**LOGICAL OPERATORS:**

* &&--- Operater delimits two commands. Second command is executed only if the first succeds.
* || --- Operater delimits two commands. Second command is executed only if the first fails.

**Ex:** $grep ‘director’ emp.lst && echo “pattern found”

$grep ‘manager’ emp.lst || echo “pattern not found”

**PROGRAM:**

if grep “^$1” |etc |password2 > |dev|null

then

echo “pattern found”

else

echo “pattern not found”

fi

**SYNTAX OF ELSE IF:**

if <condition is true>

then

<execute commands>

Elif <condition is true>

then

<execute commands>

then

<execute commands>

------

else …..

**OR**

if test $# -eq 0

then

echo “pattern found”

……

**RELATIONAL OPERATER:**

* Specify condition either using test or condition.

**Ex:** test $1 -eq $2 same as $1 -eq $2

**Relational Operaters:**

eq---- equal to

ne---- not equal to

gt----- greater than

gc----- greater than or equal to

lt------ less than

lc----- less than or equal to

**STRING OPERATERS:**

* -n --------- true, if str not a null string
* -z --------- true, if str is a null string

**Ex:** S1 = S2 , S1 != S2

**IN OPERATER:**

case <expression> in

<pattern 1> ) <execute commands>;;

<pattern 2> ) <execute commands>;;

<…..>

<…..>

**EX:**

echo “\n Enter option:\c”

Read choice

Case $choice in

<execute commands> …….

$ -wall ----- all kind of warnings

$ -w ------warning

$ -g -------we need to use when we are using source code for debugging

$ -o ------- output

$ -l ------- libraries

**EX:**

vi main.c

gcc -wall main.c -o app1

gcc -e main.c

--save temps msain.c ----- temporary compile process & save

**MAKE UTILITY:** The term "make utility" usually refers to a build automation tool that manages the compilation and linking of programs. In the context of software development, it helps automate the process of building executable programs from source code.

**EX:**

mkdir makeut --- to go into the directory

cd makeut/ ----- to get out from directory

mkdir bin obj inc src libs scripts

**vi calc.h: (IN inc)**

#ifndef CALC\_H

#define CALC\_H

int add(int,int);

int sub(int ,int);

#endif

**vi calc.c: (IN src)**

#include <calc.h>

int add(int v1,int v2)

{

return (v1+v2);

}

int sub(int v1,int v2)

{

return (v1-v2);

}

**vi main.c: (IN src)**

#include <stdio.h>

#include <calc.h>

int main()

{

printf("\n addition of two numbers: %d", add(10,20));

printf("\n sum of two numbers: %d", sub(10,20));

return 0;

}

**EXECUTE:**

gcc -c ./src/calc.c -o ./obj/calc.o ----- compile

Fatal Error----- gcc -c -I./inc ./src/calc.c -o ./obj/calc.o

cd src

src $: gcc -c -I../inc/.. /src/calc.c -o ../obj/calc.o

src $:gcc -c -I../inc/.. /src/main.c -o ../obj/main.o

cd ..

ll

day 04 $:gcc -o app ./obj/calc.o ./obj/main.o

ll

mv app ./bin/

./bin.app ----- output

**CODE COVERAGE ANALYSIS USING gcov:**

**How do we know we have tested out the code well?**

One measure is that we have executed each code statement at least once.

Code coverage tools, like gcov, help us know which lines of the code have been executed.(how many times)

Quality metrics & customer agreements often demand close to 100% code coverage reports.

1-----how many times the line executed

# ----- not executed

-(not touched or a blank space)