cs121 - software development
debugging and testing

alexandre r.j. francçois
visiting associate professor of computer science
preamble: debugging

depbugging = removing bugs (errors) from software

correctness
  make the code compile and link
  make the program run without crashing
  make the program run without losing memory
  make the program do what it is supposed to do

performance
  make the code and executable efficient
preamble: testing

verification: did you built the software right?
    the product has been built according to the requirements
    and design specifications

validation: did you built the right software?
    the product actually meets the user's needs
    the specifications were correct in the first place
outline

introduction

static debugging
  code – compile – link

runtime debugging
  test – crash – tools

testing
debugging

notice
localize
understand
repair
(learn)

best debugging tool: human brain
debugging is a cognitive activity...
debugging techniques

goal: localize and understand
   static analysis
   defensive programming
   tracing
   interactive debuggers and other tools
static debugging

good debugging practices and good coding practices
  make code readable (by humans!)
  use code editor features: indent, syntax highlighting

code review
  some mistakes the compiler cannot catch

compiler options – warning and optimization options
makefiles – compile and link dependencies
version control system – what has changed?
main causes of crash

invalid array indexing
invalid pointers
infinite loop
  if consumes memory
invalid processor operations
  e.g. division by 0

crashes often result from memory problem
might be caused by any type of error
tracing

program flow
  conditional branches
  function calls
variable values
  initialization
  indices, array elements
pointer values
  initialization
  memory allocation
“cerr debugging”

output statements track control flow and data values
not the best practice, but can be very useful

make output statements easy to switch off
inline function, compiler constant

prefer *cerr* to *cout*
unbuffered: no output lost when program crashes
defensive programming: assertions

assertions are expressions that should evaluate to true at a specific point in the code

**assert** macro

```c
#include <assert.h>
```

pass assertion expressions as argument to assert
if assertion fails, the program calls `abort()` after printing position
remove all assertion by compiling with `-DNDEBUG`

assert statements document assumptions made when coding
run-time debugging

test
  find what defects or bugs exist
stabilize
  make bugs reproducible
localize
  identify code responsible
correct
  fix bug
verify
  make sure the fix works
(repeat)
black box testing

test interface through interface
  no knowledge of implementation

+: unbiased by implementation details

-: uninformed about implementation
  insufficient or redundant tests

regression testing: compare output of running program
  with fixed inputs and expected outputs from one
  version to the next
white box testing

test code through code

+: design tests to achieve good code coverage and avoid duplication
+: can stress complicated, error-prone code
+: can stress boundary values (fault injection)

-: tester=developer may have bias
-: if code changes, tests may have to be redesigned
gray box testing

test code through interface
unit testing

original goal
  the systems meets contractual requirements
  test through public interface

xUnit architecture for unit testing frameworks
  Junit, CppUnit, GoogleTest

(what constitutes a unit / component?)
what makes a good test?

tests should be independent and repeatable

tests should be well organized and reflect the structure of the tested code

tests should be portable and reusable

when tests fail, they should provide as much information about the problem as possible

tests should be fast

**tests should be few and useful**

not as easy as it sounds...
test case basics (from googletest)

start by writing assertions, which are statements that check whether a condition is true

an assertion's result can be success, nonfatal failure, or fatal failure

if a fatal failure occurs, it aborts the current function otherwise the program continues normally

tests use assertions to verify the behavior of the tested code

the test fails if the test driver crashes or has a failed assertion

otherwise the test succeeds
test case basics (from googletest)

a test case contains one or many tests
organize tests into test cases that reflect the structure of the tested code
when multiple tests in a test case need to share common objects and subroutines, they can be put into a test fixture class

a test program can contain multiple test cases
unit testing

hierarchical testing
  test individual components at each level of the physical hierarchy
  separate test driver for each component

Incremental testing
  test only the functionality actually implemented within the component under test
  complexity of the test proportional to the complexity of the component
testing outcome/symptoms

crash

incorrect result/behavior

expected result/behavior
  program correct?
  memory leak? executable size? memory usage? speed?
interactive debuggers

interactive debugging
  step by step execution
  watchpoints
  data values and call stack

post-mortem analysis
  core dump memory snapshot (Unix-like systems only)
other tools

correctness: instrumented code libraries
  trace program flow
  trace memory allocation / deallocation
  trace variable / pointer initialization
  etc.

performance: profiling tools
  memory usage
  time spent in each function
  etc.
summary

good coding practices
editor features
compiler and linker options and error and warning messages
assertions
tracing
tools

lots of practice
easy to get, remember to learn