بسم الله الرحمن الرحيم

EE 463 : Operating Systems
Faculty Of Engineering
King Abdulaziz University
Department Of Electrical and Computer Engineering

LAB #4

The C Programming Environment in Linux
Gcc, Make, Makefiles and Gdb
A Tutorial by Example

Objective

If you are used to do your programming using only some advanced IDEs in a rich GUI environment, then, for this course, you need to know about how to do so using the CLI programming environment. This tutorial is not comprehensive or particularly detailed, but it is designed to give you enough information to get you going in your **C** programming assignments using Linux's command line interface. You will learn about some of the most useful tools for programming in this environment: **gcc**, **make**, Makefiles, and **gdb**. These programs allow you to easily compile or build, and debug your **C** projects.

Introduction

Linux is a modern, open-source, well documented and freely available operating system. For this reason, it is adopted for studying the operating systems courses in many colleges around the world. Most of our examples and your project assignments are required to be done in **C** under Linux, since it is also written in **C**. Although there are many graphical user interfaces (GUIs) for Linux, the default and the most powerful Linux user interface is the command line interface (CLI), better known as the Linux shell, and you will be using shell commands to develop your projects. So, in order to work efficiently under these conditions, you need to master more than just the syntax of the **C** language. You need to know your environment, which includes: your ***tools***, your ***libraries*** and your ***documentation***. In particular, this tutorial will help you know this required programming environment:

- The tools that are relevant to **C** compilation under Linux are: **gcc**, **make**, **gdb**, and maybe **ld**. We'll talk about each of these in more detail later on.
- There are many library routines that are available to you, but a lot of functions are found in **libc**, which is linked with all **C** programs by default. You only need to #include the right header files.
- Finally, knowing how to find library routines you need and reading man pages, is a skill you need.

Like (almost) everything worth doing in life, becoming an expert in these domains takes time. Spending the time up-front to learn more about the tools and the environment is definitely well worth the effort.

1. A Simple C Program structure

Let's start with the following **C** program, called "hw.c". Copy it into a file in your Linux home directory, and save the file as, "hw.c". Unlike Java, there is not necessarily a connection between the file name and the contents of the file; thus, use your common sense in naming files in a manner that is appropriate.

hw.c

```
/* header files go up here */
/* note that C comments are enclosed within a slash and a star, and
   may wrap over lines */
// if you use gcc, two slashes will work too (and may be preferred)
#include <stdio.h>

// main returns an integer
int main(int argc, char *argv[]) {

   // printf is our output function; by default, writes to stdout
   // printf returns an integer, but we ignore that
   printf("hello, world\n");

   // return 0 to indicate all went well
   return(0);
}
```

The first line specifies a file to include, in this case **stdio.h**, which *prototypes* many of the commonly used input/output routines; the one we are interested in is printf(). When you use the #include directive, you are telling the **C** preprocessor (**cpp**) to find a particular file (e.g., **stdio.h**) and to insert it directly into your code at the spot of the #include. By default, **cpp** will look in the directory /usr/include/ to try to find the file. The next part specifies the signature of the main() routine, namely, that it returns an integer (int), and will be called with two arguments, an integer argc, which is a count of the number of arguments on the command line, and an *array of pointers* to characters (argv), each of which contain a word from the command line, and the last of which is **null**.

The program then simply prints the string "hello, world" and advances the output stream to the next line, because of the backslash followed by an "**n**" at the end of the call to printf(). Afterwards, the program completes by returning a value, which is passed back to the shell that executed the program. A script or the user at the terminal can check this value, to see if the program exited cleanly or with an error.

2. The Compilation Process

After writing a program in a high-level language like **C**, it needs to be translated into an executable machine language program, so the computer can execute it. The translation process involves several steps, all done automatically by calling the **gcc** command. Actually, **gcc** is not a compiler, but rather a "*compiler driver*"; thus it coordinates the four to five steps of the compilation process. These steps are done using the following programs in the same order as shown:

1. The **C** pre-processor (**cpp**): Where all the directives, such as #define, #include, and other macros are processed by **cpp**. This program is just a source-to-source translator, so its end-product is still just source code (i.e., a **C** file).
2. The **C** compiler (**cc**): Where the **C** source-level code is converted by **cc** into low-level, **assembly** language code, specific to the host machine (CPU).
3. The Assembler (**as**): Where the low-level **assembly** language code is converted into machine-level **object** code (i.e. bits and things that machines can really understand).
4. The Linker (**ld**): Where all **object** code parts of the program, the libraries of used functions, and device drivers are linked (put) together to produce the final executable program.

3. Compiling and Executing

We are going to use **gcc** to compile our example program, `hw.c` above. The translation process is highly automated, since **gcc** hides all the steps we mentioned before and simply, produces an executable program. To compile our `hw.c` program, at the shell prompt, you just type:

```
$ gcc hw.c
```

The result of your compilation will be an executable, named (by default) `a.out`. To run the program, you simply type:

```
$ ./a.out
```

When the program is executed, the OS will set `argc` and `argv` properly so that the program can process the command-line arguments as need be. Specifically, `argc` will be equal to **1**, `argv[0]` will be the string "**./a.out**", and `argv[1]` will be **null**, indicating the end of the array.

The gcc command Syntax

```
gcc  [-c|-S|-E] [-std=standard]
     [-g] [-pg] [-Olevel]
     [-Wwarn...] [-Wpedantic]
     [-Idir...] [-Ldir...]
     [-Dmacro[=defn]...] [-Umacro]
     [-foption...] [-mmachine-option...]
     [-o outfile] [@file] infile...
```

Only the most useful options are listed here; see below for explanation.

Description

When you invoke **gcc**, it normally does preprocessing, compilation, assembly and linking. Some options allow you to stop this process at an intermediate stage. For example, the **-c** option says not to run the linker. Then the output consists of only the object (**.o**) files generated by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Useful Options

The student is encouraged to see a full documentation of **gcc** using "`man gcc`" command. Here, we point out some useful compilation options for **gcc**:

```
$  gcc -c hw.c          # -c:    to produce only the object code "hw.o"
$  gcc -o hw hw.c       # -o:    to specify the executable name
$  gcc -Wall hw.c       # -Wall: gives much better warnings
$  gcc -g hw.c          # -g:    to enable debugging with gdb
$  gcc -I/foo/bar hw.c  # -Idir: to specify other than the default include directory
$  gcc -L/foo/bar hw.c  # -Ldir: to specify other than the default libraries directory
$  gcc -O hw.c          # -O:    to turn on optimization
```

Of course, you may combine these options as you see fit (e.g., **gcc -o hw -g -Wall hw.c**). Of these options, you should always use **-Wall**, which gives you lots of extra warnings about possible mistakes. **Don't ignore the warnings!** Instead, fix them and thus make them blissfully disappear.

Linking with Libraries

Sometimes, you may want to use a library routine in your program. Because so many routines are available in the **C** library (which is automatically linked with every program), all you usually have to do is find the right `#include` file. The best way to do that is via the **manual pages**, called just the **man pages**.

For example, let's say you want to use the `fork()` system call1. By typing `man fork` at the shell prompt, you will get back a text description of how `fork()` works. At the very top will be a short code snippet that will tell you which files you need to `#include` in your program in order to get it to compile. In the case of `fork()`, you need to `#include` both `sys/types.h` and `unistd.h`, which would be accomplished as follows:

```
#include <sys/types.h>
#include <unistd.h>
```

Note that `fork()` is a system call, and not just a library routine. However, the **C** library provides **C** wrappers for all the system calls, each of which simply trap into the operating system.

Some library routines do not reside in the **C** library, and therefore you will have to do a little more work. For example, the math library has many useful routines, such as **sin**, **cos**, **tan**, and the like. If you want to include the routine `tan()` in your code, you should again first check the man page. At the top of the Linux man page for **tan**, you will see the following two lines:

```
#include <math.h>
...
Link with -lm.
```

The first line you already should understand—you need to `#include` the math library, which is found in the standard location in the file system (i.e., /usr/include/math.h). The last line is telling you how to "link" your program with the math library. A number of useful libraries exist and can be linked with; many of those reside in /usr/lib; it is indeed where the math library is found.

There are two types of libraries:

Statically-linked libraries (which end in **.a**):

This type is attached directly into your executable; i.e., the low-level code for the library is inserted into your executable by the linker, and results in a much larger binary object code.

Dynamically-linked libraries (which end in **.so**):

This type minimizes the resulting binary object code, by just including the reference to a library in your program executable; when the program is run, the operating system loader dynamically links in the library. This method is preferred over the static approach because it saves disk space and allows applications to share library code and static data that are already loaded in memory.

In the case of the math library, both static and dynamic versions are available, with the static version called /usr/lib/libm.a and the dynamic one /usr/lib/libm.so. In any case, to link with the math library, you need to specify the library to the link-editor; this can be achieved by invoking **gcc** with the right options.

```
$ gcc -o hw hw.c -Wall -lm
```

The `-lxxx` option tells the linker to look for file, `libxxx.so` or `libxxx.a`, probably in that order. If for some reason you insist on the static library over the dynamic one, there is another option you can use —

see if you can find out what it is. People sometimes prefer the static version of a library because of the slight performance cost associated with using dynamic libraries.

Note:

If you want the compiler to search for headers in a different path than the usual places, or want it to link with libraries that you specify, you can use the compiler option **-I**/foo/bar to look for headers in the directory /foo/bar, and the **-L**/foo/bar option to look for libraries in the /foo/bar directory. One common directory to specify in this manner is "**.**", called "dot", which is UNIX shorthand for the current directory.

Note that the **-I** option should go on a *compile* line, and the **-L** option on the *link* line.

Separate Compilation

Once a program starts to get large enough, you may want to split it into separate files, compiling each separately, and then linking them together and with the required libraries.

Typically, you may have several header (**.h**) files, and several implementation (**.c**) files; For example, say we expanded our `hw.c` program and split it into four files: `main.c`, `hello.c`, `factorial.c`, and `functions.h`, as will be shown later, and you wish to compile them individually, and then link them together. The following commands will do just that:

```
# We are using -Wall for warnings, -O for optimization
$ gcc -Wall -O -c main.c
$ gcc -Wall -O -c hello.c
$ gcc -Wall -O -c factorial.c
$ gcc -o hello main.o hello.o factorial.o -lm
```

The **-c** option tells the compiler just to produce an object file — in this case, files called `main.o`, `hello.o`, and `factorial.o`. These files are not executables, but just machine-level representations of the code within each source file. To combine the object files into an executable, you have to "link" them together; this is accomplished with the fourth line, **gcc -o hello main.o hello.o factorial.o**.

In this case, **gcc** sees that the input files specified are not source files (**.c**), but instead are object files (**.o**), and therefore skips right to the last step and invokes the link-editor (linker) **ld** to link them together into a single executable. Because of its function, this line is often called the "link line", and would be where you specify link-specific commands such as **-lm**.

Analogously, options such as **-Wall** and **-O** are only needed in the compile phase, and therefore need not be included on the link line but rather only on compile lines. Of course, you could just specify all the **C** source files on a single line to **gcc**, (e.g. **gcc -Wall -O -o hello main.c hello.c factorial.c**), but this requires the system to recompile every source-code file, which can be a time-consuming process. By compiling each individually, you can save time by only recompiling those files that have changed during your editing, and thus increase your productivity. This process is best managed by another program, **make**, which we now describe.

4. The MAKE Utility

As you have seen, to compile a project's source code can be tedious, especially when it includes many source files, header files, libraries, … etc., and you have to type the compiling command(s), by hand, every time you change any part of the project to re- Build it. The **make** utility uses specially formatted files, called **Makefiles** to help you manage and automatically build your project(s) from its **C** source files.

Preparation

For this part of the tutorial you will need the following files, copy each program code below, into a respective file, and save them in a suitable new directory in your Linux system:

- main.c

```
#include <stdio.h>
#include "functions.h"

int main(){
    print_hello();
    printf("\n");
    printf("%s %d\n", "The factorial of 5 is ", factorial(5));
    return 0;
}
```

- hello.c

```
#include <stdio.h>
#include "functions.h"

void print_hello(){
    printf("%s", "Hello World!");
}
```

- factorial.c

```
#include "functions.h"

int factorial(int n){
    if(n!=1){
        return(n * factorial(n-1));
    }
    else return 1;
}
```

- functions.h

```
void print_hello();
int factorial(int n);
```

Note:

We use **gcc** for compiling here. You are free to change it to a compiler of your choice.

If we run the command

```
$ make
```

This program will look for a file named makefile or Makefile in the current directory, and then execute it. If there are several makefiles, then we can execute the one we need with the command:

```
$ make -f MyMakefile
```

There are several other switches to the **make** utility. For more information, type:

```
$ man make
```

The Build Process

Building a project means producing an executable file from all the files comprising the project. As we stated before, there are two steps to build a project starting from its source files:

1. Compiler takes the **source** files and outputs **object** files
2. Linker takes the **object** files and creates an **executable** file

Compiling by Hand

The trivial way to compile the above **C** files, and obtain an executable file, is by running the command:

```
$ gcc main.c hello.c factorial.c -o hello
```

The Basic Makefile

The basic **makefile** is composed of:

```
target: dependencies
[tab] system command
```

This syntax applied to our example would look like the following. Copy it to your directory:

Makefile1

```
all:
    gcc main.c hello.c factorial.c -o hello
```

To run this makefile on our files, type:

```
$ make -f Makefile1
```

On this first example we see that our target is called *all*. This is the default target for makefiles. The make utility will execute this target if no other one is specified.

We also see that there are no dependencies for target *all*, so make safely executes the specified commands.

Finally, make compiles the program according to the command line we gave it.

Using Dependencies

Sometimes it is useful to use different *targets*. This is because if we modify a single file in our project, we don't have to recompile everything, only what we have modified. Here is an example, copy it to your directory:

Makefile2

```
all: hello

hello: main.o factorial.o hello.o
    gcc main.o factorial.o hello.o -o hello

main.o: main.c
    gcc -c main.c
```

```
factorial.o: factorial.c
    gcc -c factorial.c

hello.o: hello.c
    gcc -c hello.c

clean:
    rm *o hello
```

Now we see that the target *all* has only dependencies, but no system commands. In order for make to execute correctly, it has to meet ***all*** the dependencies of the called ***target*** (in this case *all*).

Each of the dependencies is searched through all the targets available in the file, and executed if found.

In this example we see a target called *clean*. It is useful to have such a target if you want to have a fast way to get rid of all the object files and executables before executing other targets.

Using Variables and Comments

We can also use *variables* when writing Makefiles. It comes in handy in situations where we want to change the compiler, or the compiler options. Copy it to your directory:

Makefile3

```
# I am a comment, and I want to say that the variable CC will be
# the compiler to use.
CC=gcc

# Hey!, I am comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.c
    $(CC) $(CFLAGS) main.c

factorial.o: factorial.c
    $(CC) $(CFLAGS) factorial.c

hello.o: hello.c
    $(CC) $(CFLAGS) hello.c

clean:
    rm *o hello
```

As you can see, variables can be very useful sometimes. To use them, just assign a value to a variable before you start to write your targets. After that, you can just use them with the dereference operator, for example: `$(VAR)`.

Where to Go From Here

With this brief introduction to Makefiles, you can create some very sophisticated mechanisms for compiling your projects, which is sufficient for our course's term-project. However, this is just the tip of

the iceberg. If you need to learn more about the make utility and makefiles, consult the proper documentation or you can run the command "**info make**" to get the make full manual, if it is installed in your system.

We don't expect anyone to fully understand the example presented below without having consulted some Make documentation or read about it in the Unix book. Copy the next Makefile to your directory, and try it out:

Makefile4

```
CC=gcc
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
    $(CC) $(CFLAGS) $< -o $@
```

If you understand this last example, you could adapt it to your own personal projects changing only 2 lines: (**SOURCES** and **EXECUTABLE**), no matter how many additional files you have!!!

5. The GNU Debugger

After writing a program, you may face some problems executing it. It is almost always the case where you need to debug the program before it is ready for presentation. There are many ways to go about debugging, from printing out messages on the screen, to using a debugger, or just thinking about what the program is doing and making an educated guess as to what the problem is.

Before a bug can be fixed, the source of the bug must be located. For example, with segmentation faults, it is useful to know on which line of code the segmentation fault is occurring. Once the line of code in question has been found, it is useful to know about the values in that method, who called the method, and why, specifically, the error is occurring. Using a debugger makes finding all of this information very simple.

In this section you will learn how to successfully debug your programs using the GNU debugger called **gdb**, which can be used for several languages, including **C** and **C++**. It allows you to inspect what the program is doing at a certain point during execution. Errors like segmentation faults or out-of-bound array indexes may be easier to find with the help of **gdb**.

Preparation

For your work in this section of the lab, we assume that you have uploaded the following buggy **C** program called "main.c" to your Linux virtual machine, preferably, into a separate new directory:

main.c

```
// main.c
// This file contains the example program used in the
// gdb debugging tutorial.
#include <stdio.h>

int number_instantiated = 0;
```

```
struct Node {
public:
  Node(const int &value, Node *next = 0) : value_(value), next_(next) {
    printf("%s%d%s\n", "Creating Node, ",
            ++number_instantiated,
            " are in existence right now");
  }
  ~Node() {
    printf("%s%d%s\n", "Destroying Node, ",
            --number_instantiated,
            " are in existence right now");
    next_ = 0;
  }

  Node* next() const { return next_; }
  void next(Node *new_next) { next_ = new_next; };
  const int& value() const { return value_; }
  void value(const int &value) { value_ = value; }

private:
  Node();
  int value_;
  Node *next_;
};

struct LinkedList {
public:
  LinkedList () : head_(0) {};
  ~LinkedList () { delete_nodes (); };

  // returns 0 on success, -1 on failure
  int insert (const int &new_item) {
    return ((head_ = new Node(new_item, head_)) != 0) ? 0 : -1;
  }

  // returns 0 on success, -1 on failure
  int remove (const int &item_to_remove) {
    Node *marker = head_;
    Node *temp = 0;        // temp points to one behind as we iterate

    while (marker != 0) {
      if (marker->value() == item_to_remove) {
        if (temp == 0) {   // marker is the first element in the list
          if (marker->next() == 0) {
            head_ = 0;
            delete marker; // marker is the only element in the list
            marker = 0;
          } else {
            head_ = new Node(marker->value(), marker->next());
            delete marker;
            marker = 0;
          }
          return 0;
        } else {
          temp->next (marker->next());
          delete temp;
          temp = 0;
          return 0;
        }
      }
      marker = 0;  // reset the marker
      temp = marker;
      marker = marker->next();
    }

    return -1;    // failure
  }

  void print (void) {
    Node *marker = head_;
    while (marker != 0) {
      printf("%d\n", marker->value());
      marker = marker->next();
    }
  }

private:
  void delete_nodes (void) {
    Node *marker = head_;
    while (marker != 0) {
      Node *temp = marker;
      delete marker;
      marker = temp->next();
    }
  }

  Node *head_;
};

int main (int argc, char **argv) {
```

```
  LinkedList *list = new LinkedList ();

  list->insert (1);
  list->insert (2);
  list->insert (3);
  list->insert (4);

  printf("%s\n", "The fully created list is:");
  list->print ();

  printf("\n%s\n", "Now removing elements:");
  list->remove (4);
  list->print ();
  printf("\n");

  list->remove (1);
  list->print ();
  printf("\n");

  list->remove (2);
  list->print ();
  printf("\n");

  list->remove (3);
  list->print ();

  delete list;

  return 0;
}
```

Normally, you would compile the above program like we have done in the previous sections, but if you need to debug this program, you need to add a **-g** option to your compile command to enable built-in debugging support, which **gdb** needs. For example:

```
$ g++ -Wall -g main.c -o main
```

To make things easier to manage, we created the following simple Makefile. Copy it to the same directory, where you have the buggy main.c program:

Makefile

```
CC = g++
FLAGS = -g -Wall

main: main.c
    ${CC} ${FLAGS} -o main main.c

clean:
    rm -f main
```

Go ahead and **make** the program for this tutorial, and **run** it. The program will print out some messages, and then it will print that it has received a *segmentation fault* signal, resulting in a program crash.

```
  ...
  ...
Segmentation fault
```

Given the information on the screen at this point, it is near impossible to determine why the program crashed, much less how to fix the problem. We will now begin to debug this program.

Starting up gdb

Type: "**gdb**" or "**gdb main**." You'll see many lines displayed then you get a prompt that looks like this:

```
(gdb)
```

If you didn't specify a program to debug, you'll have to load it in now by typing:

```
(gdb) file main
```

Here, "**main**" is the program you want to load, and "**file**" is the command to load it. **gdb** has an interactive shell, much like the one you use as soon as you log into the Linux machines. It can recall history with the **arrow** keys, auto-complete words (most of the time) with the **TAB** key, and has other nice features. If you're ever confused about a command or just want more information, use the "**help**" command, with or without an argument:

```
(gdb) help [command]
```

You should get a nice description and maybe some more useful tips.

Running the program

The debugger is now waiting for the user to type a command. You need to run the program so that **gdb** can help you see what happens when the program crashes. At the **gdb** prompt, type "**run**". Here is what happens when I run this command:

```
(gdb) run
Starting program: /root/Programs/gdb/main
Creating Node, 1 are in existence right now
Creating Node, 2 are in existence right now
Creating Node, 3 are in existence right now
Creating Node, 4 are in existence right now
The fully created list is:
4
3
2
1

Now removing elements:
Creating Node, 5 are in existence right now
Destroying Node, 4 are in existence right now
4
3
2
1


Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a50 in Node::next (this=0x0) at main.c:22
22        Node* next () const { return next_; }
(gdb)
```

This runs the program that you loaded for debugging. If it had no serious problems (i.e. the normal program didn't get a *segmentation fault*, etc.), the program should run fine here too. But our program did have issues, it crashed. So let's see what kind of information we can gather. You should get some useful information like the line number where it crashed, and parameters to the function that caused the error, as you can see from the last four lines above.

Inspecting crashes

Already you can see that the program was at line 22 of **main.c**, that **this** points to **0**, and you can see the line of code that was executed. But we also want to know who called this method and we would like to be able to examine values in the calling methods. So at the **gdb** prompt, type "**backtrace**" which gives the following output:

```
(gdb) backtrace
#0  0x0000000000400a50 in Node::next (this=0x0) at main.c:22
#1  0x0000000000400c8e in LinkedList::remove (this=0x602010,
    item_to_remove=@0x7fffffffe50c: 1) at main.c:70
#2  0x000000000040090b in main (argc=1, argv=0x7fffffffe618) at main.c:113
(gdb)
```

In addition to what we knew about the current method and the local variables, we can now see also what the calling methods, and what their parameters were. For example, we can see that the current method was called by **LinkedList::remove()**, where the parameter **item_to_remove** is at address **0x7fffffffe50c**.

It may help to understand the bug if we know the value of **item_to_remove**, so we want to see the value at its address. This can be done using the "**x**" command and the address as a parameter. ("**x**" can be thought of as being short for "examine".) Here is what happens when we run the command, type:

```
(gdb) x 0x7fffffffe50c
0x7fffffffe50c: 0x00000001
(gdb)
```

The program is crashing while trying to run **LinkedList::remove()** with a parameter of **1**. We have now narrowed the problem down to a specific function and a specific value for the parameter.

Now that we know where and when the *segmentation-fault* is occurring, we want to watch what the program was doing right before it crashes. We don't want to run the program without any stopping or breaking, etc. Otherwise, it will just rush past the error and we can never find the root of the problem.

One way to do this is to step through, one step at a time, every statement of the program until we arrive upon the exact error. This works, but it is tedious. We may want to just run to a particular section of code and stop execution at that point so we can examine data at that location. This brings us to the next set of commands.

Setting breakpoints

If you have ever used a debugger you are probably familiar with the concept of breakpoints. Basically, a breakpoint is a line in the source code where the debugger should break execution. The simplest way is the command "**break**." This sets a breakpoint at a specified file:line pair. In our example, we want to look at the code in **LinkedList::remove()** so we would want to set a breakpoint at line **45** of main.c. We may type:

```
(gdb) break main.c:45
```

Now, if the program ever reaches that location when running, the program will pause and prompt you for another command. You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them. Since we may not know the exact line number, we can also tell the debugger to break at a particular function. Here is what we want to type for our example:

```
(gdb) break LinkedList::remove
Breakpoint 1 at 0x400b39: file main.c, line 45.
(gdb)
```

So now Breakpoint 1 is set at main.c, line **45** as desired. The reason the breakpoint gets a number is so we can refer to the breakpoint later, for example if we want to delete it. Once you've set a breakpoint, you can try using the "**run**" command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point). Also, you can proceed onto the next breakpoint by typing "**continue**" (Typing **run** again would restart the Program from the beginning, which isn't very useful.)

```
(gdb) continue
```

You can single-step (execute just the next line of code) by typing "**step**." This gives you really fine-grained control over how the program proceeds. You can do this as many times as you want.

```
(gdb) step
```

Similar to "**step**," the "**next**" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.

```
(gdb) next
```

Typing "**step**" or "**next**" a lot of times can be tedious. If you just press **ENTER**, **gdb** will repeat the same command you just gave it. You can do this as many times as you need.

Querying Other Aspects of the Program

So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging. The **print** command prints the value of the variable specified, and **print/x** prints the value in hexadecimal:

```
(gdb) print my_var
(gdb) print/x my_var
```

Setting watchpoints

Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They make the program pause whenever a watched variable's value is modified. For example, the following watch command:

```
(gdb) watch my_var
```

Now, whenever my_var's value is modified, the program will interrupt and print out the old and new values. You may wonder how **gdb** determines which variable named my_var to **watch** if there is more than one declared in your program. The answer is that it relies upon the variable's scope, relative to where you are in the program at the time of the watch. This just means that you have to remember the tricky nuances of scope and extent of variables.

Conditional breakpoints

Breakpoints by themselves may seem too tedious. You have to keep stepping, and stepping, and stepping. . . But once we develop an idea for what the error could be (like dereferencing a **NULL** pointer, or going past the bounds of an array), we probably only care if such an event happens; we don't want to break at each iteration regardless. So ideally, we'd like to make a condition on a particular requirement (or set of requirements). Using conditional breakpoints allow us to accomplish this goal.

Conditional breakpoints are just like regular breakpoints, except that you get to specify some criterion that must be met for the breakpoint to trigger. We use the same **break** command as before, for example:

```
(gdb) break my_file.c:6 if i >= ARRAYSIZE
```

This command sets a breakpoint at line **6** of file my_file.c, which triggers only if the variable i is greater than or equal to the size of the array, which probably is bad if line 6 does something like **arr[i]**).

Conditional breakpoints can most likely avoid all the unnecessary stepping, etc. For our example, we know that the program crashes when **LinkedList::remove()** is called with a value of 1. So we might want to tell the debugger to only break at line **45** if **item_to_remove** is equal to 1. This can be done by typing the following command:

```
(gdb) condition 1 item_to_remove==1
(gdb)
```

This basically says "Only break at Breakpoint 1 if the value of **item_to_remove** is 1." Now we can run the program and know that the debugger will only break here when the specified condition is true.

Continuing with the example above, we have set a conditional breakpoint and now want to go through this method one line at a time and see if we can locate the source of the error. This is accomplished using the "**step**" command. **gdb** has the nice feature that when **ENTER** is pressed without typing a command, the last command is automatically used. That way we can step through by simply tapping the **ENTER** key after

the first "**step**" has been entered. Type **run**, and answer **y**, and when the prompt comes back, type **step**, then repeatedly, press **ENTER** until the program crashes. Here is what this should look like:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/Programs/gdb/main
Creating Node, 1 are in existence right now
Creating Node, 2 are in existence right now
Creating Node, 3 are in existence right now
Creating Node, 4 are in existence right now
The fully created list is:
4
3
2
1

Now removing elements:
Creating Node, 5 are in existence right now
Destroying Node, 4 are in existence right now
4
3
2
1


Breakpoint 1, LinkedList::remove (this=0x602010,
    item_to_remove=@0x7fffffffe50c: 1) at main.c:45
45          Node *marker = head_;
(gdb) step
46          Node *temp = 0;     // temp points to one behind as we iterate
(gdb)
48          while (marker != 0) {
(gdb)
49            if (marker->value() == item_to_remove) {
(gdb)
Node::value (this=0x6020b0) at main.c:24
24        const int& value () const { return value_; }
(gdb)
LinkedList::remove (this=0x602010, item_to_remove=@0x7fffffffe50c: 1)
    at main.c:68
68            marker = 0;  // reset the marker
(gdb)
69            temp = marker;
(gdb)
70            marker = marker->next();
(gdb)
Node::next (this=0x0) at main.c:22
22        Node* next () const { return next_; }
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a50 in Node::next (this=0x0) at main.c:22
22        Node* next () const { return next_; }
(gdb)
```

Note that the debugger steps into functions that are called. If you don't want to do this, you can use "**next**" instead of "**step**" which otherwise has the same behavior. The error in the program is obvious. At line **68** `marker` is set to **0**, but at line **70** a member of `marker` is accessed. Since the program can't access memory location **0**, the *segmentation fault* occurs. In this example, nothing has to be done to `marker` and the error can be avoided by simply **removing line 68 from main.c**.

Other Useful Commands

Backtrace	Produces a stack trace of the function calls that lead to a segment fault.
Where	Same as **backtrace**; you can think of this version as working even when you're still in the middle of the program.
Finish	Runs until the current function is finished.
Delete	Deletes a specified breakpoint.
info breakpoints	Shows information about all declared breakpoints.
Quit	Exits the debugger.

Using Pointers with gdb

First, let's assume we have the following structure defined:

```
struct entry {
    int key;
    char *name;
    float price;
    long serial_number;
};
```

Maybe this **struct** is used in some sort of hash table as part of a catalog for products, or something related. Now, let's assume we're in **gdb**, and are at some point in the execution after a line that looks like:

```
struct entry *e1 = <something>;
```

We can do a lot of things with pointer operations, just like we could in **C**:

See the value (memory address) of the pointer:

```
(gdb) print e1
```

See a particular field of the **struct** the pointer is referencing:

```
(gdb) print e1->key
(gdb) print e1->name
(gdb) print e1->price
(gdb) print e1->serial_number
```

You can also use the dereference (*) and dot (.) operators in place of the arrow operator (->):

```
(gdb) print (*e1).key
(gdb) print (*e1).name
(gdb) print (*e1).price
(gdb) print (*e1).serial number
```

See the entire contents of the **struct** the pointer references (you can't do this as easily in **C**!):

```
(gdb) print *e1
```

You can also follow pointers iteratively, like in a linked list:

```
(gdb) print list_prt->next->next->next->data
```

You can be exit **gdb** by typing:

```
(gdb) quit
```

Exercises: (4 points)

1. If you look at the output from running the program after fixing the bug we discovered above, you will see first of all that the program runs without crashing, but there is a memory leak somewhere in the program. Use **gdb** to locate and fix this bug. Save the fixed program and submit it to your TA.

2. There is another bug in the source code for the linked list that is not mentioned in the above code. The bug does not show up for the sequence of inserts and removes that are in the provided driver code, but for other sequences the bug shows up. Modify the driver as follows: insert 1, 2, 3, and 4, and then try to remove 2. This sequence will reveal the bug. Use **gdb** to locate and fix this bug as well. Save the fixed program and submit it to your TA with a proof it works for the above case.