Chapter 26
Testing

Bjarne Stroustrup
www.stroustrup.com/Programming

Abstract

• This lecture is an introduction to the design and testing of program units (such as functions and classes) for correctness. We discuss the use of interfaces and the selection of tests to run against them. We emphasize the importance of designing systems to simplify testing and testing from the start. Proving programs correct and performance problems are also briefly considered.

Overview

• Correctness, proofs, and testing
• Dependencies
• System tests
• Testing GUIs
• Resource management
• Unit and system tests
• Finding assumptions that do not hold
• Design for testing
• Performance

Correctness

• Questions to ask about a program
  – Is your program correct?
  – What makes you think so?
  – How sure are you?
  – Why?
  – Would you fly in a plane that depended on that code?
  • You have to be able to reason about your code to have any real certainty
  – Programming is generally unsystematic
  – Debugging is generally unsystematic
  – What are you willing to bet that you found the last bug?
  • Related interesting questions
  – Could the program run forever if the hardware didn’t fail?
  – Does it always deliver its results in a reasonable time?

Proofs

• So why not just prove mathematically that our program is correct?
  – It’s often too hard and/or takes too long
  – Sometimes proofs are wrong too (even proofs produced by computers or by experts!)
  – Computer arithmetic isn’t the same as “real” math—remember the rounding and overflow errors we saw (due to finite and limited precision)?
  – So we do what we can: follow good design principles, test, test, and then test some more!

Testing

• “A systematic way to search for errors”
• Real testers use a lot of tools
  – Unit test frameworks
  – Static code analysis tools
  – Fault injection tools
  – …
• When done well, testing is a highly skilled and most valuable activity
  • “Test early and often”
    – Whenever you write a function or a class, think of how you might test it
    – Whenever you make a significant change, re-test
    – Before you ship (even after the most minor change), re-test
Testing

- Some useful sets of values to check (especially boundary cases):
  - the empty set
  - small sets
  - large sets
  - sets with extreme distributions
  - sets where “what is of interest” happens near the ends
  - sets with duplicate elements
  - sets with even and with odd number of elements
  - some sets generated using random numbers

A Better Test Harness (stil primitive)

Put the variables into a data file, e.g., with a format of

```
{ 27 1 2 3 5 8 13 21 0 }
```

meaning

```
{test_number value {sequence} result}
```

i.e., test #27 calls our binary search to look for the value 7 in the
sequence `{ 1 2 3 5 8 13 21 }` and checks that the result is 0 (false, that
is, not found).

Now it’s (relatively) easy to write lots of test cases, or even write another
program to generate a data file with lots of (random) cases.

Dependencies

How many dependencies can you spot in this nonsense function?

```
int do_dependent(int a, int& b) // mess function
{
    int val;
    cin>>val;
    vec[|val|] = 10;
    cout << a;
    b++;
    return b;
}
```

```
int a1[] = { 1,2,3,5,8,13,21 };
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),1)) == false)    
    cout << "1 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),5)) == false)    
    cout << "2 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),8)) == false)    
    cout << "3 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),21)) == false)    
    cout << "4 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),-7)) == true)    
    cout << "5 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),4)) == true)    
    cout << "6 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),22)) == true)    
    cout << "7 failed";
```

Resource Management

What resources (memory, files, etc.) acquired may not always be
properly released in this nonsense function?

```
void do_resources1(int a, int b, const char* s)    // mess function
    // undisciplined resource use
{
    FILE* f = fopen(s,"r"); // open file (C-style)
    int* p = new int[a];    // allocate some memory
    if (b!=0) throw Bad_arg(); // maybe throw an exception
    int* q = new int[b];    // allocate some more memory
    delete[] p;            // deallocate the memory pointed to by p
}
```
Better Resource Management

```c
// less messy function
void do_resources2(int a, int b, const string& s)
{
  istream is(s.c_str(),"r"); // open file
  vector<int>v1(a); // create vector (owning memory)
  if (b<=0) throw Bad_arg(); // maybe throw an exception
  vector<int>v2(b); // create another vector (owning memory)
}
Can do_resources2() leak anything?
```

Loops

```c
// less messy function
void do_loop(vector<int>& vec) // messy function

int do_loop(vector<int>& vec) // undisciplined loop
{
  int i;
  int sum;
  while(i<vec.size()) sum+=vec[i];
  return sum;
}
```

Buffer Overflow

- Really a special type of loop error, e.g., “storing more bytes than will fit” into an array—where do the “extra bytes” go?
- The premiere tool of virus writers and “crackers” (evil hackers)
- Some vulnerable functions (best avoided):
  - `gets`, `scanf` // these are the worst: avoid!
  - `sprintf`
  - `strcat`
  - `strcpy`
  - …

Buffer overflow

- Don’t avoid unsafe functions just as a fetish
  - Understand what can go wrong and don’t just write equivalent code
  - Even unsafe functions (e.g. `strcpy()`) have uses
    * if you really want to copy a zero terminated string, you can’t do better than `strcpy()`
    
    ```c
    char buf[MAX];
    char* read_line() // harmless? Mostly harmless? Avoid like the plague?
    {
      int i = 0;
      char ch;
      while (cin.get(ch) && ch!='
')  buf(i++)=ch;
      buf[i+1]=0;
      return buf;
    }
    ```

Buffer overflow

- Don’t avoid unsafe functions just as a fetish
  - Understand what can go wrong and don’t just write equivalent code
  - Write simple and safe code

```
string buf;
getline(cin,buf); // buf expands to hold the newline terminated input
```

Branching

- In if and switch statements
  - Are all alternatives covered?
  - Are the right actions associated with the right conditions?
- Be careful with nested if and switch statements
  - The compiler ignores your indentation
  - Each time you nest you must deal with all possible alternatives
    - Each level multiplies the number of alternatives (not just add)
- For switch statements
  - remember the default case and to break after each other case
    - unless you really mean to “fall through”
System Tests

- Do unit tests first, then combinations of units, and so on, till we get to the whole system
  - Ideally, in isolation from other parts of the system
  - Ideally, in a repeatable fashion
- What about testing GUI based applications?
  - Control inversion makes GUI testing difficult
  - Human behavior is not exactly repeatable
  - Humans still needed at some point (only a human can evaluate “look and feel”)
  - Timing, forgetfulness, boredom, etc.
  - Humans still needed at some point (only a human can evaluate “look and feel”)
- Timing, forgetfulness, boredom, etc.

Testing Classes

- A type of unit test
  - but most class objects have state
  - Classes often depend on interactions among member functions
- A base class must be tested in combination with its derived classes
  - Virtual functions
  - Construction/initialization is the combined responsibility of several classes
  - Private data is really useful here (beware of protected data members)
- Take our Shape class as an example:
  - Shape has several functions
  - A Shape has a mutable state (we can add points, change color, etc.); that is, the effect of one function can affect the behavior of another function
  - Shape is not an algorithm (why not?)
- A change to a Shape can have an effect on the screen (so maybe we still need a human tester?)

Finding assumptions that do not hold

- For example, illegal input arguments
  - Should never happen, but it does
  - Check before each call or at the beginning of the function
  - Depending on which code we can modify
  - E.g., sqrt first checks that its argument is a non-negative value
- That can be difficult/problematic:
  - Consider binary_search(a,b,v); // is v in [a:b)
    - For forward iterables (e.g., for a list), we can’t test if a<b – no < operation
    - For random-access iterables, we can’t check if a and b are part of the same sequence
    - The only perfect solution involves a run-time checking library
  - Scanning the entire sequence to verify it’s sorted is much more work than actually doing the binary search
  - The purpose of binary_search() is to be faster than linear search
  - Time “interesting” test cases
    - E.g., using time or clock()
  - Repeat ≥3 times; should be ±10% to be believable

Performance

- Is it efficient enough?
  - Note: Not “Is it as efficient as possible?”
  - Computers are fast: You’ll have to do millions of operations to even notice (without using tools)
  - Accessing permanent data (on disc) repeatedly can be noticed
  - Accessing the web repeatedly can be noticed
  - Time “interesting” test cases
    - e.g., using time or clock()
    - Repeat ≥3 times; should be ±10% to be believable

Performance

- What’s wrong with this?
  - for (int i=0; i<strlen(s); ++i) {
    // do something with s[i]
  }
  - It was part of an internet message log analyzer
  - Used for files with many thousands of long log lines
Using `clock()`

```c
int n = 1000000;  // repeat `do_something()` n times

clock_t t1 = clock();
if (t1 == clock_t(-1)) {  // `clock_t(-1)` means "clock() didn't work"
    cerr << "sorry, no clock\n";
    exit(1);
}

for (int i = 0; i < n; i++) do_something();  // timing loop

clock_t t2 = clock();
if (t2 == clock_t(-1)) {
    cerr << "sorry, clock overflow\n";
    exit(2);
}

cout << "do_something() \n" << n << " times took \n" << double(t2-t1)/CLOCKS_PER_SEC << " seconds \n" (measurement granularity: \n" << CLOCKS_PER_SEC << " of a second)\n";
```

Stroustrup/Programming 28