Errors handling
Exceptions

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,
but often, we have to worry about those in real software.

Sources of errors

- Poor specification
  - “What’s this supposed to do?”
- Incomplete programs
  - “just had no time to finalize it”
- Unexpected arguments
  - “**sqrt()** isn’t supposed to be called with **-1** as its argument”
- Unexpected input
  - “the user was supposed to input an integer” (*wrong approach!*)
- Code that simply doesn’t do what it was supposed to do
  - Most difficult case! => “find the bug and fix it!”

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors (e.g. missing libraries)
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors (most difficult to find)
  - Detected by programmer

    (code runs, but produces not what we had expected)

Bad function arguments

The compiler helps:
number and types of arguments must match.
Example (function to calculate area) :

```
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 x6 = area(10, -7);            // this is a difficult case:
                                  //    the types are correct,
                                  //    but the values have 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
    - Do not expect that user of your code will do that ! Humans are lazy :-)
  - The function should check (good approach)
    - Return an "error value" (not general, problematic)
    - Set an error status indicator (e.g. "int errnum" in global scope) (not general, problematic – don't do this)
    - Throw an exception
- Note: unfortunately, sometimes we can't change a function that handles errors in a way we do not like (e.g. function from library)

Bad function arguments

- Why to worry?
  - You want your programs to be correct
  - Typically the author of a function has no control over how it is called
    - Writing “do it this way” in the manual is not a solution – usually 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 most complex

How to report an error

- Return an "error value" (not general, problematic)

```
int area(int length, int width)   // return a negative value if bad
{                                 // argument

    if(length <=0 || width <= 0) return -1;
    return length*width;
}
```

So, "let the caller beware". For example:

```
int z = area(x,y);
if (z<0) error ("bad area computation");
// error() here is some function that reports an error
```

- Problems:
  - What if caller 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)

```
int errno = 0;      // is used to indicate errors (global variable)
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 caller forget to check **errno**?
  - Global variable **errno** – not good.
  - How do user deal with that error?

How to report an error

- Report an error by throwing an exception (correct approach)

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

int area(int length, int width)
{
   if (length<=0 || width<=0) throw Bad_area();   // note the ()
   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() report
   cout << "oops! Bad area calculation – fix program"<<endl;
}
```

Exceptions

- Exception handling is general
  - You can't forget about an exception: the program will terminate even if it's not handled

    using a **try{} catch{}** construction
  - Almost every kind of error can be reported using exceptions
- You still have to figure out what to do with an exception thrown in your program.
  - Error handling is **never** really simple

“Out of range” exception

Try this:

```
vector<int> v(10);  // create 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){             // bug. Access to 10-th element
    cout << "v[" << i << "] == " << v[i]    << endl;   // run-time crash (?)
    cout << "v[" << i << "] == " << v.at(i) << endl;  // OK. Exception
}
```

Note: **operator[ ]** *(subscript operator) of standard library vector do not reports a bad index, but member function* **at()** *does it by throwing a* **out_of_range** *exception*

Exceptions

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

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

Note: standard exceptions (e.g. **out_of_range)** *are defined in* **<stdexcept>** *header*

A function error()

Here is implementation of a simple **error()** function

It works by “tagging” throws by string of comment **s**:

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

runtime_error(const string& s) *is a standard exception* (defined in ***<stdexcept>***)

Using error( )

- Example

```
cout << "please enter integer in range [1..10]"<<endl;
int x = -1;      // initialize with unacceptable value
cin >> x;        // read from the keyboard
if (!cin)        // check that cin read an integer
   error ("didn't get an integer");
if (x < 1 || x > 10) // check if value is out of range
   error ("x is out of range");

// if we reach this point,
// we can use x with confidence
```

How to look for errors

When you have written (drafted) a program, most probably it will have errors (commonly called “bugs”):

Program does something, but not what you've expected. What to do:

- Find out what it actually does.
- Correct bug(s)
- Try again

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 "looks 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:

- Write comments.
  - Explain design ideas. *Write comments for yourself!*
- Use meaningful names.
- Indent. Many text editors (as “emacs”) do it for you.
  - Use a consistent code layout
- 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 << endl;          // oops!
```

- is every character literal terminated?

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

- is every block is terminated?

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

- every set of parentheses matched?

```
if (a>0                                     // oops!
    x = f(y);
```

First get the program to compile

- is every name declared?
  Did you include needed headers?
  (e.g., **<iostream>, <vector>, <algorithm>...**)

- 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

- Look for run-time bugs, i.e. bugs not found by compiler:
  (That's much harder to do than it sounds):
  - **for (int i=0; 0 < vec.size(); ++i) {** *// oops! Infinite loop*
  - **for(int i = 0; i <= max; ++j) {** *// oops! (if* **j** *was declared before)*
- 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 debug output statements

    cerr << "x == " << x << ", y == " << y << endl; // error stream

Debugging

- When you write the program, always 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 (number_of_elements > largest_reasonable)
        error("unexpectedly large number of elements");

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

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

Debugging. Using assert()

- It is also a good practice to use **assert()** utility (defined in **<cassert>** header):
  - Put wherever you can **assert**(*bool_expression_that_always_must_be_true*);

    If (because of a bug) some **assert()** get **false** as parameter, you will see a message with line number and function name where it has happened.
- If you had debugged your code, all **assert()** could be disabled by compiler option -DNDEBUG
- Note:
  - If you have **bool function_do_something_important(...){...}**
  - Never do **assert(function_do_something_important(...));**

    as it could be switch off with -DNDEBUG by mistake

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 meaningful value?
- Did you handle the first / last element correctly?
- Did you handle the empty case (e.g. no elements, no input) correctly?
- Did you open your files correctly (check success of “open” operation?
- … and so on

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 sure is right.
    - Work backwards from some bad output
- Once you have found "the bug", carefully check if fixing solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I've found the last bug" - 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 but
  - if your code is used by your friends, uncaught errors can cause you to lose friends :-)

Pre-conditions

What a function requires as 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 ends?

- 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
{
    int S = length*width;
    // the result must be a positive as it is an area
    if(S  < 0) throw Bad_area();
    return S;
}
```

Testing

How do we test a program?

- “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 of the project
- When possible, test parts of a program in isolation: when you write a complicated function / class, write a little program (so called “test bed”) that uses it with all possible input conditions to see how it behaves in isolation before putting it into the real program

Next talk

- Small calculator program