EC327 Spring 2012
In-Class Lab #2

The GNU C++ Compiler and the GNU Debugger

This lab exercise is loosely based on Prof. Trachtenberg's materials for last year's spring 2011 EC327 class, and then heavily modified by D.Cullen on 2012-02-21. Enjoy!

Contents

Introduction

In In-Class Lab #1, you learned how to compile and run simple C++ programs. In this lab, you will learn a little more about the steps in the compilation process and some of the options that the compiler can take. We will also show you how several files can be compiled together into a single executable. The second half of the lab will introduce you to the GNU Debugger (GDB) and show you how to use it to debug programs. By the end of this lab, you will probably be more comfortable with using the Linux programming environment, which is important in preparing you for the midterm exam.

The GNU C++ Compiler

In In-Class Lab #1, you used g++ to compile your code. To compile your code, program g++ performs several steps: *preprocessing*, *parsing*, *compiling*, and *linking*. First, the *preprocessor* runs and reads in all of the header files that you specified with the *#include* statements and performs all of the textual substitutions for the constants you defined with *#define* statements.[1] After the preprocessor finishes, the *parser* interprets all of your C++ code and checks for syntax errors. Next, the *compiler* runs, generating assembly language instructions and creating *object files* that store pieces of the compiled code. Finally, the *linker* merges all of the pieces from the compiler into a single executable program.

The following exercises will help to familiarize you with each of the steps in the compilation process. Some of these exercises (given in red text) require you to use the debugger (`gdb`); for now, skip these exercises until after you have learned about the debugger in the second half of the lab.

Setup

First, use your favorite text editor to create a new source code file called `hello.cpp` containing the following:

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World!" << endl;
  return 0;
}
```

Test your program to make sure it successfully compiles and runs:

```
g++ hello.cpp
./a.out
```

Now you are ready to learn about the preprocessor.

Preprocessor

1. Type `g++ -E hello.cpp` to view the results of the GNU C++ preprocessor.
    1. `g++` is a freely available C++ computer distributed by the Free Software Foundation; it automatically links in C++ libraries. If you're compiling plain C code, you can use gcc.
    2. The `-E` option tells the compiler to only run its pre-processor, and then quit.
    3. `hello.cpp` is the file that you would like to compile.
2. Unless you can read at superhuman speed, the text will scroll by your window way too fast for you to read it.
3. Try again with the following command `g++ -E hello.cpp | less`.
    1. The `|` is called a pipe. It tells the operating system to take all the output of the previous command and provide it as input to the following command.
    2. The program `less` takes a large amount of input and lets you see it one window-full at a time (it is technically called a pager).
    3. Now you will see one page of output at a time. Press "space" to go to the next page. Use the “up” and “down” arrow keys to scroll by a line at a time. Press “q” to quit.
4. The first thing you'll see is the library files that are included with your `#include` directive. You'll see that there is a lot of code that is included before `g++` even touches your own code.
5. At the very beginning of the file, you will see something like `# 1 "hello.cpp"`. This is telling you that the next lines are the result of evaluating the first line of your code.
6. Near the end, you will see something like `# 2 "hello.cpp"`. Then your code will be at the bottom.

---

1 In fact, lines that start with # (e.g., #include, #define, #ifdef, #endif, #ifndef, etc) are called *preprocessor directives* because they are handled by the preprocessor.

Parser

1. The parser is the first compiler component to run after the preprocessor.
2. Type `g++ -c hello.cpp` to run the compiler. You should see no errors if you entered the code correctly.
3. To see what the `-c` option does, type `man g++` to get the `g++` manual. Scroll down until you get a description of the `-c flag`. What does it do? (You can type `/-c` and then "enter" to search the man pages for `-c`, then press the "n" key to go to the next match. Press "q" to quit out of the man pages.)

Compiler

1. The compiler converts your C++ code to assembly language instructions. To see the assembly instructions that it creates for your program, run `g++ -S hello.cpp`, which will create a file called `hello.s` with the assembly language instructions. Open this file with your favorite text editor to view the assembly code.
2. Use `g++ -c hello.cpp` to compile your code into the object file hello.o. Then type `gdb hello.o` to run your object code through the interactive GNU debugger.
    1. The gnu debugger has many options, which you can explore.
    2. Type `x/44b main` to view the first 44 bytes of the main procedure in memory.
        1. Type `x/44t main` to view the same memory in binary.
    3. Type `disassemble main` in the gnu debugger to get the assembly language for the same code.
        1. Can you figure out the machine code for the mov command? (not easy)

Linker

1. Type `g++ hello.o` to link all your object files together (in this case, you only have one), in addition to the standard gnu library (which has code for printing to the screen, for example).
2. Note `hello.o` is a binary file and therefore will look unintelligible if you open it in a text editor. If you are curious to see what it contains, you can try one of the following:
    1. `emacs` has a special mode for displaying binary files: simply open the file then run `M-x hexl-mode` to view the binary file. The left-hand contains the byte address followed by a colon, followed by the hex codes of the bytes in the file (grouped into several columns), followed by the ASCII equivalents of the hex code (or a period "." if no printable ASCII character exists).
    2. The Linux programs `hd` and `od` (which stand for "hex dump" and "octal dump", respectively) can also be used to display the bytes in a binary file. Try running the following:
        ```
        hd a.out | less
        ```
        This command prints out all of the bytes of the file in hex and pipes it through the program `less` for easier viewing. The program `od` does something similar, except that it displays the data in octal (base 8) instead of hexadecimal (base 16).
3. Can you find the bytes that contain the string "`Hello World!`"?
4. Similarly, `a.out` is also a binary file and can be viewed in the same way that you viewed `hello.o`.

Additional Remarks

- When we use the term "compile", we usually are referring to this overall process, even though g++ actually performs several steps (preprocessing, parsing, compiling, and linking) and only one of these steps is technically called "compiling".
- In general, at least for the programs you write in this class, you will compile and link in a separate step, rather than in two separate steps. For example, you will do this to compile and link:
    ```
    g++ main.cpp
    ```
    rather than compiling and linking as two separate steps:
    ```
    g++ -c main.cpp
    g++ main.o
    ```
    For large projects with many source code files, it is sometimes advantageous to compile and link in separate steps because you only need to recompile the code that you change, leading to faster compile times. However, for this class, it is easier to compile and link as a single step.

Compiler Options

You've already seen the `-o` compiler option in In-Class Lab #1. In the above exercises, you experimented with the `-E`, `-c`, and `-S` compiler flags. To learn more about any of the compiler options, you can always check the manual pages with the command `man g++`. Here is a list containing some of the more frequently-used compiler parameters:

- `-c` compile only (i.e., create object (`.o`) files); do not link
- `-o` specify name of output executable (default is `a.out`)
- `-g` compile with debug support (i.e., for use with GDB)
- `-pg` compile with support for the GNU profiler (gprof)
- `-Wall` show all warnings
- `-pedantic` display all warnings demanded by strict ISO C++.
- You can pass options to the linker (`ld`) program when you run the compiler. For example, you can print a linker map using any of the following lines:
  - `g++ -Wall -Xlinker --print-map main.cpp`
  - `g++ -Wall main.cpp -Wl,--print-map`
  - `g++ -Wall main.cpp -Wl,-M`
- `-E` print output of preprocessor
- `-S` generate a "`.s`" file containing the assembly code
- `-O` specify compiler optimization level. (The following is paraphrased from the gcc man pages.)
  - `-O` Same as "`-O1`".
  - `-O0` No optimization. This is the default.
  - `-O1` Optimize. Compiler tries to reduce code size and execution time, but it doesn't try performing optimizations that take a great deal of compilation time.
  - `-O2` Optimize more. Compiler performs more optimizations, but not ones that involve a space-speed tradeoff, such as loop unrolling or function inlining.
  - `-O3` Optimize even more. Performs all of the optimizations of "`-O2`", and then additionally performs loop unrolling and function inlining optimizations.
  - `-Os` Optimize to reduce code size. Basically runs all of the "`-O2`" optimizations that do not typically increase code size, and then it runs further optimizations designed to reduce code size.
- The `-D` option allows you to define a preprocessor constant at compile time. For example, if you want to define PI at compile time rather than with "`#define PI 3.14159`" in your code, you could compile with "`-DPI=3.14159`".
- `-l`*`library`* – Link in a specific library
- `-L`*`dir`* – Specify a directory in which to search for library files
- `-I`*`dir`* – Specify an include files (i.e., header files) directory.

Here is an example of using the "`-D`" option. First create a new file "`test.cpp`" containing the following code:

```
#include <iostream>
using namespace std;
int main(int argc, char **argv)
{
  cout << "PI is defined as " << PI << endl;
  return 0;
}
```

Try compiling like this:

```
g++ test.cpp
```

The compile should fail, with an error message stating that "PI" isn't defined. Now try compiling like this:

```
g++ -DPI=3.14159 test.cpp
```

The compile should succeed, and when you run the program, it should print out the value of PI that you defined at compile time. Try changing the value of PI, recompiling, and running, and you'll see that the result changes.

You will use the "`-g`" option later in this lab in order to add GDB debugger support to your compiled program.

Compiling multiple files

You have already started to learn about functions in the lectures and in HW#1. Up until this point, you probably have put all of the functions that you've written into a single source file. Consider the following example source code file, named "`main.cpp`", that contains the following lines:

```
#include <iostream>
using namespace std;

void print_hello(void)
{
  cout << "Hello, World!" << endl;
}

int main(int argc, char **arvg)
{
  print_hello();
  return 0;
}
```

It is also possible to put the `print_hello()` function definition in a separate file from the `main()` function. But first, it is important for us to review the difference between a function *declaration* and a function *definition*.

A function *declaration* provides the name of the function and the list of arguments, but it does not include the lines of code that make up the body of the function. A function declaration is also often called a function *prototype* or *signature* or *header* (some people argue that there is a slight distinction between each of these terms, but most programmers use the terms interchangeably). The declaration basically tells the compiler, "*Hey, here is the name and arguments of a function. The code that goes inside this function hasn't been given yet, but don't freak out if you see this function being executed because we promise that we'll define the rest of it later in the code, just keep reading.*" Here is what the function declaration looks like for the function from the code above:

```
void print_hello(void);
```

Note that the function declaration line ends with a semicolon. Note that the first keyword `void` means that this function doesn't return anything, and the second keyword `void` means that it doesn't take any arguments. Many programmers leave out the second `void` (i.e., `void print_hello();`) but I tend to include it just to be explicit; both ways are equivalent. The first `void` cannot be omitted because you must explicitly tell the compiler when the function returns no values.

A function *definition* provides the function declaration (without the semicolon), followed by the body of the function. Here is the definition for the function from above:

```
void print_hello(void)
{
  cout << "Hello, World!" << endl;
}
```

It is important to note that a function *definition* contains **both** the *declaration* **and** the *definition.* However, the reverse is not true; a function *declaration* does **not** contain a *definition*.

Before we move on to putting the function in a separate file, let's do a few more experiments with the original source file (`main.cpp`). First, move the `print_hello()` function definition down below `main()`, as shown below, and then try compiling.

```
#include <iostream>
using namespace std;

int main(int argc, char **arvg)
{
  print_hello();
  return 0;
}

void print_hello(void)
```

```
{
  cout << "Hello, World!" << endl;
}
```

This code will fail to compile. Why? Because the compiler reads through the code line-by-line, and it sees the function `print_hello()` being executed in `main()`, but it doesn't know what to do; it doesn't know what arguments this function takes or what value it returns because it hasn't seen a *declaration* for that function yet. The compiler is not smart enough to read to the end of the file, see the function definition, and then go back to where the function was called and finish compiling. The way you solve this problem is that you put a function *declaration* line into your code before the `main()` function, as shown below:

```
#include <iostream>
using namespace std;

void print_hello(void);

int main(int argc, char **arvg)
{
  print_hello();
  return 0;
}

void print_hello(void)
{
  cout << "Hello, World!" << endl;
}
```

Now your code should compile properly.

You are allowed to *declare* a function as many times as you want, but you can only *define* it once. Otherwise, how would the compiler know which definition to use? Note that function overloading still obeys this rule; even though the overloaded functions have the same, they take different arguments, so they are still considered to be separate functions, and the compiler can distinguish between them. Anyway, the following example shows that you can declare a function as many times as you want:

```
#include <iostream>
using namespace std;

void print_hello(void);
void print_hello(void);
void print_hello(void);
void print_hello(void);

int main(int argc, char **arvg)
{
  print_hello();
  return 0;
}

void print_hello(void);
void print_hello(void);
void print_hello(void);

void print_hello(void)
{
  cout << "Hello, World!" << endl;
}
```

This code compiles just fine, but of course it is silly to include so many redundant function declarations :-)

Now we are ready to explain how to put the `print_hello()` function in a separate file. The standard convention that all C/C++ programmers follow is to put function *declarations* into *header* files (*.h* files) and to put function *definitions* in source code files (*.cpp* files). (Sometimes the source code files are called *implementation* files because they contain the actual code that implements the functions, not just the function prototypes.)

Your file names must end with either a "`.h`" or a "`.cpp`" extension (or one of the others in the list below); otherwise, the compiler does not know that your files contain C++ code. The following list gives a few other variations that you might see, but in this class you will probably just stick with "`.h`" or "`.cpp`".

- `.c` C source code files
- `.h` Can be either C or C++ header files
- `.cpp` C++ source code files
- `.C` (capital C) Occasionally, people use this instead of ".cpp" for C++ source files.
- `.hpp` Occasionally, people use this instead of ".h" for C++ header files.

Create a new file called "`hello.h`" that contains the following lines:

```
#ifndef HELLO_H
#define HELLO_H

void print_hello(void);

#endif
```

We'll explain what the preprocessor directives (`#ifndef, #define, #endif`) mean later. Next, create a new file called "`hello.cpp`" that contains the following lines:

```
#include <iostream>
using namespace std;

#include "hello.h"

void print_hello(void)
{
  cout << "Hello, World!" << endl;
}
```

Finally, modify your `main.cpp` program so that it looks like this:

```
#include <iostream>
using namespace std;

#include "hello.h"

int main(int argc, char **arvg)
{
  print_hello();
  return 0;
}
```

Next we will compile this code. Try running the following command:

```
g++ main.cpp
```

Oh no! It didn't work! You should have gotten an error message that looks something like this:

```
/tmp/ccLfMa3U.o: In function `main':
main.cpp:(.text+0x7): undefined reference to `print_hello()'
collect2: ld returned 1 exit status
```

What does this cryptic error message mean? This is a linker error. The compiler was able to finish the preprocessing, parsing, and compilation steps just fine, but when it got to the linking step, it failed. Generally, if you get a error messages like this that contain things like temporary `.o` files (i.e., `ccLfMa3U.o`), hex codes (i.e., `(.text+0x7)`), and messages about the linker (i.e., `ld returned 1 exit status` – recall that `ld` is the name of the linker), it means that you have a linking error. Linking errors are usually harder to debug than syntax errors because they don't reference line numbers of in your code. The good news is that if you get linker errors, your code itself is probably just fine, because the `g++` finished the parsing and compile steps and failed on the final step (linking). The linker just can't find all of the compiled pieces of code that it needs to build the final program. In this example, the linker can't find the compiled code for the function `print_hello()`. Why? Because we forgot to compile it into our program! Now try the following:

```
g++ main.cpp hello.cpp
```

This time, your code should compile without problems. But why couldn't the compiler find our `print_hello()`

function earlier? After all, the file `main.cpp` contains the line `#include "hello.h"`; why couldn't it find the function? Because `hello.h` contains only the *declaration* of the function, not the function *definition*. Since our `g++ main.cpp` command was missing the file `hello.cpp`, the function body never got compiled, which explains why the linker couldn't find the compiled code for this function!

Another important thing to know is that *the `g++` command only takes the names of source code (`.cpp`) files to `g++`, never header (`.h`) files.* The reason for this is that only `.cpp` files contain actual statements that can be compiled into assembly instructions and then assembled into machine code. Header files only contain things like definitions of constants and function *declarations,* not function bodies, so they never contain any statements to be compiled or assembled. (A special type of functions called *inline functions* is one exception to this, but you will learn about them much later in the course.) In general, you should **never** include function definitions in header files. The `main.cpp` and `hello.cpp` files can find the header file because they contain the `#include "hello.h"` statement, so there is no reason why the header files would ever need to be passed as arguments to `g++`.

And now for a quick note on global variables. Our simple example program doesn't include any global variables, but you need to know where to define them in case you ever need to use them in the future. Global variables should be *defined* in `.cpp` files and *declared* in header files. Inside the header file, you use the `extern` keyword to let the compiler know that it is just a declaration, not a definition, and the actual storage is defined in a different (or "external") file. For example, here is an example of a global variable definition:

```
int myglobalvariable;         // (this definition line goes in the .cpp file)
```

and

```
extern int myglobalvariable;  // (this declaration line goes in the .h file)
```

Don't put the definition in a header file, or else you could get "multiple definition" compiler errors.

Another point worth noting is that although the `#include "hello.h"` line in `main.cpp` is necessary, but this line is technically unnecessary in `hello.cpp` because the function *definition* is also a function *declaration*. However, it's generally helpful to put the `#include` statement in the function's source code file as well, because if we were to add to this program in the future, it might be handy to have the function declaration included at the top of the source code file. For example, let's suppose that we define a new function called `print_twice()` in `hello.cpp` and this new function executes the function `print_hello()` two times. If we want `print_twice()` to be able to find `print_hello()`, we need to either define `print_twice()` after we define `print_hello()`, or we would need to declare `print_hello()` before we define `print_twice()`. (This is the exact same type situation that we described at the very start of this section, in which `main()` and `print_hello()` were both defined in `main.cpp`, and `print_hello()` was being executed from within `main()`.) When you start writing many functions, it gets harder to keep track of what function depends on what, so the easiest thing to do is to put all of their *declarations* into the header file and all of their *definitions* in the `.cpp` file (in any order). Another reason why you would put `#include hello.h` in `hello.cpp` is that you often define all of your constants (for example, PI=3.14159, etc.) inside header files, and your functions in `hello.cpp` might need to use these constants.

It is absolutely essential to understand that the preprocessor performs textual substitution. This means that when you use `#include` statements, it literally copies and pastes the lines of the header file into the source file when it compiles it. For example, the line `#include "hello.h"` in `main.cpp` tells the preprocessor to copy all of the lines from `hello.h` and to replace the line `#include "hello.h"` with the lines from `hello.h`. The `#define` preprocessor directives work the same way; for example, if your code contains the line `#define PI 3.14`, the preprocsesor essentially performs a find-and-replace operation, and everywhere it sees the text "`PI`", it replaces it with the text "`3.14`". Of course, the preprocessor doesn't actually overwrite your `main.cpp` file; instead, it performs all of these operations into a temporary file. You can view the result of this temporary file with the "`-E`" option for `g++`. You did this earlier in this lab. Try compiling with the `-E` option again and now it will probably make more sense.

What are the lines `#ifndef HELLO_H`, `#define HELLO_H`, and `#endif` used for? These lines are called ***include guards***. They prevent the preprocessor from including a header file more than once in the same source file. It is okay to include a header file in multiple source files (for example, you included `hello.h` from

both `main.h` and `hello.cpp`), but you shouldn't include a header file multiple times in the *same* source file. The `#ifndef` statement checks to make sure the preprocessor constant `HELLO_H` hasn't already been defined. If it hasn't been defined, the lines between `#ifndef` and `#endif` will be included when the header file is included with a `#include hello.h` statement. If `HELLO_H` has already been defined, these lines will be not included. The `#define HELLO_H` statement causes `HELLO_H` to be defined after the first time it is included, so that it cannot be included again in the future. By convention, we usually use the name of the header file (in this case `hello.h`), written in ALL CAPITALS, with the periods (“.”s) replaced with with underscores (“_”s) for the name of the preprocessor constant.

Why shouldn't a header file be included multiple times? Above we learned that only declarations will go into header files, and functions can be declared as many times as we want without a problem, so why does it matter if we include the header several times? One reason is that a special type of function, called an *inline function*, actually needs to be *defined* in a header file. If the inline function was included more than once, the compiler would produce a “multiple definition error” and fail to compile. The include guards guarantee that the statements from the header file only get included once, thus preventing these multiple definition errors.

In our simple example, `hello.h` is so simple that we don't need the #include guards, but we have included them simply because it is good practice. It doesn't take long before your simple programs become larger and more complex, at which point it is deceptively easy to include a header file multiple times by mistake. For example, if your source code includes several different header files, and each of these includes other header files, and two more more of these other header files are the same file, then this header file would be included multiple times if you did not have the #include guards. So the bottom line is that you should get into the habit of always using #include guards.

It may seem like a lot of unnecessary work to make this distinction between header files and source files. Why don't we just have source files? Why do we need header files? Why do we have declarations that are separate from the definitions? Why doesn't the compiler just scan through the code multiple times, so that it can resolve all of the references, rather than requiring functions to be declared before they are used? There are many reasons for this; here are just a few:

- Historical reasons. Compilers run much faster when they only need to perform one or two passes. The programs you write in this class are small so they compile in under a second, but speed becomes critical for larger projects, such as operating system kernels, which may contain hundreds of thousands or millions of lines of code. For example, compiling one of today's Linux kernels for a typical desktop system can easily take several hours. Computers are much faster today than 20 or 30 years ago; back then, designing efficient compilers was especially critical. Separating header files from implementation files helped to speed up compile time, as well as designing programs such as GNU Make, which compile pieces of the program into separate *translation units* so that only the pieces that change need to be recompiled.
- Good programming practice. One fundamental programming principle of designing good software libraries is to hide the implementation details from the other programmers who will be using those libraries. You can give a programmer a set of precompiled libraries and the header files but none of the implementation (`.cpp`) files. The programmer can then write code that use the functions from the libraries. The header files provide the compiler with everything it needs to compile the programmer's code, and the linker simply links the programmer's code with the precompiled library functions to produce the final executable. Hiding the implementation details from the programmer is important because it allows the programmer to think about the problem on a higher level, using the library function calls; the programmer doesn't need to worry about all of the internal details. Moreover, suppose that in the future, the writers of the library come up with a new technique to make the library functions run more efficiently. As long as they keep the interface the same (i.e., the function prototypes in the header file), whoever uses the libraries doesn't need to change their code; they only need to re-link against the newest version of the libraries. Separating the header files and the implementation files helps programmers to create better software libraries.

This is a lot of information all at once, but if you come back and re-read it again later in the course, it will make more sense.

The GNU Debugger (GDB)

Debugging Methodology

Most of the time, you will debug your code by peppering it with print statements to display the results of variables at intermediate steps. However, this is not always the best approach. Sometimes it isn't even possible, especially when working with embedded systems, which might not have a terminal interface with which to print. Even if terminal output is available, it might be more practical simply to pause the running program in order to examine the variables and memory contents.

The GNU Debugger, usually called just GDB, is the standard debugger for the GNU software system. It is a portable debugger that runs on many Unix-like systems and works for many programming languages.

Unfortunately, it is not always possible to use the debugger. For example, many real-time applications cannot be paused because pausing them would cause the system to stop working or would prevent the error from being reproduced. Applications that involve networking and careful timing are also difficult to debug with GDB. For more sophisticated programs, more sophisticated debugging techniques must be used. However, it is important for every good programmer to be familiar with a variety of debugging techniques, and GDB is one of these.

Of course, the best way to debug is simply to avoid debugging in the first place. There is a lot of truth behind this tongue-in-cheek statement. Extreme Programming (XP) is one style of programming that follows this the philosophy. The main idea behind XP is that you always write testbenches, which are small programs used to verify that the rest of your code works. In fact, XP recommends writing the testbenches even before you write the code itself! That way, you have a very good idea of how your program is supposed to work before you even start writing it. Moreover, by forcing yourself always to check the behavior of your code, you'll never need to spend any time debugging. The concept of self-verification or self-validation becomes even more important when you have very large projects, in which a small change in one area may break another part of the code without you realizing it. However, by running the testbenches automatically every time you make a change, you can detect and correct these errors immediately.

Using the GNU debugger (GDB)

The purpose of a debugger such as GDB is to 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.

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

- Start your program, specifying anything that might affect its behavior.
- 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.

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source code line numbers and addresses in the executable code.

To request debugging information, specify the `-g` option when you run the compiler.

GDB Quickstart

When using GDB, it is helpful to have a cheat sheet of common commands. A pretty good cheat sheet can be downloaded here: http://www.ece.utexas.edu/~adnan/gdb-refcard.pdf

To help you get started, the following is another list of some of the most common GDB commands.

Stopping Execution

- `break`
    - A *breakpoint* makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the 'break' command and its variants to specify the place where your program should stop by line number, function name or exact address in the program.
        - break FUNCTION Set a breakpoint at entry to function FUNCTION.
        - break LINENUM Set a breakpoint at line LINENUM in the current source file.
        - break FILENAME:LINENUM Set a breakpoint at line LINENUM in source file FILENAME.
        - break FILENAME:FUNCTION Set a breakpoint at entry to function FUNCTION found in file FILENAME.
        - break ... if COND Set a breakpoint with condition COND; evaluate the expression COND each time the breakpoint is reached, and stop only if the value is nonzero--that is, if COND evaluates as true. ... stands for one of the possible arguments described above (or no argument) specifying where to break.
- `watch`
    - You can use a watchpoint to stop execution whenever the value of an expression changes, without having to predict a particular place where this may happen.
        - watch EXPR Set a watchpoint for an expression. GDB will break when EXPR is written into by the program and its value changes.
- `info`
    - Print a table of all breakpoints and watchpoints set and not deleted, with the following columns for each breakpoint: info breakpoints [N]
        - `info break [N]`
        - `info watchpoints [N]`
- `clear`
    - Delete any breakpoints at the next instruction to be executed in the selected stack frame. When the innermost frame is selected, this is a good way to delete a breakpoint where your program just stopped.
        - clear FUNCTION
        - clear FILENAME:FUNCTION Delete any breakpoints set at entry to the function FUNCTION.
        - clear LINENUM
        - clear FILENAME:LINENUM Delete any breakpoints set at or within the code of the specified line.
- `delete [breakpoints] [BNUMS...]`
    - Delete the breakpoints or watchpoints of the numbers specified as arguments. If no argument is specified, delete all breakpoints (GDB asks confirmation, unless you have '`set confirm off`'). You can abbreviate this command as '`d`'.

Running/Resuming Execution

- `run`
    - Use the 'run' command to start your program under GDB. You must first specify the program name with an argument to GDB, or by using the 'file' or 'exec-file' command.
- `step`
    - Continue running your program until control reaches a different source line, then stop it and return control to GDB. This command is abbreviated s.
        - step [COUNT] (shorthand `s`) 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]` (shorthand `n`)
    - Continue to the next source line in the current (innermost) stack frame. This is similar to `step', but function calls that appear within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the original stack level that was executing when you gave the 'next' command. This command is abbreviated 'n'.
    - `continue [IGNORE-COUNT]` (shorthand `c`)
    - Resume program execution, at the address where your program last stopped; any breakpoints set at that address are bypassed. The optional argument IGNORE-COUNT allows you to specify a further number of times to ignore a breakpoint at this location.
- `finish`
    - Continue running until just after function in the selected stack frame returns. Print the returned value (if any).

Examining the Data and Source Code

- `print [EXP]` (shorthand `p`)
    - EXP is an expression (in the source language). By default the value of EXP is printed in a format appropriate to its data type; If you omit EXP, GDB displays the last value again.
- `display EXP` (shorthand `disp`)
    - Add the expression EXP to the list of expressions to display each time your program stops. 'display' does not repeat if you press RET again after using it.
- `list` (shorthand `l`)
    - To print lines from a source file, use the '`list`' command (abbreviated '`l`'). By default, ten lines are printed. There are several ways to specify what part of the file you want to print.
    - Here are the forms of the `list' command most commonly used:
        - `list LINENUM` Print lines centered around line number LINENUM in the current source file.
        - `list FUNCTION` Print lines centered around the beginning of function FUNCTION.
        - `list` Print more lines. If the last lines printed were printed with a `list' command, this prints lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (*note Examining the Stack: Stack.), this prints lines centered around that line.
        - `list -` Print lines just before the lines last printed.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there. Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a "stack frame". The stack frames are allocated in a region of memory called the "call stack".

- `backtrace` (shorthand `bt`) Print a backtrace of the entire stack: one line per frame for all frames in the stack.

GDB Example

Create a new source code file called `test.cpp` that contains the following code:

```
#include <iostream>
using namespace std;

int main(int argc, char **arvg)
```

```
{
  unsigned long int start;
  cout << "Please enter the starting value: ";
  cin >> start;
  cout << "You entered " << start << "." << endl;

  unsigned long int a = start;
  unsigned long int b = 0;
  unsigned long int c = 0;
  unsigned long int sum = a + b + c;
  while (true)
  {
    c = b;
    b = a;
    a = a + 1;
    sum = a + b + c;
  }

  return 0;
}
```

This simple example performs some useless arithmetic in a never-ending loop. Now compile the code with the following command:

```
g++ -g test.cpp
```

Note that we use the `-g` compiler flag to enable GDB support in the executable. We used unsigned long integers for the datatypes, just to

Now we will run the program with the debugger in order to analyze its behavior.

- Type `gdb a.out` to run the debugger on your executable.
- Press `r` to have it run the program.
- Type in any number, let's say `50`, and press enter.
- Let the program run for a few seconds, and then press `Control-C` to stop it so that we can inspect what is going on.
    - The debugger will display the current line that it is executing.
    - Type `list` to see some context for this line
    - Type `where` to see the call stack (we will discuss this later).
- Type `b 20` to set a *breakpoint* on line `20` of `test.cpp`. (This should be the line `sum = a + b + c;` in the loop, assuming that source code file is identical to mine.) A breakpoint tells the debugger to stop execution when it reaches a certain line.
    - Our example is simple, so it only contains one source code file, but if you have another program multiple source code files, you can specify which line of which file using the syntax `b test.cpp:20`. It is also possible to set a breakpoint on a specific function using the syntax `b functionname`.
- Press `c` to continue execution. The execution will *break* (i.e., pause) when it hits line `20`.
- You can get help with the `help` command. For example, if you want more help on breakpoints, you can type `help breakpoints`.
- *Watchpoints* cause the debugger to stop execution when a particular expression has changed its value. For example, type `watch sum` to have the debugger stop every time the variable `sum` changes.
- During execution, you can view any variables that are in scope with the `p` command. For example, `p sum` will print the variable `sum` and `p a` will print the variable `a`. You can also try `p a*b` to print the product of variables `a` and `b`, or any other expression you want. You can even change the value of any variable; try using the command `p a=10`, for example, to set the variable `a` to have the value `10`.

- To step through the code one line at a type, use the commands `step` (`s` for short) or `next` (`n` for short). These commands are equivalent as long as subroutine calls (i.e., function calls) do not happen. How do these commands differ for function calls? Use the help command on each of these commands and try to figure out the difference.

Here are a few more things that you can try with GDB:

- Temporary Breakpoints. Breakpoints generally stop the debugger every time they are encountered. If you want the breakpoint to stop the debugger just once, use a temporary breakpoint (`tbreak`).
- Backtrace. Type backtrace to see a list of the functions currently on the call stack. You can go up or down the stack with `up` and `down`. Of course, our example is so simple that there are no function calls. Modify the source code by writing a few simple functions and adding function calls to the body of the main loop. Then experiment with the `backtrace` command to see how it works.

GUI Version of GDB

GDB can be difficult to learn because it is hard to remember all of the commands if you don't use them on a regular basis. Fortunately, GUI versions of GDB do exist. One popular graphical front-end for GDB is `ddd`. See http://www.gnu.org/software/ddd/ for more information and documentation.