Testing, Timing, Profiling, & Instrumentation

CS 217

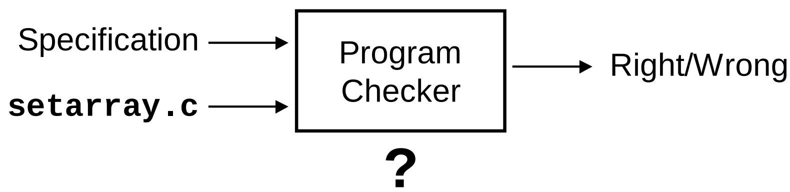
Testing, Profiling, & Instrumentation

- 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"

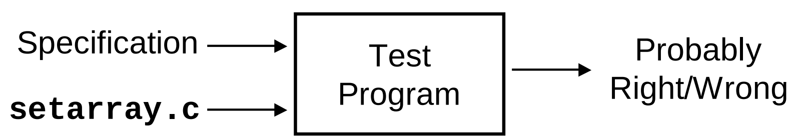
Program Verification

- 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

```
int main(int argc, char *argv[])
{
   Set_T oSet;
   SetIter_T oSetIter;
   const char *pcKey;
   char *pcValue;
   int iLength;

   /* Test Set_new, Set_put, Set_getKey, Set_getValue.  */
   oSet = Set_new(2, myStringCompare);
   Set_put(oSet, "Ruth", "RightField");
   Set_put(oSet, "Gehrig", "FirstBase");
   Set_put(oSet, "Mantle", "CenterField");
   Set_put(oSet, "Jeter", "Shortstop");
   printf("----------------------------------------------------\n");
   printf("This output should list 4 players and their positions\n");
   printf("----------------------------------------------------\n");
   pcKey = (const char*)Set_getKey(oSet, "Ruth");
   pcValue = (char*)Set_getValue(oSet, "Ruth");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Gehrig");
   pcValue = (char*)Set_getValue(oSet, "Gehrig");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Mantle");
   pcValue = (char*)Set_getValue(oSet, "Mantle");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Jeter");
   pcValue = (char*)Set_getValue(oSet, "Jeter");
   printf("%s\t%s\n", pcKey, pcValue);
```

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;
}
```

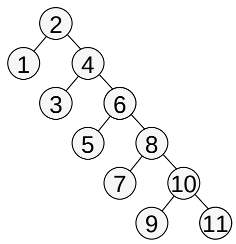
Profiling Example

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

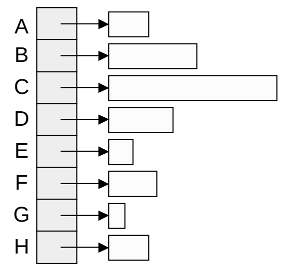
```
Summary of ideal time data (pixie-counts)--
                    3664181847: Total number of instructions executed
                    3170984513: Total computed cycles
                        16.261: Total computed execution time (secs.)
                         0.865: Average cycles / instruction
-------------------------------------------------------------------------
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?

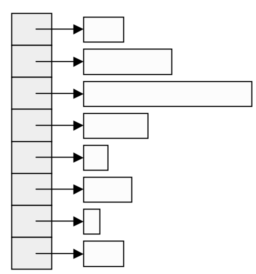



```
2,1,4,3,6,5,8,7,10,9,11
```



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”);
}
```

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