Of course, once you change the ownership to another user, you can't change the ownership or permissions anymore!
File owners are set up by the system during installation. Linux system files are owned by IDs such as root, uucp, and bin. Do not change the ownership of these files.

The chown command

Use the chown (change ownership) command to change ownership of a file. The syntax is chown <owner> <filename>. In the following example, you change the ownership of the file myfile to root:

```
$ ls -l myfile

-rw-r—r— 1 fido users 114 Dec 7 14:31 myfile

$ chown root myfile

$ ls -l myfile

-rw-r—r— 1 root users 114 Dec 7 14:31 myfile
```

To make any further changes to the file myfile, or to chown it back to fido, you must use su or log in as root.

The chgrp command

Files (and users) also belong to groups. Groups are a convenient way of providing access to files for more than one user but not to every user on the system. For instance, users working on a special project could all belong to the group project. Files used by the whole group would also belong to the group project, giving those users special access. Groups normally are used in larger installations. You may never need to worry about groups.
The chgrp command is used to change the group the file belongs to. It works just like chown.
The syntax of this command is as follows:
chgrp [-Rcfv] [--recursive] [--changes] [--silent] [--quiet] [--verbose] *group filename…*

File Permissions

Linux lets you specify read, write, and execute permissions for each of the following: the owner, the group, and "others" (everyone else).

- **read** permission enables you to look at the file. In the case of a directory, it lets you list the directory's contents using ls.
- **write** permission enables you to modify (or delete!) the file. In the case of a directory, you must have write permission in order to create, move, or delete files in that directory.
- **execute** permission enables you to execute the file by typing its name. With directories, execute permission enables you to cd into them.

For a concrete example, let's look at myfile again:

```
    -rw-r—r— 1 fido users 163 Dec 7 14:31 myfile
```

The first character of the permissions is -, which indicates that it's an ordinary file. If this were a directory, the first character would be d.
The next nine characters are broken into three groups of three, giving permissions for owner, group, and other. Each triplet gives read, write, and execute permissions, always in that order. Permission to read is signified by an r in the first position, permission to write is shown by a w in the second position, and permission to execute is shown by an x in the third position. If the particular permission is absent, its space is filled by -.
In the case of myfile, the owner has rw-, which means read and write permissions. This file can't be executed by typing myfile at the Linux prompt.
The group permissions are r—, which means that members of the group "users" (by default, all ordinary users on the system) can read the file but not change it or execute it.
Likewise, the permissions for all others are r—: read-only.
File permissions are often given as a three-digit number—for instance, 751. It's important to understand how the numbering system works, because these numbers are used to change a file's permissions. Also, error messages that involve permissions use these numbers.
The first digit codes permissions for the owner, the second digit codes permissions for the group, and the third digit codes permissions for other (everyone else).
The individual digits are encoded by summing up all the "allowed" permissions for that particular user as follows:

Read permission	4
write permission	2
execute permission	1

Therefore, a file permission of 751 means that the owner has read, write, and execute permission (4+2+1=7), the group has read and execute permission (4+1=5), and others have execute permission (1).
If you play with the numbers, you quickly see that the permission digits can range between 0 and 7, and that for each digit in that range there's only one possible combination of read, write, and execute permissions.

Note: If you're familiar with the binary system, think of rwx as a three-digit binary number. If permission is allowed, the corresponding digit is 1. If permission is denied, the digit is 0. So r-x would be the binary number 101, which is 4+0+1, or 5. —x would be 001, which is 0+0+1, which is 1, and so on.

The following combinations are possible:
0 or —-: No permissions at all
4 or r—: read-only
2 or -w-: write-only (rare)
1 or —x: execute
6 or rw-: read and write
5 or r-x: read and execute
3 or -wx: write and execute (rare)
7 or rwx: read, write, and execute
Anyone who has permission to read a file can then copy that file. When a file is copied, the copy is owned by the person doing the copying. He or she can then change ownership and permissions, edit the file, and so on. Removing write permission from a file doesn't prevent the file from being deleted! It does prevent it from being deleted accidentally, since Linux asks you whether you want to override the file permissions. You have to answer y, or the file will not be deleted.

Changing File Permissions

To change file permissions, use the chmod (change [file] mode) command. The syntax is chmod <specification> file.
There are two ways to write the permission specification. One is by using the numeric coding system for permissions:
Suppose that you are in the home directory.

```
$ ls -l myfile
-rw-r—r— 1 fido users 114 Dec 7 14:31 myfile
$ chmod 345 myfile
$ ls -l myfile
—wxr—r-x 1 fido users 114 Dec 7 14:31 myfile
$ chmod 701 myfile
$ ls -l myfile
-rwx——x 1 root users 114 Dec 7 14:31 myfile
```

This method has the advantage of specifying the permissions in an absolute, rather than relative, fashion. Also, it's easier to tell someone "Change permissions on the file to seven-five-five" than to say "Change permissions on the file to read-write-execute, read-execute, read-execute."
You can also use letter codes to change the existing permissions. To specify which of the permissions to change, type u (user), g (group), o (other), or a (all). This is followed by a + to add permissions or a - to remove them. This in turn is followed by the permissions to be added or removed. For example, to add execute permissions for the group and others, you would type:

```
$ chmod go+r myfile
```

Other ways of using the symbolic file permissions are described in the chmod man page.

Changing Directory Permissions

You change directory permissions with chmod, exactly the same way as with files. Remember that if a directory doesn't have execute permissions, you can't cd to it.

Note: Any user who has write permission in a directory can delete files in that directory, whether or not that user owns or has write privileges to those files. Most directories, therefore, have permissions set to drwxr-xr-x. This ensures that only the directory's owner can create or delete files in that directory. It is especially dangerous to give write permission to all users for directories!

Exercises

1. Explain what the following commands do (with examples) and practice them:

```
Lockfile
cksum
comm.
csplit
chattr
touch
```

2. What do the following do:

```
cat  ch1
cat ch1 ch2 ch3 > "your-practical-group"
cat note5 >> notes
cat > temp1
cat > temp2 << "yourname"
```

3. Practice the following commands and explain each:

cpio
sort
fuser
file

4. What does the z option of the tar command do? Explain with examples.

5. Differentiate between cp and cpio command?

6. Write two commands to take the backup of your home-folder and all sub-folders. The destination folder should be /home/bkup. *NOTE: size of backup should be smaller than original folder.*

7. What is the difference between the permissions 777 and 775 of the chmod command?

***NOTE:** Full details of all the commands can be found in the man pages for those commands.*

Lab Session 09

Practice Shell Scripting

When a user enters commands from the command line, he is entering them one at a time and getting a response from the system. From time to time it is required to execute more than one command, one after the other, and get the final result. This can be done with a ***shell program or a shell script***. A shell program is a series of Linux commands and utilities that have been put into a file by using a text editor. When a shell program is executed the commands are interpreted and executed by Linux one after the other.
A shell program is like any other programming language and it has its own syntax. It allows the user to define variables, assign various values, and so on.

Creating and Executing a Shell Program

At the simplest level, shell programs are just files that contain one or more shell or Linux commands. These programs can be used to simplify repetitive tasks, to replace two or more commands that are always executed together with a single command, to automate the installation of other programs, and to write simple interactive applications.
To create a shell program, you must create a file using a text editor and put the shell or Linux commands you want to be executed into that file. For example, assume you have a CD-ROM drive mounted on your Linux system. This CD-ROM device is mounted when the system is first started. If you later change the CD in the drive, you must force Linux to read the new directory contents. One way of achieving this is to put the new CD into the drive, unmount the CD-ROM drive using the Linux umount command, and then remount the drive using the Linux mount command. This sequence of steps is shown by the following commands:

```
umount /dev/cdrom

mount -t iso9660 /dev/cdrom /cdrom
```

Instead of typing both of these commands each time you change the CD in your drive, you could create a shell program that would execute both of these commands for you. To do this, put the two commands into a file and call the file remount (or any other name you want).
Several ways of executing the commands are contained in the remount file. One way to accomplish this is to make the file executable. This is done by entering the following command:

```
chmod +x remount
```

This command changes the permissions of the file so that it is now executable. You can now run your new shell program by typing remount on the command line.

Command Aliases

Another way that bash makes life easier for you is by supporting command aliases. Command aliases are commands that the user can specify. Alias commands are usually abbreviations of other commands, designed to save keystrokes.
For example, if you are entering the following command on a regular basis, you might be inclined to create an alias for it to save yourself some typing:

```
      cd /usr/X11R6/lib/X11/config
```

Instead of typing this command every time you wanted to go to the sample-configs directory, you could create an alias called goconfig that would cause the longer command to be executed. To set up an alias like this you must use the bash alias command. To create the goconfig alias, enter the following command at the bash prompt:

```
      alias goconfig='cd /usr/X11R6/lib/X11/configs
```

Now, until you exit from bash, the goconfig command will cause the original, longer command to be executed as if you had just typed it.
If you decide after you have entered an alias that you did not need it, you can use the bash unalias command to delete the alias:

```
      unalias goconfig
```

Note: When defining aliases, you can't include spaces on either side of the equal sign, or the shell can't properly determine what you want to do. Quotation marks are necessary only if the command within them contains spaces or other special characters.

If you enter the alias command without any arguments, it will display all of the aliases that are already defined on-screen.

The bang line

Bang line tells the kernel that a specific shell or language is to be used to interpret the contents of the file. This is the line which is written at the beginning of every program. It starts with the bang character (#!) and goes for example like this:

```
      #!/bin/bash
```

This is the link to the Bourne again shell(bash). The instructions written after this bang line will be interpreted using the above named shell.

Using Variables

Linux shell programming is full-fledged programming language and, as such, supports various types of variables. Variables have three major types:

- **Environment variables** are part of the system environment, and they do not need to be defined. They can be used in shell programs. Some of them such as PATH can also be modified within the shell program.
- **Built-in variables** are provided by the system. Unlike environment variable they cannot be modified.
- **User variables** are defined by the user when he writes the script. They can be used and modified at will within the shell program.

A major difference between shell programming and other programming languages is that variables are not typecast.

Assigning values to variables

A value is assigned to a variable simply by typing the variable name followed by an equal sign and the value that is to be assigned to the variable. For example, if you wanted to assign a value of 5 to the variable count, you would enter the following command in bash or pdksh:

```
count=5
```

With tcsh you would have to enter the following command to achieve the same results:

```
set count = 5
```

With the bash and pdksh syntax for setting a variable, you must make sure that there are no spaces on either side of the equal sign. With tcsh, it doesn't matter if there are spaces or not.

Notice that you do not have to declare the variable as you would if you were programming in C or Pascal. This is because the shell language is a non-typed interpretive language. This means that you can use the same variable to store character strings that you use to store integers. You would store a character string into a variable in the same way that you stored the integer into a variable. For example:

```
name=Garry - (for pdksh and bash)
```

Accessing variable values

To access the value stored in a variable precede the variable name with a dollar sign ($). If you wanted to print the value stored in the count variable to the screen, you would do so by entering the following command:

```
echo $count
```

If you omitted the $ from the preceding command, the echo command would display the word count on-screen.

Positional parameters

The shell has knowledge of a special kind of variable called a positional parameter. Positional parameters are used to refer to the parameters that were passed to a shell program on the command line or a shell function by the shell script that invoked the function. When you run a shell program that requires or supports a number of command-line options, each of these options is stored into a positional parameter. The first parameter is stored into a variable named 1, the second parameter is stored into a variable named 2, and so forth. These variable names are reserved by the shell so that you can't use them as variables you define. To access the values stored in these variables, you must precede the variable name with a dollar sign ($) just as you do with variables you define.
The following shell program expects to be invoked with two parameters. The program takes the two parameters and prints the second parameter that was typed on the command line first and the first parameter that was typed on the command line second.

```
#program reverse, prints the command line parameters out in reverse #order
```

```
    echo "$2" "$1"
```

If you invoked this program by entering

```
    reverse hello there
```

The program would return the following output:

```
    there hello
```

Built-in Variables

These are special variables that Linux provides that can be used to make decisions in a program. Their values cannot be modified.
Several other built-in shell variables are important to know about when you are doing a lot of shell programming. The following table lists these variables and gives a brief description of what each is used for.

Variable	*Usage*
$#	Stores the number of command-line arguments that were passed to the shell program.
$?	Stores the exit value of the last command that was executed.
$0	Stores the first word of the entered command (the name of the shell program).
$*	Stores all the arguments that were entered on the command line ($1 $2 ...).
"$@"	Stores all the arguments that were entered on the command line, individually quoted ("$1" "$2" ...).

The importance of quotation marks

The use of the different types of quotation marks is very important in shell programming. Both kinds of quotation marks and the backslash character are used by the shell to perform different functions. The double quotation marks (""), the single quotation marks (''), and the backslash (\) are all used to hide special characters from the shell. Each of these methods hides varying degrees of special characters from the shell.
The double quotation marks are the least powerful of the three methods. When you surround characters with double quotes, all the whitespace characters are hidden from the shell, but all other special characters are still interpreted by the shell. This type of quoting is most useful when you are assigning strings that contain more than one word to a variable. For example, if you wanted to assign the string hello there to the variable greeting, you would type the following command:

```
    greeting="hello there" (for bash and pdksh)
```

This command would store the hello there string into the greeting variable as one word. If you typed this command without using the quotes, you would not get the results you wanted. bash and pdksh would not understand the command and would return an error message. tcsh would assign the value hello to the greeting variable and ignore the rest of the command line.
Single quotes are the most powerful form of quoting. They hide all special characters from the shell. This is useful if the command that you enter is intended for a program other than the shell.

Because the single quotes are the most powerful, you could have written the hello there variable assignment using single quotes. You might not always want to do this. If the string being assigned to the greeting variable contained another variable, you would have to use the double quotes. For example, if you wanted to include the name of the user in your greeting, you would type the following command:

```
greeting="hello there $LOGNAME" (for bash and pdksh)
set greeting="hello there $LOGNAME" (for tcsh)
```

Remember that the LOGNAME variable is a shell variable that contains the Linux username of the person who is logged in to the system.

This would store the value hello there root into the greeting variable if you were logged in to Linux as root. If you tried to write this command using single quotes it wouldn't work, because the single quotes would hide the dollar sign from the shell and the shell wouldn't know that it was supposed to perform a variable substitution. The greeting variable would be assigned the value hello there $LOGNAME if you wrote the command using single quotes.
Using the backslash is the third way of hiding special characters from the shell. Like the single quotation mark method, the backslash hides all special characters from the shell, but it can hide only one character at a time, as opposed to groups of characters. You could rewrite the greeting example using the backslash instead of double quotation marks by using the following command:

```
greeting=hello\ there (for bash and pdksh)
set greeting=hello\ there (for tcsh)
```

In this command, the backslash hides the space character from the shell, and the string hello there is assigned to the greeting variable.

Backslash quoting is used most often when you want to hide only a single character from the shell. This is usually done when you want to include a special character in a string. For example, if you wanted to store the price of a box of computer disks into a variable named disk_price, you would use the following command:

```
disk_price=\$5.00 (for bash and pdksh)
set disk_price = \$5.00 (for tcsh)
```

The backslash in this example would hide the dollar sign from the shell. If the backslash were not there, the shell would try to find a variable named 5 and perform a variable substitution on that variable. Assuming that no variable named 5 were defined, the shell would assign a value of .00 to the disk_price variable. This is because the shell would substitute a value of null for the $5 variable.

The disk_price example could also have used single quotes to hide the dollar sign from the shell.

The back quote marks (`) perform a different function. They are used when you want to use the results of a command in another command. For example, if you wanted to set the value of the variable contents equal to the list of files in the current directory, you would type the following command:

```
contents='ls' (for bash and pdksh)
set contents = 'ls' (for tcsh)
```

This command would execute the ls command and store the results of the command into the contents variable. As you will see in the section "Iteration Statements," this feature can be very useful when you want to write a shell program that performs some action on the results of another command.

Exercises

1. Write a shell program that takes one parameter (your name) and displays it on the screen.

2. Write a shell program that takes a number parameters equal to the last digit of your roll number and displays the values of the built in variables such as $#, $0, and $* on the screen.

3. Write a script that displays the values of the following environment variables
   i) SHELL ii) PWD

Lab Session 10

Use conditional statements, iteration statements and functions in Shell Scripting

In bash and pdksh, a command called test is used to evaluate conditional expressions. You would typically use the test command to evaluate a condition that is used in a conditional statement or to evaluate the entrance or exit criteria for an iteration statement. The test command has the following syntax:

```
test expression
or
[ expression ]
```

Several built-in operators can be used with the test command. These operators can be classified into four groups: integer operators, string operators, file operators, and logical operators.

The shell integer operators perform similar functions to the string operators except that they act on integer arguments. The following table lists the test command's integer operators.

The Test Command's Integer Operators

Operator	Meaning
Int1 -eq int2	Returns True if int1 is equal to int2.
Int1 -ge int2	Returns True if int1 is greater than or equal to int2.
Int1 -gt int2	Returns True if int1 is greater than int2.
Int1 -le int2	Returns True if int1 is less than or equal to int2.
Int1 -lt int2	Returns True if int1 is less than int2.
Int1 -ne int2	Returns True if int1 is not equal to int2.

The string operators are used to evaluate string expressions. The table below lists the string operators that are supported by the three shell programming languages.

The Test Command's String Operators

Operator	Meaning
Str1 = str2	Returns True if str1 is identical to str2.
Str1 != str2	Returns True if str1 is not identical to str2.
str	Returns True if str is not null.
-n str	Returns True if the length of str is greater than zero.
-z str	Returns True if the length of str is equal to zero.

The test command's file operators are used to perform functions such as checking to see if a file exists and checking to see what kind of file is passed as an argument to the test command. The following is the list of the test command's file operators.

The Test Command's File Operators

Operator	Meaning
-d filename	Returns True if file, filename is a directory.
-f filename	Returns True if file, filename is an ordinary file.
-r filename	Returns True if file, filename can be read by the process.
-s filename	Returns True if file, filename has a nonzero length.
-w filename	Returns True if file, filename can be written by the process.
-x filename	Returns True if file, filename is executable.

The test command's logical operators are used to combine two or more of the integer, string, or file operators or to negate a single integer, string, or file operator. The table below lists the test command's logical operators.

The Test Command's Logical Operators

Command	Meaning
! expr	Returns True if expr is not true.
expr1 -a expr2	Returns True if expr1 and expr2 are true.
expr1 -o expr2	Returns True if expr1 or expr2 is true.

Conditional Statements

The bash has two forms of conditional statements. These are the if statement and the case statement. These statements are used to execute different parts of your shell program depending on whether certain conditions are true. As with most statements, the syntax for these statements is slightly different between the different shells.

The if Statement

All three shells support nested if...then...else statements. These statements provide you with a way of performing complicated conditional tests in your shell programs. The syntax of the if statement is the same for bash and pdksh and is shown here:

```
if [ expression ]
then
     commands
elif [ expression2 ]
then
     commands
else
     commands
fi
```

The elif and else clauses are both optional parts of the if statement. Also note that bash and pdksh use the reverse of the statement name in most of their complex statements to signal the end of the statement. In this statement the fi keyword is used to signal the end of the if statement. The elif statement is an abbreviation of else if. This statement is executed only if none of the expressions associated with the if statement or any elif statements before it were true. The commands associated with the else statement are executed only if none of the expressions associated with the if statement or any of the elif statements were true.

The following is an example of a bash or pdksh if statement. This statement checks to see if there is a. profile file in the current directory:

```
if [ -f “sar.txt” ]
then
     echo "There is a .txt file in the current directory."
else
     echo "Could not find the .txt file."
fi
```

The case Statement

The case statement enables you to compare a pattern with several other patterns and execute a block of code if a match is found. The shell case statement is quite a bit more powerful than the case statement in Pascal or the switch statement in C. This is because in the shell case statement you can compare strings with wildcard characters in them, whereas with the Pascal and C equivalents you can compare only enumerated types or integer values.
Once again, the syntax for the case statement is identical for bash and pdksh and different for tcsh. The syntax for bash and pdksh is the following:

```
case string1 in
str1)
     commands;;
str2)
     commands;;
*)
     commands;;
esac
```

String1 is compared to str1 and str2. If one of these strings matches string1, the commands up until the double semicolon (;;) are executed. If neither str1 nor str2 matches string1, the commands associated with the asterisk are executed. This is the default case condition because the asterisk matches all strings.

The following code is an example of a bash or pdksh case statement. This code checks to see if the first command-line option was -i or -e. If it was -i, the program counts the number of lines in the file specified by the second command-line option that begins with the letter i. If the first option was -e, the program counts the number of lines in the file specified by the second command-line option that begins with the letter e. If the first command-line option was not -i or -e, the program prints a brief error message to the screen.

```
case $1 in
-i)
     count='grep ^i $2 | wc -l'
     echo "The number of lines in $2 that start with an i is
     $count"
     ;;
-e)
     count='grep ^e $2 | wc -l'
     echo "The number of lines in $2 that start with an e is
     $count"
     ;;
* )
```

```
          echo "That option is not recognized"
          ;;
     esac
```

The shell languages also provide several iteration or looping statements. The most commonly used of these is the for statement.

The for Statement

The 'for' statement executes the commands that are contained within it a specified number of times. The 'bash' and 'pdksh' have two variations of the for statement.

The first form of the for statement that bash and pdksh support has the following syntax:

```
     for var1 in list
     do
          commands
     done
```

In this form, for statement executes once for each item in the list. This list can be a variable that contains several words separated by spaces, or it can be a list of values that is typed directly into the statement. Each time through the loop, the variable var1 is assigned the current item in the list, until the last one is reached.
The second form of for statement has the following syntax:

```
     for var1
     do
          statements
     done
```

In this form, the for statement executes once for each item in the variable var1. When this syntax of the for statement is used, the shell program assumes that the var1 variable contains all the positional parameters that were passed in to the shell program on the command line.
Typically this form of for statement is the equivalent of writing the following for statement:

```
     for var1 in "$@"
     do
          statements
     done
```

The following is an example of the bash or pdksh style of for statement. This example takes as command-line options any number of text files. The program reads in each of these files, converts all the letters to uppercase, and then stores the results in a file of the same name but with a .caps extension.

```
     for file
     do
          tr a-z A-Z < $file >$file.caps
     done
```

The while Statement

Another iteration statement offered by the shell programming language is the while statement. This statement causes a block of code to be executed while a provided conditional expression is true. The syntax for the while statement in bash and pdksh is the following:

```
while expression
do
     statements
done
```

Care must be taken with the while statements because the loop will never terminate if the specifies condition never evaluates to false.

The until Statement

The until statement is very similar in syntax and function to the while statement. The only real difference between the two is that the until statement executes its code block while its conditional expression is false, and the while statement executes its code block while its conditional expression is true. The syntax for the until statement in bash and pdksh is

```
until expression
do
     commands
done
```

In practice the until statement is not very useful, because any until statement you write can also be written as a while statement.

The shift Command

bash, pdksh, and tcsh all support a command called shift. The shift command moves the current values stored in the positional parameters to the left one position. For example, if the values of the current positional parameters are

```
$1 = -r $2 = file1 $3 = file2
```

and you executed the shift command

```
shift
```

the resulting positional parameters would be as follows:

```
$1 = file1 $2 = file2
```

You can also move the positional parameters over more than one place by specifying a number with the shift command. The following command would shift the positional parameters two places:

```
    shift 2
```

This is a very useful command when you have a shell program that needs to parse command-line options. This is true because options are typically preceded by a hyphen and a letter that indicates what the option is to be used for. Because options are usually processed in a loop of some kind, you often want to skip to the next positional parameter once you have identified which option should be coming next. For example, the following shell program expects two command-line options—one that specifies an input file and one that specifies an output file. The program reads the input file, translates all the characters in the input file into uppercase, then stores the results in the specified output file.

The following example was written using bash, pdksh syntax.

```
    while [ "$1" ]
    do
         if [ "$1" = "-i" ] then
              infile="$2"
              shift 2
         elif [ "$1" = "-o" ]
         then
              outfile="$2"
              shift 2
         else
              echo "Program $0 does not recognize option $1"
         fi
    done
    tr a-z A-Z <$infile >$outfile
```

This program would run by giving the following command on the terminal:

```
    $ program_name –i inputfile –o outputfile
```

The select Statement

pdksh offers one iteration statement that neither bash nor tcsh provides. This is the select statement. This is actually a very useful statement. It is quite a bit different from the other iteration statements because it actually does not execute a block of shell code repeatedly while a condition is true or false. What the select statement does is enable you to automatically generate simple text menus. The syntax for the select statement is

```
    select menuitem [in list_of_items]
    do
         commands
    done
```

where square brackets are used to enclose the optional part of the statement. When a select statement is executed, pdksh creates a numbered menu item for each element in the list_of_items. This list_of_items can be a variable that contains more than one item, such as choice1 choice2, or it can be a list of choices typed in the command. For example:

```
     select menuitem in choice1 choice2 choice3
```

If the list_of_items is not provided, the select statement uses the positional parameters just as with the for statement. Once the user of the program containing a select statement picks one of the menu items by typing the number associated with it, the select statement stores the value of the selected item in the menuitem variable. The statements contained in the do block can then perform actions on this menu item.

The following example illustrates a potential use for the select statement. This example displays three menu items, and when the user chooses one of them it asks whether that was the intended selection. If the user enters anything other than y or Y, the menu is redisplayed.

```
     select menuitem in pick1 pick2 pick3
     do
          echo "Are you sure you want to pick $menuitem"
          read res
          if [ $res = "y" -o $res = "Y" ]
          then
               break
          fi
     done
```

A few new commands are introduced in this example. The read command is used to get input from the user. It stores anything that the user types into the specified variable. The break command is used to exit a while, until, repeat, select, or for statement.

The break Statement

The break statement can be used to terminate an iteration loop, such as a for, until or repeat command.

The exit Statement

The exit statement can be used to exit a shell program. A number can be optionally used after exit. If the current shell program has been called by another shell program, the calling program can check for the code and make a decision accordingly.

Functions

The shell languages enable you to define your own functions. These functions behave in much the same way as functions you define in C or other programming languages. The main advantage of using functions as opposed to writing all of your shell code in line is for organizational purposes. Code written using functions tends to be much easier to read and maintain and also tends to be smaller, because you can group common code into functions instead of putting it everywhere it is needed.
The syntax for creating a function in bash and pdksh is the following:

```
     fname () {
          shell commands
     }
```

pdksh also allows the following syntax:

```
function fname {
     shell commands
}
```

Both of these forms behave in the exact same way. Once you have defined your function using one of these forms, you can invoke it by entering the following command:

```
fname [parm1 parm2 parm3 ...]
```

Notice that you can pass any number of parameters to your function. When you do pass parameters to a function, it sees those parameters as positional parameters, just as a shell program does when you pass it parameters on the command line. For example, the following shell program contains several functions, each of which is performing a task associated with one of the command-line options. This example illustrates many of the topics covered in this chapter. It reads all the files that are passed on the command line and—depending on the option that was used—writes the files out in all uppercase letters, writes the files out in all lowercase letters, or prints the files.

```
upper () {
     shift
     for i
     do
          tr a-z A-Z <$1 >$1.out
          rm $1
          mv $1.out $1
          shift
     done; }
lower () {
     shift
     for i
     do
          tr A-Z a-z <$1 >$1.out
          rm $1
          mv $1.out $1
          shift
     done; }
print () {
     shift
     for i
     do
          lpr $1
          shift
     done; }
usage_error () {
     echo "$1 syntax is $1 <option> <input files>"
     echo ""
     echo "where option is one of the following"
     echo "p — to print frame files"
     echo "u — to save as uppercase"
     echo "l — to save as lowercase"; }
```

```
case $1
in
p | -p) print $@;;
u | -u) upper $@;;
l | -l) lower $@;;
*) usage_error $0;;
esac
```

Exercises

1. What test command should be used to test that /usr/bin is a directory or a file?

2. Write a script that takes two strings as input compares them and depending upon the results of the comparison prints the results.

3. Write a script that takes a number (parameter) from 1-3 as input and uses case to display the name of corresponding month.

4. Write a script that calculates the average of all even numbers less than or equal to your roll number and prints the result.

5. Write a function that displays the name of the week days starting from Sunday if the user passes a day number. If a number provided is not between 1 and 7 an error message is displayed.

6. Write scripts that displays the parameters passed along with the parameter number using while and until statements.

7. Write a script that displays the following menu:
   - Quotient
   - Remainder

Depending on user's choice, the result of division must be displayed and the loop breaks. The two numbers (dividend and divisor) must be supplied at runtime as command line arguments. If user chooses an item that is not in the list, he must be prompted to make proper choice and the loop must restart (or continue).

Lab Session 11

Using gcc and gdb in Linux Operating System

In the world of personal computing, graphical user interfaces (GUIs) dominate, but there are powerful tools in the Linux terminal that shouldn't be overlooked. This lab introduces you to two essential command-line tools: GCC (GNU Compiler Collection) for compiling C and C++ programs and GDB (GNU Debugger) for debugging.

GCC - Compiling and Executing a C Program:

Getting Help:

To explore the capabilities of GCC, you can access the help manual using the --help option:

```
gcc --help
```

This command provides a wealth of information about GCC options and their usage. If you need more detailed information about a specific option, you can refer to the GCC manual pages using the man utility:

```
man gcc
```

Compiling and Running a Simple C Program:

Consider the following example program (greetings.c):

```
#include <stdio.h>

int main() {
  printf("hello, world!\n");
  return 0;
}
```

Compilation Steps:
- **Preprocessing:**

The preprocessing step involves expanding macros and inserting header file contents into the source code. The command below performs preprocessing and saves the result in greetings.i:

```
cpp greetings.c > greetings.i
```

- **Compilation:**

The compilation phase translates the preprocessed code into assembly code (greetings.s):

```
gcc -S greetings.i
```

- **Assembly:**

The assembly phase converts the assembly code into machine language instructions and generates an object file (greetings.o):

```
as greetings.s -o greetings.o
```

- **Linking:**

The linking phase combines object files and libraries to produce an executable. Here's the complex linking command:

```
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o greetings.o /usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/crtbegin.o -L /usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/ -lgcc -lgcc_eh -lc -lgcc -lgcc_eh /usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/crtend.o -o greetings
```

Explanation:

- **-dynamic-linker**: Specifies the dynamic linker.
- **/lib64/ld-linux-x86-64.so.2**: Path to the dynamic linker.
- **/usr/lib/x86_64-linux-gnu/crt1.o, /usr/lib/x86_64-linux-gnu/crti.o, /usr/lib/x86_64-linux-gnu/crtn.o:** Object files containing startup and termination routines.
- **greetings.o**: The object file containing the machine instructions for the program.
- **/usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/crtbegin.o**: Object file for initializing global objects.
- **-L /usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/:** Specifies the directory to search for libraries.
- **-lgcc -lgcc_eh -lc -lgcc -lgcc_eh:** Links with the GCC runtime library and C library.
- **/usr/lib/gcc/x86_64-linux-gnu/$(gcc -dumpversion)/crtend.o:** Object file for finalizing global objects.
- **-o greetings:** Specifies the output executable name.

Now let's have a look at the simplified linking command using gcc:

```
gcc greetings.o -o greetings
```

The gcc command simplifies the linking process by internally handling the necessary details, including specifying the dynamic linker, linking with standard libraries, and managing other configurations.

- **Running the Executable:**

```
./greetings
```

GDB - Basic Usage and Debugging:

Basic Usage:

Let's explore GDB with a simple C program (test.c):

```
#include <stdio.h>

int main() {
  int a = 10;
  int b = 5;
  float c = a / b;
  printf("%d/%d = %f\n", a, b, c);
  return 0;
}
```

Debugging with GDB:

Ensure that your program is compiled with the -g option to include debugging information:

```
gcc -g test.c -o test
```

Running and Debugging:

```
gdb ./test
```

To start execution:

```
run
```

To run with arguments:

```
run arg_1 arg_2
```

Breaking Execution:

To break at a specific line:

```
break test.c:8
```

To break at the beginning of a function:

```
break test.c:main
```

Stepping Through Code:

To execute one line of code:

```
step or s
```

To execute the entire function:

```
next or n
```

To resume execution:

```
continue or c
```

Inspecting Variables:

To inspect variable values:

```
print variable_name
```

Watchpoints:

A watchpoint is a feature in GDB that allows you to monitor the value of a variable and break the program's execution when the variable is modified. In our example, let's set a watchpoint on the variable c:

```
watch c
```

Now, continue the execution:

```
continue
```

If somewhere in your program the value of c changes, GDB will break the execution and inform you about the change.

For instance, if you modify the program to change the value of c during runtime:

```
#include <stdio.h>

int main() {
  int a = 10;
  int b = 5;
  float c = a / b;

  // Changing the value of c
  c = 3.14;

  printf("%d/%d = %f\n", a, b, c);
  return 0;
}
```

Now, when you run the program in GDB, it will break at the point where c is modified:

```
Breakpoint 1, main () at test.c:9
9         c = 3.14;
```

This allows you to investigate the state of your program when a specific variable changes, providing valuable insights during debugging.

Exercises

1- Demonstrate the compilation and execution process of the following program. Insert break point while calculating c, and print the value of c for each iteration (Attach screenshots).

Note:

- argc and argv are parameters used to handle command-line arguments passed to a program when it is executed. argc(Argument Count) is an integer parameter that represents the number of command-line arguments passed to the program. argv(Argument Vector) is an array of strings (or pointers to strings) that holds the actual command-line arguments.

- The atoi (ASCII to Integer) function is used to convert a string representation of an integer to an actual integer value. When you receive input from the command line, the input is typically in the form of strings (character arrays). To perform numerical operations on these values, you need to convert them from strings to integers.

```
 int main(int argc, char **argv)
{
int a, b,c;
a = atoi(argv[1]);
b = atoi(argv[2]);
for(int i=1;i<=b;i++)
{
c = a*i;
 printf("%d * %d = %d\n",a,i,c);
}
return 0;
}
```

2- Write a C program that initializes an array and performs some operations on its elements. Save the program in a file named array_operations.c. Compile the program with debugging information. Use GDB to set a watchpoint on a specific element of the array and observe its changes during the program execution.

Lab Session 12

Automating build processes with Makefiles in C

A Makefile is a simple and powerful tool used in software development to automate the build process of a project. It consists of a set of rules and dependencies that define how a project should be built. Makefiles are commonly used in compiled languages like C and C++, but they can be applied to other programming languages and projects as well. Makefiles serve several essential purposes in software development:

- **Automating Builds**: Makefiles automate the compilation and linking of source code files, making it easier to manage complex software projects. Instead of manually entering compilation commands for every source file, developers can rely on the Makefile to do this efficiently.

- **Dependency Management**: Makefiles specify dependencies between source code files and targets. This means that if a source file is modified, the Makefile can determine which parts of the project need to be rebuilt, thus saving time and ensuring that only the necessary parts are recompiled.

- **Consistency**: Makefiles enforce a consistent build process across different platforms and for different developers. This ensures that the project is built correctly, regardless of the development environment.

- **Documentation**: Makefiles can also serve as a form of documentation for the build process. They make it clear how the project is structured, what files are needed, and how they are linked together.

- **Reusability:** Once a Makefile is set up for a project, it can be easily reused or adapted for similar projects, saving time and effort.

Components of a Makefile

A Makefile consists of the following components:

- **Targets**: Targets are the output files or goals that the Makefile aims to build. These are usually the executable programs or libraries you want to create. A Makefile can have multiple targets, but each should have a unique name.

- **Dependencies**: Dependencies are the input files or prerequisites required to build a target. These are the source code files that the target relies on. If any of the dependencies change, the target needs to be rebuilt. Dependencies are specified for each target in the Makefile.

- **Commands**: Commands are the set of instructions or rules that define how to build a target from its dependencies. These commands are written in a specific format and are executed by the make tool. Commands often include compilation and linking instructions, like compiling C source files with a C compiler (e.g., GCC) and linking object files together.

Writing Your First Makefile

Step 1: Create a Simple C Program:

If you don't already have a C program, create a simple one in a text editor or IDE. For example, create a file named "hello.c" with the following content:

```
#include <stdio.h>

int main() {
    printf("Hello, Makefile!\n");
    return 0;
}
```

Step 2: Create a Makefile:

To create a Makefile, you'll need to create a new text file and name it "Makefile" (with a capital 'M'). The Makefile should be in the same directory as your C program. You can use terminal commands or your text editor to create the file. For example, in a Unix-based system, you can use the touch command to create the Makefile:

```
touch Makefile
```

A Makefile consists of rules and definitions. Each rule is structured like this:

```
target: dependencies
    command
```

Step 3:Define the Makefile Targets, Dependencies and commands

```
CC = gcc
CFLAGS = -Wall
TARGET = hello
SRC = hello.c

$(TARGET): $(SRC)
    $(CC) $(CFLAGS) -o $@ $^
```

Let's break down what's happening above:

- **CC**: This variable defines the compiler (GCC in this case).
- **CFLAGS**: This variable specifies compiler flags (e.g., -Wall for enabling warnings).
- **TARGET**: This variable sets the name of the output executable.
- **SRC**: This variable specifies the source code file.

The rule $(TARGET): $(SRC) indicates that the "hello" target depends on "hello.c." The command $(CC) $(CFLAGS) -o $@ $^ is used to build the "hello" target. It compiles "hello.c" with the specified compiler and flags, creating the output executable named "hello." If we write this Makefile without using variables, the following will also work:

```
hello: hello.c
    gcc -Wall -o hello hello.c
```

In this version, the variables (CC, CFLAGS, TARGET, and SRC) have been replaced with their respective values directly in the Makefile rule. Note that variables in Makefiles allow you to define and reuse values, making your Makefile more maintainable and adaptable.

Here we have also used automatic variables. Automatic variables in Makefiles provide dynamic values based on the context of the rule. They simplify writing rules by eliminating the need to explicitly list dependencies and targets. Some commonly used automatic variables include:

$@: Represents the target.
$<: Represents the first dependency.
$^: Represents all dependencies.

This Makefile will compile the "hello.c" source file with the GCC compiler and the -Wall flag to create an executable named "hello."

Step 4: Test the Makefile:

Save your "Makefile" in the same directory as your C program. Open your terminal or command prompt. Navigate to the directory where your "Makefile" and "hello.c" are located. Run the make command to build your program:

```
make
```

This command will execute the rules you defined in your Makefile and compile your program. Once the compilation is complete, you can run your program:

```
./hello
```

You should see the output: "Hello, Makefile!"

Step 5: Clean Up (Optional):

To remove the generated executable and object files, you can add a "clean" target to your Makefile like this:

```
clean:
    rm -f $(TARGET)
```

Then, you can use the make clean command to remove the executable:

```
make clean
```

Common Makefile Conventions and Best Practices:

Use meaningful target names and file extensions: Make your Makefile more readable by choosing target names that reflect the purpose of the output and using the appropriate file extensions for your platform (e.g., .out, .exe, or none).

Comment your Makefile: Provide comments to explain the purpose of targets and rules, especially if your Makefile becomes more complex.

Include a "clean" target: To remove generated files (e.g., executables and object files), create a "clean" target.

Use phony targets: Phony targets in Makefiles are typically used for defining targets that don't represent actual files. They are often used for tasks like cleaning, testing, or other non-file-related operations. Declare targets as .PHONY to specify that they don't represent actual files. This is often used for targets like "clean" or "all."

Be platform-agnostic: Makefiles should be written to work across different platforms. Avoid hardcoding platform-specific commands or paths.

Example

Let's walkthrough a Makefile for a small C library. In this example, we'll create a Makefile for a simple library named "mylib" that consists of two source files, "mylib.c" and "helper.c."

```
# Compiler and flags
CC = gcc
CFLAGS = -Wall -O2

# Directories
SRC_DIR = src
OBJ_DIR = obj
LIB_DIR = lib

# Source files
SOURCES = $(wildcard $(SRC_DIR)/*.c)
OBJECTS = $(patsubst $(SRC_DIR)/%.c, $(OBJ_DIR)/%.o, $(SOURCES))

# Library name
LIB_NAME = libmylib.a

# Targets
all: $(LIB_NAME)

$(LIB_NAME): $(OBJECTS)
	ar rcs $(LIB_DIR)/$@ $^

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c | $(OBJ_DIR)
	$(CC) $(CFLAGS) -c -o $@ $<
```

```
$(OBJ_DIR):
	mkdir -p $@

clean:
	rm -rf $(OBJ_DIR) $(LIB_DIR) $(LIB_NAME)

.PHONY: all clean
```

Now, let's walk through this Makefile step by step:

Compiler and Flags:

- CC = gcc: We define the C compiler as "gcc."
- CFLAGS = -Wall -O2: Compiler flags for warnings (-Wall) and optimization level (-O2).

Directories:

- SRC_DIR, OBJ_DIR, and LIB_DIR define source code, object files, and library directories.

Source Files:

- SOURCES is a list of source files found in the SRC_DIR directory.
- OBJECTS generate a list of corresponding object files in the OBJ_DIR directory.

Library Name:

- LIB_NAME specifies the name of the library as "libmylib.a."

Targets:

- all is the default target. It depends on $(LIB_NAME) and will build the library.
- $(LIB_NAME) builds the library using ar rcs.
- $(OBJ_DIR)/%.o is a pattern rule for compiling C source files into object files. It depends on source files and uses automatic variables.
- $(OBJ_DIR) is a target for creating the object directory using mkdir -p.
- clean is a phony target to remove generated files and directories (objects, library, and the object directory).

Phony Target:

- .PHONY: all clean declares "all" and "clean" as phony targets.

With this Makefile, you can run make to build the "mylib" library, and make clean to remove generated files and directories. This Makefile demonstrates the use of targets, dependencies, commands, variables, automatic variables, the "all" target, phony targets, and a clean target for a small C library.

Exercises

1. Write a Makefile that conditionally compiles different parts of a project based on build configurations. For example, you can have a "debug" and a "release" target with different compiler flags.
2. Develop a Makefile for creating a static library (.a file) from a set of source files. Then, create a separate Makefile to link this library with another C program.
3. Create a Makefile that works on different platforms (e.g., Windows and Linux) without modification. Show how to handle platform-specific commands or flags.

Lab Session 13

Interacting with the FileSystem using Linux System calls

In the realm of systems programming, understanding and harnessing the power of Linux system calls is fundamental. System calls act as the bridge between user-level applications and the Linux kernel. They enable applications to request services or interact with the kernel to perform various tasks. This section introduces system calls and highlights their significance in systems programming. We will also provide examples of common system calls used for file operations.

What are System Calls?

A system call is a programmatic request for a service provided by the operating system's kernel. System calls serve as an interface between user-space applications and the kernel, allowing programs to perform privileged operations that would be unsafe or impossible if executed directly by the application. Common system call functionalities include file I/O, process management, memory management, and more.

Significance in Systems Programming

System calls are the building blocks of systems programming for several reasons:

- **Low-Level Interface**: They provide a low-level interface to interact with the kernel, allowing programmers to have fine-grained control over system resources.

- **Access to Kernel Services**: System calls give user-space applications access to kernel services such as file management, process control, and hardware control.

- **Portability**: System calls are part of the POSIX standard, making them largely portable across Unix-like operating systems, including Linux.

- **Security**: They are a key part of maintaining system security, as they control access to critical resources and operations.

Interacting with the File System

Interacting with the file system is a crucial part of systems programming on Linux. Understanding how to navigate the Linux file system and how to manipulate files and directories using system calls is essential for building powerful and efficient software. In this section, we will discuss how to navigate into the Linux file system and cover system calls for file and directory manipulation.

Navigating the Linux File System

The Linux file system is organized as a hierarchical structure, similar to a tree. The root directory, denoted by '/', is the top-level directory from which all other directories and files branch. Each directory can contain files and subdirectories. To navigate the file system, you need to understand the basic directory structure and the concept of absolute and relative paths.

- **Absolute Paths**: An absolute path specifies the full directory path starting from the root directory. For example, "/home/user/documents/file.txt" is an absolute path.

- **Relative Paths:** A relative path specifies the location of a file or directory in relation to the current working directory. For example, if you are in the "/home/user" directory, "documents/file.txt" is a relative path.

System Calls for File and Directory Manipulation

open(): The open() system call is used to open a file and create a new file descriptor. Note that a file descriptor is a low-level integer representing a reference to an open file or a communication channel. File descriptors are used to uniquely identify and access files, devices, or other I/O resources in a Unix-like operating system. Open() allows specifying various options like read, write, create, or truncate the file. For example:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int fd = open("example.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
```

#include <fcntl.h>, #include <sys/types.h> and #include <sys/stat.h>: These header files provide the necessary declarations and constants for the open system call and related flags (O_RDWR, O_CREAT, S_IRUSR, S_IWUSR). The sys/types.h and sys/stat.h headers define data types and structures used in file operations, while fcntl.h provides constants and flags for file control operations. Make sure to include these headers at the beginning of your C program to use the open system call and related constants without compilation errors.

The second line of code declares an integer variable fd and initializes it with the result of the open() system call. The open() system call takes three arguments:

"example.txt": The name of the file to be opened or created.
O_RDWR | O_CREAT: Flags that specify the mode of the file open operation. In this case:
    O_RDWR: Open the file for reading and writing.
    O_CREAT: If the file does not exist, create it.
S_IRUSR | S_IWUSR: File permissions for the newly created file. Here:
    S_IRUSR: Read permission for the owner.
    S_IWUSR: Write permission for the owner.

read() and write(): The read() and write() system calls are used for reading from and writing to a file or file descriptor, respectively.

```
#include <unistd.h>

char buffer[1024];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
ssize_t bytes_written = write(fd, buffer, bytes_read);
```

#include <unistd.h>: The unistd.h header in C provides access to various POSIX operating system API functions, including file I/O, process management, and other system-related calls.

char buffer[1024];: This declares an array named buffer of type char with a size of 1024 bytes. This buffer is intended to hold data read from or written to a file.

ssize_t bytes_read = read(fd, buffer, sizeof(buffer));: This line reads data from a file using the read() system call. It takes three arguments:

> **fd:** The file descriptor of the file to be read. (Presumably, fd is previously obtained from opening a file using open().)
> **buffer:** The buffer where the read data will be stored.
> **sizeof(buffer):** The size of the buffer, indicating the maximum number of bytes to read in a single operation.

The read() system call returns the number of bytes actually read. This value is stored in the variable bytes_read, which is of type ssize_t (a signed size type).

ssize_t bytes_written = write(fd, buffer, bytes_read);: This line writes data to a file using the write() system call. It takes three arguments:

> **fd:** The file descriptor of the file to be written.
> **buffer:** The buffer containing the data to be written.
> **bytes_read:** The number of bytes to write, as obtained from the previous read() operation.

The write() system call returns the number of bytes actually written. This value is stored in the variable bytes_written, which is also of type ssize_t. In summary, this code reads data from a file into a buffer and then writes the same data back to the file. The read() and write() system calls are used for these file I/O operations.

close(): The close() system call is used to close a file descriptor, releasing associated resources.

```
#include <unistd.h>

close(fd);
```

lseek(): The lseek() system call is used to change the file offset, which allows random access within a file.

```
#include <unistd.h>

off_t new_offset = lseek(fd, 0, SEEK_SET);
```

The above line of code uses the lseek() function to set the file offset of a file descriptor (fd) to the beginning of the file, and the resulting new offset is stored in the variable new_offset. Note that off_t is a data type used to represent file offsets in the C programming language.

stat() and fstat(): The stat() and fstat() system calls provide information about a file or file descriptor, respectively.

```
#include <sys/types.h>
#include <sys/stat.h>

struct stat file_info;
int result = stat("example.txt", &file_info);
```

struct stat file_info;: This line declares a variable named file_info of type struct stat. The struct stat structure is used to store information about a file, including details such as file size, inode number, permissions, and timestamps.

int result = stat("example.txt", &file_info);: This line calls the stat() system call to retrieve information about the file named "example.txt" and stores that information in the file_info structure.

"example.txt": The name of the file for which information is being retrieved.
&file_info: The address of the struct stat variable where the file information will be stored.

If the function succeeds, it typically returns 0; otherwise, it returns -1, indicating an error. After the call, the file_info structure contains various details about the file, such as its size, permissions, and timestamps.

unlink(): The unlink() system call is used to delete a file. It removes the file from the file system.

```
#include <unistd.h>

unlink("file_to_delete.txt");
```

mkdir(): The mkdir() system call is used to create a new directory. It takes the name of the directory to be created and an optional set of permissions.

```
#include <sys/stat.h>
#include <sys/types.h>
int result = mkdir("new_directory", 0755);
```

This line declares an integer variable result and initializes it with the result of the mkdir() system call. Here 0755 is the octal permission mode for the new directory which corresponds to read, write, and execute permissions for the owner, and read and execute permissions for others. If the directory creation is successful, the system calls returns 0; otherwise, it returns -1, indicating an error.

rmdir(): The rmdir() system call is used to remove an empty directory. It takes the name of the directory to be removed.

```
#include <unistd.h>

int result = rmdir("empty_directory");
```

rename(): The rename() system call is used to rename a file or directory. It takes the old path and the new path as arguments.

```
int result = rename("old_name.txt", "new_name.txt");
```

opendir(), readdir(), and closedir(): These system calls allow you to open a directory, read its entries, and close the directory when you're done.

```
#include <sys/types.h>
#include <dirent.h>

DIR *directory = opendir("/path/to/directory");
struct dirent *entry;

while ((entry = readdir(directory)) != NULL) {
    // Process directory entry
}
closedir(directory);
```

Example

This example demonstrates creating a directory, navigating into it, creating a file, writing data to the file, listing directory entries, removing the file, and then removing the directory. It uses various system calls, such as mkdir, chdir, open, write, close, opendir, readdir, unlink, and rmdir to perform these operations. Please note that error handling is included to catch potential issues.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>

int main() {
    // Create a directory
    int dir_create_result = mkdir("example_directory", 0755);
    if (dir_create_result == -1) {
        perror("mkdir");
        exit(EXIT_FAILURE);
    }

    // Change directory
    chdir("example_directory");

    // Create a file
    int file = open("example_file.txt", O_CREAT | O_RDWR, S_IRUSR |
S_IWUSR);
    if (file == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Write data to the file
    const char *data = "Hello, Linux System Calls!";
    ssize_t bytes_written = write(file, data, sizeof(data) - 1);

    if (bytes_written == -1) {
        perror("write");
        exit(EXIT_FAILURE);
    }

    // Close the file
    close(file);

    // List directory entries
    DIR *directory = opendir(".");
    if (directory == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    struct dirent *entry;
    while ((entry = readdir(directory)) != NULL) {
        printf("Found: %s\n", entry->d_name);
    }

    closedir(directory);

    // Remove the file
    int unlink_result = unlink("example_file.txt");
    if (unlink_result == -1) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }

    // Change back to the parent directory
    chdir("..");

    // Remove the directory
    int dir_remove_result = rmdir("example_directory");
    if (dir_remove_result == -1) {
        perror("rmdir");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Understanding Linux system calls and file system interaction is critical for systems programming on Linux. These tools provide you with the means to interact with the kernel, manage files and directories, and build powerful and efficient software.

Exercises

1. Create a program that navigates through a given directory and prints the names of all files and subdirectories it contains.
2. Develop a program that takes a directory path and a keyword as command-line arguments. The program should search for files containing the keyword in their content and print the file paths.
3. Write a program that compares the content of two files. Print the first differing line and the line number where the difference occurs.

Lab Session 14

Managing Linux users and groups

In Linux, effective user and group management is essential for maintaining system security and access control. Users are individuals who interact with the system, while groups are collections of users with shared privileges. Proper management ensures a structured and secure environment.

Creating Users Programmatically

Objective: Write a C program to create a new user on the system.

Steps:

- Use system commands (sudo useradd) to create a new user.
- Dynamically construct the command to include the username and any desired options.
- Check the return status to confirm the user creation.

Example 1

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

int main() {
    const char *username = "newuser";

    char command[100];
    snprintf(command, sizeof(command), "sudo useradd %s", username);

    // Execute the command
    int status = system(command);

    // Check the execution status
    if (status == 0) {
        printf("User %s created successfully with home directory.\n",
username);
    } else {
        fprintf(stderr, "Error creating user %s.\n", username);
    }

    return 0;
}
```

Creating Groups Programmatically

Objective: Write a C program to create a new group on the system.

Steps:

- Use system commands (sudo groupadd) to create a new group.
- Dynamically construct the command to include the group name and any desired options.
- Check the return status to confirm the group creation.

Example 2

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

int main() {
    const char *groupname = "newgroup";

    // Construct the command to create a new group
    char command[50];
    snprintf(command, sizeof(command), "sudo groupadd %s", groupname);

    // Execute the command
    int status = system(command);

    // Check the execution status
    if (status == 0) {
        printf("Group %s created successfully.\n", groupname);
    } else {
        fprintf(stderr, "Error creating group %s.\n", groupname);
    }

    return 0;
}
```

Adding Users to Groups Programmatically

Objective: Write a C program to add a user to a specific group.

Steps:

- Use system commands (sudo usermod -aG) to add a user to a group.
- Dynamically construct the command to include the username, group name, and any desired options.
- Check the return status to confirm the addition.

Example 3

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

int main() {
    const char *username = "newuser";
    const char *groupname = "newgroup";
    // Construct the command to add a user to a group
    char command[100];
    snprintf(command, sizeof(command), "sudo usermod -aG %s %s",
groupname, username);

    // Execute the command
    int status = system(command);

    // Check the execution status
    if (status == 0) {
        printf("User %s added to group %s successfully.\n", username,
groupname);
    } else {
        fprintf(stderr, "Error adding user %s to group %s.\n",
username, groupname);
    }

    return 0;
}
```

Example 4

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

int main() {
    // Specify the output file name
    const char *filename = "group_users.txt";

    // Construct the command to get information about all groups and
redirect the output to a file
    char command[100];
    snprintf(command, sizeof(command), "getent group > %s", filename);

    // Execute the command
    int status = system(command);

    // Check the execution status
    if (status == 0) {
        printf("Groups and users information written to %s.\n",
filename);
    } else {
        fprintf(stderr, "Error writing groups and users
information.\n");
    }

    return 0;
}
```

The above C program is designed to fetch information about all groups and their associated users from the system and write that information into a file named "group_users.txt".

Exercises

1. Write a C program to read user information from a file and create users accordingly.
2. Write a C program to change a user's password
3. Write a C program to list all users belonging to a specific group.
4. Write a C program to delete an existing user from the system.

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: Computer Engineering Workshop CS-219

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately..	The solution exhibits redundancy and partially covers the problem..	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor's Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: Computer Engineering Workshop CS-219

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately..	The solution exhibits redundancy and partially covers the problem..	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: Computer Engineering Workshop CS-219

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately..	The solution exhibits redundancy and partially covers the problem..	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	