TATA ELXSI
engineering creativity

TATA

C Programming

Presenter : Learning & Development Team

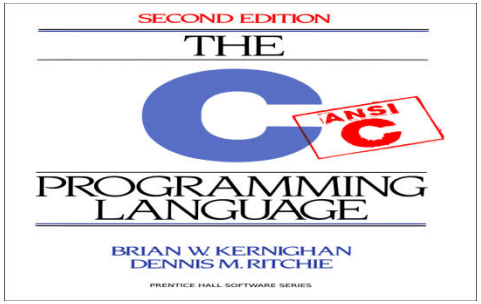
Day - 7

Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

  http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

ANSI C K&R



Day – 7 Agenda

- C Preprocessor
- #define, #include,
- #if, #ifdef, #ifndef, #else, #elif, #endif, #undef
- # and ##
- #pragma
- Inline functions
- __attribute__

C Preprocessor

C Preprocessor

- The preprocessor is a part of all C compiler packages.
- When you run the compiler, it automatically runs the preprocessor.
- The preprocessor changes your source code based on instructions, or preprocessor directives, in the source code.
- The output of the preprocessor is a modified source code file that is then used as the input for the next compilation step.

The #define Preprocessor Directive

- The #define preprocessor directive has two uses: creating symbolic constants and creating macros.

- Creating Function Macros with #define

```
#define   HALFOF(value)    ((value)/2)
```

- the program line :

printf("%f", HALFOF(x[1] + y[2])); is replaced by this line:

printf("%f", ((x[1] + y[2])/2));

The #define Preprocessor Directive

```
#define AVG5(v, w, x, y, z) (((v)+(w)+(x)+(y)+(z))/5)
```

The following macro, in which the conditional operator determines the larger of two values, also uses each of its parameters twice.

How about writing like this ? :

```
#define LARGER(x++, y++)   ((x) > (y) ? (x) : (y))
```

The #define Preprocessor Directive

- Look at the following example of a macro defined without parentheses:

#define SQUARE(x) x*x

- what if you pass an expression as an argument?

result = SQUARE(x + y);

- How to solve the expression evaluation in the above statement?

- Solution:
- #define SQUARE(x) (x) * (x)

Inbuilt Macro: Tokenizer [#]

//Converts expression into a constant string.

```
#include <stdio.h>
#define get(a) #a
int main()
{
    // GeeksQuiz is changed to "GeeksQuiz"
    printf("%s", get(GeeksQuiz));
}
```

Inbuilt Macro: Continuation operator [/]

- The macros can be written in multiple lines using '\'.
- The last line doesn't need to have '\'.

```
#define  PRINT(i, limit) while (i < limit) \
     { \
            printf("GeeksQuiz "); \
            i++; \
     }
int main()   {
      int i = 0;
      PRINT(i, 3);
      return 0;
}
```

Inbuilt Macro: concatenation operator (##)

- This operator concatenates, or joins, two strings in the macro expansion.

- It doesn't include quotation marks or special treatment of escape characters.
- For example, if you define and invoke a macro as

```
#define CHOP(x)    func ## x
salad = CHOP(3)(q, w);
```

- the macro invoked in the second line is expanded to
- salad = func3 (q, w);

Using #if, #elif, #else, and #endif

- These four preprocessor directives control conditional compilation.
- The term conditional compilation means that blocks of C source code are compiled only if certain conditions are met.
- In many ways, the #if family of preprocessor directives operates like the C language's if statement.
- The difference is that if controls whether certain statements are executed, whereas #if controls whether they are compiled.

Using #if, #elif, #else, and #endif

- The structure of an #if block is as follows:

```
#if condition_1
                    statement_block_1
#elif condition_2
                    statement_block_2
                    ...
#elif condition_n
                    statement_block_n
#else
                    default_statement_block
#endif
```

Example: #if, #elif, #else, and #endif

Suppose you're writing a program that uses a great deal of country-specific information.
This information is contained in a header file for each country.
When you compile the program for use in different countries.

```
#if   ENGLAND == 1
      #include "england.h"
#elif  FRANCE == 2
      #include "france.h"
#elif   ITALY == 3
      #include "italy.h"
#else
      #include "usa.h"
#endif
```

Conditional selection of code using #ifdef

The keywords for conditional selection are; #ifdef, #else and #endif.

#ifdef

takes a name as an argument, and returns true if the the name has a current definition. The name may be defined using a #define, the –d option of the compiler, or certain names which are automatically defined by the UNIX environment.

#else

is optional and ends the block beginning with #ifdef. It is used to create a 2 way optional selection.

#endif

ends the block started by #ifdef or #else.

Using preprocessor directives with header files.

```
    /* PROG.H – A header file with a check to prevent multiple includes! */

#if defined( PROG_H )
    /* the file has been included already */
#else
    #define PROG_H
    /* Header file information goes here... */
#endif
```

The #undef Directive

The #undef directive is the opposite of #define--it removes the definition from a name. Here's an example:

```
#define DEBUG 1
    /* In this section of the program, occurrences of DEBUG     */
    /* are replaced with 1*/
#undef DEBUG
    /* In this section of the program, occurrences of DEBUG  */
    /* are not replaced, */
```

You can use #undef and #define to create a name that is defined only in parts of your source code.

You can use this in combination with the #if directive, as explained earlier, for more control over conditional compilations.

Predefined Macros

Most compilers have a number of predefined macros.

The most useful of these are __DATE__, __TIME__, __LINE__, and __FILE__.

When the pre compiler encounters one of these macros, it replaces the macro with the macro's code.

__DATE__ and __TIME__ are replaced with the compile date and time.

By having a program display its compilation date and time, you can tell whether you're running the latest version of the program or an earlier one.

Predefined Macros

__LINE__ is replaced by the current source-file line number.

__FILE__ is replaced with the current source-code filename.

These two macros are best used when you're trying to debug a program or deal with errors.

```
printf( "Program %s: (%d) Error opening file ", __FILE__, __LINE__ );
```

Make utility

What is make file?

- A utility that determines dependencies in creation of targets
- Builds files that are out of date
- Not specific to any language
- Traditionally been used on UNIX platform
- Various ports exist
  - GNU - GNU make
  - Microsoft - nmake

Preparing and Running Make

- To prepare to use make, you must write a file called the *makefile* that describes the relationships among files in your program and provides commands for updating each file.

- Once a suitable makefile exists, each time you change some source files, this simple shell command:
- $ make

  reads and processes *makefile* in the current directory,

- $ make [-f mymake] [target]

  reads and processes my*make* in the current directory,

Make file rules

- A simple makefile consists of “rules” with the following shape:

target … : *prerequisites* …

    recipe [action]

    …

    …

- A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files.

Make file rules

- A target can also be the name of an action to carry out, such as ‘clean’.
- A *prerequisite* is a file that is used as input to create the target. A target often depends on several files
- A recipe may have more than one command, either on the same line or each on its own line.
- **note:** you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary.

What Makefiles Contain

- Makefiles contain five kinds of things:
  - *Rules*
    - *explicit rules*,
    - *implicit rules*,
  - *variable definitions*,
  - *Directives*
    - Inclusion of another make
    - Conditional directives
  - *comments*.
    - Text that follows **#** symbol is treated as comment

What Name to Give Makefile

- By default, when make looks for the makefile, it tries the following names, in order:
  - GNUmakefile,
  - makefile and
  - Makefile.

- If you want to use a nonstandard name for your makefile, you can specify the makefile name with the ‘-f’ or ‘--file’ option.

Rules

- Explicit rule
  - explicitly specify the prerequisites for a specific target
- Implicit rules
  - Take advantage of the knowledge *make* has about known patterns of files (e.g., .c, .cpp .o, .s …)

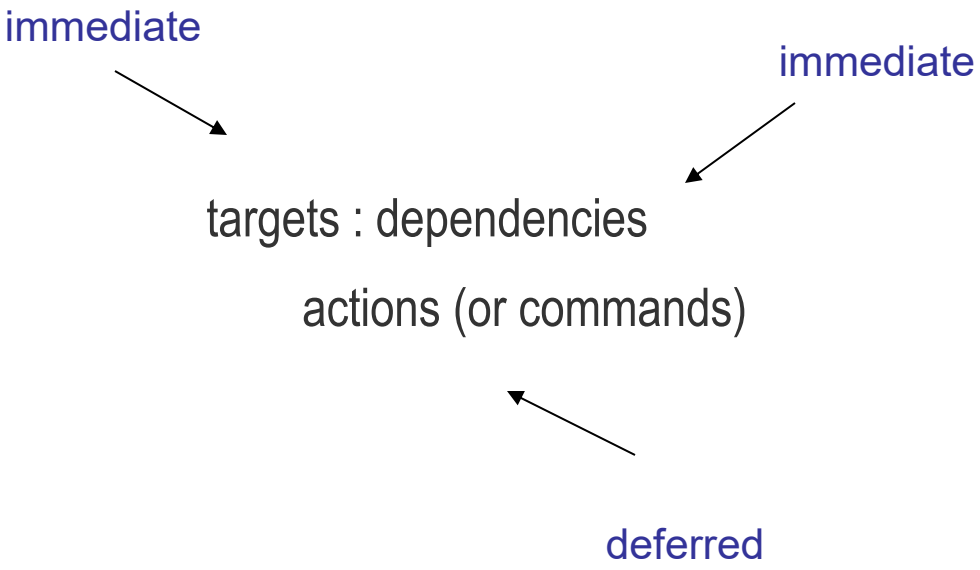
Processing of make file

- Two-phase processing
  - Phase-1
    - Read include files, if any
    - Expand variables and functions
    - Process conditional directives
    - Construct dependency graph
  - Phase-2
    - Determine which targets to be rebuild
    - Invoke necessary rules

Example

Processing of make file

- Immediate expansion
  - Expansion that happens during the first phase
- Deferred
  - Expansion that does not happen during the first phase



Multiple rules in a make file

```
target1 :
        echo processing rule 1
target2 :
        echo processing rule 2
target3 :
        echo processing rule 3
```

```
$ make  target3
echo processing rule 3
processing rule 3

$ make target1  target3
echo processing rule 1
processing rule 1
echo processing rule 3
processing rule 3
```

Prefixes to commands

Prefix	Action
-	ignore command errors, if any
@	don't' echo the command to standard output
+	execute the command even if –n option is selected

```
TARGET1 = $(SRC_DIR)
SRC_DIR = proj/src
TARGET4 := $(SRC_DIR)
show:
    +@echo this message is always
     show
     @echo TARGET1 is $(TARGET1)
      -rm   *.o
     rm   *.o
     -rm   *.o
```

How to Use Variables

- Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files

- Variable names are case-sensitive.

- To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces:

- Example:
- '$(foo)' or '${foo}' is a valid reference to the variable foo.

Variables in makefiles

- When file names are duplicated in makefile, it is good to use variables.

- *Variables* allow a text string to be defined once and substituted in multiple places later

```
CC=gcc
CFLAG=-c
OFLAG=-o
TARGETS=add.o sub.o mul.o mean.o

prog: prog.c myhead.h $(TARGETS)
        $(CC) $(OFLAG) prog prog.c  $(TARGETS)
```

Variables in makefile

- There are two ways that a variable in GNU make can have a value.

- The first flavor of variable is a *recursively expanded* variable.
  - Variables of this sort are defined by lines using '='

- For example,
- foo = $(bar)
- bar = $(ugh)
- ugh = Huh?
- All :; echo $(foo)

- will echo 'Huh?': '$(foo)' expands to '$(bar)' which expands to '$(ugh)' which finally expands to 'Huh?'.

simply expanded variables.

- To avoid all the problems and inconveniences of recursively expanded variables
- *Simply expanded variables* are defined by lines using ':=' or '::='

- It does not contain any references to other variables; it contains their values *as of the time this variable was defined*.

- Therefore,

```
x := foo
y := $(x) bar
x := later
```

Substitution References

- A *substitution reference* substitutes the value of a variable with alterations that you specify.

- When we say “at the end of a word”, we mean that *a* must appear either followed by whitespace or at the end of the value in order to be replaced;

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

sets ‘bar’ to ‘a.c b.c c.c’.

implicit rule

- It is not necessary to spell out the recipes for compiling the individual C source files.

- make can figure them out: it has an *implicit rule* for updating a '.o' file from a correspondingly named '.c' file using a 'cc -c' command.

```
edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h                         #implicit rule
kbd.o : defs.h command.h
command.o : defs.h command.h
```

TATA ELXSI
engineering creativity

Debugging Using GNU GDB

Presenter: Vikas karanth
Date: 17 – 12 -2015

Objective

- Features
- What is gdb?
- When can it be used?
- Advantages

- **Why Use A Debugger?**
- What is Syntactical errors?
- What is Semantical- errors ?
- What is Logical errors ?

Introduction to Gdb

- You can use gdb to debug programs written in C and C++.
- The purpose of a debugger such as gdb is to allow you to see what is going on “inside” another program
- Gdb functions somewhat like an interpreter for your programs.
  - Start your program, specifying anything that might affect its behaviour.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Invoking the "gdb" Debugger

- Before invoking the debugger. make sure you compiled your program (all its modules, as well as during linking) with the "-g" flag.

Step 1: $ gcc -g debug_me.c -o debug_me

Step 2: $ gdb debug_me

Step 3: (gdb) break main

Step 4: (gdb) run

help on debugger

- "help" with no arguments, you will get a list of help topics similar to the following:

- (gdb) help
- List of classes of commands:
- aliases -- Aliases of other commands
  - breakpoints -- Making program stop at certain points
  - data -- Examining data
  - files -- Specifying and examining files
  - internals -- Maintenance commands
  - obscure -- Obscure features
  - running -- Running the program
  - stack -- Examining the stack
  - status -- Status inquiries
  - support -- Support facilities
  - tracepoints -- Tracing of program execution without stopping the program
- Type "help" followed by a class name for a list of commands in that class.

- **To restart a program running in the debugger?**

- Use the **kill** command in gdb to stop execution.
- The you can use the run command as shown above to start it again.

- (gdb) kill
  Kill the program being debugged? (y or n) y
  (gdb) run ...

- **To exit the debugger**

  - Use the **quit** command.
  - (gdb) quit

- NOTE: You may be asked if you want to kill the program. Answer yes.
  - (gdb) quit The program is running. Exit anyway? (y or n) y
  - prompt >

How to run programs with the debugger

- First start the debugger with your program name as the first argument.
- $ gdb programname
- Next use the **run** command in gdb to start execution.
- Pass your arguments to this command.
- (gdb) run arg1 "arg2" ...

How do I restart a program running in the debugger?

- Use the **kill** command in gdb to stop execution.

- The you can use the run command as shown above to start it again.
- (gdb) kill

  - Kill the program being debugged? (y or n) y
  - (gdb) run ...

How to stop execution?

- You can stop execution by sending your program UNIX symbols like SIGINT.
- This is done using the **Ctrl-C** key combination.

```
(gdb) run
Starting Program: /home/trainee/a.out

Program received signal SIGINT, Interrupt. 0x80483b4 in main(argc=1,
    argv=0xbffffda4) at loop.c:5 5
while(1){
...
(gdb)
```

Continue

- **How to continue execution?**
- Use the **continue** command to restart execution of your program whenever it is stopped.
- **How to see where program stopped?**
- Use the **list** command to have gdb print out the lines of code above and below the line the program is stopped at.

How to step through the code line-by-line?

- First stop your program by sending it signals or using breakpoints.

- Then use the **next** and **step** commands.

```
5 while(1){
(gdb) next
7 }
(gdb)
```

- *NOTE* On a line of code that has a function call, next will go 'over' the function call to the next line of code, while step will go 'into' the function call.

- step *count*
  - Continue running as in step, but do so *count* times.
  - If a breakpoint is reached, or a signal not related to stepping occurs before *count* steps, stepping stops right away.

- next [*count*]
  - Continue to the next source line in the current (innermost) stack frame.
- until
- u
  - Continue running until a source line past the current line, in the current stack frame, is reached.

Examine variables

- Use the **print** command with a variable name as the argument.
- For example, if you have int x and char *s:

- (gdb) print x $1 = 900
- (gdb) print s $3 = 0x8048470 "Hello World!\n"
- (gdb)

- NOTE: The output from the print command is always formatted $## = (value).

- The $## is simply a counter that keeps track of the variables you have examined.

Modifying variables

- Use the **set** command with a C assignment statement as the argument.
- For example, to change int x to have the value 3:

```
(gdb) set x = 3 (gdb)
print x $4 = 3
```

- NOTE: in newer versions of gdb, it may be necessary to use the command 'set var', as in 'set var x = 3'

How to call functions linked into the program?

- From the debugger command line you can use the **call** command to call any function linked into the program.

- This includes your own code as well as standard library functions.
- For example, if you wish to have your program dump core:

```
(gdb) call abort()
```

How to return from a function?

- Use the **finish** command to have a function finish executing and return to it's caller.
- This command also shows you what value the function returned.

```
(gdb) finish
Run till exit from #0 fun1 () at test.c:5
main (argc=1, argv=0xbffffaf4) at test.c:17
17 return 0;
Value returned is $1 = 1
```

How to use the call stack

- The call stack is where we find the stack frames that control program flow.
- When a function is called, it creates a stack frame that tells the computer how to return control to its caller after it has finished executing.
- Stack frames are also where local variables and function arguments are 'stored'. We can look at these stack frames to determine how our program is running.
- Finding the list of stack frames below the current frame is called a backtrace

How do I get a backtrace?

- Use the gdb command **backtrace**.
- (gdb) backtrace

  #0 func2 (x=30) at test.c:5

  #1 0x80483e6 in func1 (a=30) at test.c:10

  #2 0x8048414 in main (argc=1, argv=0xbffffaf4) at test.c:19

  #3 0x40037f5c in __libc_start_main () from /lib/libc.so.6

  (gdb)

How do I change stack frames?

- Use the gdb command **frame**.
- Notice in the backtrace above that each frame has a number beside it.
- Pass the number of the frame you want as an argument to the command.

```
(gdb) frame 2
#2 0x8048414 in main (argc=1, argv=0xbffffaf4) at test.c:19
19 x = func1(x);
(gdb)
```

How do I examine stack frames?

- To look at the contents of the current frame, there are 3 useful gdb commands.
- **info frame** displays information about the current stack frame.
- **info locals** displays the list of local variables and their values for the current stack frame, and
- **info args** displays the list of arguments.

```
(gdb) info frame
Stack level 2, frame at 0xbffffa8c:
eip = 0x8048414 in main (test.c:19); saved eip 0x40037f5c
called by frame at 0xbffffac8, caller of frame at 0xbffffa5c source language c.

(gdb) info locals
x = 30
s = 0x8048484 "Hello World!\n"

(gdb) info args
argc = 1
argv = (char **) 0xbffffaf4
```

Breakpoints

How to use breakpoints?

- Breakpoints are a way of telling gdb that you want it to stop your program at certain lines of code.
- You can also have it stop when your program makes specific function calls.
- Once the program is stopped, you can poke around in memory and see what the values of all your variables are, examine the stack, and step through your program's execution.

How do I set a breakpoint on a line?

- The command to set a breakpoint is **break**.
- If you only have one source file, you can set a breakpoint like so:
  - (gdb) break 19
  - Breakpoint 1 at 0x80483f8: file test.c, line 19

- If you have more than one file, you must give the **break** command a filename as well:

  - (gdb) break test.c:19
  - Breakpoint 2 at 0x80483f8: file test.c, line 19

Set a breakpoint on a C function

- To set a breakpoint on a C function, pass it's name to break.
  - (gdb) break func1
  - Breakpoint 3 at 0x80483ca: file test.c, line 10
- Setting a breakpoint on a C++ function
  - However C++ is polymorphic, so you must tell break which version of the function you want to break on (even if there is only one).
  - To do this, you tell it the list of argument types.
- (gdb) break TestClass::testFunc(int)
- Breakpoint 1 at 0x80485b2: file cpptest.cpp, line 16.

How to use offset and address to set break point

- break +*offset*
- break –*offset*
  - Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected *stack frame*.
- break **address*
  - Set a breakpoint at address *address*.
  - You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

What happens when I give break without arguments

- **break**
  - When called without any arguments, break sets a breakpoint at the next instruction to be executed in the selected stack frame.
  - In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame
  - This is similar to the effect of a finish command in the frame inside the selected frame--except that , GDB stops the next time it reaches the current location;

- One nifty use for conditional breakpoints is to define a counter variable and break on a specified iteration.
- Perhaps the first 999 invocations of a method work fine, but something goes wrong after that.
- conditional breakpoint using a counter constructed from a **gdb** convenience variable.

(gdb) **set $count = 0**

(gdb) **break funMethod: if ++$count == 1000**

- supply an expression that's evaluated each time the breakpoint is crossed.

- Control stops at the breakpoint only if the expression is true.

(gdb) break 10 if i > 25

(gdb) break cut: if sender == NXApp

(gdb) break [MyTextField setStringValue:] if (!strcmp (newString, "Hello world"))

(gdb) break malloc if !NXMallocCheck()

How to list breakpoints?

- Use the **info breakpoints** command.
- (gdb) info breakpoints

```
Num Type           Disp Enb Address    What
2 breakpoint keep y 0x080483c3 in func2 at test.c:5
3 breakpoint keep y 0x080483da in func1 at test.c:10
```

- **set a temporary breakpoint**

- Use the **tbreak** command instead of break.
- A temporary breakpoint only stops the program once, and is then removed.

How do I disable breakpoints?

- Use the **disable** command.
- Pass the number of the breakpoint you wish to disable as an argument to this command.
- (gdb) disable 2
- (gdb) info breakpoints
- Num Type Disp Enb Address What
- 2 breakpoint keep n 0x080483c3 in func2 at test.c:5
- 3 breakpoint keep y 0x080483da in func1 at test.c:10

How to skip breakpoints?

- To skip a breakpoint a certain number of times, we use the **ignore** command.
- The **ignore** command takes two arguments: the breakpoint number to skip, and the number of times to skip it.
- (gdb) ignore 2 5
- Will ignore next 5 crossings of breakpoint 2.

How to use watchpoints

- Watchpoints are similar to breakpoints.
- Watchpoints are not set for functions or lines of code.
- Watchpoints are set on variables.
- When those variables are read or written, the watchpoint is triggered and program execution stops.

How to set a write watchpoint for a variable?

- watch *expr*
  - Set a watchpoint for an expression. GDB will break when *expr* is written into by the program and its value changes.
- This implies that the variable you want to set a watchpoint on must be in the current scope.
- So, to set a watchpoint on a non-global variable, you must have set a breakpoint that will stop your program when the variable is in scope.
- You set the watchpoint after the program breaks.

Example : Watch point

```
(gdb) watch x
Hardware watchpoint 4: x

(gdb) c
Continuing.
Hardware watchpoint 4: x

Old value = -1073743192
New value = 11
main (argc=1, argv=0xbffffaf4) at test.c:10 10 return 0;
```

How to disable watchpoints?

- Use the **info breakpoints** command to get this list.
- Then use the **disable** command to turn off a watch point, just like disabling a breakpoint.

- (gdb) info breakpoints

  Num Type Disp Enb Address What

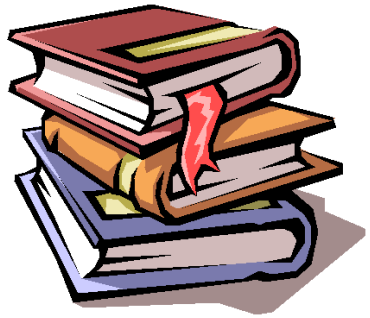
  1 breakpoint keep y 0x080483c6 in main at test.c:5

  breakpoint already hit 1 time

  4 hw watchpoint keep y x

  breakpoint already hit 1 time
- (gdb) disable 4

references



- The GNU C Reference Manual : https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

- The GNU C Reference Manual : http://www.cprogramming.com/

- **GDB: The GNU Project Debugger**
  https://www.gnu.org/software/gdb/

- **GNU make**
- **https://www.gnu.org/software/make/manual/make.html**

Thank you

TATA ELXSI

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com