BACKTRACKING FROM DISASTER: REVISITING GDB

As you have most likely learned in the last week, programs involving dynamic memory allocation are often prone to errors, and they can be much harder to troubleshoot. By now, you have probably all seen your homework 2 crash once or twice. In the UNIX environment, you see relatively uninformative messages like "Segmentation fault" and perhaps also "Bus error" when a program crashes. Whenever you get an error like this, *it is most likely to be a memory problem*.

The most important thing to remember about dynamic memory is that an error and its consequences can be very distant from each other in the program. After the memory error, your program may seem to be running fine. Perhaps later on, some peculiar things begin to happen (like when you assign a coefficient to one polynomial and it changes a coefficient in another polynomial). Perhaps the behavior of the program varies from machine to machine. You may even get an error message only when you quit the program at the end. You know how chickens supposedly run around for a while after their heads are cut off? The same idea holds here. After a memory error, your program may be able to keep running for a short time, but the fatal damage has been done to it, and eventually it's going to fall over dead.

Here's an example of a memory problem. We have allocated an array of something (characters here, but the contents aren't important) from the heap memory.

```
char *letters;
letters = new char[5];  // reserves heap memory for the array
letters[0] = 'A';       // sets the array characters
letters[1] = 'G';
letters[2] = 'C';
letters[3] = 'Z';
letters[4] = 'Q';
```

Now the array looks like this:

A	G	C	Z	Q

Now what happens when we try to add to a slot that isn't actually in the array (that is, the array subscript inside the [ ] is larger than the largest slot in the array):

```
letters[5] = 'X';
```

This will compile fine, and when you run the program, it may seem as if nothing blows up at first. But at this point our program becomes like the headless chicken; sooner or later, it's just going to fall over dead. When and how it dies is a matter of circumstance. The out-of-bounds assignment can overwrite memory being used by the operating system, or your program. It will also often crash when you say:

```
delete [] letters;
```

This is because the letters array is now pointing to a section of memory that doesn't really belong to it.

How do we track down errors in memory? It's not easy. We don't have any tools that can examine the code and catch even the simplest mistakes, like saying letters[5] = 'X' for a letters array of length <= 5. The best strategy is to use the gdb debugger to run your program, step by step. You can watch the program execute each of your methods, and you can use the display option for variables in gdb to tell you how the code is changing things. Methods like constructors are the first things our program will run, and they set up all our dynamic memory initially, so it makes sense to check them first. If one of our constructors is bad, then when we call add_coef() or reserve(), which change the coefficient array, we're likely to see problems even if these modification routines work fine. Other places where nasty bugs may emerge is in things like the copy constructor and assignment operator, particularly when we try to free up memory we don't need anymore.

In emacs, hitting Control-g can back you out of almost any mistaken command you give. You can split an emacs window into 2 with Control-x 1; Control-x o moves you to the other half of the window (the half that you're not currently using).

Here are the general steps.
Make a directory for this:

```
mkdir ~/2270/lab6
cd ~/2270/lab6
```

Copy all the files from me:

```
cp ~ekwhite/2270SP04/lab6/*.cxx .
cp ~ekwhite/2270SP04/lab6/*.h .
cp ~ekwhite/2270SP04/lab6/Makefile .
```

Open the polyexam1.cxx program in emacs:

```
emacs polyexam1.cxx
```

Split the window in 2

```
Control-x 2
```

Start the debugger:

```
Escape-x
gdb
```

Run gdb (like this):

```
gdb polyexam1
```

Switch to your code window:

```
Control-x o
```

Move your cursor to the first line of the main program, on the line after the {

```
Control-x Space
```

You should see a message about the breakpoint in the debug window.
Switch to the debug window:

```
Control-x o
```

And type

```
run
```

at the prompt. You should see a message in the gdb part of the window indicating that the code has executed up to the breakpoint line. In the code window, there should be an arrow pointing to the next line. At the gdb prompt, you can now type

next

or just

n

to go to the next line. If the next line calls a function you've written and you want to see it working in more detail, you can type:

step

or just

s

to execute this function line by line until it returns to the polyexam1 code. Remember that you don't really want to step into code for << or >> very often!

EXERCISE 1:

Copy the poly2_bug1.cxx file (a buggy version) to your poly2.cxx file.

```
cp ~ekwhite/2270SP04/lab6/poly2_eliz_bug1.cxx poly2.cxx
make clean
make all
polyexam
```

If you want to restore the working copy of poly2.cxx later, you can type

```
cp ~ekwhite/2270SP04/lab6/poly2_eliz_ok.cxx poly2.cxx
```

What happens? This bug is in the default constructor, but where it appears to you in the program's behavior is unpredictable. Try setting a breakpoint in the code before the polynomial declaration and stepping into the constructor. You are free to ask for the values of certain variables (they'll be weird and random until your code explicitly sets them).

When you want to display a member variable, like capacity, type

display capacity

The gdb display prints it as

this->capacity: 40

to demonstrate that it's a member variable.

If you have a function like the operator + with a local variable like answer, you can display its coefficients by using their array subscripts:

display answer.data[0]

When you want to display a reference parameter, like iterations when you are stepping through the find_root function, if you say

display iterations

you'll get a pointer address for an answer:

iterations = (unsigned int *) 0xffbef648

but if you say,

display *iterations

you'll get the current value of iterations in the gdb window:

*iterations = 0

What display variables help you see what's going on in the program?
MORE EXERCISES (2-9)

Copy, build, and track down the errors for poly2_eliz_bug2.cxx through poly2_eliz_bug9.cxx, using the debugger to help you search for telltale weird values in member variables. Below is a key: the first line after the filename is the routine the bug is in; the other lines describe the error. What display statements in gdb help to pin the problem down for you?

1. poly2_eliz_bug1.cxx
polynomial::polynomial(value_type c)
The line
```
data = new value_type();
```
should be
```
data = new value_type[capacity];
```

2. poly2_eliz_bug2.cxx
polynomial::polynomial(value_type c)
The line
```
capacity = DEFAULT_CAPACITY;
```
is missing
so the polynomial capacity is never set before use

3. poly2_eliz_bug3.cxx
polynomial::polynomial(value_type c)
The line
```
clear();
```
is missing; garbage values will collect and confuse degree

4. poly2_eliz_bug4.cxx
void polynomial::reserve(size_type new_capacity)
The line
```
copy(data, data + new_capacity, new_data);
```
should be
```
copy(data, data + capacity, new_data);
```

5. poly2_eliz_bug5.cxx
void polynomial::reserve(size_type new_capacity)
The line
```
delete [] data;
```
occurs before before copying data to new_data

6. poly2_eliz_bug6.cxx
void polynomial::assign_coef(double coefficient, unsigned int exponent)
The line:

while (exponent > capacity) reserve(capacity*2+1);
should be
while (exponent >= capacity) reserve(capacity*2+1);
This results in an off by one error

7. poly2_eliz_bug7.cxx
void polynomial::find_root(double& answer, bool& success, unsigned int& iterations,
double starting_guess = 1, unsigned int maximum_iterations = 100, double
epsilon = 1e-8) const
The line
new_guess = current_guess - (f_prime.eval(current_guess)/(eval(current_guess));
should be
new_guess = current_guess - (eval(current_guess)/f_prime.eval(current_guess));

8. poly2_eliz_bug8.cxx
void polynomial::find_root(double& answer, bool& success, unsigned int& iterations,
double starting_guess = 1, unsigned int maximum_iterations = 100, double
epsilon = 1e-8) const
The line
iterations++;
is at the start of the do loop instead of at the end; this throws off the iterations count

9. poly2_eliz_bug9.cxx
polynomial::polynomial(const polynomial& source)
The line:
data = NULL;
is missing before the assignment operator is invoked
The assignment operator only checks if (data != NULL) before deleting it, so this permits deletion of memory that was never allocated with new.