CS2211a Lab No. 6
Introduction to C
Tuesday November 1, 2016 (sections 2, 3 and 4),
Wednesday November 2, 2016 (sections 6 and 8),
Thursday November 3, 2016 (sections 5 and 9)

Location: ***MC10*** lab

The objective of this lab is:

- to practice the *xxgdb* debugging tool
- to practice the use of loop statements, as well as calling some C standard functions

If you would like to leave, and at least 30 minutes have passed, raise your hand and wait for the TA.
Show the TA what you did. If, and only if, you did a reasonable effort during the lab, he/she will give you the lab mark.

==========

Debugging C Programs

When you develop a C program, you need to compile it and make it syntax error-free before you can run it. After doing so, you may get an unexpected output. You may also find that your program occasionally crashes. This is what we call runtime errors. Such errors are not always easy to locate. The process of identifying the runtime errors in your code and fixing them is called debugging your program.

- The first step in debugging your program is to attempt to reproduce the problem, .i.e., identifying the set of inputs that lead to the problem.
- After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug. Such simplification can be made manually, using a divide-and-conquer approach, where you will try to remove some parts of original test case and check if the problem still exists.
- After the test case is sufficiently simplified, you can carefully examine program states (values of variables and the order of execution) and track down the origin of the problem(s).

The simplest way to do so is to insert some `printf` statements here and there. This is sometimes called *printf* debugging. While it is simple to do, it becomes a boring and confusing process when you have a complex bug. Alternatively, there are software tools that allow you to inspect what the program is doing at a certain point ***during execution***. These programs include, *gdb*, *dbx*, *xxgdb*. Errors like *segmentation faults* may be easier to find with the help of such tools.

gdb and *dbx* debuggers allow you to see what is going on *inside* another program while it executes -- or what another program was doing at the moment it crashed.

They can do four main things (plus other things in support of these) in order to help you catch bugs in the act:

- ***Start*** your program.
- ***Stop*** your program on specified conditions.
- ***Examine*** what has happened, when your program has stopped.
- ***Change*** things in your program, so you can experiment the effects of one bug and go on to learn about another.

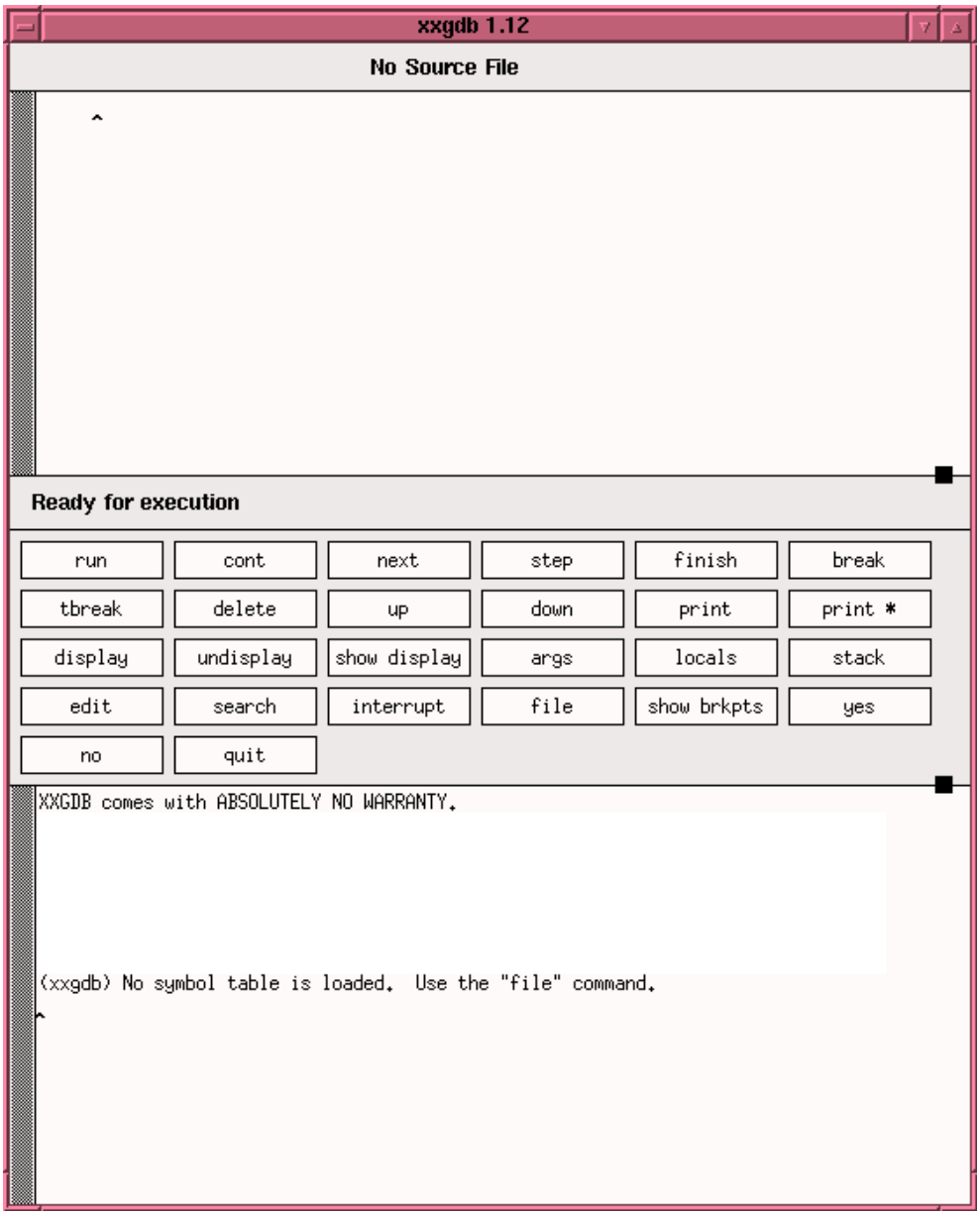
xxgdb is a source code debugging system. It is a graphical user interface to the *gdb* debugger under the X Window System. It provides visual feedback and mouse input for the user to control program execution through breakpoints, to ***examine*** and ***traverse*** the function call stack, to ***display*** values of variables as well as data structures, and to ***browse*** source files as well as functions.

To use such *debuggers*, the program must be compiled using the `-g` compiler flag set. For example, the command:

```
gcc -g -o prog prog.c
```

compiles the file `prog.c`. The `-g` flag causes the compiler to *insert information* into the executable file (named `prog`) that *gdb* and *xxgdb* require in order to work.

To invoke *xxgdb* in the background, execute `xxgdb&` in the command prompt which will open the following screen:



The main *xxgdb* window can be thought of as three subdivided windows:

- The first part is called the *source window* because the source file is displayed in this part.
- The second part is called the *command window* where essentially all the *xxgdb* commands are present. To invoke any of them, simply click on the appropriate button.
- The third part is the *dialog window* which provides a typing interface to the *xxgdb*.

It is worth mentioning that once you click on a button in the command window, the equivalent *gdb* command is generated and executed in the dialog window.

To load a file to the debugger, click the *file* command button in the *Command Window* and select the *executable* file `prog`. The source code of `prog.c` appears on the window. You can also invoke *xxgdb* followed by the name of the executable file as an argument to the command.

To know how to use the various *xxgdb* commands, key in *help <command name>*. For example, if you key in `help run` in the *Dialog Window*, you will get the description of the *run* command.

Practicing *xxgdb*

To learn how to debug a C program, let us create the following C program that calculates and prints the factorial of a number. This C program contains some errors for our debugging purpose.

```
#include <stdio.h>

int main()
{
        int i, num, j;

        printf ("Enter the number: ");
        scanf ("%d", &num );

        for (i=1; i<num; i++)
                j=j*i;

        printf("The factorial of %d is %d\n",num,j);

        return 0;
}
```

Below is what you will get if you attempt to execute this program:

```
$ ./a.out
Enter the number: 3
The factorial of 3 is -8392248
```

In order to debug this program, you should complete the following steps:

1. Compile your C program with `-g` option. This allows the compiler to collect the debugging information.
   `$ gcc -g factorial.c`
   Note: The above command creates an `a.out` file which will be used during the debugging process, as shown below.
2. Launch the C debugger (*xxgdb*) as shown below.
   `$ xxgdb a.out &`
3. Place break points in the C program where you suspect errors. While executing the program, the debugger will stop at the break points, and give you the prompt to debug.
   So before starting up the program, let us place a break point in our program by clicking on the *main* function and then clicking on the *break* button. You can also set break points in your program by clicking at the beginning of a line you wish to break at and then clicking on the *break* button.
   You can show the current break points by clicking on *show brkpts* button.
   Let us put another breakpoint *at* `j=j*i;` line.
4. Run the program by clicking on the *run* button, or type *run* in the dialog window. You can also give the command line arguments to the program via *run args*. *N.B: the example program we used here does not require any command line arguments.*
   Once you have executed the C program, it will execute until the first break point, and then give you the prompt for debugging.
   You can use various *xxgdb* commands to debug the C program.
   For the time being, as we are at the beginning of the program, we will proceed to the second break point by clicking on the *cont* button. Yes, it is the *cont* button not the *run* button, as *run* will restart the program from the beginning.
   To enter the value to the `scanf` statement, you have to click on the dialog window and then enter the data.
5. Print the values of various variables that you want checked by highlighting the variable and then clicking on the *print* button, or just executing *print* <*variable_name*> in the dialog window, or simply *p* <*variable_name*>. In our case, print the values of `i`, `j`, and `num`.
   As you see, in the program, we have not initialized the variable `j`. So, it gets a garbage value resulting in a big number as the factorial value.
   Fix this issue by initializing variable `j` with `1`, compile the C program and execute it again.

6. Keep looping in this cycle until having a program which is free from any semantic errors. This will eventually produce a program that gives the correct answer every time.
   In our example, even after performing the fix above, there seems to be some problems in the program, as it still gives a wrong factorial value. So, you need to recompile it again with `-g`, run *xxgdb* again, put suitable *breakpoins* here and there, and start running the program again.

When you stop at a break point, you may choose to do one of the following actions to resume execution:

- Click on the *cont* button: debugger will continue executing until the next break point.
- Click on the *next* button: debugger will execute the next line as single instruction.
- Click on the *step* button: same as *next*, but does not treats a function as a single instruction, instead goes into the function and executes it line by line.

By *continuing* or *stepping through*, you would have found that the issue in the program is because we have not used the `<=` in the `for` loop condition checking. So changing `<` to `<=` will solve the issue.

During this lab you should get familiar with *xxgdb* so that you can *fluently* use it from now on whenever developing any C program in the future.

Below, you will find a summary of all *xxgdb* commands and a brief description for each one of them.

Command buttons

Execution Commands

Run	Begin program execution.
Cont	Continue execution from where it stopped.
Next	Execute one source line, without stepping into any function call.
Step	Execute one source line, stepping into a function if the source line contains a function call.
Finish	Continue execution until the selected procedure returns; the current procedure is used if none is selected.

Breakpoint Commands

Break	Stop program execution at the line or in the function selected. To set a breakpoint in the program, place the caret at the start of the source line or on the function name and click the *break* button
Tbreak	Set a breakpoint enabled only for one stop. This is the same as the *break* button except the breakpoint is automatically disabled the first time it is hit.
Delete	Remove the breakpoint on the source line selected or the breakpoint number selected.
Show brkpts	Show the current breakpoints (both active and inactive).

Stack Commands

Stack	Show a stack trace of the functions called.
Up	Move up one level on the call stack.
Down	Move down one level on the call stack.

Data Display Commands

Print Print the value of a selected expression.

Print * Print the value of the object the selected expression is pointing to.

Display Display the value of a selected expression in the display window, updating its value every time the execution stops.

Undisplay Stop displaying the value of the selected expression in the display window. If the selected expression is a constant, it refers to the display number associated with an expression in the display window.

Args Print the arguments of the selected frame.

Show display Show the names of currently displayed expressions

Locals Print the local variables of the selected frame.

Stack Print a backtrace of the entire stack.

Miscellaneous Commands

Search Pop up a search panel which allows both forward (>>) and reverse (<<) searches of text strings in the source file. Hitting carriage return after entering the search string will begin a forward search and pop down the search panel.

File Pop up a directory browser that allows the user to move up and down in the directory tree, to select a text file to be displayed, to select an executable file to debug, or to select a core file to debug. Directory entries are marked with a trailing slash (‘/’) and executables with a trailing asterisk (‘*’). Filenames beginning with a dot (‘.’) or ending with a tilde (‘~’) are not listed in the menu.

Yes Send ‘y’ (yes) to *gdb* (to be used when *gdb* requires a yes/no response).

No Send ‘n’ (no) to *gdb* (to be used when *gdb* requires a yes/no response).

Quit Exit *xxgdb*.

Practicing loops and calling some standard functions

1. There are 9,870 people in a town whose population increases by 10 percent each year. Write a loop that displays the annual population and determines how many years it will take for the population to surpass 30,000. Your program should accept the current number of people as an input.

2. The following program uses the `sqrt` function from the math library; hence, `math.h` must be included at the beginning of the program (Read `man 3 sqrt`). To compile such program, you have to inform the compiler to use the math library during the linking step. To do so, you should use "**gcc prog.c –lm**" Unfortunately, there are some errors in this code. Type the program, compile it, fix its errors, and run it correctly.

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

int main()
{
  int i;

  printf("Number \t Square Root\n\n");

  for (i=0, i<=30, ++i);
       printf("%d \t\t %d \n",i, sqrt(1.0 * i));

  return 0;
}
```

3. Write a program to read a positive integer value, and compute the following sequence:
   - If the number is even, halve it;
   - if it is odd, multiply by 3 and add 1.

   Repeat this process until the value becomes 1. Print out each intermediate value. Finally print out how many of these operations you have performed. If the input value is less than 1, print an error message and perform an `exit(EXIT_FAILURE);` Note that, the function `exit()` is defined in `stdlib.h`. Hence, `stdlib.h` must be included at the beginning of the program.