

Programmazione Avanzata per il Calcolo Scientifico

Advanced Programming for Scientific Computing

Lab 03 - 27/03/2015

Carlo de Falco

CMake

CMake Tutorial

CMake is a **cross-platform**, open-source build system.

CMake is a family of tools designed to build, test and package software.

CMake is used to control the software compilation process using simple platform and compiler independent configuration files.

CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

Let's follow the tutorial at <http://www.cmake.org/cmake-tutorial/>

CMake Tutorial

Step I: Build executable from source code files

Step II: Add a (static) Library

Step III: Installing and Testing (CTest)

Step IV: Adding System Introspection

Step V: Adding a Generated File and Generator

Step VI: Building an Installer (CPack)

Step VII: Test Results submission

STEP VII :

**DIY, convert the preliminary example
on Makefiles to use CMake**

Autotools

Autotools

The GNU build system, also known as the Autotools, is a suite of programming tools designed to assist in making source code packages portable to many **Unix-like systems**.

It consists of:

- GNU Autoconf
- GNU Automake
- GNU Libtool
- Gnulib

GDB

Debugging Under Unix: `gdb` Tutorial

Contents

1. [Introduction](#)
 1. [Who should read this?](#)
 2. [Source code](#)
 2. [Preparations](#)
 1. [Environment settings](#)
 2. [Debugging symbols](#)
 3. [Debugging](#)
 1. [When to use a debugger](#)
 2. [Loading a program](#)
 3. [Inspecting crashes](#)
 4. [Conditional breakpoints](#)
 5. [Stepping](#)
 4. [Further information](#)
 5. [Notes](#)
-

Introduction

This tutorial was originally written for [CS 342](#) at [Washington University](#). It is still maintained by [Andrew Gilpin](#).

Who should read this?

This tutorial is written to help a programmer who is new to the Unix environment to get started with using the `gdb` debugger. This tutorial assumes you already know how to program in C++ and you can compile and execute programs. It also sort of assumes that you basically know what debugging is and that you have used a debugger on another system.

Source code

To help illustrate some of the debugging principles I will use a running example of a buggy program. As you progress through this tutorial, you will use the debugger to locate and fix errors in the code. The code can be downloaded [here](#) and a simple Makefile for the program can be downloaded [here](#).

The code is very simple and consists of two class definitions, a node and a linked list. There is also a simple driver to test the list. All of the code was placed into a single file to make illustrating the process of debugging a little easier.

Preparations

Environment settings

`gdb` is in the `gnu` package on CEC machines. If you don't have this package loaded then type `pkgadd -u gnu` at a shell prompt. If you can run `g++`, then you will be able to run `gdb`.

Debugging symbols

`gdb` can only use debugging symbols that are generated by `g++`. For Sun CC users, there is the `dbx` debugger which is very similar to `gdb`.

`gdb` is most effective when it is debugging a program that has debugging symbols linked in to it. With `g++`, this is accomplished using the `-g` command line argument. For even more information, the `-ggdb` switch can be used which includes debugging symbols which are specific to `gdb`. The makefile for this tutorial uses the `-ggdb` switch.

Debugging

When to use a debugger

Debugging is something that can't be avoided. Every programmer will at one point in their programming career have to debug a section of code. There are many ways to go about debugging, from printing out messages to the screen, 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 seg 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.

Go ahead and make the program for this tutorial, and run the program. 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. 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.

Loading a program

So you now have an executable file (in this case `main`) and you want to debug it. First you must launch the debugger. The debugger is called `gdb` and you can tell it which

file to debug at the shell prompt. So to debug main we want to type `gdb main`. Here is what it looks like when I run it:

```
aggl@sukhoi aggl/.www-docs/tutorial> gdb main
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb)
```

(Note: If you are using Emacs, you can run `gdb` from within Emacs by typing `M-x gdb`. Then Emacs will split into two windows, where the second window will show the source code with a cursor at the current instruction. I haven't actually used `gdb` this way, but I have been told by a very reliable source that this will work. :)

`gdb` is now waiting for the user to type a command. We need to run the program so that the debugger can help us see what happens when the program crashes. Type `run` at the `(gdb)` prompt. Here is what happens when I run this command:

```
(gdb) run
Starting program: /home/cec/s/a/aggl/.www-docs/tutorial/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.
Node<int>::next (this=0x0) at main.cc:28
28      Node<T>* next () const { return next_; }
(gdb)
```

The program crashed so lets see what kind of information we can gather.

Inspecting crashes

So already we can see that the program was at line 28 of main.cc, that `this` points to 0, and we 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, we type `backtrace` which gives me the following output:

```
(gdb) backtrace
#0  Node<int>::next (this=0x0) at main.cc:28
#1  0x2a16c in LinkedList<int>::remove (this=0x40160,
    item_to_remove=@0xffbef014) at main.cc:77
#2  0x1ad10 in main (argc=1, argv=0xffbef0a4) at main.cc:111
(gdb)
```

So in addition to what we knew about the current method and the local variables, we can now also see what methods called us and what their parameters were. For example, we can see that we were called by `LinkedList<int>::remove ()` where the parameter `item_to_remove` is at address `0xffbef014`. It may help us to understand our bug if we know the value of `item_to_remove`, so we want to see the value at the address of `item_to_remove`. This can be done using the `x` command using the address as a parameter. ("x" can be thought of as being short for "examine".) Here is what happens when I run the command:

```
(gdb) x 0xffbef014
0xffbef014:      0x00000001
(gdb)
```

So the program is crashing while trying to run `LinkedList<int>::remove` with a parameter of 1. We have now narrowed the problem down to a specific function and a specific value for the parameter.

Conditional breakpoints

Now that we know where and when the segfault is occurring, we want to watch what the program is doing right before it crashes. One way to do this is to step through, one at a time, every statement of the program until we get to the point of execution where we want to see what is happening. This works, but sometimes you may want to just run to a particular section of code and stop execution at that point so you can examine data at that location.

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. In our example, we want to look at the code in `LinkedList<int>::remove ()` so we would want to set a breakpoint at line 52 of main.cc. Since you may not know the exact line number, you can also tell the debugger which function to break in. Here is what we want to type for our example:

```
(gdb) break LinkedList<int>::remove
Breakpoint 1 at 0x29fa0: file main.cc, line 52.
(gdb)
```

So now Breakpoint 1 is set at main.cc, line 52 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.) So when the program is run, it will return control to the debugger everytime it reaches line 52. This may not be desirable if the method is called many times but only has problems with certain values that are passed. Conditional breakpoints can help us here. For our example, we know that the program crashes when `LinkedList<int>::remove()` is called with a value of 1. So we might want to tell the debugger to only break at line 52 if `item_to_remove` is equal to 1. This can be done by issuing 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.

Stepping

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. Here is what this looks like:

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

Starting program: /home/cec/s/a/aggl/.www-docs/tutorial/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<int>::remove (this=0x40160,
    item_to_remove=@0xffbef014) at main.cc:52
52      Node<T> *marker = head_;
(gdb) step
53      Node<T> *temp = 0;  // temp points to one behind as we iterate
(gdb)
55      while (marker != 0) {
(gdb)
56          if (marker->value() == item_to_remove) {
(gdb)
Node<int>::value (this=0x401b0) at main.cc:30
30      const T& value () const { return value_; }
```

```

(gdb)
LinkedList<int>::~remove (this=0x40160, item_to_remove=@0xffbef014)
    at main.cc:75
75         marker = 0; // reset the marker
(gdb)
76         temp = marker;
(gdb)
77         marker = marker->next();
(gdb)
Node<int>::~next (this=0x0) at main.cc:28
28     Node<T>* next () const { return next_; }
(gdb)

Program received signal SIGSEGV, Segmentation fault.
Node<int>::~next (this=0x0) at main.cc:28
28     Node<T>* next () const { return next_; }
(gdb)

```

After typing `run`, `gdb` asks us if we want to restart the program, which we do. It then proceeds to run and breaks at the desired location in the program. Then we type `step` and proceed to hit enter to step through the program. 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 75 `marker` is set to 0, but at line 77 a member of `marker` is accessed. Since the program can't access memory location 0, the seg fault occurs. In this example, nothing has to be done to `marker` and the error can be avoided by simply removing line 75 from `main.cc`.

If you look at the output from running the program, you will see first of all that the program runs without crashing, but there is a memory leak somewhere in the program. (Hint: It is in the `LinkedList<T>::remove()` function. One of the cases for `remove` doesn't work properly.) It is left as an exercise to the reader to use the debugger in locating and fixing this bug. (I've always wanted to say that. ;)

`gdb` can be exited by typing `quit`.

Further information

This document only covers the bare minimum number of commands necessary to get started using `gdb`. For more information about `gdb` see the `gdb` man page or take a look at a very long description of `gdb` [here](#). Online help can be accessed by typing `help` while running `gdb`. Also, as always, feel free to ask questions on the newsgroup or you can ask me during lab hours.

Notes

- 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. For example, inserting 1, 2, 3, and 4, and then trying to remove 2 will show

the error. Special thanks to Linda Gu and Xiaofeng Chen for locating this bug. The bug fix is pretty simple and is left as an exercise.

- Special thanks to Ximmo da Jazz for providing valuable fixes for some typos and erroneous output.
- Special thanks to Raghuprasad Govindarao for discovering a broken link.

Please send comments, suggestions, and bug reports to [Andrew Gilpin](#).

Page last modified: April 7, 2004