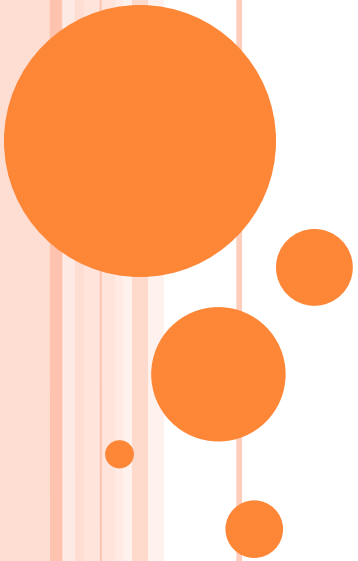
CS5354
UNIX TOOL PROGRAMMING



Program Development Tools

- A. Sawarkar

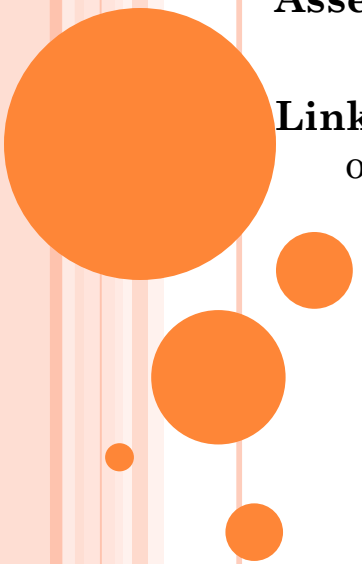
Program flow in backend

There are 3 phases a C program pass though before executabe is created

Compiling : Souce code (.c file) to assembly lang (.s file)

Assembling: Assembler code is transformed into object code (.o) by assembler

Linking: Object code of program is finally linked by the linker or loader with other object file and libraries that contain code used by function



CC compiler

cc or in GNU **gcc** is actually combine three phases.

E.g cc first.c second.c third.c

It create **a.out** not **.o** files

CC calls assembler (name **as)** to create the .o file before it invoke the linker (name **ld**) to create a single executable .

Using **-c** option you can create only the object files and without creating **a.out**

E.g **cc -c** first.c second.c third.c >>>>Creates first.o second.o third.o

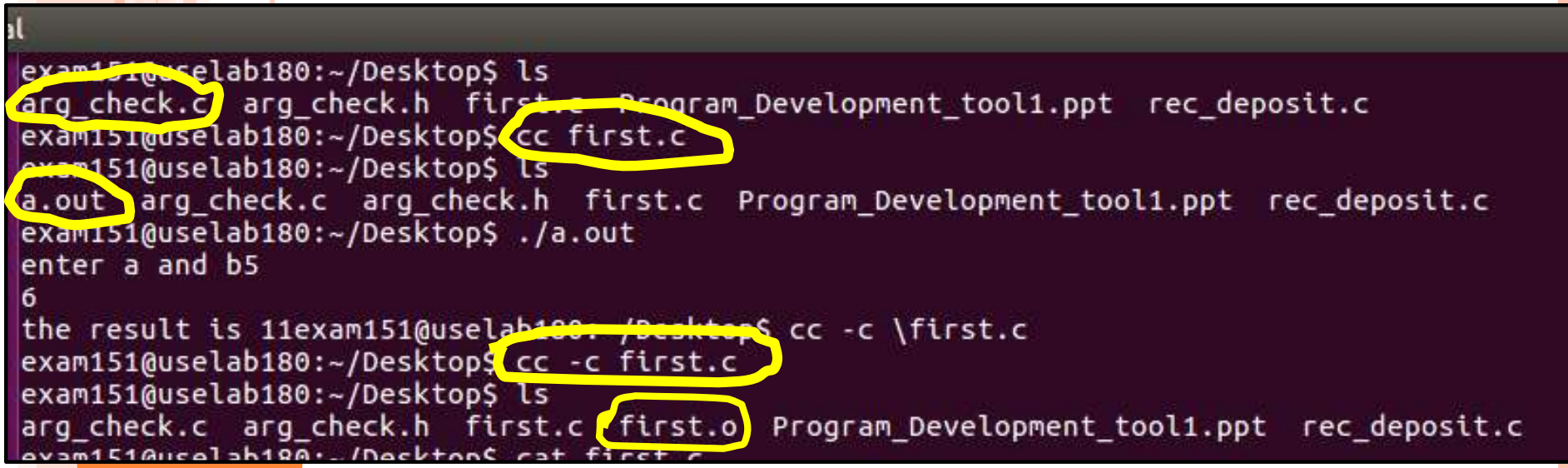
We can create our own executable file name

E.g **cc -o utp** first.c second.c third.c >>>> create a excutable name **utp**

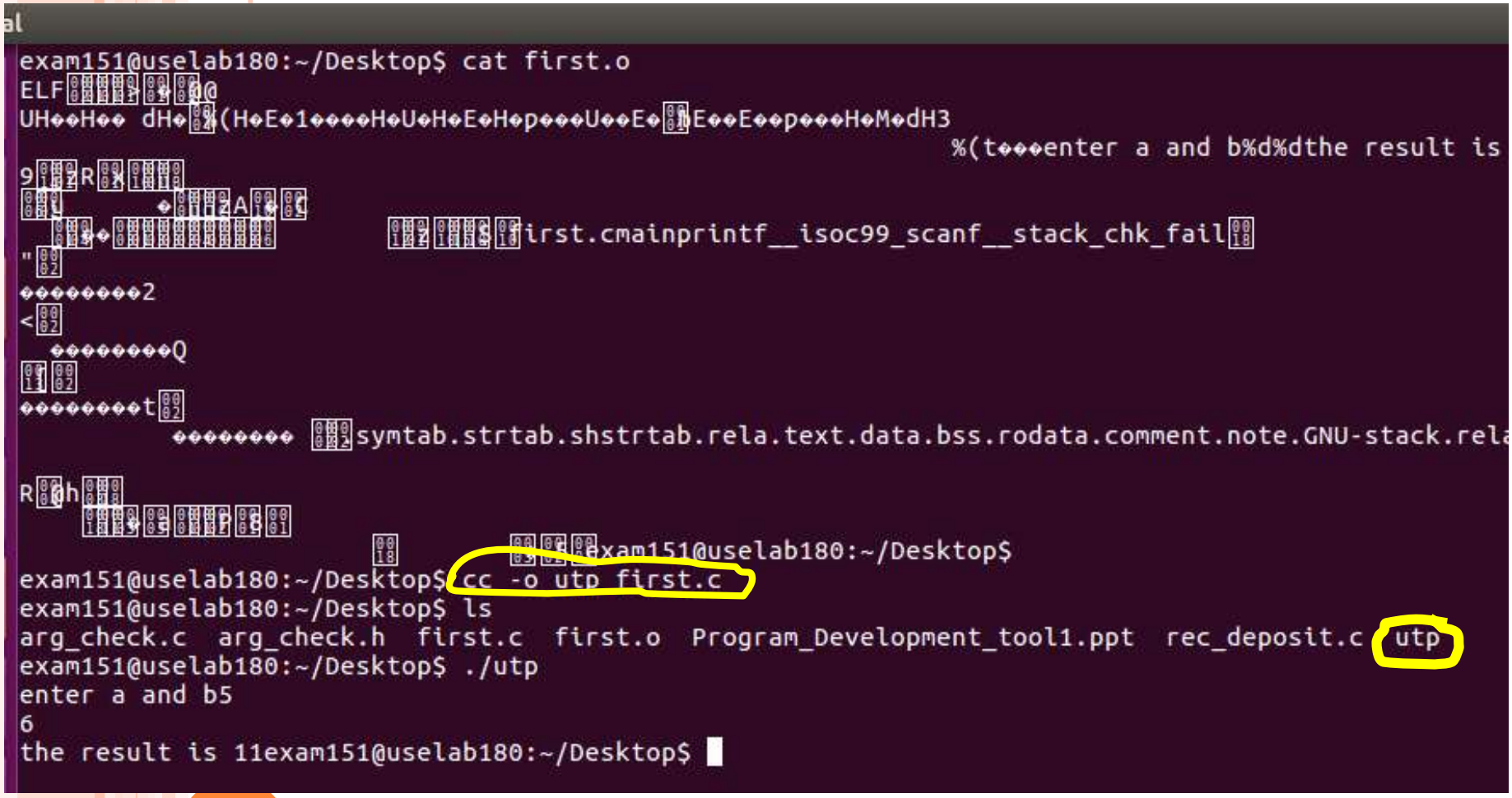
Note: ./a.out >>>>> ./utp

first.c

```
#include<math.h>
#include<stdio.h>
int main()
{
 int a,b,c;
  printf("enter a and b");
  scanf("%d%d",&a,&b);
  c=a+b;
 printf("the result is %d",c);
 return 0;
}
```



-c means create an object file .o extension



Content of object file is in not readable form

-o create an executable file that name given by you

Work with .c, .o and .h

Here we take an example for Recurring Deposit(RD)

Parameter you have to know.

Principal , Interest Rate, Term(no. Of year), Maturity Amount

E.g Principal =100, Rate =5%, Year 2, Maturity ?

Rate= (1+Rate/100) = 1.05

1st year Maturity= principal*pow(rate,1); >> 100*pow(1.05,1)>>105

2nd year Maturity= principal*pow(rate,2); 100*pow(1.05,2) >>100*1.1025
>>110.25

Total Maturity= 215.25

Using Loop

```
Maturity =0;
rate=1+rate/100;
for(i=1; i <= term; i++)
maturity=maturity + principal*pow(rate,i)

// compute function
Float compute(float principal, float rate, float term)
{
   int i;
   float maturity =0;
  rate=1+rate/100;
 for(i=1; i <= term; i++)
maturity=maturity+principal*pow(rate,i);
 retrun maturity;
}
```

Function **sscanf**

Syntax >>> **int sscanf (char * s, char * format, ...);**

Read formatted data from string

Reads data from **s** and stores them according to parameter **format** into the locations given by the additional arguments,

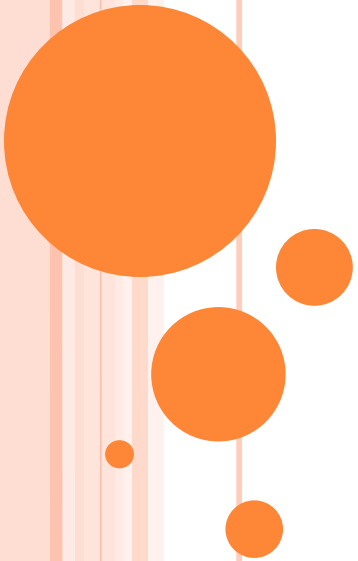
```
int main ()
 {
  int day, year;
   char weekday[20], month[20], dtm[100];
   strcpy( dtm, "Saturday March 25 2019" );
   sscanf( dtm, "%s %s %d  %d", weekday, month, &day, &year );
     printf("%s %d, %d = %s\n", month, day, year, weekday );
   return(0);
}
```

OUTPUT

March 25, 1989 = Saturday

Program with command line...

$ program_name.ext arg1 arg2 ....
$rec_deposit.c 100 5   2
Total Maturity= 215.25



Here we also seen how we create a header file
And how it will include in your program

*rec_deposit.c (~/Desktop) - gedit

Open Save

```c
#include<math.h>
#include "quit.h"
#include "arg_check.h"
float compute(float,float,float);

int main( int argc, char **argv)
{
   float pri,rate,term,sum;
    char *mesg="three argu req\n";
    char *mesg2="all must +ve";
arg_check(4,argc,mesg,1);
sscanf(argv[1],"%f",&pri);
sscanf(argv[2],"%f",&rate);
sscanf(argv[3],"%f",&term);

if(pri <=0 || rate <=0 || term <=0)
quit(mesg2,2);
sum=compute(pri,rate,term);
printf("maturity amount %f\n", sum);
exit(0);
}

float compute(float pri,float rate,float term)
{
int i;
float maturity=0;
rate=1+rate/100;
for(i=1;i<=term;i++)
maturity+=pri*pow(rate,i);
return maturity;
}
```

C Tab Width: 8 Ln 4, Col 1 INS

arg_check.c (~/Desktop) - gedit

Open Save

```c
#include "arg_check.h"
void arg_check (int args, int argc,
char *message, int exit_status)
{
   if(argc!=args)
   {
   fprintf(stderr,message);
   exit(exit_status);
   }
}
```

quit.c (~/Desktop) - gedit

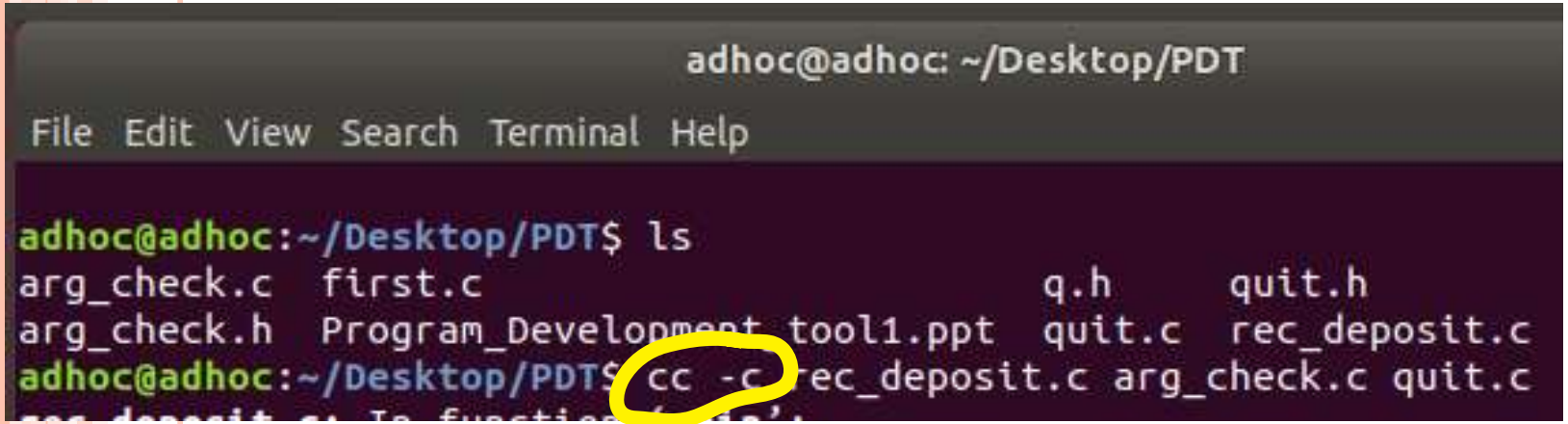
Open Save

```c
#include "quit.h"
void quit (char *message, int
exit_status)
{
   fprintf(stderr,message);
   exit(exit_status);
}
```

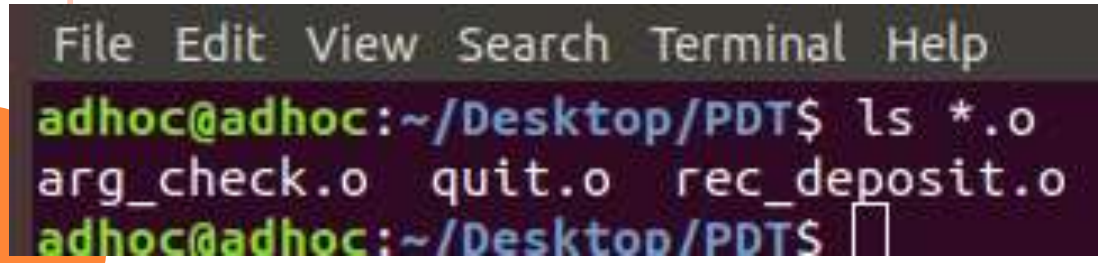
b Width: 8 Ln 6, Col 2 INS


Open
arg_check.h
~/Desktop/PDT
Save
1 #include<stdio.h>
2 void arg_check (int,int,char *, int);

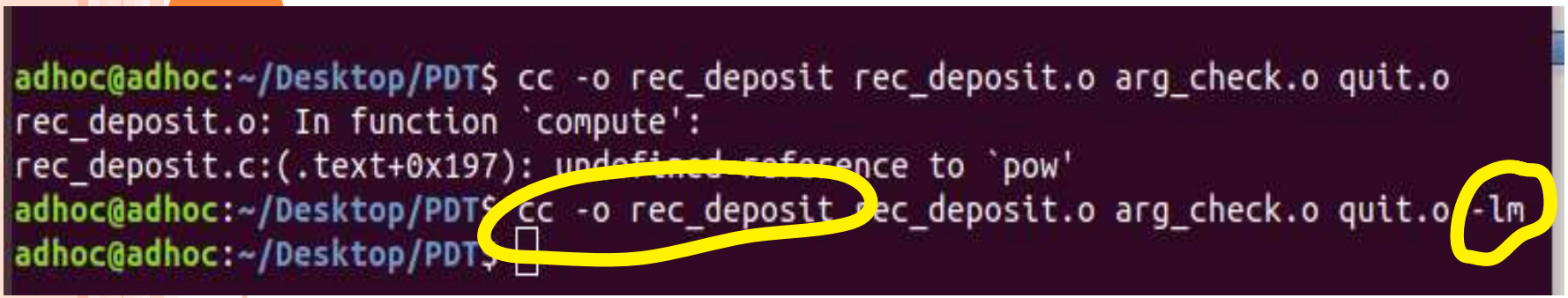

Open
quit.h
~/Desktop/PDT
Save
1 #include<stdio.h>
2 void quit (char*, int);
Tab Width: 8
Ln 1, Col 1
INS

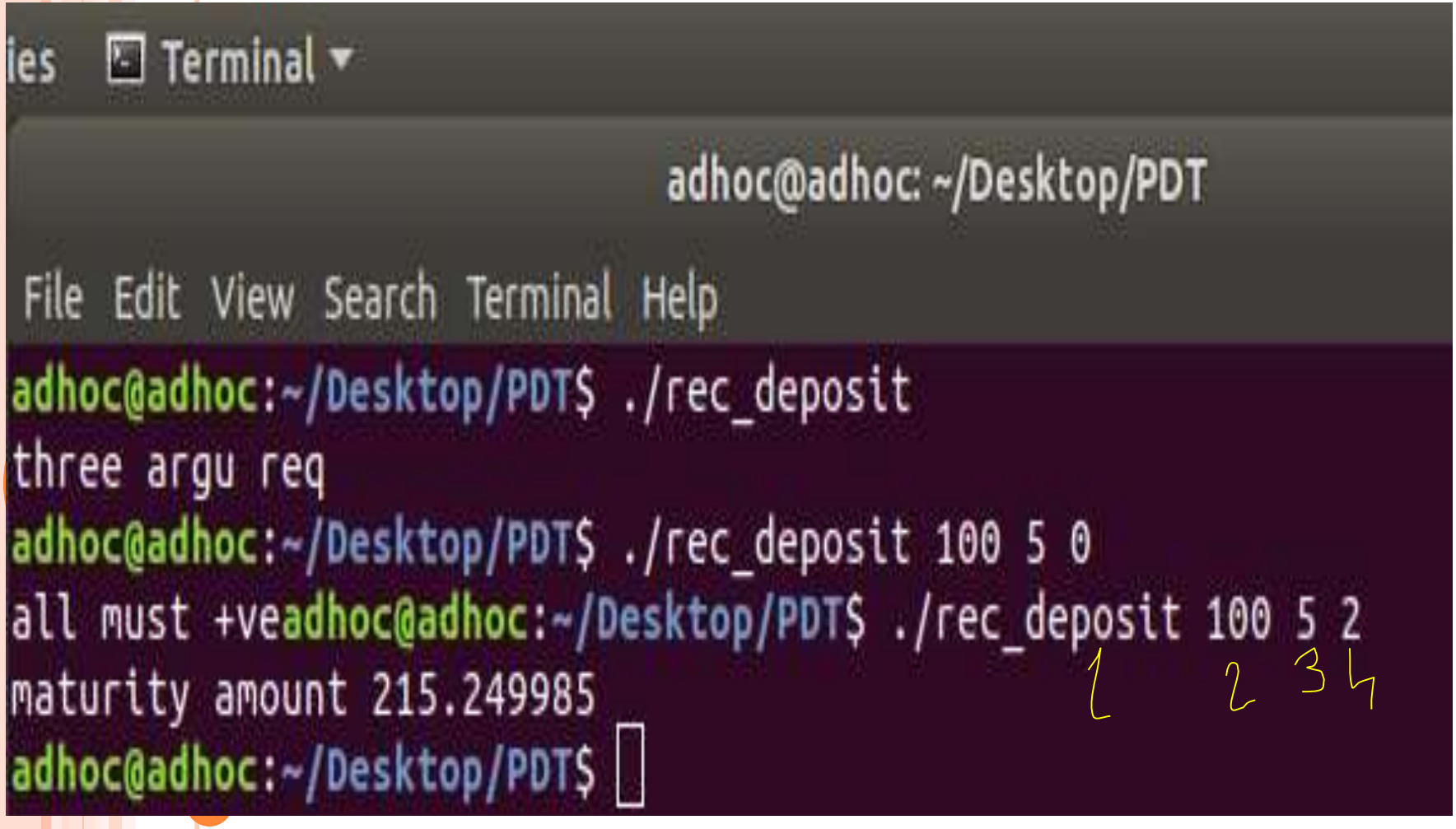


It create 3 object files



- Create a own exectable name **rec_deposit**




ies
Terminal
adhoc@adhoc: ~/Desktop/PDT
File Edit View Search Terminal Help
adhoc@adhoc:~/Desktop/PDT$ ./rec_deposit
three argu req
adhoc@adhoc:~/Desktop/PDT$ ./rec_deposit 100 5 0
all must +veadhoc@adhoc:~/Desktop/PDT$ ./rec_deposit 100 5 2
maturity amount 215.249985
adhoc@adhoc:~/Desktop/PDT$
1 2 3 4

make : Keeping program up-to-date

quit.o depends on >>> **quit.c** and **quit.h**

Now,if **quit.c or quit.h** is modify then quit.c must be **recompiled** to recreate **quit.o**.

i.e. **rec_deposit** is also depends on **quit.o**, so you have rebuilt as well. Same for **arg_check** module also.

Keeping track of these much dependencies in large application which have several file is impossible without **tool** to assist us.

For that we use **make** command

If program is short, recompiling only those sources (.c and .h) that have changed. It use **makefile** command.

makefile have two things.

1.How a program or object file has dependencies on other file

2. The command to execute when a file, on which another file depends, changes. (run cc command to do these job)

make by default uses a file named ***makefile*** in the user current directory

This file contains set of *rules* , rule is in the form of:

target: dependency_list >> Dependency

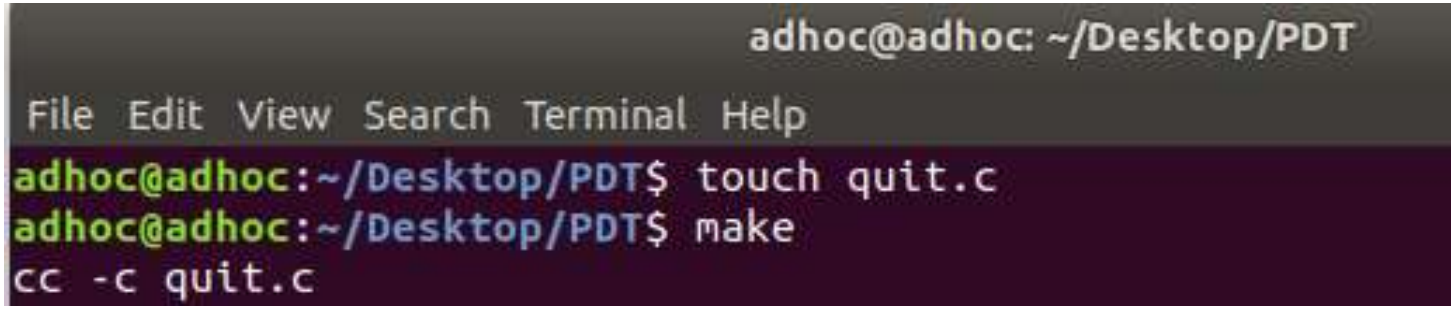
command_list >> the command to execute

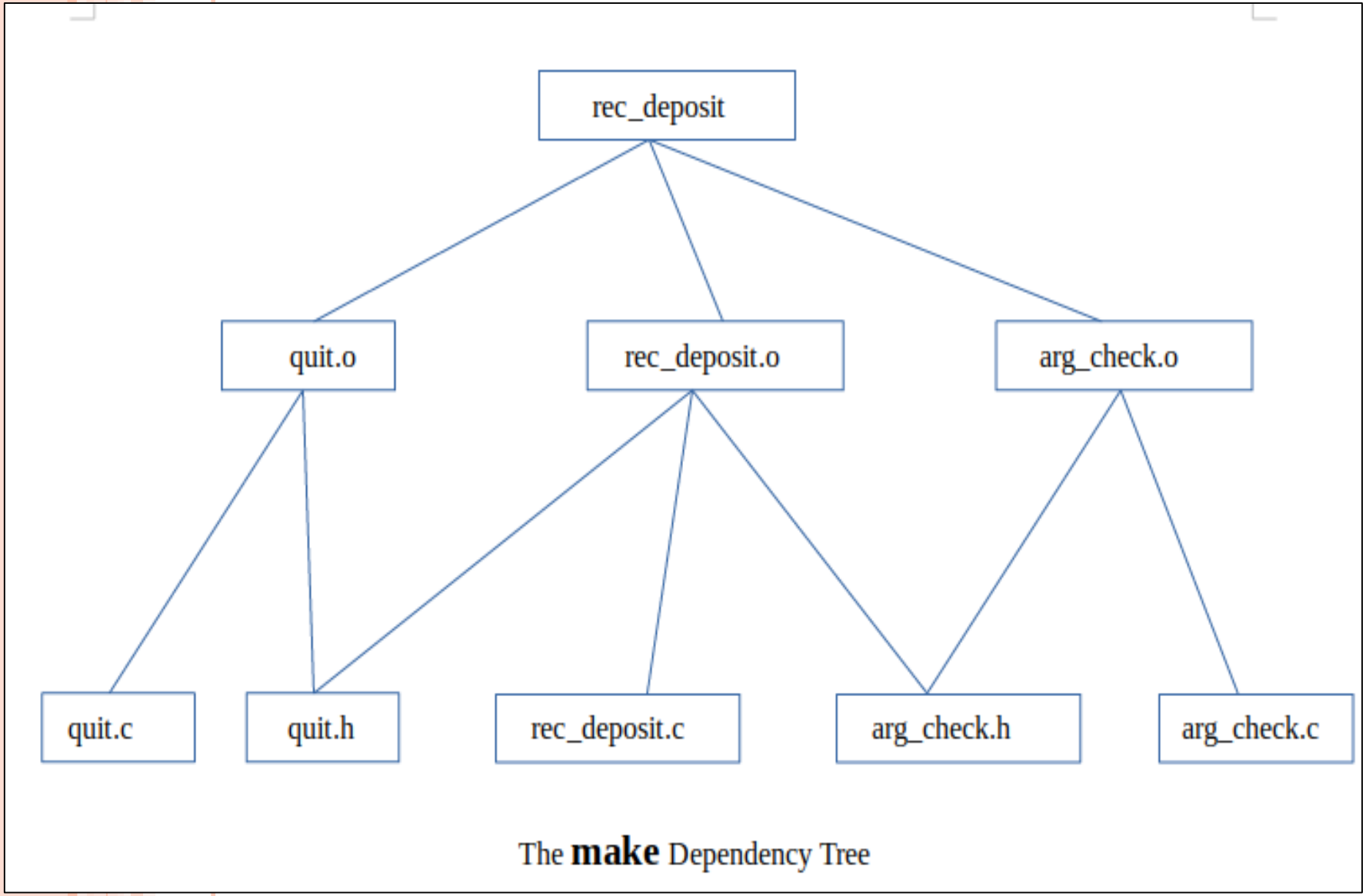
Note: **target** is executable or object file which depends on **dependency_list** .
If any file from **dependency_list** change , then target have to rebuilt using **command_list** preceded by **tab**(space)

- E.g

make rule for quit.o

```
quit.o: quit.c quit.h
	cc -c quit.c
```





The **make** Dependency Tree

makefile with all rules.

```
#rule1
rec_deposit: rec_deposit.o arg_check.o quit.o
        cc -o rec_deposit rec_deposit.o arg_check.o quit.o -lm

#rule2
rec_deposit.o: rec_deposit.c  quit.h arg_check.h
                        cc -c rec_deposit.c

#rule3
quit.o:  quit.c quit.h
        cc -c quit.c

#rule4
arg_check.o: arg_check.c  arg_check.h
        cc -c  arg_check.c
```

Removing Redundancies

1. If target and dependency have same **basename**, then *command_list* need not be specified

E.g

If **quit.o** has **quit.c** as its dependency, then need not specify

Cc -c quit.c.

i.e **quit.o : quit.c quit.h** >> No *command_list* required

2. If source file itself is omitted in the dependency , **make** assume that the base source filename is the object file.

i.e **quit.o: quit.h**

Function of make : cleaning and Backup

Make doesn't always need a dependency to work on ;
It can run a UNIX command depending on the command line arguments.
#instruct **make** to remove all object files or even a perform a backup with **tar**

$cat mkefile2

```
#All redundancies removed
rec_deposit: rec_deposit.o arg_check.o quit.o
cc -o rec_deposit rec_deposit.o arg_check.o quit.o -lm

rec_deposit.o:  quit.h arg_check.h      #>> Rule2
quit.o: quit.h                          #>>Rule3
arg_check.o: arg_check.h                #>>Rule4
clean:                                  #>>No dependecy list
        rm *.o
tar:                                    #>>No dependecy list
        tar -cvf progs.tar  *. c  *.h
```

Function of make : cleaning and Backup

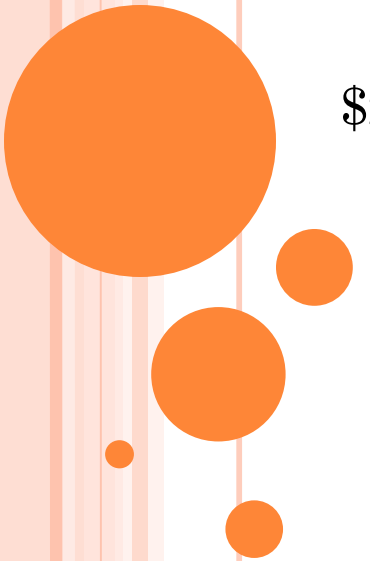
When **make** runs with clean as argument, it removes all .o files:

$make -f makefile2 clean

rm *.o

makefile clean remove all object files in current directory

$make -f makefile2 tar



Macros

make support **macros,** sometime called variables.

A **macro** is of the form *macroname=value* and is define at the beginning of the makefile.

A **macro** is invoked in a rule with $(*macroname) or ${macroname}*

$cat makefile3

```
CC = gcc
SOURCE = rec_deposit.c   arg_check.c  quit.c
OBJECTS = rec_deposit.o  arg_check.o  quit.o
 HEADERS = arg_check.h  quit.h

 rec_deposit: $(OBJECTS)
     $(CC) -o rec_deposit $(OBJECTS) –lm
rec_deposit.o : $(HEADERS)
arg_check.o: arg_check.h
          quit.o: quit.h
clean:
        rm $(OBJECTS)
tar:
    tar -cvf progs.tar  $(SOURCE) $(HEADERS)
```

Syllabus for Minor2

- Filters
- Regular Expression
- grep, pr, head ,tail, cut
- paste, sort, uniq, tr
- sed
- awk
- Backup: using tar and cpio
- Programming development tool: make

Ar: Buildinga library (Archive)

ar can manipulate an archive in the same way **tar** does, except that an **ar** archive has the extension, **.a**

Option:

-r: Add a file if it is not present in the archive or replace an existing one.

-q: Appends a file at the end of the archive.

-x: Extracts a file from the archive.

-d : Deletes a file in the archive.

-t : Display the table of contents of the archive

-v : Verbose output

E.g

name of archive

$ar -rv *librec.a* quit.o arg_check.o

r- quit.o

r- arg_check.o

ar: writing librec.a

Ar: Buildinga library (Archive)

We can add other object files with -q and then check table of contents with -t

$ar -qv librec.a compute.o

a- compute.o

ar: writing librec.a

$ar -tv librec.a

rw-r—r-- 1027 / 10 676 Mar 21 14:37 2019 quit.o

rw-r—r-- 1027 / 10 724 Mar 21 14:37 2019 arg_check.o

rw-r—r-- 1027 / 10 952 Mar 21 14:38 2019 compute.o

We can delete all three object file from the directory. We can **extract** a file from archive using **-x** or **remove** it from archive with **-d**

$ar -xv librec.a compute.o

x- compute.o

$ar -dv librec.a compute.o

d- compute.o

ar: writing librec.a

GDB : GNU Debugger

Gdb is a debugger for C (and C++).

- A good debugger is one of the most important tools in a programmer's toolkit.
- On a UNIX or Linux system, GDB (the GNU debugger) is a powerful and popular debugging tool.
- It lets you do whatever you like with your program running under GDB.
- It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point.
- Step through the program one line at a time and print out the values of each variable after executing each line.
- It uses a command line interface.

Compiling

- You have a general idea of programming with C or C++.
- You put a lot of cout or printf statements in the code if something goes wrong.
- You have used a debugger with an IDE, and are curious about how the command line works.
- You've just moved to a Unix-like operating system and would like to know about the tool chain better.
- To prepare your program for debugging with gdb, you must compile it with the **-g flag.**
- So, if your program is in a source file called **factmain.c** and you want to put the executable in the file **fmain**, then you would compile with the following command:

gcc -**g** factmain.c -**o** fmain

Invoking and Quitting GDB

To start gdb,

just type **gdb** at the unix prompt.

gdb will give you a prompt that looks like this: (gdb).

From that prompt you can run your program, look at variables, etc., using the commands listed below (and others not listed). Or, you can start gdb and give it the name of the program executable you want to debug by saying

gdb executable_filename

E.g **gdb fmain**

To exit the program just type **quit** at the (gdb) prompt (actually just typing q is good enough).

Basic GDB Commands

General Commands:

file [<file>]	selects <file> as the program to debug
run [<args>]	runs selected program with arguments <args>
attach <pid>	attach gdb to a running process <pid>
kill	kills the process being debugged
quit	quits the gdb program
help [<topic>]	accesses the internal help documentation

Stepping and Continuing:

c[ontinue]	continue execution (after a stop)
s[tep]	step one line, entering called functions
n[ext]	step one line, without entering functions
finish	finish the function and print the return value

Basic GDB Commands

help

Gdb provides online documentation. Just typing help will give you a list of topics. Then you can type help topic to get information about that topic Or you can just type help command and get information about any other command.

file

file executable specifies which program you want to debug.

run

run will start the program running under gdb. (The program that starts will be the one that you have previously selected with the file command, or on the unix command line when you started gdb.)

run 5 2 5 (3 argument for addition)

You can give command line arguments to your program on the gdb command line the same way you would on the unix command line, except that you are saying run instead of the program name:

You can even do input/output redirection: run > outfile.txt.

Basic GDB Commands

continue(c)

continue will set the program running again, after you have stopped it at a breakpoint.

step(s)

step will go ahead and execute the current source line, and then stop execution again before the next source line.

next(n)

next will continue until the next source line in the current function (actually, the **current innermost** stack frame, to be precise).

This is similar to *step*, **except** that if the line about to be executed is a function call, then that function call will be completely executed before execution stops again, whereas with *step* execution will stop at the first line of the function that is called.

GDB Breakpoints

A "**breakpoint**" is a spot in your program where you would like to temporarily stop execution in order to check the values of variables, or to try to find out where the program is crashing, etc. To set a breakpoint you use the break command.

Useful breakpoint **commands**:

b[reak] [<where>]	sets breakpoints. <where> can be a number of things, including a hex address, a function name, a line number, or a relative line offset
[r]watch <expr>	sets a watchpoint, which will break when <expr> is written to [or read]
info break[points]	prints out a listing of all breakpoints
clear [<where>]	clears a breakpoint at <where>
d[elete] [<nums>]	deletes breakpoints by number

GDB Breakpoints

break(b)

break ***function*** sets the breakpoint at the beginning of *function*. If your code is in multiple files, you might need to specify *filename*:*function*.

break ***linenumber*** or break ***filename:linenumber*** sets the breakpoint to the given line number in the source file. Execution will stop before that line has been executed.

delete(d)

delete will delete all breakpoints that you have set.

delete *number* will delete breakpoint numbered *number*. You can find out what number each breakpoint is by doing *info breakpoints*. (The command info can also be used to find out a lot of other stuff. Do *help info* for more information.)

clear

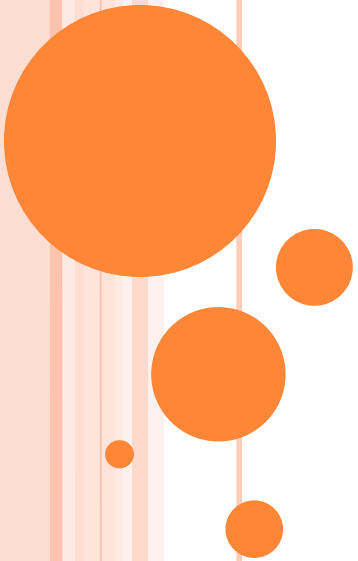
clear *function* will delete the breakpoint set at that *function*. Similarly for *linenumber*, *filename*:*function*, and *filename*:*linenumber*.

Commands

until

until is like next, except that if you are at the end of a loop, *until* will continue execution until the loop is exited, whereas next will just take you back up to the beginning of the loop.

This is convenient if you want to see what happens after the loop, but don't want to step through every iteration.



Playing with Data in GDB

Commands for looking around:

list [<where>]	prints out source code at <where>
search <regexp>	searches source code for <regexp>
backtrace [<n>]	prints a backtrace <n> levels deep
info [<what>]	prints out info on <what> (like local variables or function args)
p[rint] [<expr>]	prints out the evaluation of <expr>

Commands for altering data and control path:

set <name> <expr>	sets variables or arguments
return [<expr>]	returns <expr> from current function
jump <where>	jumps execution to <where>

list(l)

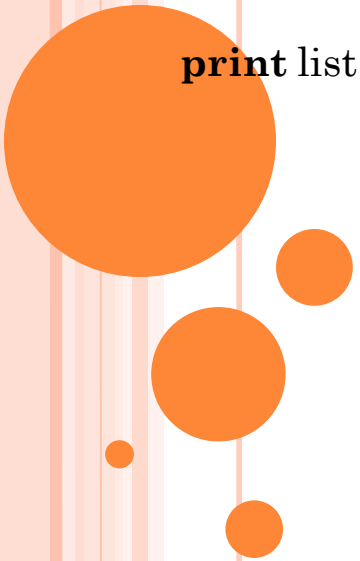
list linenumber will print out some lines from the source code around *linenumber*.

If you give it the argument *function* it will print out lines from the beginning of that function. Just list without any arguments will print out the lines just after the lines that you printed out with the previous *list* command.

print(p)

print expression will print out the value of the expression, which could be just a variable name. To print out the first 25 (for example) values in an array called list, do

print list[0]@25



Open ▾ fact... ~/Des... Save

```c
#include<stdio.h>
int factorial(int n);
 int main()
{
    int n,val;
    printf("enter n");
    scanf("%d",&n);
    val=factorial(n);
    printf("%d",val);
   return 0;
}

int factorial(int n)
{
    int f=1,i;
   for(i=1;i<=n;i++)
    {
        f=f*i;
    }
    return f;
}
```

Tab Width: 8 ▾ Ln 21, Col 2

File Edit View Search Terminal Help

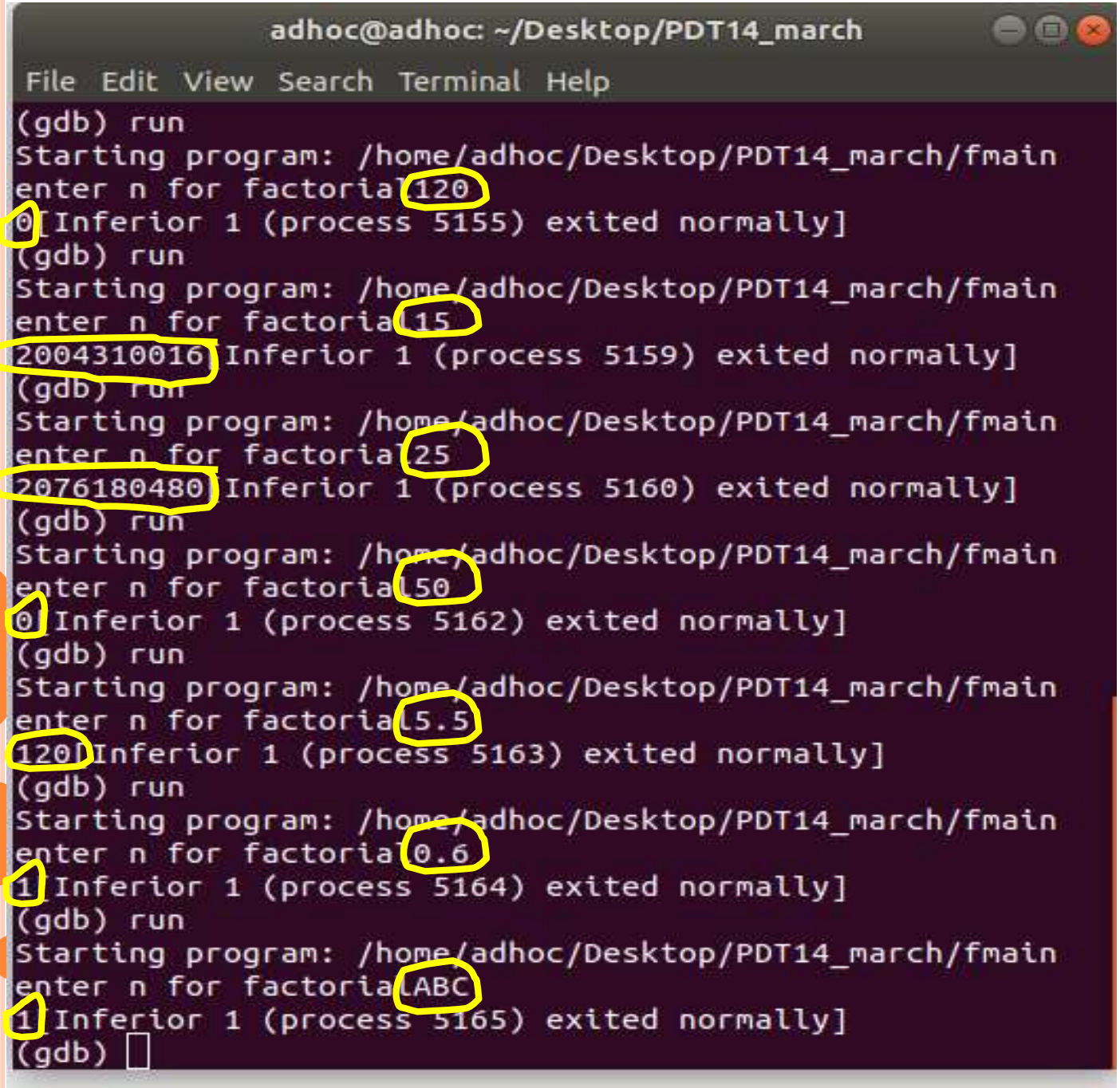
```
adhoc@adhoc:~/Desktop/PDT14_march$ gcc factorial_main.c
adhoc@adhoc:~/Desktop/PDT14_march$ ls
a.out          factorial_main.c  Program_Development_tool1.ppt  quit.h
arg_check.c    first.c           q.h                            rec_deposit.c
arg_check.h    main.cpp          quit.c                         reversefirst.c
adhoc@adhoc:~/Desktop/PDT14_march$ ./a.out
enter n for factorial5
120adhoc@adhoc:~/Desktop/PDT14_march$ gcc factorial_main.c -o factmain
adhoc@adhoc:~/Desktop/PDT14_march$ ls
a.out          factorial_main.c              q.h            reversefirst.c
arg_check.c    first.c                       quit.c
arg_check.h    main.cpp                      quit.h
factmain       Program_Development_tool1.ppt rec_deposit.c
adhoc@adhoc:~/Desktop/PDT14_march$ ./factmain
enter n for factorial5
120adhoc@adhoc:~/Desktop/PDT14_march$ gcc -g factorial_main.c -o fmain
adhoc@adhoc:~/Desktop/PDT14_march$ ./fmain
enter n for factorial5
adhoc@adhoc:~/Desktop/PDT14_march$
```

To run **gdb** use “run command”.

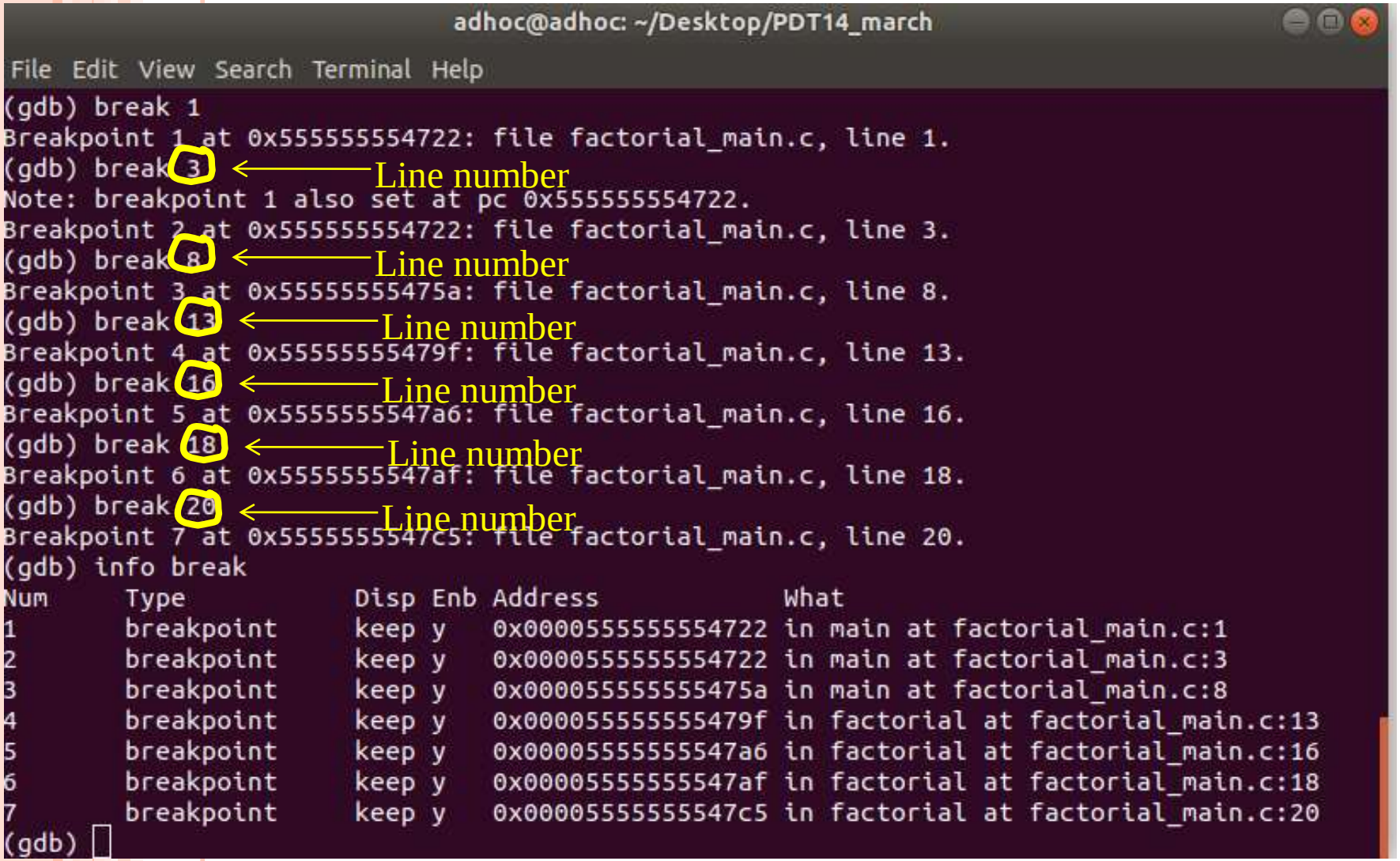
If there is no error then it show correct output.

If error , then

```
adhoc@adhoc:~/Desktop/PDT14_march$ gcc -g factorial_main.c -o fmain
adhoc@adhoc:~/Desktop/PDT14_march$ gdb fmain
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from fmain...done.
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial5
120 [Inferior 1 (process 5041) exited normally]
(gdb)
```


adhoc@adhoc: ~/Desktop/PDT14_march
File Edit View Search Terminal Help
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial120
0[Inferior 1 (process 5155) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial15
2004310016[Inferior 1 (process 5159) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial25
2076180480[Inferior 1 (process 5160) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial50
0[Inferior 1 (process 5162) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial5.5
120[Inferior 1 (process 5163) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorial0.6
1[Inferior 1 (process 5164) exited normally]
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain
enter n for factorialABC
1[Inferior 1 (process 5165) exited normally]
(gdb)

Now I am creating breakpoints



```
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fma

Breakpoint 1, main () at factorial_main.c:4
4       {
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb) next
6         printf("enter n for factorial");
(gdb) n
7           scanf("%d",&n);
(gdb) 5
Undefined command: "5".  Try "help".
(gdb) continue
Continuing.
enter n for factorial5

Breakpoint 3, main () at factorial_main.c:8
8           val=factorial(n);
(gdb) c
Continuing.

Breakpoint 4, factorial (n=5) at factorial_main.c:15
15          int f=1,i;
(gdb) c
Continuing.

Breakpoint 5, factorial (n=5) at factorial_main.c:16
16        for(i=1;i<=n;i++)
(gdb) print n
$1 = 5
(gdb) p i
```

```
$2 = 0
(gdb) c
Continuing.

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18              f=f*i;
(gdb) print i
$3 = 1
(gdb) jump 20
Continuing at 0x5555555547c5.

Breakpoint 7, factorial (n=5) at factorial_main.c:20
20          return f;
(gdb) print f
$4 = 1
(gdb) print n
$5 = 5
(gdb) n
21      }
(gdb) n
main () at factorial_main.c:9
9           printf("%d",val);
(gdb) n
10          return 0;
(gdb) n
11      }
(gdb) n
__libc_start_main (main=0x555555555471a <main>, argc=1, arg
v=0x7fffffffdf78,
    init=<optimized out>, fini=<optimized out>, rtld_fini=
<optimized out>,
    stack_end=0x7fffffffdf68) at ../csu/libc-start.c:344
344     ../csu/libc-start.c: No such file or directory.
(gdb)
```

By using contiue we go bebug program step by step i.e all break point with sequnace

```
Breakpoint 6, factorial (n=5) at factorial_main.c:18
18              f=f*i;
(gdb) print f
$4 = 1
(gdb) c
Continuing.

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18              f=f*i;
(gdb) c
Continuing.

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18              f=f*i;
(gdb) print f
$5 = 6
(gdb) c
Continuing.

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18              f=f*i;
(gdb) c
Continuing.

Breakpoint 7, factorial (n=5) at factorial_main.c:20
20          return f;
(gdb) print f
$6 = 120
(gdb) p i
$7 = 6
(gdb) c
Continuing.
120[Inferior 1 (process 5414) exited normally]
(gdb)
```

By using next we go line by line

```
The program is not being run.
(gdb) run
Starting program: /home/adhoc/Desktop/PDT14_march/fmain

Breakpoint 1, main () at factorial_main.c:4
4       {
(gdb) n
6         printf("enter n for factorial");
(gdb) n
7             scanf("%d",&n);
(gdb) n
enter n for factorial5

Breakpoint 3, main () at factorial_main.c:8
8             val=factorial(n);
(gdb) n

Breakpoint 4, factorial (n=5) at factorial_main.c:15
15            int f=1,i;
(gdb) n

Breakpoint 5, factorial (n=5) at factorial_main.c:16
16          for(i=1;i<=n;i++)
(gdb) n

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18                f=f*i;
(gdb) n
16          for(i=1;i<=n;i++)
(gdb) n

Breakpoint 6, factorial (n=5) at factorial_main.c:18
18                f=f*i;
(gdb) n
16          for(i=1;i<=n;i++)
```