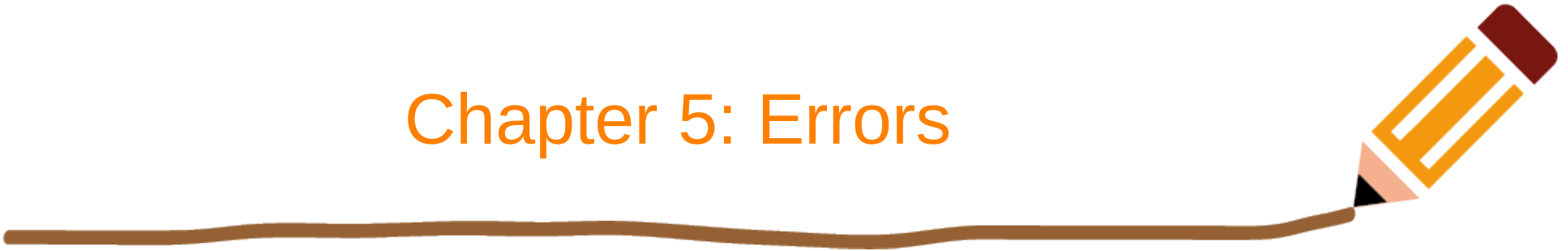
Chapter 5: Errors



Abstract

- When we program, we have to 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.

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

- 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.
- Author’s guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
- We can do much better for small programs.
  - or worse, if we’re sloppy

Our 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 have to worry about those in real software.S

Sources of errors

- Poor specification

  if not all details are clear about what a program should do, we are “unlikely” to adequately examine the “dark corners” and make sure that all cases are handled

Sources of errors

- Poor specification

  if not all details are clear about what a program should do, we are “unlikely” to adequately examine the “dark corners” and make sure that all cases are handled

- Incomplete programs

  during the development, there are cases that haven’t yet take care of. What we must aim for is to know when we have handled all cases.

Sources of errors

- Unexpected arguments

  Functions take arguments.

  If a function is given an argument we don’t handle, we have a problem.

Sources of errors

- Unexpected arguments

  Functions take arguments.

  If a function is given an argument we don’t handle, we have a problem.

- Unexpected input

  Programs typically read data (from …). A program makes many assumptions about such input.

Sources of errors

- Unexpected state

  Most programs keep a lot of data around for use by different parts of the system. Examples: address lists, phone directories, vectors of temperature readings, etc.

  What if such data is incomplete or wrong? The various parts of the program must still manage.

- Logical errors

  The code simply doesn't do what it is supposed to do. We'll have to find and fix such problems.

Sources of errors

- Unexpected state

  Most programs keep a lot of data around for use by different parts of the system. Examples: address lists, phone directories, vectors of temperature readings, etc.

  What if such data is incomplete or wrong? The various parts of the program must still manage.

Kinds of errors

- Compile-time errors (compiler is our first line of defence)
  - 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)

Compile-time errors

- Syntax errors

  an example:

  int s1=area(7;

- Type errors

Exceptions

- The fundamental idea is to separate detection of an error (should be done in a called function) from the handling of an error (should be done in the calling function), while ensuring that a detected error cannot be ignored
- The basic idea: a function finds an error it cannot handle, then it doesn’t return normally, instead it throws an exception indicating what when wrong. Any indirect or direct caller can catch the exception and specify what to do. A function expresses the interest in exceptions by using a try-block. If no caller catches an exception, the program terminates.

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
  - 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

Reporting an error

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

```
int area(int length, int width)   // return -1 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");
// …
```

Error function by default terminates the program with a system error message + the string we pass as an argument to it

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

Reporting an error

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

```
int errno = 0; // used to indicate errors
int area(int length, int width)
{
   if (length<=0 or width<=0) errno = 7;
   return length*width;
}
```

- So, "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 do I deal with that error?

Reporting an error

- Report an error by throwing an exception

```
class Bad_area { };  // a class is a user defined type
        // Bad_area is a type to be used as an exception

int area(int length, int width)
{
   if (length<=0 or width<=0) throw Bad_area{};
   return length*width;                    // note the {} – a value
}
```

- Catch and deal with the error (e.g., in main())

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

cerr is meant for error output

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**)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is never really simple

Out of range

- Consider this code fragment:

```
vector<int> v(10);  // a vector of 10 ints,
                    // each initialized to the default value, 0,
                                // referred to as v[0] .. v[9]
for (int i = 0; i < v.size(); ++i) v[i] = i;
for (int i = 0; i <= 10; ++i)
   cout << "v[" << i << "] == " << v[i] << endl;
```

Out of range

- Consider this code fragment:

```
vector<int> v(10);  // a vector of 10 ints,
                    // each initialized to the default value, 0,
                    // referred to as v[0] .. v[9]
for (int i = 0; i < v.size(); ++i) v[i] = i;
for (int i = 0; i <= 10; ++i)
   cout << "v[" << i << "] == " << v[i] << endl;
```

- vector's `operator[ ]` (subscript operator) reports a bad index (its argument) by throwing a `Range_error` if you use #include "std_lib_facilities.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 std_lib_facilities.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 or 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
   }
```

invalid_argument and out_of_range

- invalid_argument exception is thrown when we have invalid arguments
- out_of_range exception is thrown by the at() member funcion of vector class and similar entities

grab the file ThrowingAndCatchingExample.cpp from Blackboard or our website

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- C++ How To Program, 10th edition, by P. Deitel and H. Deitel
- Class textbook