Chapter 5: Errors (continues)

Plan for today

- We will talk about:
  - Logic errors
  - Estimation
  - Debugging
  - Pre- an post-conditions
  - Testing

Logic errors

- Once we removed the compiler and linker errors, the program runs

Logic errors

- Once we removed the compiler and linker errors, the program runs
- Typically, next we get that
  - no output is produced ,or
  - the output produced is not what we were expecting

Logic errors

- Once we removed the compiler and linker errors, the program runs
- Typically, next we get that
  - no output is produced ,or
  - the output produced is not what we were expecting
- Various reasons
  - Our understanding of program logic flawed
  - We didn’t write what we though we wrote
  - We made some silly mistakes in our if-statement or somewhere else

Logic errors

- Logic errors are usually the <u>most difficult to find and eliminate</u>
- At this stage the computer does what we asked it to
- Our job is to figure out why that wasn’t really what we meant

Logic error: example

- Consider the following code fragment that calculates the smallest age

```
int smallestAge = 0;
for (age : ages){
   if (smallestAge < age)
      smallestAge = age;
}
```

On the vector of ages = {12, 8, 19, 2, 1, 5, 23} the code will produce the smallest age of 0. This is a wrong result.

What is wrong with the code?

Debugging

- Errors are called “bugs” and the process of finding them and correcting them is called debugging

Debugging

- Errors are called “bugs” and the process of finding them and correcting them is called debugging
- Debugging works roughly like this:
  - get the program to compile
  - get it to link
  - get it to do what it is supposed to do

Debugging

- Errors are called “bugs” and the process of finding them and correcting them is called debugging
- Debugging works roughly like this:
  - get the program to compile
  - get it to link
  - get it to do what it is supposed to do
- How not to debug:

```
while (the program doesn’t appear to work) {
  randomly look through the code for something that “looks odd”
  change it to look better}
```

Debugging

- How not to debug:

```
while (the program doesn’t appear to work) {
  randomly look through the code for something that “looks odd”
  change it to look better}
```

“It is obviously a poor algorithms with little guarantee of success. Unfortunaely, that description is only a slight caricature of what many people find themselves doing late at night when feeling particularly lost and clueless, having tried “everything else.”

Debugging

- *How would I know if the program actually worked correctly?*

If we can’t answer this question, then we are in for a long and tedious debug session, and most likely our users are in for some frustration.

Debugging

- *How would I know if the program actually worked correctly?*

If we can’t answer this question, then we are in for a long and tedious debug session, and most likely our users are in for some frustration.

Anything that helps to answer this questions minimized debugging and helps produce correct and maintainable programs.

Debugging

- *How would I know if the program actually worked correctly?*

If we can’t answer this question, then we are in for a long and tedious debug session, and most likely our users are in for some frustration.

Anything that helps to answer this questions minimized debugging and helps produce correct and maintainable programs.

Basically, we’d like to design our programs so that bugs have nowhere to hide. That’s usually too much to ask for...

Debugging

- *How would I know if the program actually worked correctly?*

If we can’t answer this question, then we are in for a long and tedious debug session, and most likely our users are in for some frustration.

Anything that helps to answer this questions minimized debugging and helps produce correct and maintainable programs.

Basically, we’d like to design our programs so that bugs have nowhere to hide. That’s usually too much to ask for...but we try to minimize the chance of error and maximize the chance of finding errors that do creep in.

Practical debugging advises: program structure

- *Make the program easy to read so that you have a chance of spotting the bugs*
  - *Comment (including explanations of design ideas)*
  - *Use meaningful names*
  - *Indent*
    - *Use a consistent layout*
    - *Your IDE tries to help (but it can’t do everything) - you are the one responsible*
  - *Break code into small functions*
    - *Try to avoid functions longer than a page*
  - *Avoid complicated code sequences*
    - *Try to avoid nested loops, nested if-statements, etc. (obviously, you sometimes need those)*
  - *Use library facilities*

Practical debugging advises: get it to compile

- Is every string literal terminated?

```
cout << "Hello, << name << '\n';   // oops!
```

- Is every character literal terminated?

```
cout << "Hello, "  << name << '\n;  // oops!
```

- Is every block terminated?

```
if (a>0) { /* do something */
 else      {  /* do something else */ }  // oops!
```

- Is every set of parentheses matched?

```
if (a    // oops!
   x = f(y);
```

- The compiler generally reports this kind of error “late”
  - It doesn’t know you didn’t mean to close “it” later

Practical debugging advises: get it to compile

- Is every name declared?
  - Did you include needed headers? (e.g., std_lib_facilities.h)
- Is every name declared before it’s used?
  - Did you spell all names correctly?

```
int count;  /* … */ ++Count;  // oops!
char ch;    /* … */ Cin>>c;   // double oops!
```

- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y)+2 // oops!
z = x+3;
```

Debugging

- Carefully follow the program through the specified sequence of steps
  - Pretend you're the computer executing the program
  - Does the output match your expectations?
  - If there isn't enough output to help, add a few debug output statements

```
cerr << "x == " << x << ", y == " << y << '\n';
```

- Be very careful
  - See what the program specifies, not what you think it should say
    - That's much harder to do than it sounds
    - `for (int i=0; 0<month.size(); ++i) {   // oops!`
    - `for( int i = 0; i<=max; ++j) {       // oops! (twice)`

Debugging

- When you write the program, insert some checks (“sanity checks”) that variables have “reasonable values”
  - Function argument checks are prominent examples of this

```
if (number_of_elements<0)
   error("impossible: negative number of elements");

if (largest_reasonable<number_of_elements)
   error("unexpectedly large number of elements");

if (x<y) error("impossible: x<y");
```

- Design these checks so that some can be left in the program even after you believe it to be correct
- It’s almost always better for a program to stop than to give wrong results

Debugging

- Pay special attention to "end cases" (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
  - Did you open your files correctly?
    - more on this in chapter 11
  - Did you actually read that input?
    - Write that output?

Debugging

- "If you can't see the bug, you're looking in the wrong place"
  - It's easy to be convinced that you know what the problem is and stubbornly keep looking in the wrong place
  - Don't just guess, be guided by output
    - Work forward through the code from a place you know is right
      - so what happens next? Why?
    - Work backwards from some bad output
      - how could that possibly happen?
- Once you have found "the bug" carefully consider if fixing it solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I found the last bug"
    - is a programmer's joke

Note

- Error handling is fundamentally <u>more difficult and messy</u> than “ordinary code”
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- The more people use a program, the better the error handling must be
  - If you break your own code, that’s your own problem
    - And you’ll learn the hard way
  - If your code is used by your friends, uncaught errors can cause you to lose friends
  - If your code is used by strangers, uncaught errors can cause serious grief
    - And they may not have a way of recovering

Pre- and post-conditions

- The call of a function is basically the best point to think about correct code and to catch errors: this is where a logically separate computation starts (and ends on the return).
- Consider the example:

```
double func1(int a, int b, double c, double d)
// integers a and b should be non-negative
// decimals c and d should be negative
{
   if (a < 0 or b < 0 or c >= 0 or d <= 0)
      error(“bad arguments for func1”);
   // …
}
```

Pre-conditions must be true for the function to perform its work correctly.

Pre- and post-conditions

- *Pre-condition* is a requirement on function argument
- Always document the pre-conditions in comments
- Sometimes it’s a good idea to check it

Pre- and post-conditions

- *Post-condition* is what must be true when a function completes its work

```
int area(int length, int width)
// calculate area of a rectangle
// length and width must be positive
{
        if (length<=0 || width <=0) throw Bad_area{};
        // the result must be a positive int that is the area
        // no variables had their values changed
        return length*width;
}
```

Pre- and post-conditions

- Always think about them
- If nothing else write them as comments
- Check them “where reasonable”
- Check a lot when you are looking for a bug

Testing

- How do we test a program?
  - Be systematic
    - “pecking at the keyboard” is okay for very small programs and for very initial tests, but is insufficient for real systems
  - Think of testing and correctness from the very start
    - When possible, test parts of a program in isolation
      - E.g., when you write a complicated function write a little program that simply calls it with a lot of arguments to see how it behaves in isolation before putting it into the real program (this is typically called “unit testing”)
- You can read more about it in Chapter 26

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html

- Class textbook