

TP 5

# GDB

The purpose of this course is to study and to use a debugging tool of C programs : **gdb**.

## 1 GDB Debugger

A debugger such as GDB allows to see what is going on "inside" a program while it executes. It help catch bugs in the act :

- Start a program, specifying anything that might affect its behavior.
- Make a program stop on specified conditions.
- Examine and change the content of variables.

### • Invoking gdb

Before invoking the debugger. Programs (all its modules) must be compiled with the "-g" flag. This option produce debugging information required to debug. An example :

```
$ gcc -g -o file.exe file.c
```

Now we can **gdb** on the result :

```
$ gdb file.exe
```

The debugger launching provides information about used version and GNU licence. Next, the gdb prompt is displayed :

```
(gdb)
```

The program debugger begins now.

The debugger can be launched without specifying an executable file. Type just **gdb** to start. The executable file to be debugged is specified by **exec-file** command :

```
(gdb) exec-file file.exe
```

### • Quitting gdb

To exit gdb use the **quit** command (abbreviated **q**) :

```
(gdb) quit
```

For safe **gdb** can ask for confirmation :

```
The program is running. Exit anyway? (y or n)
```

### • Starting program

The execution of a program starts under gdb by using **run** command. For example :

```
(gdb) run [args]
```

starts the execution of **file.exe** and specifies the arguments (**args**) to give it.

Arguments can be given in file and redirected using shell redirection in the run command line :

```
(gdb) run < data.txt
```

**data.txt** is data file (contains arguments).

Similarly, the program's output can be redirected to an output file :

```
(gdb) run > result.txt
```

**result.txt** is an output file.

Sometimes, we want to show the arguments given to the program when it is started. This is done by **show** command. For example,

```
(gdb) show args
```

Then we'll get a message like :

```
Argument list to give program being debugged when it is started is "6".
```

**6** is the argument used to launch the program.

If no error detected, the program will run normally until the end, **gdb** will display :

```
Program exited normally.
```

```
(gdb)
```

## Debugging

In order to handle with the **gdb**, we implement a circular linked list.

### Exercise

- Define declaration type.
- Write the basis functions (create a node, check whether a list is empty or no..).
- Write a function to count the nodes in a circular list.

### • Printing variables and expressions

During a program debugging the content of its variables can be printed with :

```
(gdb) print result
```

**result** being a variable defined in the program.

Then we'll get a message like :

```
$1=8
```

that means **result** variable contains the number **8**. The symbol **\$1** is the name given to this variable and can be reused. For example the command :

```
(gdb) print $1*2
```

will display

```
$2=16
```

By default, **gdb** prints a value of a variable according to its data type. We might specify an output format when we print a value. Like :

```
(gdb) print /f result
```

**f** specifies that the output is a float, (other format letters are **c** a character, **d** an int, **o** a byte, **x** an hexa).

To check the type of variables we can use the command **whatis**. For example,

**(gdb) whatis result**

**Print** allows also to display contiguous memory zones. The command,

**(gdb) print result@length**

will display the value of **result** and the value of **length-1** memory zones. Obviously, this command is used to handle with arrays.

To avoid ambiguities between local and global variable names. They can be specified with their function names where they are defined. Like :

**(gdb) print func-name : :var-name**

**func-name** is a function name where **var-name** is defined.

## • Variables modification

In order to set the program's variable, use the **set variable** command :

**(gdb) set variable var-name = expr**

this command will assign the value of **expr** to the variable **var-name**.

This assignment can be done with **print** command :

**(gdb) print var-name = expr**

## • Functions

The function prototype (currently debugging) can be obtained by :

**(gdb) info func func-name**

For example,

**(gdb) info func fact**

We'll get a message like :

**All functions matching regular expression "fact" :**

**File exp.c :**

**double fact(double);**

which means the function **fact** is defined in **exp.c** file, takes one argument a **double** and return a **double**.

Functions can be called either by **print** command as :

**(gdb) print func-name(args)**

or by **call** command as :

**(gdb) call func-name(args)**

**func-name** is the name of the function and **args** is the list of call arguments.

## • Function call stack

Sometimes we wish to inspect what function is being executed now, which function called it, examine the content of variables, and so on. A **backtrace** command allows that. It gives a summary of how our program got where it is. It shows one line per frame (the data associated with one call to one function), for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack. As an example :

**(gdb) backtrace**

We will get a message like :

```
#0 fact (x=1) at exp.c :5
#1 0x0804846b in exp (x=2) at exp.c :36
#2 0x080484d8 in main () at exp.c :50
#3 0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
```

in that case the currently executing function is **fact()** with the argument **1**, at file **exp.c**, line 5. The function that called it is **exp(x=2)**, which is called by **main()** function. A similar command is **where**.

If we try to see the content of local variables, for example the variable named **resultat** defined in function **exp** :

```
(gdb) print result
No symbol "result" in current context.
```

Indeed the current function is **fact**. So, we can see contents of variables local to the calling function, and to any other function on the stack by typing the following two commands :

```
(gdb) frame 1
(gdb) print result
```

The **frame** command tells the debugger to switch to the given stack frame ('0' is the frame of the currently executing function). At that stage, any print command invoked will use the context of that stack frame.

We might also wish to move on the stack. For example to reach the function **exp**, we type :

```
(gdb) up
```

We will get :

```
#1 0x0804846b in exp (x=2) at exp.c :36
36 resultat += puiss(x, i) / fact(i);
```

The first line gives the function address, its name with the call parameters and the file where it is defined. The second gives the its body.

More generally, we can move *n* positions with **up n**. In the same way, we can move in opposite direction by **down n** command.

## • Break points

The problem with just running the code is that it keeps on running until the program exits. Precisely, by debugging we want to assess partial results before exiting. **Break** is a command for the debugger to stop the execution of the program before executing a specific source line (point). So we can set break points using two methods :

1. by specifying a function name to break every time it is being called, for example :

```
(gdb) break fact
Breakpoint 2 at 0x804836e : file exp.c, line 5.
```

this will set a break point right when starting function **fact()**. This point is the second break point and 5th line source file **exp.c**.

2. by specifying a specific line of code to stop in, for example :

```
(gdb) break 23
Breakpoint 3 at 0x80483fd : file exp.c, line 23.
```

the third break point is the line 23.

When we execute a program (after launching **run** command) a program stops whenever the first breakpoint is reached ; we can then check or modify the content of its variables.

To resume a program execution until it completes normally, we can use **continue** command :

```
(gdb) continue
Continuing.
```

```
...
```

The table of breakpoints can be listed by **info breakpoints**. For example :

```
(gdb) info breakpoints
Num Type      Disp Enb Address  What
1  breakpoint keep y  0x080484a3 in main at exp.c :46
2  breakpoint keep y  0x0804836e in fact at exp.c :5
3  breakpoint keep y  0x080483fd in exp at exp.c :23
```

For each breakpoints the following columns : breakpoint number, type, disposition (whether the breakpoint is marked to be disabled or deleted when hit), enabled or disabled (yes or no), address (memory address), what (in the source for your program, as a file and line number).

Breakpoint can be retrieved from the table thanks to **delete num-point-stop** command. For example :

```
(gdb) delete 2
```

deletes the second breakpoint. The breakpoint ranges specified as arguments. If no argument is specified, delete all breakpoints.

Rather than deleting a breakpoint, we can disable it by **disable number**. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can enable it again later.

## • Stepping

When we want to go through a program one line at a time and see if we can locate the source of the error. This is accomplished using the **step** command. Note that the debugger steps into functions that are called. If we don't want to do this, we can use **next** command instead of **step** which otherwise has the same behavior.

**Step** command will enter functions for debugging. However **finish** command will run to the end of the current function and display return value.

Another way to display the content of variables is **display** command (same syntax as **print**). This command allows to print out the value of an expression at each break point.

For example, for displaying the value of **result** we type :

```
(gdb) display result
```

To each displayed expression is associated a number. The command **info display** prints out the list of displayed expressions and corresponding number.

As for **disable** command, **display** can be cancelled by using **undisplay** command followed by the corresponding number (without the numbers, all display are deleted).

## • Commands execution at stop points

Often we have to compute the same list of commands at each stop point. We can run automatically the list by using **commands** command as :

```
(gdb) commands num-poin-stop
commande-1
...
commande-n
end
```

**num-point-stop** denote the number of stop point where we want run the list of commands. For example :

```
(gdb) commands 4
Type commands for when breakpoint 4 is hit, one per line.
End with a line saying just "end".
>silent
>echo valeur de x \ n
>print x
>continue
>end
```

which means at stop point **4** will issued the commands : **silent**, **echo...** When the program is issued, these commands are run at each crossing stop point ; particularly the command **continue** which allows to come to the next stop point.

Note command **commands** terminates with **ends**.

The command **silent** is used for deleting the message **Breakpoint ...** produced by the gdb when it reaches a stop point.

- **Command history**

As under Unix the history command can be activated by :

```
(gdb) set history expansion
```

Symbols **!!** calls up the lastest run command and **!char** calls up the last command beginning by **char**.

- **Interface with Shell**

Under gdb someone can run the shell commands (**cd**, **pwd** et **make**) :

```
(gdb) shell shell-command
```

- Basic gdb commands

Commande	Abv <sup>1</sup>	Action
backtrace	bt	indique où l'on se situe dans la pile des appels (synonyme de where)
break	b	pose un point d'arrêt à une ligne définie par son numéro ou au début d'une fonction.
clear	cl	détruit tous les points d'arrêt sur une ligne ou dans une fonction
commands		définit une liste de commandes à effectuer automatiquement à un point d'arrêt.
cond		ajoute une condition à un point d'arrêt
continue	c	continue l'exécution (après un point d'arrêt)
delete	d	détruit le point d'arrêt dont le numéro est donné
disable		désactive un point d'arrêt
disable disp		désactive un display
display		affiche la valeur d'une expression à chaque arrêt du programme
down		descend dans la pile des appels
enable		réactive un point d'arrêt
enable disp		réactive un display
file		redéfinit l'exécutable
finish		termine l'exécution d'une fonction
frame		permet de se placer à un endroit donné dans la pile des appels et affiche le contexte
help	h	fournit de l'aide à propos d'une commande
info breakpoints	i b	affiche les points d'arrêt
info display		donne la liste des expressions affichées par des display
info func		affiche le prototype d'une fonction
next	n	exécute l'instruction suivante (sans entrer dans les fonctions)
run	r	lance l'exécution du programme (par défaut avec les arguments utilisés précédemment)
print	p	affiche la valeur d'une expression
ptype		détaille un type structure
quit	q	quitte gdb
set history expansion		active l'historique des commandes
set variable		modifie la valeur d'une variable
shell		permet d'exécuter des commandes shell
show args		affiche les arguments du programme
show values		réaffiche les 10 dernières valeurs affichées
step	s	exécute l'instruction suivante (en entrant dans les fonctions)
undisplay		supprime un display
up		monte dans la pile des appels
whatis		donne le type d'une expression
where		indique où l'on se situe dans la pile des appels (synonyme de backtrace)