Lecture 14:
Debugging
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
New operators: `<<` and `>>`

- "`<<`": Insertion operator
- "`>>`": Extraction operator
  - Operator overloading: you can define what it means to insert or extract your object.

- Often used in conjunction with "streams"
  - Recall our earlier experience with C streams
    - `stderr`, `stdout`, `stdin`
  - Streams are communication channels
C++ streams example

```c
fawcett:330 childs$ cat print.c
#include <stdio.h>

int main()
{
    printf("The answer is: ");
    printf("%d", 8);
    printf("\n");
}
fawcett:330 childs$ gcc print.c
fawcett:330 childs$ ./a.out
The answer is: 8
```

```cpp
fawcett:330 childs$ cat printCPP.C
#include <iostream>

int main()
{
    std::cout << "The answer is: ";
    std::cout << 8;
    std::cout << "\n";
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ ./a.out
The answer is: 8
```

```c
fawcett:330 childs$ cat print.C
#include <stdio.h>

int main()
{
    printf("The answer is: %d\n", 8);
}
fawcett:330 childs$ g++ print.C
fawcett:330 childs$ ./a.out
```

```cpp
fawcett:330 childs$ cat printCPP.C
#include <iostream>

using std::cout;

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```

(cascading & endl)
Three pre-defined streams

- `cout` <= => `fprintf(stdout, ...`
- `cerr` <= => `fprintf(stderr, ...`
- `cin` <= => `fscanf(stdin, ...`
cin in action

```cpp
fawcett:330 childs$ cat cin.C
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int main()
{
    int X, Y, Z;
    cin >> X >> Y >> Z;
    cout << Z << "", " << Y << "", " << X << endl;
}
fawcett:330 childs$ ./a.out
3 5
4
4, 5, 3
```
cerr

• Works like cout, but prints to stderr
• Always flushes everything immediately!

```
fawcett:330 childs$ cat cerr.C
#include <iostream>

using std::cerr;
using std::cout;
using std::endl;

int main()
{
    int *X = NULL;
    stream << "The value is ";
    stream << *X << endl;
}
```

```
fawcett:330 childs$ g++ -Dstream= cerr cerr.C
fawcett:330 childs$ ./a.out
The value is Segmentation fault
```

“See the error”
fstream

• ifstream: input stream that does file I/O
• ofstream: output stream that does file I/O

• Not lecturing on this, since it follows from:
  – C file I/O
  – C++ streams

http://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
#1 complaint I hear from employers

• “students can’t debug”
  – If you can debug, you can progress on their projects (even if slowly)
  – If not, they will have to hand hold you

• Think about your debugging approach.
  – How would you describe it during an interview?

This lecture describes how I would answer that question
Debugging Strategy

• (#1) Figure out _where_ the error is occurring
• (#2) Figure out _why_ the error is occurring
• (#3) Form a plan to fix the error
Terrible debugging strategy

- fix it by repeatedly making random changes
  - typically, code is pretty close to working in the first place
  - each random change creates a new problem that then has to be debugged
  - code ends up as a complete mess

This is a “bugging” strategy.

Always make sure you feel confident about what the problem is before you start changing code.
Debugging as the scientific method

• Debugging involves the scientific method:
  – You have a hypothesis
    • ("the bug is because of this")
  – You form an experiment
    • ("I will test my theory by examining the state of some variables")
  – You confirm or deny the hypothesis
    • ("the state was OK, my hypothesis was wrong")
    • ("the state was bad, my hypothesis was right")
Backups

• The “scientific method” of debugging – which is good – can leave your code as a mess

• My recommendation:
  – when you have a bug, immediately make a copy of your program
  – apply the scientific method-style of debugging until you understand the problem and how to fix
  – then go back to your original program and make the fix there
Debugging Overview

• To me, effective debugging is about two things:
  – Challenging your own assumptions
  – Divide-and-conquer to find the problem
Challenging Assumptions

• you thought the code would work and it doesn’t
  – so something you did is wrong, and you have to figure out what

• I find students are often turning under the wrong stones, since there are some stones they don’t see

• the way to find these “hidden stones” is to start with the bad outcome and search backwards
  – why is this pointer NULL? why is this value bad?
Divide-and-Conquer

• There are lots of things that could be wrong with your program
  – This is a search problem!!
• Divide-and-Conquer: try to focus on hypotheses that will eliminate half the possibilities, no matter what the outcome is
Divide-and-Conquer

• “Halving” hypotheses:
  – The problem is in this module
  – The problem is in this function

• Non-halving hypotheses:
  – The problem is in this line of code
  – The problem is with this variable

As you divide the space smaller and smaller, you will eventually end up hypothesis that are small in scope
Good practice / Poor practice

• Good practice:
  – Write a few lines of code and test that it does what you want

• Poor practice:
  – Write a bunch of code and compile it all at the end

Why is it better to write smaller portions of code at a time?
Why is it better to write smaller portions of code at a time?

• If you have one bug
  – it is easier to figure out where that bug is
    • searching through tens of lines of code, not hundreds

• If you have many bugs
  – this is a disaster and you end up chasing your tail
    • and you are still searching through hundreds of lines of code, not tens

The extra effort of modifying the main program to test your new code pays off ten-fold (WAG)
Final thought: always initialize your variables!!

• Many crashes in your HW assignments due to uninitialized pointers.
  – If you assign it to NULL, then at least you know that is set to something not valid
  – Otherwise, you see an address and think it might have valid memory

• Initialize non-pointers too!

• Classic point that employers look for.

This practice becomes increasingly essential when you work in teams on large projects.
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
Print statements

• Print statements (cerr, printf):
  – the world’s most used debugger
  – very successful SW professionals are able to debug large SW projects using only print statements

• Pitfalls:
  – output buffering
  – too many print statements
Pitfall #1: output buffering

• output is sometimes buffered, and buffered output is dropped when a program crashes
• if you don’t see a print statement
  – your program may have crashed before it got to that print statement
  – your program may have gotten to that print statement, but crashed before it got outputted

cerr: automatically flushes & mostly prevents this problem

cerr << "*(NULL) = " << *NULL << endl; // still doesn’t work
Output buffering and cerr

- cerr: automatically flushes & mostly prevents output buffering problem
- Exception:
  - cerr << "*(NULL) = " " << *NULL << endl;
  - (crashes before anything is printed out)
- Work-around:
  - cerr << "*(NULL) = " ";
  - cerr << *NULL << endl;
Pitfall #2: millions of print statements

```c
void make_black(unsigned char *b, int width, int height, int buffer_size)
{
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < height; j++)
        {
            int pixel_index = j*width+i;
            int buffer_index = 3*pixel_index;
            cerr << "About to write to index" << endl;
            b[buffer_index+0] = 0;
            b[buffer_index+1] = 0;
            b[buffer_index+2] = 0;
        }
    }
}
```

This will result in millions of print statements ... hard to debug.
Reducing print statements

```c
void make_black(unsigned char *b, int width, int height,  
    int buffer_size)
{
    for (int i = 0 ; i < width ; i++)
    {
        for (int j = 0 ; j < height ; j++)
        {
            int pixel_index = j*width+i;
            int buffer_index = 3*pixel_index;
            if (buffer_index < 0 || buffer_index >= buffer_size)
            {
                cerr << "About to write to index"  
                << buffer_index << endl;
                exit(EXIT_FAILURE);
            }
            b[buffer_index+0] = 0;
            b[buffer_index+1] = 0;
            b[buffer_index+2] = 0;
        }
    }
}```
Make it easy on yourself...

#include <iostream>

using std::cerr;
using std::endl;

#define PL cerr << "(PL): " << __FILE__ << ": " << __LINE__ << endl;

int main()
{
    PL
    int width = 100, height = 100;
    PL
    int buffer_size = width*height;
    PL
    unsigned char *b = new unsigned char[3*buffer_size];
    PL
    for (int i = 0 ; i < width ; i++)
    {
        PL
        for (int j = 0 ; j < height ; j++)
        {

128-223-223-73-wireless:330 hank$ ./a.out
(PL): big_print.C: 10
(PL): big_print.C: 12
(PL): big_print.C: 14
(PL): big_print.C: 16
(PL): big_print.C: 19
About to write to index10200
Make it easy on yourself to run too...

128-223-223-73-wireless:330 hank$ cat r
./proj3C 3C_input.pnm 3C_output.pnm
128-223-223-73-wireless:330 hank$ ./r
Other ways to make life easier

- tab from shell: auto-completes
- Ctrl-R: searches backwards in your shell history
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
Debuggers

• Allow you to set breakpoints
• Allow you to inspect variables
• Show you where a program crashed
Debuggers

• gdb:
  – GNU debugger
  – goes with gcc/g++

• lldb:
  – CLANG debugger
    • Mavericks doesn’t really have gcc, and gdb is not possible on Mavericks
Debugging Symbols

• Object files:
  – by default, are compact and contain minimal information that connects to your original program
  – optionally, can be instructed to contain increased linkage
    • what line of code did these instructions come from?
    • what variable name is in this register?

You enable debugging symbols by adding “-g” to compile line “gcc –c file.C” → “gcc –g –c file.C”
Running with gdb

```
hank@ix: ~ 7$ cat myprogram.C
#include <stdlib.h>
int main()
{
    int *p = NULL;
    int X = *p;
}
hank@ix: ~ 8$ g++ -g myprogram.C
hank@ix: ~ 9$ gdb a.out
```
Running with gdb

(gdb) run
Starting program: /home/users/hank/a.out

Program received signal SIGSEGV, Segmentation fault.
0x000000000004004c4 in main () at myprogram.C:5
5 int X = *p;
(gdb) where
#0 0x000000000004004c4 in main () at myprogram.C:5
(gdb)
Arguments

• You are running “./proj3A 3A_input.pnm 3A_output.pnm”
• In gdb, you would do:
  % gdb proj3A
  (gdb) run 3A_input.pnm 3A_output.pnm
“core” files

• When a program crashes, it can create a “core” file
  – This file contains the state of memory when it crashes
  – It is very large, and can fill up filesystems
    • So system administrators often disable its generation
      – “ulimit –c 0” → “ulimit –c 10000000”
  – You can run a debugger on a program using a core file (tells you where it crashed)
    • gdb a.out core
Valgrind: a memory checker

hank@ix: ~ 14$ valgrind a.out
==13623== Memcheck, a memory error detector
==13623== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13623== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13623== Command: a.out
==13623==
==13623== Invalid read of size 4
==13623==  at 0x4004C4: main (myprogram.C:5)
==13623==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==13623==
==13623== Process terminating with default action of signal 11 (SIGSEGV)
==13623==  Access not within mapped region at address 0x0
==13623==  at 0x4004C4: main (myprogram.C:5)
==13623==  If you believe this happened as a result of a stack overflow in your program's main thread (unlikely but possible), you can try to increase the size of the main thread stack using the --main-stacksize= flag.
==13623==  The main thread stack size used in this run was 8388608.
==13623==
==13623== HEAP SUMMARY:
==13623== in use at exit: 0 bytes in 0 blocks
==13623== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13623== All heap blocks were freed -- no leaks are possible
==13623== For counts of detected and suppressed errors, rerun with: -v
==13623== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
Segmentation fault (core dumped)
Valgrind and GDB

- Valgrind and gdb are available on ix
  - Older versions of Mac are possible, newer are not
  - Linux is easy

- You will have an assignment to have a memory error-free and memory leak-free program with valgrind.
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
Pitfall #1

```c
void AllocateBuffer(int w, int h, unsigned char **buffer) {
    *buffer = new unsigned char[3*w*h];
}

int main() {
    int w = 1000, h = 1000;
    unsigned char *buffer = NULL;
    AllocateBuffer(w, h, &buffer);
}
```

This is using call-by-value, not call-by-reference.
struct Image
{
    int width;
    int height;
    unsigned char *buffer;
};

Image *ReadFromFile(char *filename) {
    Image *rv = NULL;

    /* OPEN FILE, descriptor = f */
    /* ... */
    /* set up width w, and height h */
    /* ... */

    rv = malloc(sizeof(Image));
    rv->width = w;
    rv->height = h;
    fread(rv->buffer, sizeof(unsigned char), w*h, f);
}
Pitfall #3

- int *s = new int[6*sizeof(int)];
int main()
{
    // new black image
    int height = 1000, width = 1000;
    unsigned char *buffer = new unsigned char[3*width*height];
    for (int i = 0 ; i < sizeof(buffer) ; i++)
    {
        buffer[i] = 0;
    }
}

• Assume:
  int *X = new int[100];
• What is sizeof(X)?
• What is sizeof(*X)?
Pitfall #5

/* struct definition */
struct Image
{
    /* data members */
};

/* prototypes */
void WriteImage(Image *, const char *);

/* main */
int main()
{
    Image *img = NULL;
    /* set up Image */
    const char *filename = "out.pnm";
    WriteImage(img, filename);
}

/* WriteImage function */
void WriteImage(char *filename, Image *img)
{
    /* code to write img to filename */
}
Top requires memory allocation / deletion, and does extra copy.
Pitfall #7

• For objects on the stack, the destructors are called when a function goes out of scope
  – You may have a perfectly good function, but it seg-faults on return

• Especially tricky for main
  – program ran to completion, and crashed at the very end
#include <stdlib.h>

class Image {
    public:
        Image() { width = 0; height = 0; buffer = NULL; }
        virtual ~Image() { delete [] buffer; }

        void ResetSize(int width, int height);
        unsigned char *GetBuffer(void) { return buffer; }

    private:
        int width, height;
        unsigned char *buffer;
};

void Image::ResetSize(int w, int h) {
    width = w;
    height = h;
    if (buffer != NULL)
        delete [] buffer;
    buffer = new unsigned char[3*width*height];
}

int main() {
    Image img;
    unsigned char *buffer = img.GetBuffer();
    img.ResetSize(1000, 1000);
    for (int i = 0; i < 1000; i++)
        for (int j = 0; j < 1000; j++)
            for (int k = 0; k < 1000; k++)
                buffer[3*(i*1000+j)+k] = 0;
}
Outline

• Review
• Debugging Philosophy
• Print Statements
• Debuggers
• Recently Observed Coding Pitfalls
• Project 3E
Project 3E

• You will need to think about how to accomplish the data flow execution pattern and think about how to extend your implementation to make it work.

• This prompt is vaguer than some previous ones
  — ... not all of the details are there on how to do it
Project 3E

- Worth 5% of your grade
- Assigned today, due May 22\textsuperscript{nd}