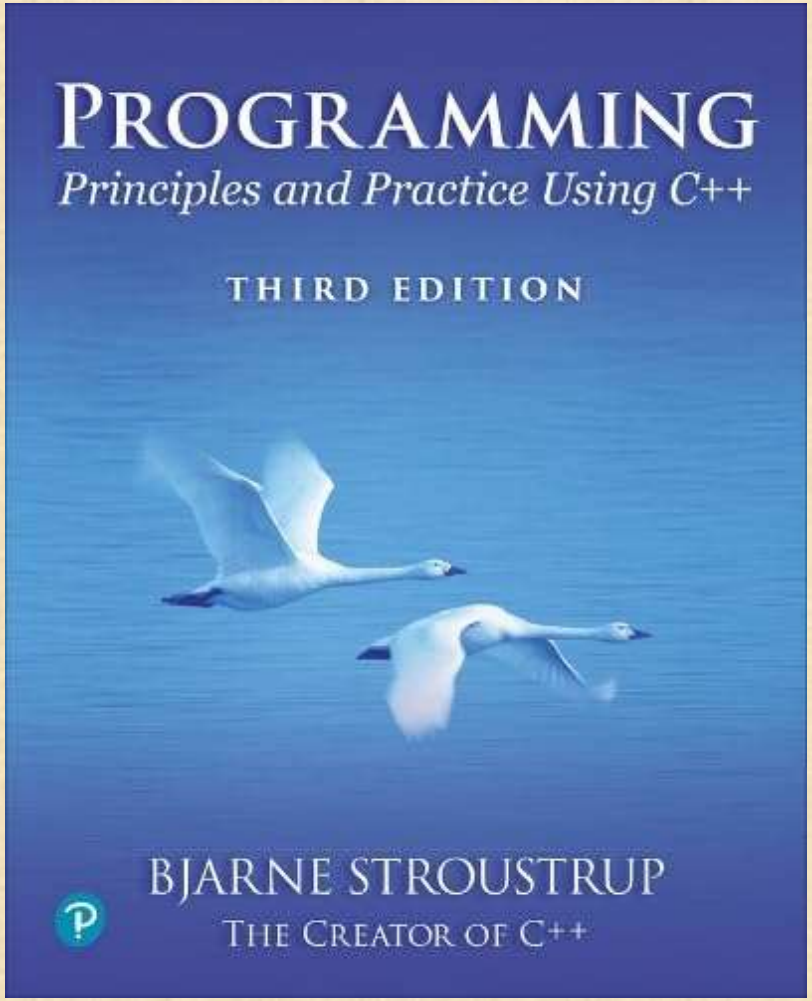
Chapter 5 – Errors



I realized that from now on a large part of my life would be spent finding and correcting my own mistakes.
– Maurice Wilkes, 1949

Abstract

- When we program, we must deal with errors. Our most basic aim is correctness, but we must deal with incomplete problem specifications, incomplete programs, and our own errors. Here, we'll concentrate on a key area: how to deal with unexpected function arguments. We'll also discuss techniques for finding errors in programs: debugging and testing.

Overview

- Kinds of errors
- Bad function arguments
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Avoiding and finding errors

Errors

- " … I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  - Maurice Wilkes, 1949 (2nd Turing award winner)
- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors.
  - Eliminate most of the errors we made anyway.
    - Debugging
    - Testing
  - Make sure the remaining errors are not serious.
- My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
  - You can do much better for small programs.
    - or worse, if you're sloppy

Your Program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

3, 4, and 5 are true for beginner's code; often, we must worry about those in real software.

Sources of errors

- Poor specification
  - “What’s this supposed to do?”
- Incomplete programs
  - “but I’ll not get around to doing that until tomorrow”
- Unexpected arguments
  - “but **sqrt()** isn’t supposed to be called with **-1** as its argument”
- Unexpected input
  - “but the user was supposed to input an integer”
- Unexpected state
  - But my database has bad data
- Logic errors: Code that simply doesn’t do what it was supposed to do
  - “so fix it!”

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Check your inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (istream)
- For example

```
int x = 0;
cin >> x;
if (!(0<x && x<max))
        error("input out of range");
```

- We'll get back to that

Bad function arguments

- The compiler helps:
  - Number and types of arguments must match

```
int area(int length, int width)
{
    return length*width;
}

int x1 = area(7);                 // error: wrong number of arguments
int x2 = area("seven", 2);        // error: 1st argument has a wrong type
int x3 = area(7, 10);             // ok
int x5 = area(7.5, 10);           // ok, but dangerous: 7.5 truncated to 7;
                                  //      most compilers will warn you
int x = area(10, -7);             // this is a difficult case:
                                  // the types are correct, but the values make no sense
```

Bad function arguments

- So, how about **int x = area(10, -7);** ?
- Alternatives
  - Just don't do that
    - Rarely a satisfactory answer
  - The caller should check
    - Hard to do systematically
  - The function should check
    - Return an "error value" (not general, problematic)
    - Set an error status indicator (not general, problematic – don't do this)
    - Throw an exception
- Note: sometimes we can't change a function that handles errors in a way we do not like
  - Someone else wrote it and we can't or don't want to change their code

Bad function arguments

- Why worry?
  - You want your programs to be correct
    - Especially when you write code for others to use
  - Typically, the writer of a function has no control over how it is called
    - Writing “do it this way” in the manual (or in comments) is no solution. Many people don’t read manuals
  - The beginning of a function is often a good place to check
    - Before the computation gets complicated
- When to worry?
  - If it doesn’t make sense to test every function, test some
  - ***Always*** consider the possibility of bad arguments

How to report an error

- Return an “error value” (not general, problematic)

```
int area(int length, int width)                    // return a negative value for bad input
{
   if (length <=0 || width <= 0)
         return -1;
   return length*width;
}
```

- So, “let the caller beware”

```
int z = area(x,y);
if (z<0)
   error("bad area computation");
// …
```

- Problems
  - What if I forget to check that return value?
  - For some functions there isn’t a “bad value” to return (e.g., **max()**)

How to report an error

- Set an error status indicator (not general, problematic, don't!)

```
int errno = 0;      // used to indicate errors (as used in the C standard library)
int area(int length, int width)
{
   if (length<=0 || width<=0)
            errno = 7;
   return length*width;
}
```

- And let the caller check

```
int z = area(x,y);
if (errno==7)
   error("bad area computation");
// ...
```

- Problems
  - What if I forget to check **errno**?
  - How do I pick a value for **errno** that's different from all others?

How to report an error

- Report an error by throwing an exception

```
int area(int length, int width)
{
   if (length<=0 || width<=0)
         throw Bad_area{};   // Bad_area is a type used to report the error; note the {} - a value
   return length*width;
}
```

- And let a caller catch the error (e.g., in **main()**)
  - if you don't "the system" reports the error: you can't ignore an exception

```
try {
   int z = area(x,y);        //  if area() doesn't throw an exception; make the assignment
}
catch(Bad_area) {            // if area() throws Bad_area{}, respond:
   cerr << "oops! Bad area calculation - fix program\n";
}
```

- We have more elegant ways of using exceptions; see ???

Exceptions

- Exception handling is general
  - You can't forget about an exception: the program will terminate if someone doesn't handle it (using a **try** … **catch**)
  - Use exceptions for rare ("exceptional") errors
    - E.g., a network failure in the middle of a file transfer
  - Use return values to indicate errors that can be considered normal
    - E.g., a file that cannot be opened
- You still have to figure out what to do about exceptions
  - Every exception that you want your program to handle
    - Even if only to give a good error message
- Error handling is ***never*** really simple

Range error

- Try this

```
vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
for (int i = 0; i<=10; ++i)                    // print 10 values (???)
    cout << "v[" << i << "] == " << v[i] << '\n';
```

- vector's operator[] (subscript operator) reports a bad index (its argument) by throwing a out_of_range if you use PPP.h"
  - The default behavior can differ
- You can't make this mistake with a range-for

Exceptions – for now

- For now, just use exceptions to terminate programs gracefully, like this

```
int main()
try
{
    // …
}
catch (out_of_range&) {    // out_of_range exceptions
    cerr << "oops – some vector index out of range\n";
}
catch (…) {                // all other exceptions
    cerr << "oops – some exception\n";
}
```

A function **error()**

- Here is a simple **error()** function as provided in **PPP.h**
- This allows you to print an error message by calling **error()**
- It works by disguising throws, like this:

```
void error(string s)          // one error string
{
        throw runtime_error(s);
}

void error(string s1,  string s2)    // two error strings
{
        error(s1 + s2);    // concatenates
}
```

Using **error( )**

- Example

```
cout << "please enter integer in range [1..10]\n";
int x = -1;        // initialize with unacceptable value (if possible)
cin >> x;
if (!cin)          // check that cin read an integer
    error("didn't get a value");
if (x < 1 || 10 < x)       // check if value is out of range
    error("x is out of range");
// if we get this far, we can use x with confidence
```

How to report an error

- In general, I prefer for the callee to check for errors
  - It's done in one place
  - It cannot be forgotten
  - It's often simpler (see longer example in §4.5)
  - If cost/efficiency is an issue, there are ways of suppressing tests
    - But that's not relevant for this book/course
  - See assertions (§4.7.3)

How to look for errors

- When you have written (drafted?) a program, it'll have errors (commonly called "bugs")
  - It'll do something, but not what you expected
  - How do you find out what it actually does?
  - How do you correct it?
  - This process is usually called "debugging"

Debugging

- How ***not*** to do it

```
while (program doesn't appear to work) {          // pseudo code
    Randomly look at the program for something that "looks odd"
    Change it to "look better"
}
```

- A key question

  How would I know if the program worked correctly?

- A useful question
  - Is the result plausible?

Program structure

- Make the program easy to read
  - so that you have a chance of spotting the bugs
  - Comment
    - Explain design ideas
  - Use meaningful names
    - Appropriate for its context
  - 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
    - A function should do on logical action or computation
    - If you can't give it a meaningful name, you probably have a conceptual problem
    - Try to avoid functions longer than a page
  - Avoid complicated code sequences
    - Try to avoid nested loops, nested if-statements, etc. (But, we sometimes need those)
  - Use library facilities

First get the program 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

First get the program to compile

- Is every name declared?
  - Did you include needed headers? (e.g., **PPP.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';
    ```

  - Eventually, you'll learn to use a debugger to help with that
- 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 later
  - 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?
- If you can’t find a bug, try explaining to a friend why it couldn’t be there
- 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 more difficult and messier 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-conditions

- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
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{};
        return length*width;
}
```

Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
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”
  - Use **expect()** from PPP
- Check a lot when you are looking for a bug
  - And as systematically as you can
- This can be tricky
  - How could the post-condition for area() fail after the pre-condition succeeded (held)?

Expect()

- Many systems have pre- and post-conditions checks
  - Sometimes called assertions; sometimes called contracts
  - None are (yet) standard
  - PPP has **expect()**

```
int area(int length, int width)
    // calculate area of a rectangle
    // length and width must be positive
{
        expect ([&] { return 0<=length && 0<width; }, “bad argument for area()”)
        return length*width;
}
```

- Calls **error()** if the condition (here, **0<=length && 0<width**) is false
  - Sorry for the “boilerplate”

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”)
- See PPP2 Chapter 26 and the literature on test frameworks
    - But not yet

The next lecture

- In the next two lectures, we'll discuss the design and implementation of a complete small program – a simple "desk calculator."