GNU Debugger (GDB)

CSCI 1061U — Programming Workshop 2

faisal.qureshi@uoit.ca
Faculty of Science
University of Ontario Institute of Technology

GDB

- Enables you to peek into the state of a program (executable) at run-time
- A very important tool that can help find logical errors, segmentation faults, etc.
- GDB can be installed on a linux machine using some sort of package manager

For ubuntu, try

```
$ sudo apt-get install libc6-dbg gdb valgrind
```

Preparing an Executable for use with GDB

- Use gcc **-g** option to enable debugging support

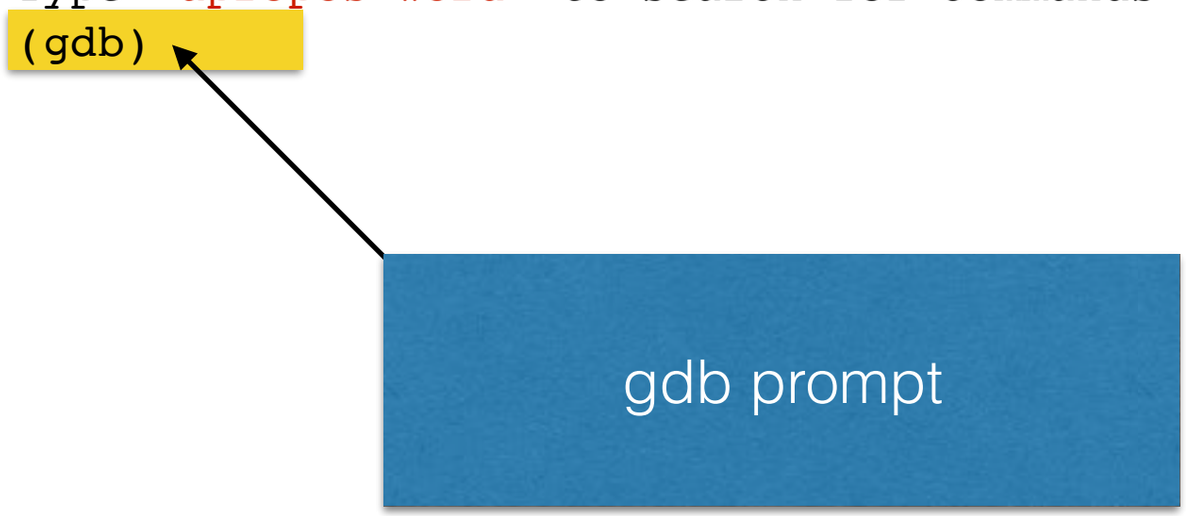
```
g++ -g arr.cpp -o arr
```

GDB Usage

- GDB is a command-line utility
  - Upon starting, it will provide you the (gdb) prompt
- GDB is interactive and it behaves similar to the terminal (bash shell) that you have been using in Linux.
  - It can recall history using arrow keys, supports auto-complete using TAB key
  - **`help`** command is available and can provide useful information about various features available within gdb

```
$ gdb
GNU gdb (GDB) 7.12.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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-apple-darwin16.4.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
```

```
(gdb)
```



The help command. [commandname] is optional.

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

Attaching an Executable

Use **`file (f)`** command to attach an executable

```
(gdb) file arr [optional command line arguments]
```

Executable

Use **`run (r)`** command to run an executable

```
(gdb) run
Starting program: /media/psf/Home/Dropbox/Teaching/2016-2017/Winter
2017/2017-winter-csci-1061u/course-material/lectures/week-12/cpp-gdb/src-
arr/arr
12 2 3 4 5
sum = 26
144 4 9 16 25
[Inferior 1 (process 3954) exited normally]
```

Breakpoints

- Use **`break (b)`** command to create breakpoints

  `break [filename]:[linenumber]` or `break [filename]:[funtionname]`

- Breakpoints allow program execution to stop at a particular line of code, providing an opportunity to inspect the state (e.g., variables and call stack) of the program

filename : line number
filename is optional

```
(gdb) break arr.cpp:7
Breakpoint 1 at 0x400923: file arr.cpp, line 7.
```

function name

```
(gdb) break arr_sum
Breakpoint 4 at 0x400979: file arr.cpp, line 18.
```

Conditional Breakpoints

- Often times you would like to use conditional break points. These pause the program only when a certain condition is met.
- These are extremely useful in large programs, where breakpoints quickly get out of hand and tedious to use

```
(gdb) break arr.cpp:19 if sum > 20
Breakpoint 4 at 0x400980: file arr.cpp, line 19.
(gdb) finish
Run till exit from #0  arr_sum (a=0x7fffffffdce0, n=5) at arr.cpp:18
0x0000000000400a43 in main () at arr.cpp:34
34    cout << "sum = " << arr_sum(arr, 5) << endl;
Value returned is $2 = 26
```

Continuing after breakpoints

- Use **`continue`** or (**`c`**) command to continue with the execution after the breakpoint. Don't use the **`run`** command, since it will restart the program.

- Use **`step`** or (**`s`**) command to execute the next program line. This will take you into a subroutine (function).

- Use **`next`** or (**`n`**) command to execute the next program, side-stepping the subroutines. Meaning it won't go into a subroutine (function), rather it will treat the function call as a single program line

Inspecting variables

- Use **`print (p)`** and **`print/x`** commands to check the value of a variable

```
(gdb) file arr
Reading symbols from arr…done.
(gdb) break arr_sum
Breakpoint 1 at 0x400979: file arr.cpp, line 18.
(gdb) run
Starting program: /media/psf/Home/Dropbox/Teaching/2016-2017/Winter 2017/2017-winter-
csci-1061u/course-material/lectures/week-12/cpp-gdb/src-arr/arr
12 2 3 4 5

Breakpoint 1, arr_sum (a=0x7fffffffdce0, n=5) at arr.cpp:1818
int sum = 0;
(gdb) print sum
$1 = -139055554
(gdb) print/x sum
$2 = 0xf7b62e3e(gdb)
```

Inspecting variables

- Use **`print`** and **`print/x`** commands to check the value of a variable

- Only variables that are in the current scope are available

```
(gdb) print foo
No symbol "foo" in current context.
```

Inspecting Pointers

- **`print`** and **`print/x`** commands also work with pointers
  - Operators **`->`** and **`*`** are also available (similar to how these are used in C and C++)

```
(gdb) break square
Breakpoint 5 at 0x40090e: file arr.cpp, line 6.
(gdb) continue
Continuing.
sum = 26

Breakpoint 5, square (v=0x7fffffffdce0) at arr.cpp:6
6     *v = (*v) * (*v);
(gdb) print v
$3 = (int *) 0x7fffffffdce0
(gdb) print *v
$4 = 12
```

v is a pointer

Setting watchpoints

- Use **`watch`** command to set watchpoints
- Watchpoints pause the program execution whenever a particular variable is changed.
- These are similar to breakpoints, which pause the program at a particular line of code
- *Watch uses scoping rules to determine which variable to watch*

Watchpoints

```
(gdb) watch sum
Hardware watchpoint 2: sum
(gdb) continue
Continuing.
```

Program stopped when value of sum is changed

```
Hardware watchpoint 2: sum

Old value = -139055554
New value = 0
arr_sum (a=0x7fffffffdce0, n=5) at arr.cpp:19
19   for (int i=0; i<n; ++i) {
```

Printing Call Stack

- **where** command is used to print current call stack

```
(gdb) where
#0  arr_sum (a=0x7fffffffdce0, n=5) at arr.cpp:19
#1  0x0000000000400a43 in main () at arr.cpp:34
```

Getting out of subroutines (functions)

- Use finish command to execute till the end of the current subroutine or function

```
(gdb) finish
Run till exit from #0  arr_sum (a=0x7fffffffdce0, n=5) at arr.cpp:18
0x0000000000400a43 in main () at arr.cpp:34
34    cout << "sum = " << arr_sum(arr, 5) << endl;
Value returned is $1 = 26
```

Miscellaneous Commands

- Use **`delete`** command to remove a breakpoints and watch
- Use **`info breakpoints`** to list information about all break points
- Use **`info watchpoints`** to list information about current watchpoints
- Use **`info`** for more information
- Use **`list [linenumber]`** to see code listing with line numbers
- Use **`quit`** to close gdb
- Press **ENTER** to issue the last used command

Beyond GDB

- Once the code is working "correctly," the next item on the *todo* list is to make it go as fast as possible. We want to write correct code that is efficient.

- C++ provides mechanisms to measure the execution speed of a particular piece of code.

Code Profiling and Optimization

- Profiling measures the runtime characteristics (also known as *dynamic code analysis*) of a program
  - Memory usage
  - Execution speed
  - Frequency of function calls
  - Usage of a particular line of code
- The goal is to use this analysis to improve the runtime performance of the program, i.e., use less memory, achieve faster execution speed, etc.
- A key step in *code profiling* is the ability to measure the execution speed of a particular set of instructions.

Measuring Execution Speed

Use **std::clock()** available in **<ctime>** header

```
#include <ctime> // std::clock, CLOCKS_PER_SEC


std::clock_t c_start = std::clock();
arr_sum(arr, n);
std::clock_t c_end = std::clock();
```

Record time before the start of instruction(s)

Record time at the end of the instruction(s)

The difference between the two is the *elapsed time*

The elapsed time is in clock ticks, convert it into **milliseconds** as follows

```
1000.0 * ((c_end - c_start) / CLOCKS_PER_SEC)
```

Time in ms

Measuring Execution Speed

Use **std::clock()** available in **<ctime>** header

```
#include <ctime> // std::clock, CLOCKS_PER_SEC
```

```
std::clock_t c_start = std::clock();
arr_sum(arr, n);
std::clock_t c_end = std::clock();
```

Modern CPUs are very fast, and oftentimes the execution time falls below the measuring ability of the **std::clock()**.

```
int ntries = 10000;
std::clock_t c_start = std::clock();
for (int j=0; j<ntries; ++j) {
  arr_sum(arr, n);
}
std::clock_t c_end = std::clock();
```

Run the same instruction multiple times and compute the average time

timing.cpp

```cpp
#include <iostream>
#include <iomanip> // std::setprecision
#include <ctime> // std::clock, CLOCKS_PER_SEC
using namespace std;

int arr_sum(int a[], int n)
{
  int sum = 0;
  for (int i=0; i<n; ++i) {
    sum += a[i];
  }
  return sum;
}

int main()
{
  int n = 1000000;
  int* arr = new int[n];
  for (int i=0; i<n; ++i) {
    arr[i] = i;
  }
  cout << endl;

  int ntries = 10000;
  std::clock_t c_start = std::clock();
  for (int j=0; j<ntries; ++j) {
    arr_sum(arr, n);
  }
  std::clock_t c_end = std::clock();

  cout << std::fixed
       << std::setprecision(2)
       << "Time (using clock()) = "
       << ((1000.0 * ((c_end - c_start) / CLOCKS_PER_SEC)) / (double) ntries)
       << " ms"
       << endl;

  delete [] arr;

  return 0;
}
```

```
g++ timing.cpp -o timing
```

Measuring Execution Speed: C++11

Use **`std::chrono::high_resolution_clock()`** available in **`<chrono>`** header

```
#include <chrono>
```

```
auto t_start = std::chrono::high_resolution_clock::now();
arr_sum(arr, n);
auto t_end = std::chrono::high_resolution_clock::now();
```

The difference between the two is the *elapsed time*

Elapsed (i.e., execution) time in **milliseconds**

```
std::chrono::duration<double, std::milli>(t_end-t_start).count()
```

timing2.cpp

```cpp
#include <iostream>
#include <iomanip> // std::setprecision
#include <chrono>
using namespace std;

int arr_sum(int a[], int n)
{
  int sum = 0;
  for (int i=0; i<n; ++i) {
    sum += a[i];
  }
  return sum;
}

int main()
{
  int n = 1000000;
  int* arr = new int[n];
  for (int i=0; i<n; ++i) {
    arr[i] = i;
  }
  cout << endl;

  auto t_start = std::chrono::high_resolution_clock::now();
  arr_sum(arr, n);
  auto t_end = std::chrono::high_resolution_clock::now();

  cout << std::fixed
       << std::setprecision(2)
       << "Time (using chrono()) = "
       << std::chrono::duration<double, std::milli>(t_end-t_start).count() / ntries
       << " ms"
       << endl;

  delete [] arr;

  return 0;
}
```

We often don't have to run the instruction whose time we want to estimate multiple time.

```
g++ -std=c++11 timing2.cpp -o timing2
```

g++ Optimization Options

- GNU C/C++ compiler provides a number of optimization options

- Without any optimization, the compiler attempts to reduce compilation time and include debugging information

- When optimization is turned, compiler attempts to increase performance or reduce memory; however, it limits a programmer's ability to debug the code

g++ Optimization Options

Flag	Effect
`-0, -01`	The compiler attempts to reduce code size and increase execution speed
`-02`	Increase compilation time and increased code performance
`-03`	Turns on all optimization features, increased compilation time, increase performance, increased program size
`-Os`	Attempts to reduce program size in conjunction with -O2
`-Ofast`	Enables all -O3 in addition to optimization that are not standard compliant
`-Og`	Turns on optimization that do not effect debugging

Use **`g++ -Q –help=optimizers`** to see the exact set of optimizations that are enabled for each level.

References

- https://www.gnu.org/software/gdb/
- http://valgrind.org/
- https://docs.microsoft.com/en-us/visualstudio/
  - One of the most feature-rich C++ IDE
  - Only for windows
- Xcode debugging tools
  - Only for OSX