DM560

Introduction to Programming in C++

Error Handling

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on slides by Bjarne Stroustrup*]

Outline

1. Error Handling

2. Debugging

Outline

Outline

- When we program, 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:
  - how to deal with unexpected function arguments
  - techniques for finding errors in programs: debugging and testing.

Outline

- Kinds of errors
- Argument checking
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Testing

Errors

- 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.

- Avoiding, finding and correcting errors is estimated to be 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 have to worry about those in real software.

Source of Errors

- Poor specification
  "What is 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"
- 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

1. Function arguments
2. Data from input (istream)

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 because someone else wrote it and we can't or don't want to change their code

Bad Function Arguments

The beginning of a function is often a good place to check (before the computation gets complicated)

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

When to worry?

- If it doesn't make sense to test every function, test some

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
int area(int length, int width)
{
  if (length<=0 || width<=0) errno = 7; // || means or
  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 is different from all others?
- How do I deal with that `error`?

How to Report an Error

The right way

- Report (Throw) 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 || width<=0) throw Bad_area{};   // note the {} - a value
  return length*width;
}
```

- 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";
}
```

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

Try this:

```
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;       // set values
for (int i = 0; i<=10; ++i)                      // print 10 values (???)
        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" (#include<stdexcept>)

The default behavior can differ: in -std-lib=c++14 compare [] with at()

You can't make this mistake with a range-for

Exceptions

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";
}
```

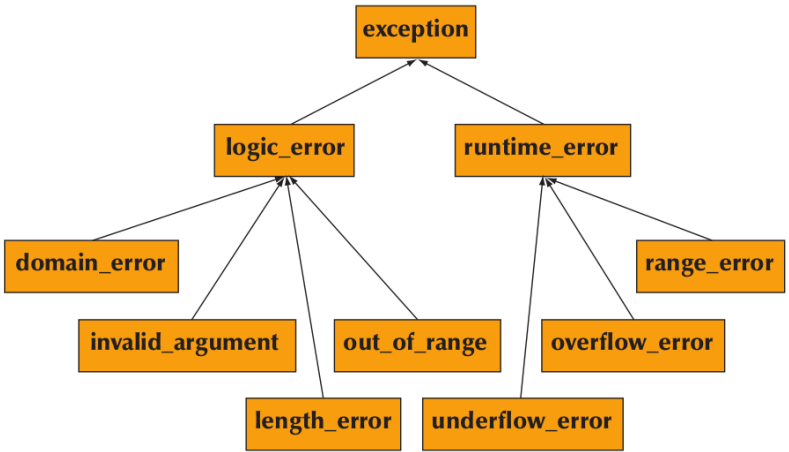
A function error()

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
}
```

Exception Class Hierarchy



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("I did not 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
```

```
please enter integer in range [1..10]
-1
terminate called after throwing an instance of 'std::runtime_error'
  what():  x is out of range
Aborted
```

Link-time errors

Example

```
int area(int length, int width);

double area(double x, double y) { };

int main() {
  int x = area(2,3);
}
```

Outline

1. Error Handling

2. Debugging

How to Look for Errors

When you have written (drafted?) a program, it will have errors (commonly called bugs). It will 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 **do**
- Randomly look at the program for something that "looks odd"
- Change it to "look better"

Key question: How would I know if the program actually worked correctly?

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
- Indent
  - Use a consistent layout
  - Your IDE tries to help - look for "format" (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.
  (But, obviously, you 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., `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");
```

- Alternatively, use `assert` which can be disabled in production with `#define NDEBUG` or option `-DNDEBUG` when compiling.

- Consider also to use the options `-fsanitize=address` and or `-fsanitize=undefined` when compiling

- 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 (with a reasonable value)?
- Did the function get the right arguments?
  Did the function return the right value?
- Did you handle the first/last element correctly?
- 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/write that input?

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 more difficult and messy 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
  - 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

Pre-conditions

What a function requires of its arguments 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-condition

What must be true when a function returns 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
  int a = length*width;
  if (a < 0) error(``area()  post-condition'');
  return a;
}
```

Can you find inputs that satisfy the pre-condition but not the post-condition?

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
- This can be tricky
  - How could the post-condition for `area()` fail after the pre-condition held?

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 Chapter 26

Performance

- Computation time

```
> time primes  # only in Linux
```

Otherwise:

```
#include<iostream>
#include<chrono>

using namespace std;
using namespace std::chrono;

int main (int argc, char **argv) {
  int iterations = atoi(argv[1]);
  auto t1=system_clock::now();

  for (int i = 0; i<iterations; i++) system("primes");

  auto t2 = system_clock::now();
  cout << iterations << "iterations took "
       << duration_cast<milliseconds>(t2-t1).count()<<" milliseconds\n";
}
```

- Memory usage

Summary

1. Error Handling

2. Debugging