Computer Science I

Error Handling

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction

Error Handling

- Errors in computer systems are *inevitable*
- Bugs vs. errors
- Bug: a flaw or defect in a computer program that causes it to produce an incorrect or unexpected result, or to behave in an unintended way
- Error: potential condition or state that can be reasonably anticipated by a programmer
- Bugs are flaws that should be resolved with rigorous *testing*

Errors

- Errors cannot be "prevented" only mitigated, anticipated, and *handled*
- Common errors:
  - Bad input leads to bad output (GIGO)
  - Illegal operations: dividing by zero
  - Dereferencing `NULL` pointers
  - More general problems: missing file, limited resources (memory), bad password, no network connection,
- Some errors may be unexpected/catastrophic/fatal
- Others are *recoverable*

Error Handling

- Dealing with error conditions is called *error handling*
- Two general approaches
- Defensive programming:
  - Check for dangerous/illegal/invalid operations before doing them; if an error would result, we "choose" not do to them
  - We can then "fail silently" or communicate the type of error and let the calling function decide *how to handle it*
  - "Look before you leap"
- Exception handling

Exception Handling

- Modern programming support *Exceptions*
- Exception: an *event* during the execution of a program that disrupts the normal control flow of the program
- Exceptions are *thrown* and may be *caught* (and handled)
- "Go ahead and leap without looking, you'll be *caught* if you fall"
- Many advantages to exception handling over defensive programming
- Not supported in C

Error Handling in C

- C generally uses defensive programming
- Error handling is generally on the function-level
- Functions validate input, check for error conditions, etc. before proceeding
- If an error is detected, the function aborts and returns
- Error condition is communicated to the calling function via an *error code*
- Error code: a number (integer) indicating the type of error (or none)

Part II: Error Codes

Error Codes

- C provides a standard error library: `errno.h` (error number)
- Defines standard *errors codes* and some (limited) utilities
- A global `int` variable named `errno` can be set by standard functions in the event of an error
- Value can be checked for an error "state"
- Zero: no error
- Only three "standard" error codes:
  - `EDOM` indicates an error in the domain of a function
  - `ERANGE` indicates an error in the range of a function
  - `EILSEQ` illegal byte sequence
- Error codes defined via macros `#define`

EDOM

`EDOM`

- Error in the domain value of a function
- Functions map a *domain* to a *range*
- Domain is the set of all possible inputs
- In other words: illegal input
- Example: $\sqrt{x}$ is only defined for values ≥ 0
- `sqrt(-1)` would result in an `EDOM` error

ERANGE

`ERANGE`

- Error in the range value of a function
- Range is the set of all possible outputs
- Illegal or aberrant output value from a function
- Example: $\log(0)$ is undefined (but converges to $-\infty$)
- `log(0)` would result in an `ERANGE` error
- Demonstration

POSIX Error Codes

- Portable Operating System Interface (POSIX) standard defines many more error codes
- Mostly for systems programming
- Examples:
  - No such file or directory
  - Out of memory
  - Network is down
- Demonstration

Exit Codes

- Similar: *exit codes*
- When a program quits, it can "return" a value to the operating system
- Can be used externally to determine if a program was successful
- Example: Segmentation Faults usually exit with an error code of 139
- Actual numbers are not standardized
- Two standard flags defined in `stdlib.h`
- `EXIT_FAILURE` (usually 1)
- `EXIT_SUCCESS` (usually 0, no error)
- Demonstration

Part III: Defensive Programming

Defensive Programming

- Don't generally use standard error codes for user-defined functions
- Can use the same approach: defensive error checking with error codes
  1. Look before you leap: check for invalid state before a dangerous operation
  2. If invalid, return an error code to communicate the *type of error*

Defensive Programming

General design philosophy:

- You *communicate* the error to the calling function
- You *don't* decide (dictate) what how to *handle* the error
- The calling function is responsible for deciding what to do

Defensive Programming

Advantages:

- Makes your functions more flexible
- Leaves the decision making process to the user of the library
- Different error codes means the calling function can decide to apply different solutions to different errors
- Avoids unrecoverable state

Common Implementations

- Input validation (ranges)
- Null pointer checks
- Outputs are "returned" via pass-by-reference variables
- Preserve the return value to return an error code
- Convention: use zero for success
- Similar to booleans: 0 = no error, non-zero = some kind of error

Demonstration

Modify the `euclideanDistance` and `computeLine` functions to use error codes.

Pitfalls

- In general: functions should *not exit*
  - Takes the decision away from the calling function
  - Makes all errors fatal errors
  - Defeats the purpose of error *handling*
- In general: functions should *not print error output*
  - Most programs are not interactive, messages are pointless
  - Standard error/output may not be monitored
  - Proper logging systems should be used in practice
- Error checking should always *come first*
  - Look *before* you leap
  - Dangerous operations could leave a program in an illegal state, unable to actually *handle* an error

Part IV: Enumerated Types

Enumerated Types

- Some pieces of data have a limited number of possible values
- Examples: days of the week, months in a year, *error codes*
- You can define an *enumerated type* with pre-defined human-readable values
- An *enumeration* is a complete, ordered listing of all items in a collection

Syntax & Style

```c
typedef enum {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
} DayOfWeek;
```

Syntax:

- `typedef` (type definition) and `enum` (enumeration)
- Opening/closing curly brackets
- Comma-delimited list of possible values
- Name of the enumerated type followed by a semicolon

Syntax & Style

```
typedef enum {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
} DayOfWeek;
```

Style:

- `UPPER_UNDERSCORE_CASING` for values
- One value per line for readability
- Name: `UpperCamelCasing` (modern convention)
- Typically declared in a header file

Using

- Once declared you can use an enumerated type like other built-in variable types

```
DayOfWeek today;
today = TUESDAY;
```

```
if(today == FRIDAY) {
  printf("Have a good weekend!\n");
}
```

Pitfall

- In reality, C uses `int` values for enumerated types
- Default usually starts at 0
- `SUNDAY = 0`, `MONDAY = 1`, . . . , `SATURDAY = 6`
- You *can* perform integer arithmetic on enumerated types

```
DayOfWeek today = FRIDAY;
today = today + 1;
today++;
```

```
DayOfWeek someday = 99999;
```

- *Can*, but **shouldn't**

Use Cases

- Using enumerated types allows you to use human-readable terms
- Without enumerated types, you are forced to use *magic numbers*
- Makes your code more readable and easily understood
- Slight advantage over `#define` "constants": understood by debuggers; name conflicts are compile-time errors.
- Demonstration

Part V: Unit Testing

Software Testing

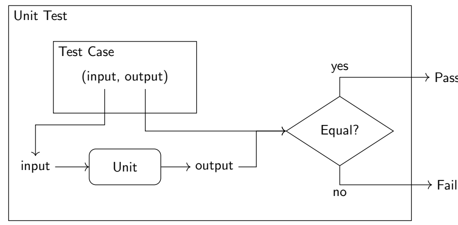
Many different types of *software testing*

- Functional vs. non-functional testing
- Acceptance testing
- Performance and load testing
- Integration testing
- Regression testing
- Unit testing

Unit Testing

- Unit testing involves testing a *unit* of code
- Units:
  - A module, submodule, or library
  - A *class* or a single header/source file
  - An individual function
- A unit test can involve several *test cases* comprising a *test suite*
- A test case is an input-output pair that is *known* to be correct that we can test the unit (function) against
- If the function produces the same (or sufficiently similar) output, it *passes* the test, otherwise it *fails* the test

Illustration



Importance of Testing

- Testing provides some *assurance of quality*
- Provide a reasonably high confidence that our software is *correct*
- Correct: it conforms to our specifications or expectations
- Never a guarantee: testing only gives assurances for what we test, not for what we do not (or cannot) test
- Still possible to have false positives and false negatives if our tests are wrong
- Prevents/reduces costly bugs that manifest themselves “in production”
- Informs good design

Goals of Testing

- Should strive for good or high *code coverage*
- Test as many types of input(s) as we can
- Edge cases: testing “extreme” inputs or input values at the edge of extreme
- Corner cases: outside normal operating procedures
- Try to break our code; be *adversarial*
- Don’t just test what you expect should work

Costs of Testing

- Testing code is often larger than the code it tests
- May require just as much or more time and effort as the code itself
- Example: SQLite is small (128.9 kLOC) but has 91,772 kLOC of testing code and scripts (711 times larger)
- Example: International Space Station has 1.8 mLOC vs 3.3 + 11 mLOC for simulation and testing
- Worth it: reduces *technical debt*
- Testing is an *investment*

Demonstration

- Ad-hoc Testing
- Designing a suite of automated tests
- Unit testing with a formal testing framework: cmocka