Robust Programming + Testing, Profiling, & Instrumentation

CS 217

Program Errors

- Programs encounter errors
  - Good programmers handle them gracefully
- Types of errors
  - Compile-time errors
  - Link-time errors
  - Run-time user errors
  - Run-time program errors
  - Run-time exceptions

Compile-Time Errors

- Code does not conform to C specification
  - Forgetting a semicolon
  - Forgetting to declare a variable
  - etc.
- Detected by compiler

```
int a = 0;
int b = 3
int c = 6;

a = b + 3;
d = c + 3;
```

```
cc-1065 cc: ERROR File = foo.c, Line = 2
  A semicolon is expected at this point.

    int c = 6;
    ^

cc-1020 cc: ERROR File = foo.c, Line = 6
  The identifier "d" is undefined.

    d = c + 3;
    ^
```

Link-Time Errors

- Error in linking together the .o files to make an a.out
  - Symbol referenced (used) in one module, not defined in another

```
extern int not_there;
.
.
.
main() {
 printf("%d", not_there);
}
```

```
Undefined        first referenced
 symbol              in file
not_there          foo.o
ld: fatal: Symbol referencing errors.
No output written to a.out
```

Run-Time User Errors

- User provides invalid input
  - User types in name of file that does not exist
  - User provides program argument with value outside legal bounds
  - etc.
- Detected with “if” checks in program
  - Program should print message and recover gracefully
  - Possibly ask user for new input
- Your program should anticipate and handle EVERY possible user input!!!

```
int ReadFile(const char *filename)
{
  FILE *fp = fopen(filename, “r”);
  if (!fp) {
    fprintf(stderr, “Unable to open file: %s\n”, filename);
    return 0;
  }
...
```

Run-Time Program Errors

- Internal error from which recovery is impossible (bug)
  - Null pointer passed to **`Array_getData()`**
  - Invalid value for array index (k = -7)
  - Invariant is violated
  - etc.
- Detected with conditional checks in program
  - Program should print message and abort

```
void Array_getData(Array_T array, int k)
{
  return array->elements[k];
}
```

Run-time Exceptions

- Rare error from which recovery may be possible
  - User hits interrupt key
  - Arithmetic overflow
  - etc.
- Detected by machine or operating system
  - Program can handle them with signal handlers (later)
  - Not usually possible/practical to detect with conditional checks

```
#include <limits.h>
...
int a = MAX_INT;
int b = MAX_INT;
int c = 6;
int d = 0;
...
a = a + d;
d = a + b;
b = a - c;
...
```

Robust Programming

- Your program should never terminate without either ...
  - Completing successfully, or
  - Outputting a meaningful error message
- How can a program terminate?
  - Return from main
  - Call exit
  - Call abort

Robust Programming

- Your program should never terminate without either ...
  - Completing successfully, or
  - Outputting a meaningful error message
- How can a program terminate?
  - > **Return from main**
  - Call exit
  - Call abort

```
#include <stdio.h>
#include "stringarray.h"

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, strcmp);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

Robust Programming

- Your program should never terminate without either ...
  - Completing successfully, or
  - Outputting a meaningful error message
- How can a program terminate?
  - Return from main
  - > **Call exit**
  - Call abort

```
...
#include <stdlib.h>

void ParseArguments(int argc, char **argv)
{
  argc--; argv++;

  while (argc > 0) {
    if (!strcmp(*argv, "-filename")) {
       ...
    }
    else if (!strcmp(*argv, "-help")) {
      PrintUsage();
      exit(0);
    }
    else {
      fprintf(stderr, "Unrecognized argument: %s\n", *argv);
      PrintUsage();
      exit(1);
    }
    argv++; argc--;
  }
}
```

Robust Programming

- Your program should never terminate without either ...
  - Completing successfully, or
  - Outputing a meaningful error message
- How can a program terminate?
  - Return from main
  - Call exit
  - > **Call abort**

```
...
#include <stdlib.h>

void *Array_getData(Array_T array, int k)
{
  if (!array) {
    fprintf(stderr, “array=NULL in Array_getData\n”);
    abort();
  }

  if ((k < 0) || (k >= array->nelements)) {
    fprintf(stderr, “k=%d in Array_getData\n”, k);
    abort();
  }

  return array->elements[k];
}
```

Assert

- **`void assert(int expression)`**
  - Issues a message and aborts the program if ***expression*** is 0
  - Activated conditionally
    - While debugging: **`gcc foo.c`**
    - After release: **`gcc –DNDEBUG foo.c`**
- Typical uses
  - Check function arguments
  - Check invariants!!!

`assert.h`

```
#ifdef NDEBUG
#define assert(_e) 0
#else
#define assert(_e) \
  if (_e) { \
    fprintf(stderr, “Assertion failed on line %d of file %s\n”, __LINE__, __FILE__); \
    abort(); \
  }
  0
#endif
```

Assert

- **`void assert(int expression)`**
  - Issues a message and aborts the program if ***expression*** is 0
  - Activated conditionally
    - While debugging: **`gcc foo.c`**
    - After release: **`gcc -DNDEBUG foo.c`**
- Typical uses
  - > **Check function arguments**
  - Check invariants!!!

```
#include <assert.h>

void *Array_getData(Array_T array, int k)
{
  assert(array);
  assert((k >= 0) && (k < array->nelements));

  return array->elements[k];
}
```

Assert

- **`void assert(int expression)`**
  - Issues a message and aborts the program if ***expression*** is 0
  - Activated conditionally
    - While debugging: **`cc foo.c`**
    - After release: **`cc -DNDEBUG foo.c`**
- Typical uses
  - Check function arguments
  - > **Check invariants!!!**

```
#include <assert.h>

void Array_remove(Array_T array, int index)
{
  int i;

  for (i = index+1; i < array->num_elements; i++)
    array->elements[i-1] = array->elements[i];

  array->num_elements--;

  assert(array->nelements >= 0);
}
```

What `assert` is not best for

- Assert is meant for *bugs*, conditions that "can't" occur (or if they do, it's the programmer's fault)
  - File-not-present happens all the time, *beyond the control of the programmer*
  - Instead of an assert, print a nice error message to the user, then exit or retry

```
FILE *f;
f = fopen(filename, "r");
assert(f);
```

C Preprocessor

- Invoked automatically by the C compiler
  - try `gcc –E foo.c`
- C preprocessor manipulates text prior to C compiling
  - file inclusion
  - conditional compilation
  - macros

File Inclusion

- Header files contain declarations for modules
  - Names of header files should end in **`.h`**
- User-define header files **“ ... ”**

```
#include “mydefs.h”
```

- System header files: **< ... >**

```
#include <stdio.h>
```

Conditional Compilation

- Removing macro definitions

```
#undef plusone
```

- Conditional compilation

#ifdef *name*
#ifndef *name*
#if *expr*
#elif *expr*
#else
#endif

- Why use?

```
#ifndef FOO_H
#define FOO_H

#ifdef WINDOWS_OS
#include <windows.h>
#endif

.
.
.
#endif
```

```
gcc -DWINDOWS_OS foo.c
```

Macros

- Provide parameterized text substitution
- Macro definition

```
#define MAXLINE 120
#define lower(c) ((c)-`A'+'a')
```

- Macro replacement

```
char buf[MAXLINE+1];
```

becomes

```
char buf[120+1];
```

```
c = lower(buf[i]);
```

becomes

```
c = ((buf[i])-`A'+'a');
```

Macros (cont)

- Always parenthesize macro parameters in definition

```
#define plusone(x) x+1

i = 3*plusone(2);
```

becomes

```
i = 3*2+1
```

```
#define plusone(x) ((x)+1)

i = 3*plusone(2);
```

becomes

```
i = 3*((2)+1)
```

Macros (cont)

- Always avoid side-effects in parameters passed to macros

```
#define max(a, b) ((a)>(b)?(a):(b))

y = max(i++, j++)
```

becomes

```
y = ((i++)>(j++)?(i++):(j++));
```

Robust Programming Summary

- Programs encounter errors
  - Good programmers handle them gracefully
- Types of errors
  - Compile-time errors
  - Run-time user errors
  - Run-time program errors
  - Run-time exceptions
- Robust programming
  - Complete successfully, or
  - Output a meaningful error message

Different execution times

1. Preprocessing time
2. Compile time
3. Link time
4. Run time

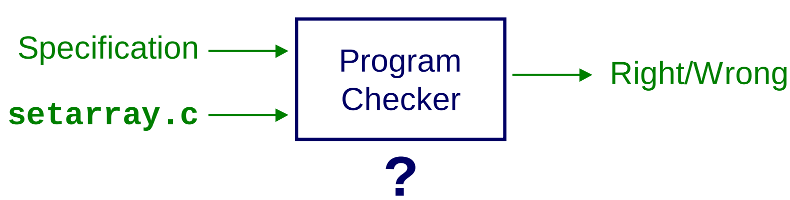
Testing, Profiling, & Instrumentation

- How do you know if your program is correct?
  - Will it ever core dump?
  - Does it ever produce the wrong answer?
    - Testing
- How do you know what your program is doing?
  - How fast is your program?
  - Why is it slow for one input but not for another?
  - Does it have a memory leak?
    - Timing
    - Profiling
    - Instrumentation

See Kernighan & Pike book:
"The Practice of Programming"

Program Verification

- How do you know if your program is correct?
  - Can you **prove** that it is correct?
  - Can you **prove** properties of the code?
    - e.g., it terminates



Program Testing

- Convince yourself that your program probably works

How do you write a test program?

Test Programs

- Properties of a good test program
  - Tests boundary conditions
  - Exercise as much code as possible
  - Produce output that is known to be right/wrong

How do you achieve all three properties?

Program Testing

- Testing boundary conditions
  - Almost all bugs occur at boundary conditions
  - If program works for boundary cases, it probably works for others
- Exercising as much code as possible
  - For simple programs, can enumerate all paths through code
  - Otherwise, sample paths through code with random input
  - Measure test coverage
- Checking whether output is right/wrong?
  - Match output expected by test programmer (for simple cases)
  - Match output of another implementation
  - Verify conservation properties
  - Note: real programs often have fuzzy specifications

Example Test Program

```
#define ASSURE(b) assure(b, __LINE__)

void testSmallTable(void)
/* Test a SymTable that contains a few bindings. */
{
   . . .

   /* Test SymTable_new. */
   printf("Testing SymTable_new.\n");
   oSymTable = SymTable_new();
   ASSURE(oSymTable != NULL);

   /* Test SymTable_put. */
   printf("Testing SymTable_put.\n");
   bSuccessful = SymTable_put(oSymTable, pcRuth, pcRightField);
   ASSURE(bSuccessful);
   bSuccessful = SymTable_put(oSymTable, pcGehrig, pcFirstBase);
   ASSURE(bSuccessful);
   bSuccessful = SymTable_put(oSymTable, pcMantle, pcCenterField);
   ASSURE(bSuccessful);

   /* Test SymTable_getLength. */
   printf("Testing SymTable_getLength.\n");
   iLength = SymTable_getLength(oSymTable);
   ASSURE(iLength == 4);

   /* Test SymTable_contains. */
   printf("Testing SymTable_contains.\n");
   bSuccessful = SymTable_contains(oSymTable, pcRuth);
   ASSURE(bSuccessful);
   . . .
```

Systematic Testing

- Incremental testing
  - Test as write code
  - Test simple cases first
  - Test code bottom-up
- Stress testing
  - Generate test inputs procedurally
  - Intentionally create error situations for testing
  - Run tests as batch processes … often

```
void *testmalloc(size_t n)
{
  static int count = 0;
  if (++count > 10) return 0;
  else return malloc(n);
}
```

Timing, Profiling, & Instrumentation

- How do you know what your code is doing?
  - How slow is it?
    - How long does it take for certain types of inputs?
  - Where is it slow?
    - Which code is being executed most?
  - Why am I running out of memory?
    - Where is the memory going?
    - Are there leaks?
  - Why is it slow?
    - How imbalanced is my binary tree?

Timing

- Most shells provide tool to time program execution
  - e.g., bash "**time**" command

```
bash> tail -1000 /usr/lib/dict/words > input.txt

bash> time sort5.pixie < input.txt > output.txt
real    0m12.977s
user    0m12.860s
sys     0m0.010s
```

Timing

- Most operating systems provide a way to get the time
  - e.g., UNIX "**gettimeofday**" command

```
#include <sys/time.h>

struct timeval start_time, end_time;

gettimeofday(&start_time, NULL);
  <execute some code here>
gettimeofday(&end_time, NULL);

float seconds = end_time.tv_sec - start_time.tv_sec +
     1.0E-6F * (end_time.tv_usec - start_time.tv_usec);
```

Profiling

- Gather statistics about your program's execution
  - e.g., how much time did execution of a function take?
  - e.g., how many times was a particular function called?
  - e.g., how many times was a particular line of code executed?
  - e.g., which lines of code used the most time?
- Most compilers come with profilers
  - e.g., **`pixie`** and **`prof`**

Profiling Example

```
#include <stdio.h>
#include <string.h>
#include "stringarray.h"

int CompareStrings(void *s1, void *s2)
{
  return strcmp(s1, s2);
}

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, CompareStrings);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

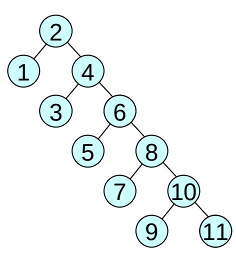
Profiling Example

```
bash> cc -o sort5.c etc.
bash> pixie sort5
bash> sort5.pixie < input.txt > output.txt
bash> prof sort5.Counts
```

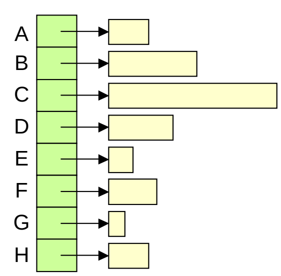
```
Summary of ideal time data (pixie-counts)--
                    3664181847: Total number of instructions executed
                    3170984513: Total computed cycles
                        16.261: Total computed execution time (secs.)
                         0.865: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
excl.secs  excl.%   cum.%      cycles  instructions    calls  function  (dso: file, line)
-------------------------------------------------------------------------
  8.935    54.9%    54.9%  1742355689    1778629217        1  Array_sort (sort5: array.c, 110)
  5.897    36.3%    91.2%  1149885000    1299870000 49995000  CompareStrings (sort5: sort5.c, 7)
  1.386     8.5%    99.7%   270290536     575736340 49995000  strcmp (libc.so.1: strcmp.s, 34)
  0.010     0.1%    99.8%     1879873       2279949    10000  _doprnt (libc.so.1: doprnt.c, 227)
  0.004     0.0%    99.8%      746528        584896    20000  strlen (libc.so.1: strlen.s, 58)
  0.004     0.0%    99.8%      700059        880214    10001  fgets (libc.so.1: fgets.c, 26)
  0.003     0.0%    99.9%      494950        666600    10018  _memccpy (libc.so.1: memccpy.c, 29)
  0.002     0.0%    99.9%      420000        510000    10000  Array_addKth (sort5: array.c, 72)
  0.002     0.0%    99.9%      417401        411003    10000  strcpy (libc.so.1: strcpy.s, 103)
  0.002     0.0%    99.9%      340000        450000    10000  fprintf (libc.so.1: fprintf.c, 23)
  0.002     0.0%    99.9%      310028        250028        1  StringArray_write (sort5: str...c, 22)
  0.001     0.0%    99.9%      267789        296579     2680  resolve_relocations (rld: rld.c, 2636)
  0.001     0.0%    99.9%      264264        345576    10164  cleanfree (libc.so.1: malloc.c, 933)
  0.001     0.0%    99.9%      263196        329639    10038  memcpy (libc.so.1: bcopy.s, 329)
  0.001     0.0%    99.9%      262829        413379    10000  _smalloc (libc.so.1: malloc.c, 127)
```

Instrumentation

- Gather statistics about your data structures
  - e.g., how many nodes are at each level of my binary tree?
  - e.g., how many elements are in each bucket of my hash table?
  - e.g., how much memory is allocated from the heap?

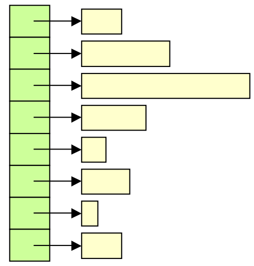


2,1,4,3,6,5,8,7,10,9,11



Instrumentation Example

Hash table implemented as array of sets



```
typedef struct Hash *Hash_T;

struct Hash {
  Set_T *buckets;
  int nbuckets;
};

void Hash_PrintBucketCounts(Hash_T oHash, FILE *fp)
{
  int i;

  /* Print number of elements in each bucket */
  for (i = 0; i < oHash->nbuckets; i++)
    fprintf(fp, “%d “, Set_getLength(oHash->buckets[i]), fp);
  fprintf(fp, “\n”);
}
```

Testing Summary & Guidelines

- Test your code as you write it
  - It is very hard to debug a lot of code all at once
  - Isolate modules and test them independently
  - Design your tests to cover boundary conditions
  - Test modules bottom-up
- Instrument your code as you write it
  - Include asserts and verify data structure sanity often
  - Include debugging statements (e.g., #ifdef DEBUG and #endif)
  - You’ll be surprised what your program is really doing!!!
- Time and profile your code **only** when you are done
  - Don’t optimize code unless you have to (you almost never will)
  - Fixing your algorithm is almost always the solution
  - Otherwise, running optimizing compiler is usually enough