CME 212 Debugging Notes

The need to debug a program is inevitable. When programming, we embed assumptions (known and unknown) in almost every line of code. Problems occur when the state of the computer violates those assumptions. Debugging is the process of figuring out what went wrong. It is difficult, because we have to find the location of the actual error (position in code) and why the error occurred. This often involves tracing the state of the program backwards in time from the point of the error. These notes will cover some tools useful in the process of debugging. The most important tool is your brain and systematic intervention and instrumentation.

Simple debugging techniques

Print statements

The most basic and widely used debugging methodology involves inserting print statements (`cout`) in your code to determine the state of various things as the program is running. It is likely that you are already doing this! Here are some things to keep in mind:

- Make sure to commit your code to version control before embarking on a print statement fueled bug hunt. After you find the bug, it is nice to be able to revert the code to clean up all of the print statements.
- Make sure to commit your code before you make changes in an attempt to fix the bug. It can be very difficult to remember what to undo without version control. You want to be able to quickly undo a mess created in an attempt to fix a bug.
- You can use `__FILE__` and `__LINE__` preprocessor macros to output the filename and line number. For example, the code:

  ```
  cout << "__FILE__: " << __FILE__ << endl;
  cout << "__LINE__: " << __LINE__ << endl;
  ```

  will output the file name and line number. See `src/print.cpp`.

Debug macro

It is often handy to use a preprocessor macro for debugging statements. This can alleviate some of the burden of always writing `std::cout << x << std::endl;`. Let's look at a simple example in `src/debug_macro.hpp`:

```
#ifdef PRINT_DEBUG
#define BUGPRINT(x)                                    \
  do {                                                 \
    std::cerr << __FILE__ << ":" << __LINE__ << ": ";  \
    std::cerr << #x << " -> " << (x) << std::endl;     \
  } while (0)
#else
#define BUGPRINT(X)
#endif
```

This code defines a "macro" called `BUGPRINT`. It can be used like a normal C++ function. See `src/debug_macro.cpp`:

```
#include "debug_macro.hpp"
#include <iostream>

int main() {
  using std::cout;
```

```
  using std::endl;

  int a = 5;
  int b = 6;

  BUGPRINT(a*b);

  cout << "a = " << a << endl;
  cout << "b = " << b << endl;

  return 0;
}
```

Under normal compilation, the call to `BUGPRINT` has no effect:

```
$ clang++ -std=c++14 debug_macro.cpp -o debug_macro
$ ./debug_macro
a = 5
b = 6
```

We add a compiler flag `-DPRINT_DEBUG` to enable the macro:

```
$ clang++ -DPRINT_DEBUG -std=c++14 debug_macro.cpp -o debug_macro
$ ./debug_macro
debug_macro.cpp:11: a*b -> 30
a = 5
b = 6
```

In the output we get a lot of useful information!

IMPORTANT NOTE: if you use a macro like `BUGPRINT`, it is important that the input have no side-effects. That is, no variables should be changed. If a call to `BUGPRINT` caused variables to change, then you would have different behavior when the macro is enabled and make your bug even harder to find.

Assert statements

Assert statements are a useful tool to check conditions in code. It looks like this:

```
assert(condition);
```

If `condition` evaluates to `true`, nothing happens. If `condition` evaluates to false, the program will terminate and print a (hopefully useful) error message. Let's look at the example from http://en.cppreference.com/w/cpp/error/assert (see `src/assert_1.cpp`):

```
#include <cassert>
#include <iostream>

int main() {
  assert(2 + 2 == 4);
  std::cout << "Execution continues past the first assert\n";
  assert(2 + 2 == 5);
  std::cout << "Execution continues past the second assert\n";
}
```

If we compile and run, we see:

```
$ clang++ -std=c++14 assert_1.cpp -o assert_1
$ ./assert_1
Execution continues past the first assert
```

```
assert_1: assert_1.cpp:7: int main(): Assertion `2 + 2 == 5' failed.
Aborted (core dumped)
```

`assert` is a preprocessor macro. All `assert` statements can be disabled by defining `NDEBUG`:

```
$ clang++ -DNDEBUG -std=c++14 assert_1.cpp -o assert_1
$ ./assert_1
Execution continues past the first assert
Execution continues past the second assert
```

Assert statements in functions

It is often a good idea to test preconditions for functions using `assert` statements. It could go something like this (see: `assert_2.cpp`):

```
/** add two non-negative integers
 * @param x integer to add
 * @param y integer to add
 * @return the value of x+y
 *
 * @pre 0 <= x and 0 <= y
 */
int non_negative_add(int x, int y) {
  assert(0 <= x);
  assert(0 <= y);
  return x+y;
}
```

Let's write a simple main function:

```
int main() {
  using std::cout;
  using std::endl;

  cout << non_negative_add(2,2) << endl;
  cout << non_negative_add(2,-5) << endl;
}
```

And use it:

```
$ clang++ -std=c++14 assert_2.cpp -o assert_2
$ ./assert_2
4
assert_2: assert_2.cpp:13: int non_negative_add(int, int): Assertion `0 <= y' failed.
Aborted (core dumped)
```

We can also disable the `assert` statements at compile time:

```
$ clang++ -DNDEBUG -std=c++14 assert_2.cpp -o assert_2
$ ./assert_2
4
-3
```

Simple pause

It is often very handy to pause a program so a human can read terminal output before continuing. This may help you debug a loop, for example. This is easy to do in C++ with `std::cin.get()` (in `<iostream>`). See src/pause.cpp:

```
#include <iostream>

int main() {
  std::cout << "before pause" << std::endl;
  // will wait for user to hit enter
  std::cin.get();
  std::cout << "after pause" << std::endl;

  // pause in a loop
  for (int i=0; i != 5; ++i) {
    std::cout << "iter: " << i << std::endl;
    std::cin.get();
  }
}
```

Usage:

```
$ clang++ -std=c++14 pause.cpp -o pause
$ ./pause
before pause
# (user hit enter)
after pause
iter: 0
# (user hit enter)
iter: 1
# (user hit enter)
iter: 2
# (user hit enter)
iter: 3
# (user hit enter)
iter: 4
# (user hit enter)
```

Be sure to remove all pauses before releasing your code into the wild. It is often good to provide a prompt:

```
std::cout << "hit enter to continue." << std::endl;
std::cin.get();
```

Using tools: debugger & memory checker

We are going to discuss the use of two very useful debugging tools:

- `gdb`: the GNU Debugger
- `valgrind`: a memory error checker

Hello world, `gdb` style

Let's start with a very simple C++ program, compile it with appropriate flags, and run it with the debugger. Here is the program (see `src/hello_gdb.cpp`):

```
#include <iostream>

int main() {
  int a = 42;
  std::cout << "Hello gdb!" << std::endl;
  std::cout << "a = " << a << std::endl;
  return 0;
}
```

Let's compile it:

```
$ g++ -g -O0 -std=c++14 hello_gdb.cpp -o hello_gdb
```

Details:

- It is currently best to change the compiler to `g++` before entering a debugging session with `gdb`. It is possible to debug a program using `clang++` and the `-g` flag, however `gdb` knows more about how `g++` compiles C++ code compared to `clang++`.
- The `-g` flag tells the compiler to enable debugging symbols in the output executable.
- The `-O0` flag turns off compiler optimization. This can be important if the compiler eliminates variables or inlines function calls.

We can run the program from the terminal:

```
$ ./hello_gdb
Hello gdb!
a = 42
```

Now let's run it in the debugger:

```
$ gdb ./hello_gdb
[gdb initialization text omitted]
(gdb) run
Starting program: /home/nwh/Dropbox/icme/cme212/2017-nwh-notes/debug/src/hello_gdb
Hello gdb!
a = 42
[Inferior 1 (process 24019) exited normally]
(gdb) quit
$
```

Details:

- Run a program in the debugger with `$ gdb <program-name>`
- The debugger is an interactive prompt that allows you to control execution and print variables. The debugger prompts you with `(gdb)`
- The command `run` begins the program
- The command `quit` quits the program
- Many `gdb` commands have short versions. `r` is short for `run`. `q` is short for `quit`. I will use long commands in the notes for clarity.
- `gdb` has a `help` command that you can use to learn about other commands

Let's run a slightly more interesting debugging session:

```
$ gdb ./hello_gdb
[gdb initialization text omitted]
(gdb) break main
Breakpoint 1 at 0x40089e: file hello_gdb.cpp, line 4.
(gdb) run
Starting program: /home/nwh/Dropbox/icme/cme212/2017-nwh-notes/debug/src/hello_gdb
```

```
Breakpoint 1, main () at hello_gdb.cpp:4
4     int a = 42;
(gdb) next
5     std::cout << "Hello gdb!" << std::endl;
(gdb) next
Hello gdb!
6     std::cout << "a = " << a << std::endl;
(gdb) print a
$1 = 42
(gdb) quit
```

Details:

- The `break` command sets a *break point*, where execution will pause.
- `break main` says put a break point at the start of the `main` function
- To break on a specific line of code use `(gdb) break file:lineno`. For example, `(gdb) break hello_gdb.cpp:7`
- `next` executes a line of code and moves forward
- `print` is used to print the state of variables

Simple debugging example

Let's work through another simple debugging session. We are going to look at the following code (see `src/sequence_gdb.cpp`):

```
#include <iostream>

int sequence(int* x, int nvals) {
  int sum = 0;
  for (int i = 0; i <= nvals; ++i) {
    x[i] = i;
    sum += x[i];
  }
  return sum;
}

int main() {
  int nvals = 1000;
  int* sequence_data;
  int sum = sequence(sequence_data, nvals);
  std::cout << "sum = " << sum << std::endl;
}
```

Step 1 - fixing a crash I bet you can already see the errors. Let's compile and run:

```
$ g++ -g -O0 -std=c++14 sequence_gdb.cpp -o sequence_gdb
$ ./sequence_gdb
Segmentation fault (core dumped)
```

That's a problem. We can see from the code that we are trying to dereference an invalid pointer. Let's boot it up in `gdb` and see what happens:

```
$ gdb ./sequence_gdb
[gdb initialization text omitted]
```

```
(gdb) run
Starting program: /home/nwh/Dropbox/icme/cme212/2017-nwh-notes/debug/src/sequence_gdb

Program received signal SIGSEGV, Segmentation fault.
0x00000000004008ce in sequence (x=0x0, nvals=1000) at sequence_gdb.cpp:6
6       x[i] = i;
(gdb) backtrace
#0  0x00000000004008ce in sequence (x=0x0, nvals=1000) at sequence_gdb.cpp:6
#1  0x0000000000400914 in main () at sequence_gdb.cpp:16
(gdb) list
1   #include <iostream>
2
3   int sequence(int* x, int nvals) {
4     int sum = 0;
5     for (int i = 0; i <= nvals; ++i) {
6       x[i] = i;
7       sum += x[i];
8     }
9     return sum;
10  }
(gdb) info args
x = 0x0
nvals = 1000
(gdb) print i
$1 = 0
(gdb) quit
```

Details:

- `gdb` is able to trap a segmentation fault and stop the program at the time of the error. We can then use the `(gdb)` prompt to inspect the state of the program.
- `gdb` immediately tells us that line 6 caused the problem (`sequence_gdb.cpp:6`).
- The command `backtrace` (or `bt`) for short lists the series of function calls to get to the current point. You can also see the argument values used in the call. `x=0x0` indicates that `x` is a null pointer.
- The command `list` shows surrounding lines of code.
- `info args` shows the values of function arguments.
- `print i` prints the value of the loop iteration counter `i`. From this we see that the error occurs in the first trip through the loop.

Let's fix this bug by properly allocating memory with `new`. We add the line of code before the call to `sequence`:

```
  sequence_data = new int[nvals];
```

And run:

```
$ g++ -g -O0 -std=c++14 sequence_gdb.cpp -o sequence_gdb
$ ./sequence_gdb
sum = 500500
```

It seems to work, but there are still errors.

Step 2 - checking for memory errors We can use `valgrind` to look for out of bounds memory accesses as well as memory leaks (our program has both). Let's run it:

```
$ valgrind ./sequence_gdb
==5973== Memcheck, a memory error detector
==5973== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5973== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5973== Command: ./sequence_gdb
==5973==
==5973== Invalid write of size 4
==5973==    at 0x40098E: sequence(int*, int) (sequence_gdb.cpp:6)
==5973==    by 0x4009FE: main (sequence_gdb.cpp:16)
==5973==  Address 0x5ab6c20 is 0 bytes after a block of size 4,000 alloc'd
==5973==    at 0x4C2E80F: operator new[](unsigned long)
==5973==    by 0x4009E9: main (sequence_gdb.cpp:15)
==5973==
==5973== Invalid read of size 4
==5973==    at 0x4009A4: sequence(int*, int) (sequence_gdb.cpp:7)
==5973==    by 0x4009FE: main (sequence_gdb.cpp:16)
==5973==  Address 0x5ab6c20 is 0 bytes after a block of size 4,000 alloc'd
==5973==    at 0x4C2E80F: operator new[](unsigned long)
==5973==    by 0x4009E9: main (sequence_gdb.cpp:15)
==5973==
sum = 500500
==5973==
==5973== HEAP SUMMARY:
==5973==     in use at exit: 76,704 bytes in 2 blocks
==5973==   total heap usage: 3 allocs, 1 frees, 77,728 bytes allocated
==5973==
==5973== LEAK SUMMARY:
==5973==    definitely lost: 4,000 bytes in 1 blocks
==5973==    indirectly lost: 0 bytes in 0 blocks
==5973==      possibly lost: 0 bytes in 0 blocks
==5973==    still reachable: 72,704 bytes in 1 blocks
==5973==         suppressed: 0 bytes in 0 blocks
==5973== Rerun with --leak-check=full to see details of leaked memory
==5973==
==5973== For counts of detected and suppressed errors, rerun with: -v
==5973== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Take some time to read the error message. `valgrind` identifies:

- `Invalid write of size 4` (that is 4 bytes, should lead you to think `int` or `float`)
- `Invalid read of size 4`
- In the `LEAK SUMMARY`, we see `definitely lost: 4,000 bytes in 1 blocks`. This indicates memory leak. We've neglected to free up memory allocated by `new`.

Let's fix the errors:

- In `sequence()` change `<=` to `<` in the `for` loop
- Add `delete [] sequence_data;` after the call to `sequence()`

And run it:

```
$ g++ -g -O0 -std=c++14 sequence_gdb.cpp -o sequence_gdb
$ valgrind ./sequence_gdb
==6058== Memcheck, a memory error detector
==6058== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
```

```
==6058== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==6058== Command: ./sequence_gdb
==6058==
sum = 499500
==6058==
==6058== HEAP SUMMARY:
==6058==     in use at exit: 72,704 bytes in 1 blocks
==6058==   total heap usage: 3 allocs, 2 frees, 77,728 bytes allocated
==6058==
==6058== LEAK SUMMARY:
==6058==    definitely lost: 0 bytes in 0 blocks
==6058==    indirectly lost: 0 bytes in 0 blocks
==6058==      possibly lost: 0 bytes in 0 blocks
==6058==    still reachable: 72,704 bytes in 1 blocks
==6058==         suppressed: 0 bytes in 0 blocks
==6058== Rerun with --leak-check=full to see details of leaked memory
==6058==
==6058== For counts of detected and suppressed errors, rerun with: -v
==6058== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Everything looks good!

C++ STL containers

For a long time, it has been tricky to use `gdb` with C++. Recent additions to `gdb` include addition of "pretty printers" for C++ STL containers (like `vector` and `map`). Let's look at an example of this (see: `src/containers_gdb.cpp`):

```cpp
#include <iostream>
#include <map>
#include <vector>
#include <string>

int main() {
  std::vector<int> x = {1,3,5,2,6};
  std::map<std::string,int> m = {{"one",1},{"two",2}};

  std::cout << "x[0] = " << x[0] << std::endl;
  std::cout << "m[\"two\"] = " << m["two"] << std::endl;
}
```

Let's compile and run with `gdb`:

```
$ g++ -g -O0 -std=c++14 containers_gdb.cpp -o containers_gdb
$ gdb ./containers_gdb
[omit prompt]
(gdb) break main
Breakpoint 1 at 0x4011bc: file containers_gdb.cpp, line 11.
(gdb) run
Starting program: /home/nwh/Dropbox/icme/cme212/2017-nwh-notes/debug/src/containers_gdb

Breakpoint 1, main () at containers_gdb.cpp:11
11  int main() {
(gdb) n
12    std::vector<int> x = {1,3,5,2,6};
```

```
(gdb) n
13    std::map<std::string,int> m = {{"one",1},{"two",2}};
(gdb) n
15    std::cout << "x[0] = " << x[0] << std::endl;
(gdb) print x
$1 = std::vector of length 5, capacity 5 = {1, 3, 5, 2, 6}
(gdb) print m
$2 = std::map with 2 elements = {["one"] = 1, ["two"] = 2}
(gdb) print x[3]
$3 = 2
(gdb)
```

Details:

- We can print an STL container with `print x`
- We can print an element of a `std::vector` with `op[]` (`(gdb) print x[3]`)
- This does not work when attempting to print an element of a map:

  ```
  (gdb)
  print m["one"]
  Cannot resolve function operator[] to any overloaded instance
  (gdb)
  ```

  This can be resolved by adding a simple function to print an element from a `map`:

  ```
  // simple function to print map value designed for use in GDB
  void mv(const std::map<std::string,int>& m, const char* a) {
    std::cout << m.at(a) << std::endl;
  }
  ```

  Now from `gdb`:

  ```
  (gdb) print mv(m,"two")
  2
  ```

Debugging check list

- Switch compiler to `g++`
- Enable debugging symbols with `-g`
- Disable optimization with `-O0`
- Possibly write a simple helper function if dealing with STL containers
- Return to desired compiler and settings after debugging is complete

For CME212 homeworks this can be achieved by modifying 2 lines in `Makefile.inc`:

- Line 24 change to: `CXX := $(shell which g++)`
- Line 39 change to: `CXXFLAGS  := -std=c++11 -g -O0 -W -Wall -Wextra #-Wfatal-errors`

Debugging references

- http://www.dirac.org/linux/gdb/
- http://moss.csc.ncsu.edu/~mueller/GDBReferenceCard.pdf

- http://proquest.safaribooksonline.com/book/programming/c/9781491904428/2dot-debug-test-document/debug_html
- http://pages.tacc.utexas.edu/~eijkhout/istc/html/debug.html