Testing

- **Testing**: Examination of the behavior of a program by executing the program on sample inputs.
  - Testing is a technique used at the implementation stage for checking correctness of a software

- There are other techniques for checking correctness of software such as
  - Manual review and inspection of the code
  - Automated analysis tools that analyze code and look for bugs

- Testing is a common and effective technique
  - Requires executable program and data sets (test cases)
### Software Testing

- **Correctness**
  - software should match its specifications
  - software should meet its functional requirements

- Testing is necessary because we cannot guarantee correctness in the software development process

- Testing: Checking software correctness by executing the software on some sample input values

### Software Testing

- **Goal of testing**
  - finding faults in the software
  - demonstrating that there are no faults in the software

- It is not possible to *prove* that there are no faults in the software using testing

- Testing should help locate errors, not just detect their presence
  - a “yes/no” answer to the question “is the program correct?” is not very helpful

- Testing should be repeatable
  - could be difficult for distributed or concurrent software
  - effect of the environment, uninitialized variables

- A **test case** provides all input values for one execution of the program. A **test set** is a set of test cases.
Exhaustive Testing is Hard

- Number of possible test cases (assuming 32 bit integers)
  - $2^{32} \times 2^{32} = 2^{64}$

- Do bigger test sets help?
  - Test set $\{(x=3,y=2), (x=2,y=3)\}$ will detect the error
  - Test set $\{(x=3,y=2),(x=4,y=3),(x=5,y=1)\}$ will not detect the error although it has more test cases

- It is not the number of test cases
- For two test sets $T_1$ and $T_2$, if $T_1 \subseteq T_2$, then $T_2$ will detect every fault detected by $T_1$

```c
/* a max function with an error */
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return x;
}
```

Exhaustive Testing

- Assume that the input for the $\text{max}$ procedure was a list $n$ integer values
  - Number of test cases: $2^{32} \times n$

- Assume that the size of the input is not bounded
  - Number of test cases: $\infty$

- The point is exhaustive testing is pretty hopeless
Random Testing

- Use a random number generator to generate test cases
- Derive estimates for the reliability of the software using some probabilistic analysis
- Coverage is a problem

Generating Test Cases Randomly

```c
/* an equality check function with an error */
bool equal(int x, int y)
{
    bool result;
    if (x == y)
        result = false;
    else
        result = false;
    return result;
}
```

- If we pick test cases randomly it is unlikely that we will pick a case where x and y have the same value
- If x and y can take $2^{32}$ different values, there are $2^{64}$ possible test cases. In $2^{32}$ of them x and y are equal
  - probability of picking a case where x is equal to y is $2^{-32}$
- It is not a good idea to pick the test cases randomly (with uniform distribution) in this case
How to generate test cases?

• Identify the tasks which software is expected to perform

• Create test cases which will check whether these tasks are performed by the software

• The tests must include the test input and the expected output for each test input

• For testing a function, you can use the pre and post-conditions of the function to generate the test cases and figure out the expected output.

How to generate test cases?

• Partition the input domain to equivalence classes based on the functionality of the program
  – Generate one test case for each equivalence class

• For example, testing the max function:
  – One test case where first input is higher
  – One test case where second input is higher
  – One test case where they are equal
Testing Boundary Conditions

• For each range \([R_1, R_2]\) listed in either the input or output specifications, choose five cases:
  – Values less than \(R_1\)
  – Values equal to \(R_1\)
  – Values greater than \(R_1\) but less than \(R_2\)
  – Values equal to \(R_2\)
  – Values greater than \(R_2\)
• For unordered sets select two values
  – 1) in, 2) not in
• For equality select 2 values
  – 1) equal, 2) not equal
• For sets, lists select two cases
  – 1) empty, 2) not empty

Unit vs. Integration Testing

• Unit testing
  – Testing each function (each unit) separately

• Integration testing
  – Testing parts of the system by combining functions
Unit Testing

- Testing a single unit at a time
  - We can consider each function to be a unit

- Note that unit testing allows us to isolate the errors to a single function
  - We know that if we find an error during unit testing it is in the function we are testing

- Functions in a program are not isolated, they interact with each other. Possible interactions:
  - Function calls, global variables

- For unit testing we need to isolate the function we want to test, we do this using two things
  - drivers and stubs

Drivers and Stubs

- **Driver**: A program that calls the functions that is being tested and reports the results
  - A driver simulates how the function being tested will be called during program execution

- **Stub**: A simple function used to replace a function that might not be implemented yet, or is not being tested
  - A stub simulates a function called by the function currently being tested
Drivers and Stubs

Driver and Stub should have the same interface as the functions they replace

Driver and Stub should be simpler than the functions they replace

Testing and Debugging Functions: Drivers

Driver programs allow testing of individual functions

Once a function is tested, it can be used in the driver program to test other functions

For example, the function get_input can be tested with the driver program shown in the next slide
Stubs

- When a function being tested calls other functions that are not yet tested, use a stub
- A stub is a simplified version of a function
  - Stubs are usually provide values for testing rather than perform the intended calculation
  - Stubs should be so simple that you have confidence they will perform correctly
  - For example, function a stub for the price function can be used as a stub to test the rest of the supermarket pricing program as shown in the next slide
Rule for Testing Functions

• Fundamental Rule for Testing Functions
  – Test every function in a program in which every other function in that program has already been fully tested and debugged.
Integration Testing

- Integration testing: Integrated collection of functions tested as a group or partial system

- Different approaches to integration testing
  - Bottom-up: Start with functions that do not call other functions, use drivers and combine the functions incrementally
  - Top-down: Start with functions that are not called by other functions, use stubs and combine functions incrementally

Test Driven Development

1. Write a test
2. Test passes
3. Run the test
4. Test fails
5. Write code
6. Some test fails
7. Run all tests
8. Cleanup code
Regression testing

• You should preserve all the test cases for a program
• During the maintenance phase, when a change is made to the program, the test cases that have been saved are used to do **regression testing**
  – figuring out if a change made to the program introduced any faults

• Regression testing is crucial during maintenance
  – It is a good idea to automate regression testing so that all test cases are run after each modification to the software

• When you find a bug in your program you should write a test case that exhibits the bug
  – Then using regression testing you can make sure that the old bugs do not reappear

Assertions

• Assertions are conditions that should **all be true for all executions** for the program to be considered correct
  – Assertions can be used for testing correctness
  – When an assertion fails, it means that there is a bug

• The assert macro
  – Can be used to test pre or post conditions
    ```
    #include <cassert>
    assert(boolean expression)
    ```
  – If the boolean is false then the program will abort
### Assertions as function contract

- A function defines a contract between the caller and the callee
  - The caller (the code that calls the function) must provide the arguments that are acceptable by the function
  - The callee (function implementation) must provide the desired result

- This contract can be defined using assertions:
  - **Pre-conditions** – conditions about the function parameters that must be true on function entry
  - **Post-conditions** – conditions on the return value and function parameters that must be true on function exit, *if the pre-conditions were true beforehand*

### Assertions for loops

- **loop invariant** – a condition that must be true on each iteration of the loop

- Best way to write a loop:
  - Identify the loop invariant before writing the loop
  - Initialize the variables so that the loop invariant holds in the first iteration of the loop
  - Write the loop body so that the loop invariant holds in each iteration of the loop
  - Make sure that the loop terminates (termination condition must eventually evaluate to true)
  - The conjunction of the termination condition and the loop invariant should correspond to the goal
Assertions as a design tool

• Assertions can help you in developing a solution to a programming problem

• Before you write a loop you should think about what is the loop invariant for that loop

• Before you write a function, you should think about what are the pre and post-conditions of that function

More assertions

• Assertions can also be used to check post-conditions
  – In this case, assertion violation means that there is a bug in the function body!

• Asserting loop invariants is useful for debugging

• Q. Why assert to check your own code?
  – Answer: catch bugs early and effectively
    • Bugs appear as soon as testing begins
    • Also know where bug occurred, and maybe where to fix it

• Note: use assert as a development tool ONLY
  – Easy to turn off all assertions for final product
    • #define NDEBUG before
    #include <cassert>
A pre-condition example

```c
int factorial(int k) {
    int j;
    int term;

    assert(k >= 0);
    /* pre-condition for the function */
    /* caller must make sure that the pre-condition holds */

    term = 1;
    for (j=2; j<=k; j++)
        term *=j;
    return term;
}
```

A post-condition example

```c
int max(int a, int b) {
    int result;
    if (a > b)
        result = a;
    else
        result = b;

    /* post-conditions */
    assert(result >= a && result >= b);
    assert(result == a || result == b);
    /* equivalently, we can write a single assertion
     * that is the conjunction of the above two */

    return result;
}
```
A loop invariant example

```c
int compute_sum(int n)
{
    int i, sum;
    sum = 0;
    i = 1
    // loop invariant: sum = 0 + 1 + ... + (i-1)
    // loop invariant: sum >= 0
    // loop invariant: i >= 1 && i <= n+1
    while (i <= n)
    {
        sum += i;
        i++;
    }
    return sum;
}
```

Debugging Loops

- Common errors involving loops include
  - Off-by-one errors in which the loop executes one too many or one too few times
  - Infinite loops usually result from a mistake in the Boolean expression that controls the loop
Fixing Off By One Errors

• Check your comparison: should it be < or <=?

• Check that the initialization uses the correct value

• Does the loop handle the zero iterations case?

Infinite Loops

• Loops that never stop are infinite loops

• The loop body should contain a line that will eventually cause the boolean expression to become false

• Example: Print the odd numbers less than 12
  
  ```
  x = 1;
  while (x != 12)
  {
    cout << x << endl;
    x = x + 2;
  }
  ```

• Better to use this comparison: `while (x < 12)`
Fixing Infinite Loops

• Check the direction of inequalities: 
  \(<\ or\ >\ ?\)

• Test for \(<\ or\ >\) rather than equality (==)
  – Remember that double type variables are most of the time only approximations

More Loop Debugging Tips

• Be sure that the mistake is really in the loop

• Trace the variable to observe how the variable changes
  – Tracing a variable is watching its value change during execution
    • Many systems include utilities to help with this

  – cout statements can be used to trace a value
Loop Testing Guidelines

• Every time a program is changed, it must be retested (regression testing)
  – Changing one part may require a change to another

• Every loop should at least be tested using input to cause:
  – Zero iterations of the loop body
  – One iteration of the loop body
  – One less than the maximum number of iterations
  – The maximum number of iterations

Starting Over

• Sometimes it is more efficient to throw out a buggy program and start over
  – The new program will be easier to read
  – The new program is less likely to be as buggy
  – You may develop a working program faster than if you repair the bad code
    • The lessons learned in the buggy code will help you design a better program faster
Debugging Example

• The following code is supposed to conclude with the variable product containing the product of the numbers 2 through 5

```java
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
```

Tracing Variables

• Add temporary cout statements to trace variables

```java
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
    cout << "next = " << next
         << "product = " << product
         << endl;
}
```
First Fix

- The `cout` statements added to the loop show us that the loop never multiplied by 2
  - Solve the problem by moving the statement `next++`

```cpp
int next = 2, product = 1;
while (next < 5)
{
    product = product * next;
    next++;  
    cout << "next = " << next
         << "product = " << product
         << endl;
}

- There is still a problem!
```

Second Fix

- Re-testing the loop shows us that now the loop never multiplies by 5
  - The fix is to use `<=` instead of `<` in our comparison

```cpp
int next = 2, product = 1;
while (next <= 5)
{
    product = product * next;
    next++;  
}
```
General Debugging Techniques

• We already discussed stubs, drivers, test cases
• When you find a bug, keep an open mind
  – Don’t assume the bug is in a particular location
• Don’t randomly change code without understanding what you are doing until the program works
  – This strategy may work for the first few small programs you write but is doomed to failure for any programs of moderate complexity
• Discuss the program with someone else (TA, or the instructor)

General Debugging Techniques

• Check for common errors, e.g.
  – Local vs. Reference Parameter
  – = instead of ==

• Localize the error
  – The temperature conversion program in the next slide has a bug
  – We can narrow down the bug using cout statements
```cpp
#include <iostream>
using namespace std;

int main()
{
    double fahrenheit;
    double celsius;

cout << "Enter temperature in Fahrenheit." << endl;
    cin >> fahrenheit;
    celsius = (5 / 9) * (fahrenheit - 32);
    cout << "Temperature in Celsius is " << celsius << endl;
    return 0;
}
```

**Sample Dialogue**

Enter temperature in Fahrenheit.
100
Temperature in Celsius is 0

```cpp
#include <iostream>
using namespace std;

int main()
{
    double fahrenheit;
    double celsius;

cout << "Enter temperature in Fahrenheit." << endl;
    cin >> fahrenheit;

    // Comment out original line of code but leave it
    // in the program for our reference
    // celsius = (5 / 9) * (fahrenheit - 32);

    // Add cout statements to verify (5 / 9) and (fahrenheit - 32)
    // are computed correctly
    double conversionFactor = 5 / 9;
    double tempFahrenheit = (fahrenheit - 32);

cout << "fahrenheit - 32 = " << tempFahrenheit << endl;
    cout << "conversionFactor = " << conversionFactor << endl;
    celsius = conversionFactor * tempFahrenheit;
    cout << "Temperature in Celsius is " << celsius << endl;
    return 0;
}
```

**Sample Dialogue**

Enter temperature in Fahrenheit.
100
Fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0
General Debugging Techniques

• Use a debugger
  – Tool typically integrated with a development environment that allows you to stop and step through a program line-by-line while inspecting variables

• The assert macro
  – Can be used to test pre or post conditions
    #include <cassert>
    assert(boolean expression)
  – If the boolean is false then the program will abort

Assert Example

• Denominator should not be zero in Newton’s Method

```c
// Approximates the square root of n using Newton’s Iteration.
// Precondition: n is positive, num_iterations is positive
// Postcondition: returns the square root of n
double newton_sqrt(double n, int num_iterations)
{
    double answer = 1;
    int i = 0;

    assert(n > 0) && (num_iterations> 0));
    while (i < num_iterations)
    {
        answer = 0.5 * (answer + n / answer);
        i++;
    }
    return answer;
}
```