Testing, Testing, 1.2.3, Testing

CS3081 Program Design and Development

The Enemy and How to Combat It

Good design and good development practices :

1. Simplify and isolate future modifications.

2. Find errors (now) before they permeate your code!

bug fixes, added features, requirement changes, new hardware, system migration, ...

Modifications are your enemy.
Modifications are necessary.

Organizing Code for Large-Scale Projects

- There will be A LOT of code, thus you need a way to break it down and organize it, so that each piece is managable (modularize).
- You will be writing only a portion of the software, not the whole thing, thus you need a way to share code (integrate).
- There is a good chance that multiple versions will be generated, thus you need to reuse code among projects (reuse).

And what about those errors at construction ...

Most construction errors are YOUR fault.

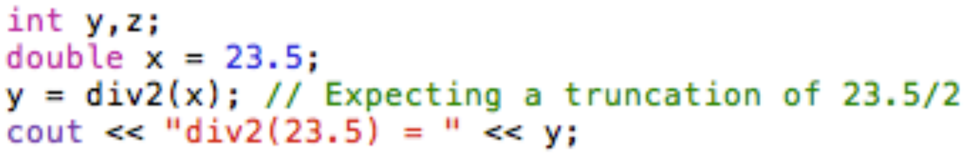
- Why YOU construct code with errors:
  - typos.
  - bad logic.
  - not a sufficient understanding of the design.
  - not a sufficient understanding of the requirements.

- The Good News ...
  - Most errors are easy to fix.
  - Errors tend to have limited scope.
  - Common errors are, well, common and predictable.

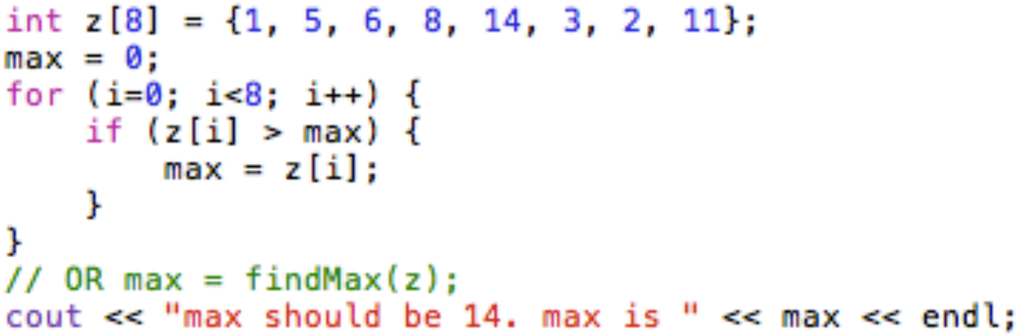
Back Up a Bit … The Big Picture of Testing

Type of Testing	What	Who
Unit	**Class, Routine, or Small Program**	**Single Programmer or Team (You!)**
Component / Module	**Class, Package, Small Program**	**Team (You and Your Team)**
Component Integration	Combined Execution of Classes, Packages, Components, or Subsystems	Team (Maybe You, Maybe Not)
System	On-Site, Installed System (Functional and Non-Functional)	Not You, probably QA professionals.
“Use”	System from User’s Perspective	QA Professionals.
Regression	Everything previously tested.	Anyone who uses your code. (You and everyone else.)

Some Tests

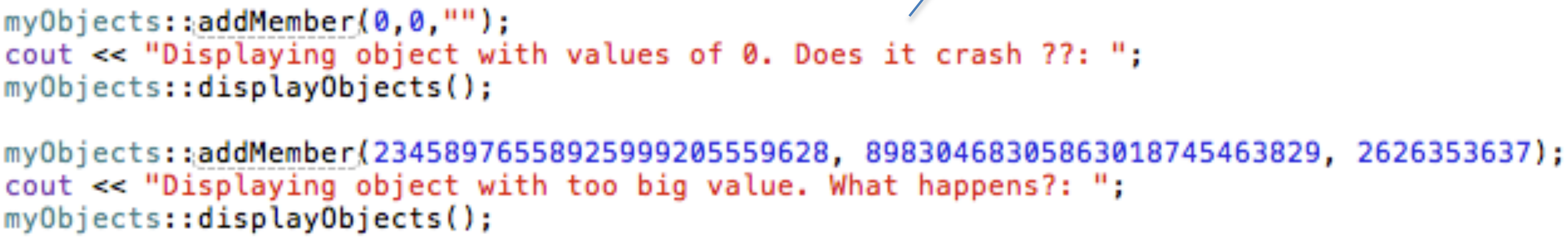


```
int y,z;
double x = 23.5;
y = div2(x); // Expecting a truncation of 23.5/2
cout << "div2(23.5) = " << y;
```



```
int z[8] = {1, 5, 6, 8, 14, 3, 2, 11};
max = 0;
for (i=0; i<8; i++) {
    if (z[i] > max) {
        max = z[i];
    }
}
// OR max = findMax(z);
cout << "max should be 14. max is " << max << endl;
```

If we don’t know what to expect, maybe it really isn’t a test .?.



```
myObjects::addMember(0,0,"");
cout << "Displaying object with values of 0. Does it crash ??: ";
myObjects::displayObjects();

myObjects::addMember(23458976558925999205559628, 89830468305863018745463829, 2626353637);
cout << "Displaying object with too big value. What happens?: ";
myObjects::displayObjects();
```

What Is A Test *During* Construction?

A Test is a comparison of
the results that you expect to get against the actual results.

Expected Results == Actual Results → Test Succeeded

Expected Results != Actual Results → Test Failed

Expectations are based on the requirements,
as YOU understand them.

What else ...

- A test “succeeds” when it “fails” (i.e. it breaks your code).
- Testing can never prove your code is without errors.
- Testing does not improve the quality of your software (although it might demonstrate the presence or absence of it).
- You must want and hope to find errors in your code through testing (if you don’t, somebody else will)!

“Yeah, I had a great day. I found 20 errors in my code!
Isn’t that fantastic!!! I hope I find some more tomorrow.”

What to Test

What do you mean I can’t find my errors?
I can do it …

- I’m going to test every possible input to my code.
- I’m going to test every path through my code.
- I’m going to test every line of code.

Black and White Box Testing

“Black-box” Testing

I put in X, and I get back Z. I don’t care how.

- Tests in which the inner workings of the method are not taken into consideration.
- Tests are based on the requirements.

“White-box” Testing

Is my logic correct? Will control go where I think?

- Tests developed with an awareness of the inner working of the code.
- Tests that consider code and/or path coverage.

Different Ways to Think About Testing

Equivalence Partitioning: Find representative test cases for equivalent data values (i.e. same code coverage).

Boundary Analysis: Test around boundaries of conditions and input values.

Bad Data: Unacceptable values or input.

Good Data: Acceptable but extreme (similar to boundary).

More Testing ...

- Static (Path) Analysis (*does not execute code*).
  - Finds errors like unitialized variables, unreachable code.
  - Compilers do this to find warnings (you can change the warning level).
  - Sometimes referred to as "lint" tools.

- Structured Basis Testing : Test every line of code.

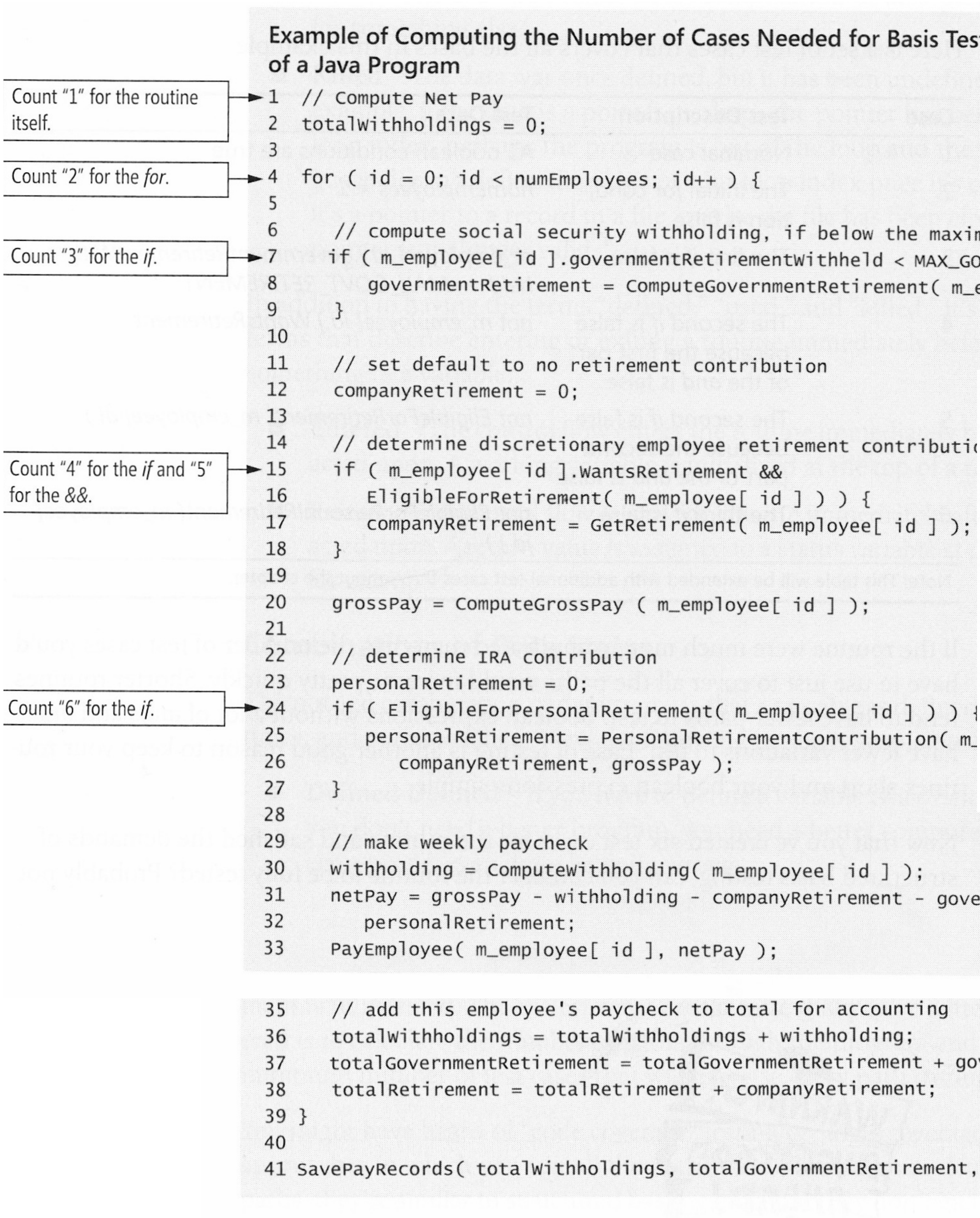
How many test cases will you need, at a **minimum**?

1. Start with 1 for the straight path through code.
2. Add 1 for each keyword { *if, while, repeat, for, and, or* }.
3. Add 1 for each *case* in a *switch* statement

Structured Basis Testing

Test Every Line

McConnel p. 507-508



Example of Computing the Number of Cases Needed for Basis Testing of a Java Program

Count "1" for the routine itself.

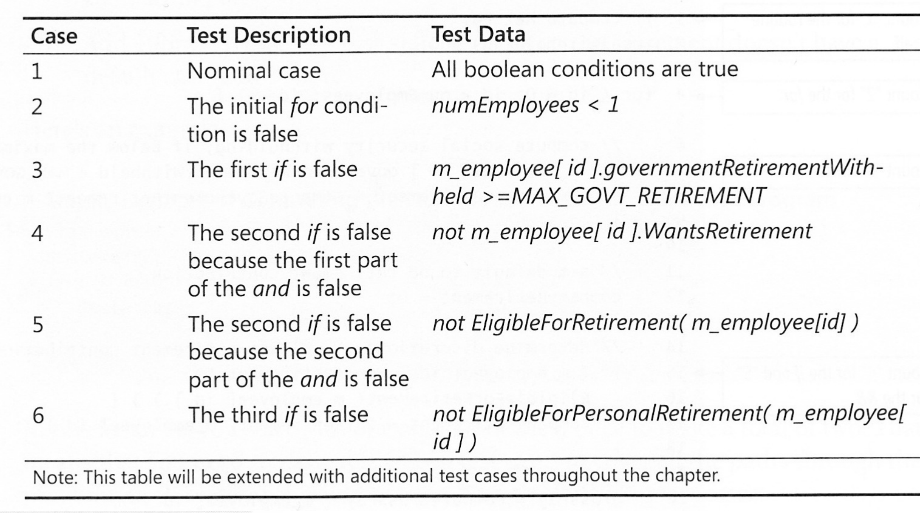
Count "2" for the *for*.

Count "3" for the *if*.

Count "4" for the *if* and "5" for the *&&*.

Count "6" for the *if*.

```
// Compute Net Pay
totalWithholdings = 0;

for ( id = 0; id < numEmployees; id++ ) {

   // compute social security withholding, if below the maximum
  if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT ) {
     governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
  }

   // set default to no retirement contribution
   companyRetirement = 0;

   // determine discretionary employee retirement contributi
   if ( m_employee[ id ].WantsRetirement &&
      EligibleForRetirement( m_employee[ id ] ) ) {
      companyRetirement = GetRetirement( m_employee[ id ] );
   }

   grossPay = ComputeGrossPay ( m_employee[ id ] );

   // determine IRA contribution
   personalRetirement = 0;
   if ( EligibleForPersonalRetirement( m_employee[ id ] ) ) {
      personalRetirement = PersonalRetirementContribution( m_
         companyRetirement, grossPay );
   }

   // make weekly paycheck
   withholding = ComputeWithholding( m_employee[ id ] );
   netPay = grossPay - withholding - companyRetirement - governmentRetirement -
      personalRetirement;
   PayEmployee( m_employee[ id ], netPay );

   // add this employee's paycheck to total for accounting
   totalWithholdings = totalWithholdings + withholding;
   totalGovernmentRetirement = totalGovernmentRetirement + governmentRetirement;
   totalRetirement = totalRetirement + companyRetirement;
}

SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```



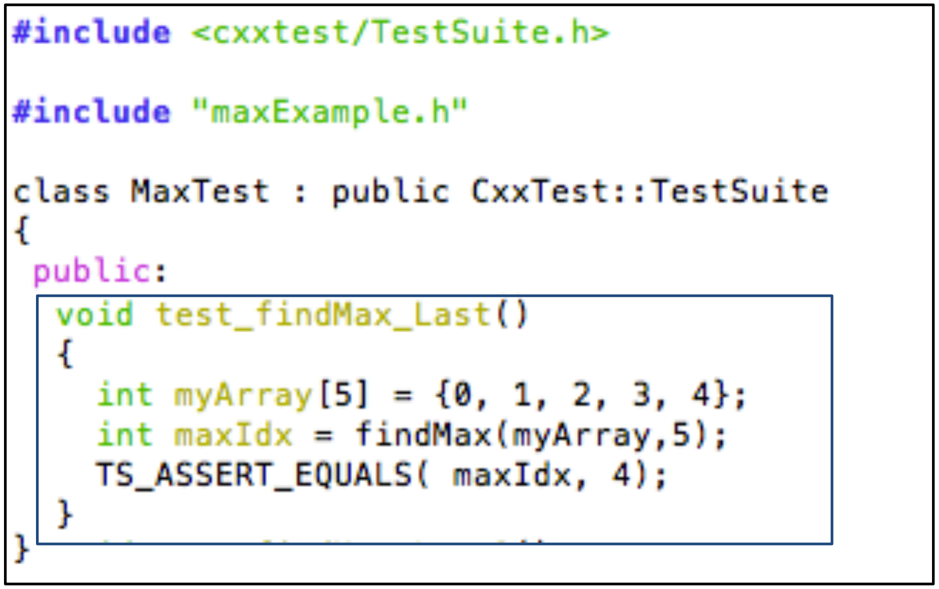
Case	Test Description	Test Data
1	Nominal case	All boolean conditions are true
2	The initial *for* condition is false	*numEmployees < 1*
3	The first *if* is false	*m_employee[id].governmentRetirementWithheld >=MAX_GOVT_RETIREMENT*
4	The second *if* is false because the first part of the *and* is false	*not m_employee[id].WantsRetirement*
5	The second *if* is false because the second part of the *and* is false	*not EligibleForRetirement(m_employee[id])*
6	The third *if* is false	*not EligibleForPersonalRetirement(m_employee[id])*

Note: This table will be extended with additional test cases throughout the chapter.

Testing Frameworks

It provides a process and framework for writing tests.

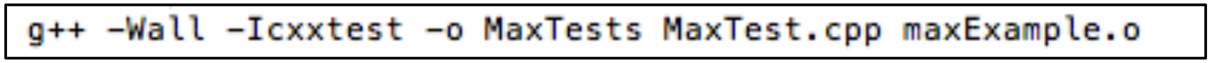
- Part of the programming language (JUnit, cxxTest, CPPUnit,Google Test)
- Tests defined using special assert statements.



```
#include <cxxtest/TestSuite.h>

#include "maxExample.h"

class MaxTest : public CxxTest::TestSuite
{
 public:
  void test_findMax_Last()
  {
    int myArray[5] = {0, 1, 2, 3, 4};
    int maxIdx = findMax(myArray,5);
    TS_ASSERT_EQUALS( maxIdx, 4);
  }
}
```

- Test suites (collection of single tests) defined in a file external to code.

- Tests are compiled with the code, but run separately.



```
g++ -Wall -Icxxtest -o MaxTests MaxTest.cpp maxExample.o
```

Testing Frameworks

It provides a process and framework for writing tests.

WHY use one:

- A universal language for testing.
- Easy to generate tests.
- Tests and debugging statements do not clutter your code.
- Tests travel with code, providing easy regression tests.
- Tests are a form of documentation.

What to Test with the Framework ?

Let Testing Strategies guide your test writing.

- –Structured Basis Testing (test every line)
- –Equivalence Partitioning (test general categories)
- –Boundary Analysis (test values at ”boundary”)
- –Bad Data
- –Good Data