Testing

CS2023 Winter 2004

Why Test?

- *The Practice of Programming*, Chapter 6
- Demonstrate the presence of bugs, not their absence
- How to write bug-free code?
  - Think about potential problems as you code
  - Test systematically and use automated tests
  - Generate the code with a program
  - Use functions, and test them individually

Test as You Write the Code

- Test code at its boundaries

```
int i;
char s[MAX];

for ( i = 0; (s[i] = getchar()) != '\n'
        && i < MAX-1; i++)
;
s[--i] = '\0';
```

Test as You Write the Code

Use idiom!

```
int i;
char s[MAX];

for ( i = 0; i < MAX-1; i++)
   if ((s[i] = getchar()) == '\n')
       break;
s[i] = '\0';
```

Test as You Write the Code

```
int i;
char s[MAX];

for ( i = 0; i < MAX-1; i++)
    if ((s[i] = getchar()) == '\n' ||
          s[i] == EOF)
         break;
s[i] = '\0';
```

Test as You Write the Code

```
int c, i;
char s[MAX];

for ( i = 0; i < MAX-1; i++)
{
   if ((c = getchar()) == '\n' ||
          c == EOF)
         break;
   s[i] = c;
}
s[i] = '\0';
```

- What about outer boundary?
  What happens if line is longer than MAX?

Test pre- and post- conditions

- Verify that expected or necessary properties hold before and after some piece of code executes
- Pre-condition example:

Test pre- and post- conditions

```
double avg(double a[], int n)
{
  int i;
  double sum;

  sum = 0.0;
  for (i = 0; i < n; i++)
    sum += a[i];
  return sum/n;
}
```

what if n is zero or negative?

Test pre- and post- conditions

```
double avg(double a[], int n)
{
    int i;
    double sum;

    sum = 0.0;
    for (i = 0; i < n; i++)
        sum += a[i];
    return n <= 0? 0.0 : sum/n;
}
```

Preconditions

- **`i >=0`** precondition to computing **`sqrt(i)`**
  - postcondition is the desired square root of **`i`**
- **`b*b - 4*a*c >=0`** precondition to finding real roots of a quadratic equation
- **`0 <= i < size`** precondition for using **`x[i]`** when **`x`** declared as **`x[size]`**

Assertions

- Pre- and postconditions are types of assertions
- A piece of code is considered correct if all the precondition assertions will lead to the postcondition assertions once the code is executed.
- C provides **`assert(int e)`** macro (assert.h)
  - If **`e`** == 0, error message displayed and execution of program aborted
  - If **`e`** != 0, **`assert(e)`** does nothing

Assertions

- **`assert(i >=0)`** before calling **`sqrt(i)`**
- **`assert (b*b - 4*a*c >=0)`** before finding real roots of a quadratic equation
- **`assert(0 <= i < size)`** before using **`x[i]`**

Assertions

```
#include <assert.h>
double avg(double a[], int n)
{
  int i;
  double sum;

  assert(n > 0);
  sum = 0.0;
  for (i = 0; i < n; i++)
    sum += a[i];
  return sum/n;
}
```

Assertions

- call avg with n <= 0, program aborts:

```
assert: assert.c:9: avg: assertion 'n > 0' failed
```

- Assertions slow down execution
- Can turn them off by defining **NDEBUG** prior to including <assert.h>:

```
#define NDEBUG
#include <assert.h>
```

- Can also define **NDEBUG** on compilation line:

```
gcc -DNDEBUG ...
```

When to use assertions

- Assertions useful for validating properties of parameters passed to functions
  - Can draw attention to inconsistencies between caller and callee
- Assertions can indicate who's at fault
  - If assertion of precondition fails, fault is with the caller of the function
  - If assertion of postcondition fails, fault is with the function itself

Defensive Programming

- Test for "can't happen" cases, such as previous avg example

```
if (grade < 0 || grade > 100)
  letter = '?';
else if (grade <= 90)
  letter = 'A';
else
...
```

- What to test for: null pointers, out of range subscripts, division by zero,....

Check Error Returns

- Check error returns from library functions

```
int i;
scanf("%d", &i);
printf("%d", i);

int i;
if(scanf("%d", &i) != 1) {
  fprintf(stderr, "Invalid input\n");
  return 1;
}
printf("%d", i);
```

Example

```
int factorial(int n)
{
  int fac;
  fac = 1;
  while (n--)
    fac *= n;
  return fac;
}
```

Example

```
int factorial(int n)
{
  int fac;
  fac = 1;
  while (n){
    fac *= n;
    n--;
  }
  return fac;
}
```

Example

```
int factorial(int n)
{
  int fac;
  if(n < 0) return 0;
  fac = 1;
  while (n){
    fac *= n;
    n--;
  }
  return fac;
}
```

Another Example

- Print characters of a string one per line

```
i = 0;
do {
  putchar(s[i++]);
  putchar('\n');
} while (s[i] != '\0');
```

Another Example

```
i = 0;
while (s[i] != '\0'){
  putchar(s[i++]);
  putchar('\n');
}
```

Systematic Testing

- Test incrementally
  - Don't write large program then test it all at once
- Test each function
- Eg. function that performs binary search on array of integers. Try searching:
  - array with no elements
  - array with one element, and trial value that is
    - less than element
    - equal to element
    - greater than single element

Systematic Testing

- array with two elements and trial values that check all five possible positions
- ....

- Build a *test scaffold*

```
int i, key, nelem, arr[1000];

while(scanf("%d %d", &key, &nelem)!=EOF){
  for (i = 0; i < nelem; i++)
    arr[i] = 2*i + 1;
  printf("%d\n", binsearch(key, arr, nelem));
}
return 0;
```

Systematic Testing

- Know what output to expect!
  - not always obvious
    - compilers
    - numerical algorithms (are output properties sane?)
- Important to validate output by comparing it with known values
- If program has an inverse, check that input recovered. (eg, encryption-decription)

Regression Testing

- Compare new version of output with old version
  - compare old (old_ka) and new (new_ka) versions of ka program for a large number of different test files

```
for i in ka_data.* #loop over test data files
do
    old_ka $i > out1 # run old version
    new_ka $i > out2 # run new version
    if ! cmp -s out1 out2 #compare output
    then
      echo $i: BAD #different: print message
    fi
done
```