



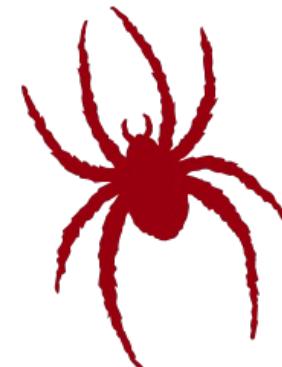
UNIVERSITY OF
RICHMOND

CMSC 240 Lecture 21

CMSC 240 Software Systems Development
Fall 2023

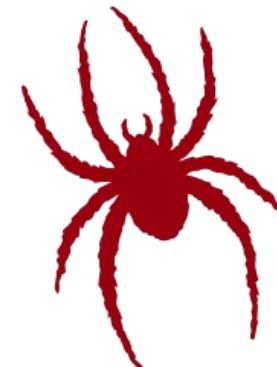
Today – Build Pipeline

- Generating Documentation
- Static Analysis
- Unit Testing



Today – Build Pipeline

- Generating Documentation
- Static Analysis
- Unit Testing



Generating Documentation From Code

- **Consistency & Accuracy**
 - Keeps documentation synchronized with code changes
 - Reflects the true state of the system
- **Efficiency & Time-Saving**
 - Reduces manual documentation effort
 - Frees up developer time for core tasks
- **Maintainability**
 - Simplifies updates as code evolves
 - Facilitates knowledge transfer and onboarding
- **Standardization**
 - Enforces uniform documentation practices
 - Enhances code readability and team collaboration

Doxygen: Automated Documents for C++

- What is Doxygen?
 - Tool for generating reference documentation from source code comments
- Key Features
 - Supports multiple programming languages, including C++
 - Generates documentation in HTML, LaTeX, RTF, and XML formats
- Benefits
 - Streamlines the documentation process
 - Ensures documentation consistency with the codebase
- Integration
 - Easily integrates with coding environments and version control systems
 - Supports collaboration by providing up-to-date code documentation

Doxygen: Automated Documents for C++

- **Doxygen Overview**
 - A documentation generator for writing software reference documentation from annotated source code
- **Key Annotations**
 - `@file`: Describes the name and a brief description of the file
 - `@class`: Documents a class and provides a brief class description
 - `@brief`: A concise description of the following element
 - `@param`: Documents one parameter of a function
 - `@return`: Describes what a function returns
 - `@throw` or `@exception`: Describes what exceptions are thrown by a function
- **How Do They Work?**
 - Doxygen scans the source code, parsing the annotations to generate the corresponding documentation sections

Doxxygen: Example

```
/**  
 * @class SimpleMath  
 * @brief A class that offers basic mathematical functions.  
 *  
 * This class can perform simple mathematical operations such as  
 * addition, subtraction, multiplication, and division.  
 */  
class SimpleMath  
{  
public:  
    /**  
     * @brief Adds two numbers.  
     * @param a First number to add.  
     * @param b Second number to add.  
     * @return The sum of a and b.  
     */  
    int add(int a, int b);  
  
    /**  
     * @brief Subtracts one number from another.  
     * @param a Number to be subtracted from.  
     * @param b Number that is to subtract.  
     * @return The difference of a and b.  
     */  
    int subtract(int a, int b);
```



Doxygen: Configuration File

```
☰ doxyfile U X
```

```
lecture21 > doxy > ⌂ doxyfile
```

```
1 PROJECT_NAME = "SimpleMath"
2 INPUT = ./
3 RECURSIVE = YES
4 OUTPUT_DIRECTORY = ./docs
5 GENERATE_HTML = YES
6 GENERATE_LATEX = NO
7
```

Doxygen: Generating Docs

```
$ doxygen doxyfile
```

Add Document Generation to the Build Pipeline

```
M Makefile U X
lecture21 > docgen > M Makefile
1 all: main docs
2
3 main: main.o SimpleMath.o
4     g++ main.o SimpleMath.o -o main
5
6 main.o: main.cpp SimpleMath.h
7     g++ main.cpp -c
8
9 SimpleMath.o: SimpleMath.cpp SimpleMath.h
10    g++ SimpleMath.cpp -c
11
12 docs: main.cpp SimpleMath.cpp SimpleMath.h
13     doxygen doxyfile
14
15 clean-code:
16     rm -f main.o SimpleMath.o main
17
18 clean-docs:
19     rm -r -f ./docs
20
21 clean: clean-code clean-docs
```

SimpleMath Class Reference

A class that offers basic mathematical functions. [More...](#)

```
#include <SimpleMath.h>
```

Public Member Functions

int add (int a, int b)

Adds two numbers. [More...](#)

int subtract (int a, int b)

Subtracts one number from another. [More...](#)

int multiply (int a, int b)

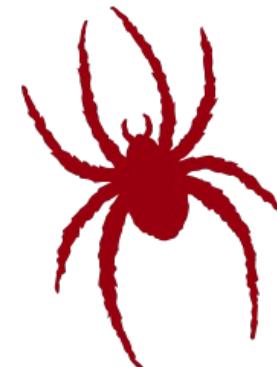
Multiplies two numbers. [More...](#)

double divide (int a, int b)

Divides one number by another. [More...](#)

Today – Build Pipeline

- Generating Documentation
- Static Analysis
- Unit Testing



Introduction to Static Code Analysis

- **What is Static Code Analysis?**
 - A method of debugging by examining code without executing it
- **Purpose of Static Code Analysis**
 - To detect code quality issues, security vulnerabilities, and coding standard violations early in development
- **Key Benefits**
 - Improves code quality and maintainability
 - Identifies potential security risks
 - Saves time and resources by catching issues before runtime
- **How It Works**
 - Uses tools to analyze the source code for patterns of known issues
 - Can be integrated into IDEs and continuous integration pipelines

Introduction to Static Code Analysis

- What is CPPCheck?
 - An open-source static analysis tool for C and C++ code
 - Designed to detect various kinds of bugs in your code
- Key Features
 - Checks for memory leaks, mismatching allocation-deallocation, and more
 - Detects undefined behavior and dangerous coding constructs
- Using CPPCheck
 - Run it from the command line: `cppcheck [options] [file(s)]`
 - Incorporate it into your build pipeline for regular analysis

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

void printVector(vector<int>& v)
{
    for (size_t i = 0; i <= v.size(); ++i)
    {
        cout << v[i] << endl; // Potential out-of-bounds access
    }
}

int main()
{
    char* p = new char[10];

    vector<int> numbers = {1, 2, 3, 4, 5};
    printVector(numbers);

    delete p; // Should be 'delete[] p;' to match 'new[]'
    return 0;
}
```

Defects Not Found During Compile or Run

```
$ g++ -Wall main.cpp -o main
$ ./main
1
2
3
4
5
0
```

Run Static Analysis With `cppcheck`

```
$ cppcheck *.cpp
Checking main.cpp ...
main.cpp:20:12: error: Mismatching allocation and deallocation: p [mismatchAllocDealloc]
    delete p; // Should be 'delete[] p;' to match 'new[]'
               ^
main.cpp:15:15: note: Mismatching allocation and deallocation: p
    char* p = new char[10];
               ^
main.cpp:20:12: note: Mismatching allocation and deallocation: p
    delete p; // Should be 'delete[] p;' to match 'new[]'
               ^
main.cpp:9:18: error: When i==v.size(), v[i] is out of bounds. [stlOutOfBounds]
    cout << v[i] << endl; // Potential out-of-bounds access
               ^
```

Add Static Analysis to the Build Pipeline

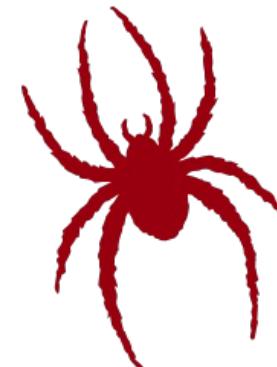
M Makefile U X

lecture21 > static > M Makefile

```
1 all: main static-analysis
2
3 main: main.o
4     g++ main.o -o main
5
6 main.o: main.cpp
7     g++ -Wall main.cpp -c
8
9 static-analysis:
10    cppcheck *.cpp
11
12 clean:
13    rm -f main.o main
14
```

Today – Build Pipeline

- Generating Documentation
- Static Analysis
- Unit Testing



The Two Approaches to Programming

- Approach #1
 - “I wrote ALL of the code, but when I tried to compile and run it, nothing seemed to work!”
- Approach #2
 - Write a little code (e.g., a method or small class)
 - Test it
 - Write a little more code
 - Test it
 - Integrate the two verified pieces of code
 - Test it
 - ...

Introduction to Unit Testing

- **What is Unit Testing?**
 - Unit testing is a software testing method where individual units of source code are tested to determine if they are fit for use
- **Key Characteristics**
 - Isolates the smallest parts of a program, (i.e. functions or methods), for testing
 - Usually automated to run as part of the development process
- **Objective**
 - To ensure that each unit operates correctly
- **Importance in Software Development**
 - Catches bugs early in the development cycle
 - Helps maintain and refactor code with confidence
 - Vital for ensuring the reliability and quality of the final product

Types of Software Testing

- Unit Testing
 - Testing individual components or functions
- Integration Testing
 - Testing combined components to determine if they function together
- System Testing
 - Testing a complete and integrated software system

Unit Testing Process

1. **Identify Units:** Determine the smallest testable components of the software to be tested
2. **Write Test Cases:** Create test cases that cover various scenarios and edge cases for each unit
3. **Execute Tests:** Run the test cases and verify the actual output against the expected output
4. **Analyze Results:** Identify failures, debug issues, and fix the failing units
5. **Repeat and Automate:** Continuously write and execute unit tests as part of the development pipeline

Code Coverage

- **Code coverage** is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs
- Types of Coverage:
 - **Statement Coverage:** each statement in the code is run at least once
 - **Branch Coverage:** every branch from each decision point is executed
 - **Path Coverage:** all the paths of execution are taken within each function
 - **Condition Coverage:** all Boolean expressions evaluated both to true and false
- Best Practices: Strive for high coverage percentage

1. Write down all the **inputs** that you would provide to completely test this function.

2. Write the corresponding **expected outputs**.

```
#include <stdexcept>

bool isLeapYear(int year)
{
    if (year <= 0)
    {
        throw std::invalid_argument("Year must be greater than 0.");
    }

    bool leapYear = false;

    if (year % 4 == 0)
    {
        if (year % 100 != 0)
        {
            leapYear = true;
        }
        else if (year % 400 == 0)
        {
            leapYear = true;
        }
    }

    return leapYear;
}
```

Example: `isLeapYear()` function

Input	Expected Output
<code>isLeapYear(1996)</code>	true
<code>isLeapYear(2000)</code>	true
<code>isLeapYear(1900)</code>	false
<code>isLeapYear(2019)</code>	false
<code>isLeapYear(0)</code>	invalid_argument
<code>isLeapYear(-100)</code>	invalid_argument

Unit Testing With doctest

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>
#include "leap.h"

TEST_CASE("Testing isLeapYear function")
{
    CHECK(isLeapYear(2000) == true); // Divisible by 400
    CHECK(isLeapYear(1996) == true); // Divisible by 4 but not by 100
    CHECK(isLeapYear(1900) == false); // Divisible by 100 but not by 400
    CHECK(isLeapYear(2019) == false); // Not divisible by 4

    CHECK_THROWS_AS(isLeapYear(0), std::invalid_argument); // Invalid argument 0
    CHECK_THROWS_AS(isLeapYear(-100), std::invalid_argument); // Invalid argument less than 0
}
```

Unit Testing With doctest

```
=====
leapTest.cpp:5:
TEST CASE: Testing isLeapYear function

leapTest.cpp:7: SUCCESS: CHECK( isLeapYear(2000) == true ) is correct!
    values: CHECK( true == true )

leapTest.cpp:8: SUCCESS: CHECK( isLeapYear(1996) == true ) is correct!
    values: CHECK( true == true )

leapTest.cpp:9: SUCCESS: CHECK( isLeapYear(1900) == false ) is correct!
    values: CHECK( false == false )

leapTest.cpp:10: SUCCESS: CHECK( isLeapYear(2019) == false ) is correct!
    values: CHECK( false == false )

leapTest.cpp:12: SUCCESS: CHECK_THROWS_AS( isLeapYear(0), std::invalid_argument )
threw as expected!"Year must be greater than 0."

leapTest.cpp:13: SUCCESS: CHECK_THROWS_AS( isLeapYear(-100), std::invalid_argument
) threw as expected!"Year must be greater than 0.

=====
[doctest] test cases: 1 | 1 passed | 0 failed | 0 skipped
[doctest] assertions: 6 | 6 passed | 0 failed |
[doctest] Status: SUCCESS!
```

```
#include <stdexcept>

bool isLeapYear(int year)
{
    if (year <= 0)
    {
        throw std::invalid_argument("Year must be greater than 0.");
    }

    bool leapYear = false;

    if (year % 4 == 0)
    {
        if (year % 100 != 0)
        {
            leapYear = true;
        }
        else if (year % 400 == 0)
        {
            leapYear = false;
        }
    }

    return leapYear;
}
```

Unit Testing With doctest

```
=====
leapTest.cpp:5:
TEST CASE: Testing isLeapYear function

leapTest.cpp:7: ERROR: CHECK( isLeapYear(2000) == true ) is NOT correct!
    values: CHECK( false == true )

leapTest.cpp:8: SUCCESS: CHECK( isLeapYear(1996) == true ) is correct!
    values: CHECK( true == true )

leapTest.cpp:9: SUCCESS: CHECK( isLeapYear(1900) == false ) is correct!
    values: CHECK( false == false )

leapTest.cpp:10: SUCCESS: CHECK( isLeapYear(2019) == false ) is correct!
    values: CHECK( false == false )

leapTest.cpp:12: SUCCESS: CHECK_THROWS_AS( isLeapYear(0), std::invalid_argument )
threw as expected!"Year must be greater than 0."

leapTest.cpp:13: SUCCESS: CHECK_THROWS_AS( isLeapYear(-100), std::invalid_argument
) threw as expected!"Year must be greater than 0."
```

```
=====
[doctest] test cases: 1 | 0 passed | 1 failed | 0 skipped
[doctest] assertions: 6 | 5 passed | 1 failed |
[doctest] Status: FAILURE!
```

Add Unit Testing to the Build Pipeline

```
M Makefile U X
lecture21 > unit > leap > M Makefile
1 all: main run-unit-tests
2
3 main: main.o leap.o
4     g++ main.o leap.o -o main
5
6 main.o: main.cpp leap.h
7     g++ -Wall main.cpp -c
8
9 leap.o: leap.cpp leap.h
10    g++ -Wall leap.cpp -c
11
12 leapTest: leapTest.cpp leap.cpp leap.h
13     g++ leapTest.cpp leap.o -o leapTest
14
15 run-unit-tests: leapTest
16     ./leapTest
17
18 clean:
19     rm -f leap.o main.o main leapTest
20
```

Test-Driven Development (TDD)

- What is TDD?
 - Test-Driven Development is a software development approach where tests are written before the code that is to be tested
- Red → Green → Refactor
 - **Red:** Write a failing test
 - **Green:** Write the minimal amount of code to make the test pass
 - **Refactor:** Clean up the code while keeping the tests green
- Benefits: More maintainable code, encourages better design

Write This FIRST!

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>
#include "leap.h"

TEST_CASE("Testing isLeapYear function")
{
    CHECK(isLeapYear(2000) == true); // Divisible by 400
    CHECK(isLeapYear(1996) == true); // Divisible by 4 but not by 100
    CHECK(isLeapYear(1900) == false); // Divisible by 100 but not by 400
    CHECK(isLeapYear(2019) == false); // Not divisible by 4

    CHECK_THROWS_AS(isLeapYear(0), std::invalid_argument); // Invalid argument 0
    CHECK_THROWS_AS(isLeapYear(-100), std::invalid_argument); // Invalid argument less than 0
}
```