CS252 Lab 1: C/C++ Programming and GDB

Lab Slides

Goal

The goal of this lab is to help you better understand both the tools that are available to you as a programmer, and the UNIX environment. A substantial part of being a good programmer is understanding the tools available to you and using them correctly. The lab also serves as a review for C/C++.

Advice

The Editor

Picking an editor that can work for you is critical to efficient programming. This does not include such simple text editors as pico or the CDE text editor. There are plenty of programmers' editors out there, and several are available on the sslab machines (sslab01.cs.purdue.edu to sslab20.cs.purdue.edu). If you are iin a different machine you may ssh to any of these machines. We recommend using either XEmacs or vim. If you are not already familiar with vim, XEmacs will probably present you with the simplest learning curve.

XEmacs

XEmacs is a fork of the GNU emacs text editor with a comprehensive graphic interface. Virtually all of the operations you would want to do are available from drop-down menus and toolbars, letting you learn the keybindings and complex commands at your own pace. **You do not need to learn complex commands in order to get started using XEmacs.**

To start XEmacs, type:

```
csh> xemacs &
```

Once you have started XEmacs, you can open files, copy, paste, save, and perform other standard operations using the pull-down menus and toolbars. You can open more than one file simultaneously and switch between them using the *Buffers* pull-down menu. You probably won't want to open more than one copy of XEmacs, choosing to use multiple buffers instead.

As you become more familiar with the editor, you will probably want to learn the keybindings to often-used commands. For instance:

Ctrl-x Ctrl-s	Save the current buffer
Ctrl-x Ctrl-f	Find a file by name and open it in a buffer
Ctrl-x 2	Split the screen into two windows (to see two files simultaneously
Ctrl-x 1	Return to only one window
Ctrl-x o	Switch from one window to another

Ctrl-g	Abort the command prompt if you accidentally open it

We strongly suggest you take a look at the UNIX and XEmacs tutorials at: UINX and XEmacs Tutorials.

More information on how to use XEmacs is available through the Help menu or from your TA.

vim

vim is an enhanced version of the `vi` editor, which has been the "standard" editor on UNIX for years. Vim adds features such as syntax hilighting, extended macros, and smart indenting to the same vi that UNIX users have always had. If you're already a vi user, you should consider trying vim.

To start vim, type:

```
csh> vim filename
```

The majority of vim commands are the same as their vi counterparts, with some additions. An addition you may be immediately interested in is `:syntax on`, which will turn on syntax hilighting. The `vimtutor` command provides a nice beginner's tutorial, and the `:help` command is quite extensive.

Assignment

Part 1: C/C++ Review

In this part of the lab you will review C strings, double pointers, and pointers to functions.

Instructions

1. Download the file lab1-src.tar.gz to your home directory on the sslab machines. It contains everything you will need for this part of the assignment. Extract it as follows:

   ```
   csh> gzip -dc lab1-src.tar.gz | tar xvf -
   ```

   This will create a `lab1-src` subdirectory containing the files you will use in this part of the assignment. This subdirectory will be created under the directory where you downloaded the file (and unzipped and untarred it), e.g., your home directory.

2. Complete the functions in the files `mystring.c`, `List.cc`, and `mysort.cc`. ***Important:** use C or C++ according to the files given to you-- do NOT change from one to the other*. You may not use any of the `str*` functions (`strdup`, `strlen`, `strcpy`, etc.) in `mystring.c`, and you may not use the `qsort` function in `mysort.cc`. Use your best judgment for other functions -- if it seems like cutting corners, it probably is.

3. You are provided with very simple test programs (`test_mystring`, `test_mysort`, and `test_list`. Make sure all your tests compile correctly by typing `make`. You will have to edit the Makefile, since it is missing one target. **Please note, however, that the tests given are just simple example tests. Part of the assignment is for you to develop more sophisticated tests. For example, you may want to check the error handling you implement by ensuring your program does not crash even when provided with erroneous parameters.**

Part 2: Memory layout of a program

The image of a running program (its memory layout) consists of the following sections:

Memory Section Name	Description
text (or code segment)	This is the area of memory that contains the machine instructions corresponding to the compiled program. This area is READ ONLY and is shared by multiple instances of a running program.
data	This area in the memory image of a running program contains storage for **initialized global** variables. This area is separate for each running instance of a program.
bss	This is the memory area that contains storage for **uninitialized global** variables. It is also separate for each running instance of a program.
stack	This region of the memory image of a running program contains storage for the automatic (local) variables of the program. It also stores context-specific information before a function call, *e.g*. the value of the Instruction Pointer (Program Counter) register before a function call is made. On most architectures the stack grows from **higher memory to lower memory addresses**.
heap	This memory region is reserved for dynamically allocating memory for variables, at run time. Dynamic memory allocation is done by using the `malloc` or `calloc` functions.
shared libraries	This region contains the executable image of shared libraries being used by the program.

Instructions

1. Use XEmacs or vim to enter the following program as `lab1.c`. Compile and run it.

```
/* lab1.c*/

int a = 25;   /*initialized global var*/
int b[45000]; /*uninitialized global var*/
int d[45000]; /*uninitialized global var*/

void donothing();

int main()
{
    char c=25;
    int  *ptr;

    printf("Address of the function donothing() is 0x%lx\n", donothing);
    printf("Address of integer a is 0x%lx\n", &a);
    printf("Address of array b is 0x%lx\n", b);
    printf("Address of array d is 0x%lx\n", d);
    printf("Address of character c is 0x%lx\n", &c);

    ptr = (int *)malloc(sizeof(int));
    printf("Address of integer pointer ptr is 0x%lx\n", &ptr);
```

```
        printf("Address of the integer pointed to by ptr is 0x%lx\n", ptr);

        donothing();
        sleep(60);
        return 0;
    }

    void donothing()
    {
        int local;

        printf("\nAddress of integer local is 0x%lx\n", &local);

        return;
    }
```

To compile and run this, type:

```
csh> gcc -o lab1 lab1.c
csh> ./lab1
```

2. Use the command `/usr/proc/bin/pmap` to print the memory mappings of the `lab1` program while it is executing. To do this, run the command `lab1` in one window, and type the following command in another window to determine the process ID of the `lab1` program (be careful to note the PID, not the PPID):

   ```
   csh> ps -ef | grep lab1
   ```

   Now that you know the PID of your `lab1` program, run:

   ```
   csh> /usr/proc/bin/pmap PID
   ```

   If you have problems understanding the output (or options) of ps or pmap, check the man pages by typing "man ps" or "man pmap".

3. Use the output of the `pmap` program to fill out a simple table in a file named `mappings.txt` showing the location of the following symbols:

   - donothing
   - &a
   - b
   - d
   - &c
   - &ptr
   - ptr
   - &local

   The only valid locations are `text`, `data`, `bss`, `stack`, `heap`, and `shared library` (watch your case!). The file should be of the format:

   ```
   symbol location
   symbol location
   ```

   Where the symbol and location are separated by one or more spaces, and each symbol, location pair is on a line by itself. For example:

```
&somevariable stack
&errno shared library
```

Important: *It is critical that the format of this file is correct. NOTE THAT THE LOCATIONS WILL BE CHECKED USING A CASE-SENSITIVE MATCH. If you have any questions, please ask your TA.*

Part 3: GDB

In this part of the lab you will learn about the `gdb` debugger, a powerful interactive debugger for many compiled languages. We will of course be focusing on C and C++.

GDB can be used in one of two major fashions -- at runtime (gdb is started before there is a problem) or post-mortem (gdb is run to determine why a program that has already died did so). Runtime debugging is desirable whenever possible, but post-mortem debugging is useful when you have a process that fails in some unexpected way and leaves a core.

These two major modes of operation are invoked in different fashions. To debug a program at run time, you can either start the program from within gdb or attach to an already running program. To start a program within gdb, you would do the following:

```
csh> gdb progname
(gdb) run
```

Note that you might want to set some breakpoints or watchpoints (discussed later) before running the program. The second common form of runtime debugging is attaching gdb to a process that is already running. This requires you to know the PID of the running process you are interested in:

```
csh> gdb progname pid
```

After connecting in this fashion the program can be debugged normally, as if you had started gdb and run the program from there.

To invoke gdb for post-mortem debugging, you must have both the binary you wish to debug and the core file from where it failed. (FYI, you can often force a core dump with Ctrl-\ [control backslash]. However, it is often more useful to attach gdb to the running process than force a core.)

```
csh> gdb progname corefile
```

Core files can also be loaded at runtime with the `core-file` command, and executables can be loaded with the `exec-file` command.

Once you have your program open in gdb, you are likely going to want to do some debugging on it. Here is a brief list of commands you are likely to find useful.

Command syntax	Description

`break`
`break linenumber`
`break filename:linenumber`
	These commands set a "breakpoint", or a place where the debugger should suspend operation of the debugger and ask for further input. Play

`break` *`function`*	with their arguments a little to see how flexible `break` is.
`bt` `where`	The `bt` and `where` commands are functionally identical. They cause gdb to print a "backtrace", or listing of the function call stack of the current program. This is very useful for determining where and why a program died. You should play with this command on both programs that have crashed and programs that are running normally to get a feel for how it works.
`cont`	This command continues execution where it was last stopped, either by `break`, `watch`, Ctrl-C, a signal, or some other action.

`help`
`help` *`topic`*
`help` *`command`*
	Gdb includes an extraordinarily comprehensive help system. The bare command `help` will generate a list of high-level topics which can be researched in more depth by giving arguments to the `help` command. If you think gdb should probably be able to do some particular thing, look here -- it probably can.

`info` *`category`*	This prints information about some part of your program or gdb's configuration; you may find the categories `locals`, `variables`, `functions`, and `display` useful.
`next`	Execute the current line of code and proceed to the next, treating subroutine calls as one line.
`print` *`expression`*	`print` is one of the most powerful commands in gdb; it is used to print the value of an arbitrary expression, such as a simple variable, a calculation, or a function call. The power of `print` stems from the fact that *expression* can be a quite complex C or C++ expression, and it will be executed just as if your program had done it; for instance, if the variable `x` is equal to 5 and you know that it should be six, the command `print x = 6` will print the number "6", but will more importantly change the actual value of the variable `x`. This can be used to both determine why a section of code is failing and to prevent it from doing so, as well as to test corner cases and bounds in your code that may not be easily reached through test cases.
`step`	Like `next`, only `step` enters subroutines, treating the first line of a called subroutine as the next line of code.
`watch` *`expression`*	`watch` is like `break`, except that it will stop the program's execution whenever the value of the given expression changes, rather than at a specific point in program flow. Watchpoints can make your program run significantly slower if they have to be constantly checked, so set them carefully.

Remember that the source code line printed to the screen when the program stops is the *next* line to be executed, it has not yet run.

To find other resources google "gdb tutorial".

Instructions

Under your `lab1-src/` directory you will find a program called `debug` and partial source to this program. (**NB:** the `debug` executable is a x86 executable and will only run correctly on the sslab01-sslab20 machines) This program contains several bugs, some of which are serious enough to make it crash. Your task is to fix its crashes and more subtle bugs. At several points during its execution, it will print lines like:

```
     Array token: grqpnerjXyVdM
```

These lines are different for each student.

Copy these lines (and only these lines!) and paste them into a file called `gdbtokens.txt` in your `lab1-src` directory, to be submitted with the rest of the project. Use the commands listed above as well as other gdb commands you may learn looking through the help to make the program execute correctly. Note well that you are to copy the *entire line*, not just the token itself.

Your goal is to find what the bugs are and fix them temporally inside gdb so the execution can reach the point where these lines are printed.

For example, follow this gdb interaction:

```
bash-2.05$ gdb debug
GNU gdb (Gentoo 7.5 p1) 7.5
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
For bug reporting instructions, please see:
...
Reading symbols from /u/data/u3/grr/cs252/lab1-gdb-x86/lab1-src/debug...done.
(gdb)
```

Now set a breakpoint in main so the execution will stop there.

```
(gdb) break main
Breakpoint 1 at 0x10bb8: file public.c, line 19.
```

Now run the program.

```
(gdb) run
Starting program: /u/data/u3/grr/cs252/lab1-gdb-x86/lab1-src/debug
warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, main (argc=1, argv=0x7fffffffe058) at public.c:19
19        printf ("Starting tests.\n");
```

Now the program has stopped in the first line of main.

In an editor open the file "public.c" so you can see line 19. To go to line 20 in"vi" type ":19".

To go to line 20 in xemacs type "esc-g 19" and enter.

Now type "next" ("n" for short) to execute line by line without entering to functions, or "step" ("s" for short) to enter into a function.

```
(gdb) n
```

```
Starting tests.
20        fflush (stdout);
(gdb) n
22        initialize_array ();
(gdb) s
initialize_array () at public.c:37
37        int *numbers = NULL, i = 0;
(gdb) n
39        for (i = 0; i < 4; i++) {
(gdb) n
40          numbers[i] = i + 1;
(gdb) n

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400ce8 in initialize_array () at public.c:40
40          numbers[i] = i + 1;
```

Check what is causing the crash. First print the value of "i"

```
(gdb) print i
$1 = 0
```

Since "i" has a correct value, now check the value of "numbers".

```
(gdb) print numbers
$2 = (int *) 0x0
```

The array "numbers" has not been initialized. To fix it temporally we need to initialize it with a call to malloc. Run the program again but this time initialize "numbers" with a call to malloc of the right size. By looking at the program listing we can see that it needs 5 elements.

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

Starting program: /u/data/u3/grr/cs252/lab1-gdb-x86/lab1-src/debug
warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, main (argc=1, argv=0x7fffffffe058) at public.c:19
19        printf ("Starting tests.\n");
(gdb) n
Starting tests.
20        fflush (stdout);
(gdb)
22        initialize_array ();
(gdb) s
initialize_array () at public.c:37
37        int *numbers = NULL, i = 0;
(gdb) s
39        for (i = 0; i < 4; i++) {
(gdb) print numbers=(int*)malloc(5*sizeof(int))
$2 = (int *) 0x603010
(gdb) n
40          numbers[i] = i + 1;
(gdb) n
39        for (i = 0; i < 4; i++) {
(gdb)
40          numbers[i] = i + 1;
(gdb)
39        for (i = 0; i < 4; i++) {
```

```
(gdb)
40          numbers[i] = i + 1;
(gdb)
39        for (i = 0; i < 4; i++) {
(gdb)
40          numbers[i] = i + 1;
(gdb)
39        for (i = 0; i < 4; i++) {
(gdb)
42        numbers[4] = 0;
(gdb)
44        private_array_check (numbers);
45      }
(gdb)
```

The problem seems to be solved and you are about to execute the procedure that will print the line that you will paste in the file gdbtokens.txt. Type "next".

```
(gdb) s
Array token: arxG92Rj/DpZ2
(gdb) next
45      }
(gdb)
```

Copy and paste this line in the file gdbtokens.txt. These lines are different for each student.

There are three procedures that you have to fix including the one mentioned above so you will need to paste three lines in the file gdbtokens.txt. Remember that the "print" statement in gdb can be used to temporally fix the code in the program you are debugging. Also, another hint is that if you type the enter key in gdb it will execute the last command you typed.

More help on gdb can be found at the gdb homepage, as well as in many online tutorials. (Try searching for `gdb tutorial` on Google.)

Turning in the project

This project is due on Monday, January 28th, 2013 before 11:59 PM.

1. Make sure the files `mappings.txt` and `gdbtokens.txt` are in your `lab1-src` directory.
2. Change directories to your `lab1-src` directory.
3. Type '`make`' and ensure that everything built correctly.
4. Type '`make clean`' to clean up the object files.
5. Use '`cd ..`' to change to the parent directory of your `lab1-src` directory.
6. Type '`turnin -c cs252 -p lab1 lab1-src`' to submit your work for lab 1.
7. Type '`turnin -c cs252 -p lab1 -v`' and verify that the files you submitted are correct.