COMP2401A – Tutorial 11
Building a library and using GDB

Learning Objectives

After this tutorial, you will be able to:

- Build your own library
- Use the GDB debugger

Download the file t11.tar and extract the tutorial files.

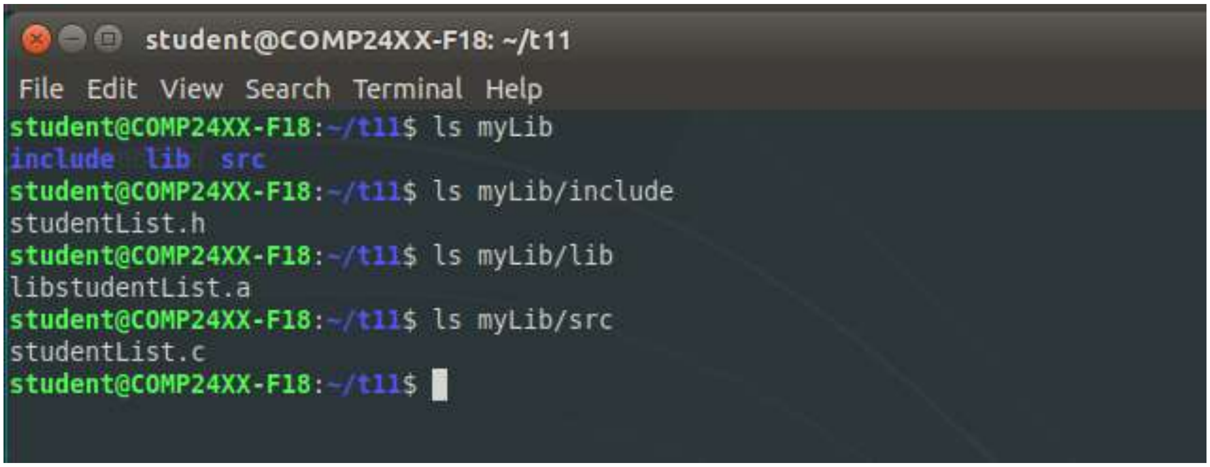
As always, at the end of the tutorial submit your work including any files you worked on, even if you haven't changed them.

1 A Student Library

In this section you will learn how to build your own library by taking the code from completeStudentList.c and converting it into the library format described in Chapter 7.2.

Step 1: Review Chapter 7.2 "Libraries".

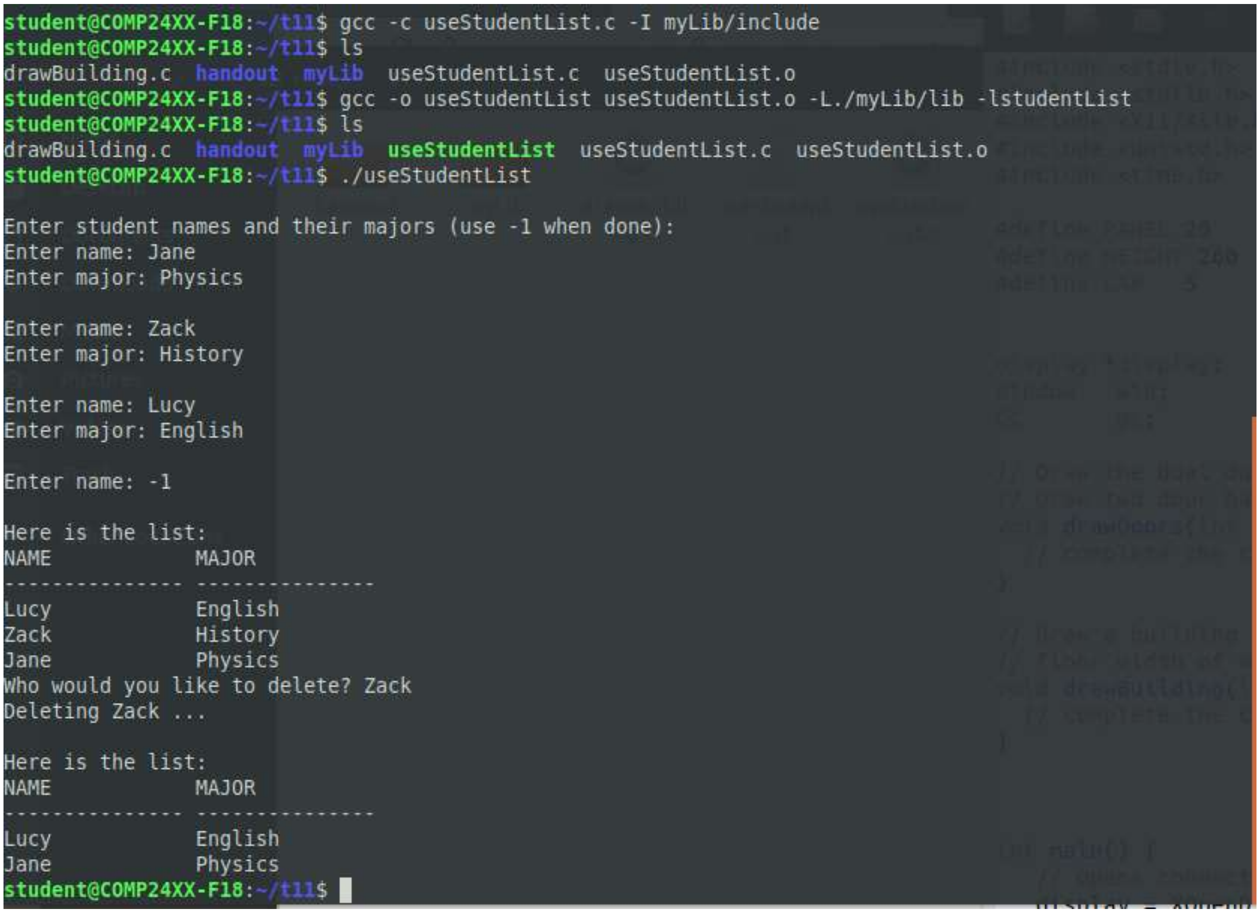
Step 2: Following the example on p264 of Chapter 7 build a library libstudentList.a from completeStudentList.c on p141-144 of Chapter 3 so that the directory structure looks like this


student@COMP24XX-F18: ~/t11
File Edit View Search Terminal Help

```
student@COMP24XX-F18:~/t11$ ls myLib
include  lib  src
student@COMP24XX-F18:~/t11$ ls myLib/include
studentList.h
student@COMP24XX-F18:~/t11$ ls myLib/lib
libstudentList.a
student@COMP24XX-F18:~/t11$ ls myLib/src
studentList.c
student@COMP24XX-F18:~/t11$ 
```

Let's call the program that uses this library useStudentList.c (in t11.tar).

Once you separated the linked list code into the above directory structure you should be able to do the following:



```
student@COMP24XX-F18:~/t11$ gcc -c useStudentList.c -I myLib/include
student@COMP24XX-F18:~/t11$ ls
drawBuilding.c  handout  myLib  useStudentList.c  useStudentList.o
student@COMP24XX-F18:~/t11$ gcc -o useStudentList useStudentList.o -L./myLib/lib -lstudentList
student@COMP24XX-F18:~/t11$ ls
drawBuilding.c  handout  myLib  useStudentList  useStudentList.c  useStudentList.o
student@COMP24XX-F18:~/t11$ ./useStudentList

Enter student names and their majors (use -1 when done):
Enter name: Jane
Enter major: Physics

Enter name: Zack
Enter major: History

Enter name: Lucy
Enter major: English

Enter name: -1

Here is the list:
NAME            MAJOR
--------------- ---------------
Lucy            English
Zack            History
Jane            Physics
Who would you like to delete? Zack
Deleting Zack ...

Here is the list:
NAME            MAJOR
--------------- ---------------
Lucy            English
Jane            Physics
student@COMP24XX-F18:~/t11$ 
```

2 Using GDB commands

Developing a computer program takes time and effort. The development process is usually long and spans over several development phases from requirements, to program design, to code development, and finally testing and acceptance.

In this document we focus on examining and fixing code using the GDB debugger. When coding and testing a program different types of errors may arise: syntax errors, inconsistencies throughout the code, missing variables and functions, logical errors, etc.

During program creation the gcc compiler captures a large number of errors. For examples, syntax errors such as missing brackets, missing ';' at the end of a statement, typos of key words such as *longe* instead of *long*. The compiler also checks inconsistencies between function declarations and parameters that are passed to functions.

For example: the function swap is declared as

Int swap(int * x, int *y);

But it is used in the code as

long numX;
long numY;
swap(&numX, &numY);

Here the compiler will “complain” that there is a mismatch between the parameters of the function swap, which are declared as *int,* and the type of the variables that were passed to the function, which are declared as *long*.

Last the compiler during a linking phase checks for missing variables or functions. For example, if the function swap() was not coded then the compiler will “complain” that the function swap() is missing.

These types of errors are usually easy to fix because the compiler provides enough information about the errors such as file name, line number and description of the error.

However, the compiler cannot detect logical errors and errors that may occur during run time such as division by 0, array out of bounds, and uninitialized variables. Detecting these errors can be time consuming (from minutes to days) depending on the nature of the problem, size of the program and our knowledge of the program. These errors are termed run-time errors because they only occur when the program is executing. Often these types of errors are consistent. Namely, the error will occur every time the program is executed, and the error will occur at the same location. However, this is not always the case as some errors only occur from time to time and usually there is no pattern.

To assist in detecting and fixing the errors one can use a debugger. A debugger is a program that executes the erroneous program and can provide the programmer with insight about their program. For example, if a programmer would like to find run time errors in program A. The programmer would then execute program A in the GDB debugger and check if the program execution meets the intended logic.

A debugging program allows a programmer to control the execution of the program by navigating the code (stopping at different code lines), inspect variables and parameters, change values of variables, etc. A partial list of commands is shown in **Error! Reference source not found.**

In this part of the tutorial you will learn to use some of the basic commands of GDB. This section is divided into several parts: code preparation for GDB; navigating the code; setting breakpoints; investigating variables.

Here we first use the following simple code that computes the average of two numbers (`average.c from t11.tar`)

The function average:

```
int average(int x, int y, int *average) {
    int sum;
    int i;

    // compute the sum
    sum = x;
    for (i = 0; I < y; i++) {
          sum += 1;
    }

    // compute the average
    *average = sum /2;
    return (0);
}
```

We will test the code using the following code (in file `average.c`)

```
int main(int argc, char **argv)
{
    int x;
    int y;
    int result;

    printf("Testing the average function \n");

    // test 1
    x = 5;
    y = 7;
    printf("test 1: testing average(%d, %d) answer should be 6\n",x,y);
    average(x,y, &result);
    printf("average(%d, %d)=%d\n \n",x, y, result);

    // test 2
    x = 5;
    y = 4;
    printf("test 2: testing average(%d, %d) answer should be 4.5\n",x,y);
    average(x,y, &result);
    printf("average(%d, %d)=%d\n \n",x, y, result);
    return(0);
}
```

2.2 Preparing the code for GDB

1. Compile the code and create an executable file average
   gcc -o average average.c

2. Fix any errors that may arise
3. Run the code in GDB using
   gdb average

4. You will get the gdb copyright notice and it will end with a prompt informing us that GDB is ready for execution.
   (gdb)

5. Try the list command to show the code
   (gdb) list

6. A message will appear saying there is no symbol table.

7. Quit GDB by typing quit (or q for short)

The problem is that a debugging program such as GDB requires additional information in order to allow the program to navigate through the code. The additional information is termed a symbol table ().

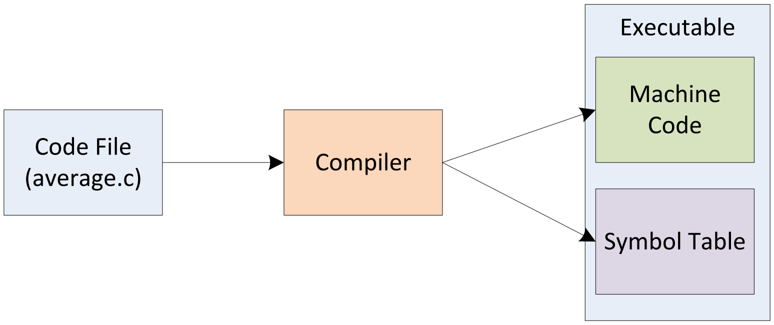


Figure 1: compiler creating code file and symbol table

This symbol table is created by the compiler by using the ***-g*** option. For example

```
gcc -g -o average average.c
```

The result is a larger executable code size. Below you can see the difference between compiling the code with the -g option and without. Compiling the file average.c without the -g option results in an executable size of 8448 bytes and in code size of 12536 when compiling with the -g option.

```
>gcc -o average average.c

>l -la average

-rwxr-xr-x 1 student student 8448 Sep 24 14:05 average*
```

```
>gcc -g -o average average.c

>l -la average

-rwxr-xr-x 1 student student 12536 Sep 24 14:05 average*
```

2.3 Navigating the source code

Commands used in this part of the tutorial: ***start, next, list, continue***

In this part of the tutorial you will navigate the code. Here you will start inspecting the code line by line

1. Begin the gdb execution by using the ***start*** command – gdb will stop at the fist line of code in the function main()
2. Execute the next line of code by typing ***next*** (or n for short).
3. Note that at anytime during the execution you can look at the code this can be done by
    a. Type ***list*** (or l for short) to see the lines of code. Typing ***l*** again will present the next few lines.
    b. Note that if you wish to see other lines of code or go back you can select a line number and the code around this line number will be presented. Try the following to see the code around line 60
    list 60

4. Repeat the command next until you reached the code line average(x, y, &result);
5. Type ***next*** again – note that the function the computes the average was skipped.
6. Type ***continue*** to terminate the execution of the program

2.4 Exploring the function `average()`

Commands used in the part of the tutorial: ***step, finish***

Here you will explore the function average() by entering it.

1. Repeat steps 1-4 of Section 2.3
2. This time instead of skipping the function average() we will enter it by typing
   step

   You should see the parameters and the values that are passed to the function average().

3. Type ***next***
4. Type ***next*** several times until you are in the middle of the for loop (line 39).
5. Instead of typing ***next, next…*** until the code reaches the end of the line, one can leave the function by typing ***finish***.
6. Type ***c*** to finish the program execution.

2.5 Managing breakpoints

Commands used in the part of the tutorial: ***breakpoint, info, delete***

Breakpoints are commands that instruct the debugger to stop at particular lines of code while the code is running. Namely, instead of walking through the code one line at a time one can tell the debugger to execute the code until it reaches the first break point. Breakpoints can be set by defining lines of code or by functions.

2.5.1 Setting breakpoints:

1. Type ***break main***. This will set a breakpoint at the first line of the function main().
2. Run the program by typing ***r***. The program will stop at the first line of main

3. Type ***list 56*** to see the program around code line 56. Inspect the code
4. Type ***break 56*** to set up a break point at line 56.
5. Type ***continue or c*** to continue the execution. The program should stop at line 56.
6. Type ***c*** again. The program should terminate.

Managing breakpoints

7. To navigate directly to the function average one can set a breakpoint at the beginning of the function. This is done by typing ***break average***.
8. Start the program again by typing ***r***.

The program stops at the first breakpoint that we set and not at the function `average()`. We will remedy this by disabling some of the breakpoints. First we need to see which breakpoints are set.

9. Type ***info breakpoints***. This will list all the breakpoints. There should be three breakpoints – the first two are in main and one in average.
10. Type ***disable 1 2*** to disable breakpoints 1 and 2. (your numbers may be different so use those instead)
11. Type ***r*** to restart the program. You will be asked to confirm it. The program should stop at the beginning of the function `average()`.
12. To enable the breakpoint at line 56 use the command enable.
    a. List the breakpoints using ***info breakpoints***
    b. Identify the number of the breakpoint at line 56. Assuming that the breakpoint number is 2 then
    c. Enable the breakpoint using ***enable 2***
13. Test the code by typing ***r***

2.5.2 Deleting breakpoints

Deleting breakpoints is done using the command delete and the breakpoint number.

14. List the breakpoints using info breakpoints (assuming that the breakpoint at average() is the breakpoint number 3)
15. Type ***delete 3*** the breakpoint at average()
16. Run the code using ***r.*** The program should stop at line 56
17. Type ***c*** and the program should terminate.
18. Type ***delete*** to delete all breakpoints

2.6 Exploring variables

Commands used in this part of the tutorial: ***print, set, display, info***

Here you will explore the function `average()` and its variables and parameters.

1. Set a breakpoint at the function `average()` using ***break average***
2. Run the code and then list the code around line 39.
3. Set up a breakpoint at line 39 (***b 39***).
4. Continue to line 39 by typing **c**
5. View the value of *i* by typing ***print i***
6. View the value of *sum* by typing ***print sum***
7. Type ***c*** Advance to the next breakpoint (which is line 39)
8. View the value of *sum* by typing ***print sum***
9. Type ***c*** again. The program will stop at line 39 again.

It can be quite annoying and time consuming to type ***print sum*** every time the program stops at line 39. We can ask GDB to display the values of *i* and *sum* every time the program stops using the command display.

10. Type ***display sum*** – the program will display the value of *sum*
11. Type c again. This time the program will display the values of *sum* as program stopped at line 39.
12. Type ***info locals*** to see all the local variables of the function `average()` and their values (in this case the local variables are *i* and *sum*)

Changing the value of a variable

13. Change the value of sum to 0 using ***set sum = 0***
14. Type ***print sum*** to check the assignment

Managing displayed points

To stop the display of the sum at every line of code one can either delete it or disable it in a similar way to managing breakpoints.

15. Type ***info display*** to list the displayed variables. Find the id of sum in the list (it should be 1)
16. Type ***disable display 1*** to disable the display of sum
17. Type ***c*** to continue. The value of sum should not be displayed.
18. Type ***delete display 1*** to delete the display of sum
19. Check it by typing info display
20. Quit GDB

2.7 Use GDB on r-fact.c

Use GDB to observe the behavior of r-fact.c from t11.tar. In particular, try the following commands:

- bt - show the stack
- info frame - show the information about the current frame
- up - to go up one level in the stack (namely to the line of the calling function)
- down - go down one level in the stack (namely to the code line of thecalled function)

2.8 Use GDB on the stackExample.c and singlyLinkedList.c

Use GDB to observe the behavior of the stackExample.c on p108 and singlyLinkedList.c on pp128-130.

2.9 Use DDD on singlyLinkedList.c

Use DDD (GDB with a GUI) to observe the behavior of singlyLinkedList.c on pp128-130 and display the linked list as it is being created. See ‘man ddd’ for how to use it.