* **Introduction to Testing**

**The Origin of this Course**

This course was developed due to the observation that students lacked proficiency in testing and debugging, skills that are increasingly demanded by the industry. The course aims to achieve several goals:

1. Distinguishing between testing and debugging.
2. Teaching proper software testing methodologies.
3. Providing students with various debugging techniques and tools.
4. Introducing students to industry-standard testing practices and tools.

The fourth goal emerged from discussions with a test lead in the industry, who highlighted the gap in new graduates' knowledge regarding software testing processes and tools. Many newcomers begin their careers in Quality Assurance (QA) roles, which underscores the importance of addressing this knowledge gap. Consequently, the course's second half focuses on using tools to manage small projects, emphasizing testing and debugging.

By the course's end, students will have acquired the skills necessary to effectively test and debug code, making them well-prepared to join industry teams that incorporate QA in their software development life cycle.

**Introduction to Testing**

Software development is a highly complex endeavor, and modern software systems push the limits of human cognitive abilities. Ensuring that software functions correctly involves considering various factors:

1. Does the software produce accurate results?
2. Can the software handle edge cases, such as no data inputs?
3. Does the software meet performance requirements?
4. Will the software fail under heavy loads?
5. How does the software handle erroneous input?
6. Are there any hidden interactions within the system that could cause errors?
7. Can the software recover from unanticipated errors and return to a stable state?

Software testing is the process of evaluating software to verify its functionality and confirm that it meets performance and functional requirements. In the early days of software development, testing received little attention. However, as software grew more complex and users demanded higher quality, testing became a critical component of the software development life cycle.

**Debugging**

Testing is just one part of the software development process. When issues are identified during testing, the next step is debugging. Debugging is a multifaceted process that involves:

1. Locating the source of a bug.
2. Determining how to fix the bug.
3. Implementing the solution.

Locating the source of a bug is often the most time-consuming step, resembling detective work. Developers gather evidence and look for clues to identify the bug's cause. Debugging often requires inserting print statements or using debugging tools to gain additional insights, making it akin to solving a puzzle.

**What Does Testing Prove?**

Testing proves that the executed tests are functioning correctly; however, it does not guarantee that the software is entirely bug-free. Even if all tests pass, the software may still harbor undetected bugs. The number of discovered bugs is directly related to the effort invested in debugging. Initially, numerous bugs are identified, but as testing progresses, fewer are found. Eventually, the returns diminish, and further testing becomes impractical.

It is crucial to recognize that passing tests do not equate to a perfectly correct software. Achieving absolute correctness is almost impossible for complex software. Despite efforts to prove software's correctness, this remains a challenge, especially for larger programs.

Additionally, the belief that frequently used software must be bug-free is a fallacy. Most users do not explore all features or push software to its limits daily. Many applications are released with known bugs, deemed unworthy of fixing because they are unlikely to affect users or have workarounds.

**Testing Versus Debugging**

Testing and debugging are distinct processes. Testing involves running the software to verify that it functions as expected and produces the correct results. In contrast, debugging is the process of identifying and rectifying issues in code known to contain bugs. While these processes often occur sequentially, they should not be confused.

In larger organizations, quality assurance teams typically conduct testing, and developers address identified bugs. In smaller companies, developers might perform both testing and debugging. Regardless of the roles, it is vital to understand that testing and debugging are separate tasks in the software development lifecycle.

* **The Importance of Testing**

This is just a small selection of famous software bugs. You are probably aware of many more bugs that have far less devastating consequences. WHile we are getting better at producing high quality software, we are far away from perfection.

What we should learn from this is that it is easy to get bugs in software and it is very hard to discover and fix them. We also see that many bugs occur not for a single reason but due to many things happening at the same time. These situations make it even more difficult to detect a bug in advance unless you have the ability to anticipate everything that is going to happen in the real world. Software is usually written by developers sitting in rooms filled with computers. They are asked to anticipate everything that can happen in the real world and to program for it. However, they cannot anticipate everything which can happen in the real world.

* **Types of Testing**

There is not one single type of software testing, but rather, different types that test the software in different ways. The following sections will investigate each of these different types of testing and show how it contributes to complete testing of your code. This is not an exhaustive list and you might find other types of tests listed elsewhere. This does, however, capture most of the important testing types.

## Unit Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#unit-testing)

Unit testing is one of the most basic types of testing. It breaks the code into small units, usually functions, and tests them to determine if they produce the correct results. The idea is to test small units of code and make sure that they work perfectly before they are combined with other pieces of code into much larger programs. It is easier to find a bug in a small piece of code than it is to find it in larger program. Further, we like to build on code which is known to be solid and bug free. If we know that the individual units that we use in a program are free from bugs, we do not have to look at the low level code to check for bugs but rather look at the way it has been integrated to find the bugs.

Unit tests normally test functions by passing them a known piece of data and checking to see that the result is what is expected. In most cases, this is done by automated tools that run a series of tests and report the results. Later in the course, we will be looking at tools to aid in the running of unit tests.

## Black Box Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#black-box-testing)

Black box testing is a way of designing unit tests. It treats every functional unit as a black box. This means that we have no idea what is inside the black box but we know that if we put a certain value into the box then a certain value should come out of the box. Because it is a black box, we have no idea what is happening inside the box and therefore cannot structure our tests to take advantage of this. Black box testing relies entirely on knowing that a certain input should produce a certain output. Black box testing is normally the first approach to designing unit tests.

## White Box Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#white-box-testing)

White box testing is almost the opposite of black box testing. Whereas with black box testing we know nothing about the internal structure of the code, with white box testing we take advantage of the internal structure of a code to design tests that test every path through the code. If we simply rely on black box testing there, is no guarantee that we would have tested every single path through the code. There could be rarely used pieces of code which have never been tested by black box testing and therefore we have no proof that they work correctly. White box testing produces additional tests which test every path through the code. This results in much higher confidence that the code works correctly.

## Integration Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#integration-testing)

While unit testing tests the low level functionality, we also need to test to make sure that the individual units were combined correctly and that they work as a whole. This is the job of integration testing which does not look at the individual functions but looks at groups of functions to ensure that they work together correctly. A unit testing framework can be used to carry out integration testing or it can be carried out in other ways.

## Functional Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#functional-testing)

Functional testing is similar to integration testing but focuses on the business requirements of the application. Functional testing verifies that the output is correct without checking the internal state of the system. An integration test might check that you accessed the database, updated some information, and committed your change. The functional test for the same thing might simply check to see that the correct value had been stored in the database without checking to see exactly how it was done.

## End to End Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#end-to-end-testing)

End to end tests replicate the user behavior and make sure that all the tasks the user would perform work as expected. In some cases, these can be difficult to automate. They can also be difficult to maintain as scenarios of how the user employs the software change over time.

## Load Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#load-testing)

Load testing checks that the software functions correctly under high loads. For a web application it might test to see if it can handle 100 simultaneous connections. A large data application might want to ensure that the database can handle 1000 simultaneous queries and give the required response time. Load testing is often combined with stress testing which tries to take it beyond the normal operating requirements to see when it will actually break. If stress testing reveals that your software will not break until well beyond your expected usage of it, then you are assured your software is in good shape.

## Security Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#security-testing)

Security testing determines if the software meets the security requirements. It might involve checking that every web page can only be reached after signing on to the website. It might involve checking the level of user access to make sure that users can only access the information they are allowed to. It could also involve attempts to break into the system and either steal or alter information stored within it.

## Acceptance Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#acceptance-testing)

Acceptance testing verifies that the application meets all the business requirements. The entire finished application is tested, usually by running examples of how the user would actually use the system. If all of the acceptance tests are met, then it indicates that the customer requirements have been met.

## Regression Testing[​](https://software-testing.sdds.ca/A-Introduction/testing-types-3#regression-testing)

Regression testing means we perform our existing tests after every change to the code. The goal of regression testing is to make sure that any changes we made to the code do not break the existing code. Regression testing is used in many software development methodologies where, after a change is made to the code, the regression tests are run to ensure that it is bug free before being checked into the repository. The goal of regression testing is to make sure that the code checked into the shared repository is free of bugs. This prevents other developers building upon code which has bugs in it.

## Test Data Selection

You can test hard or you can test smart. Testing hard is known as exhaustive testing. Consider the case of testing the addition of two integers. To be sure that every case works, we need to try every combination of numbers. If we are working with 32 bit integers, each integer can have 4,294,967,296 values. Combining these in all possible ways will require 18,446,744,073,709,551,616 tests. This is exhaustive testing.

The problem with exhaustive testing is the vast majority of these tests are just duplicates of other tests. Smart testing says we should select the test data more carefully to eliminate the duplication and reduce the number of tests down to a manageable number.

This type of test data selection is for black box testing. White box testing will add onto the black box test values to ensure that every path through the software has been properly tested. This will add test values that drive the code through parts of the code that would not be executed by the black box testing data.

## Black Box Data Selection[​](https://software-testing.sdds.ca/B-DataSelection2/data-selection-1#black-box-data-selection)

Black box test data should pick values that are not duplicates of other tests and test the areas where errors commonly occur. Testing 3 + 3 differs little from 7 + 7 since they are both the same values and they are both odd values. To test different values we would try:

* adding two identical even values (eg. 12 + 12),
* adding an even and an odd value (eg. 14 + 19),
* adding an odd and an even value (eg. 23 + 12),
* adding two identical odd values (eg. 23 + 23).

This tests adding routine values but does nothing to test the special values. Special values are those where errors frequently occur such as:

* transition points from one area of data to another (moving from positive to negative values)
* end points of the data values (the maximum and minimum integer values),
* special values (0 being a special value for addition),
* values immediately adjacent to other special values.

For our example of adding two numbers we could add:

* adding 0 to a number (12 + 0, -13 + 0)
* adding 1 to a number (12 + 1, -13 + 1)
* adding -1 to a number (12 + -1, -13 + -1)
* adding MAXINT to a number (12 + MAXINT, -13 + MAXINT)
* adding MININT to a number (12 + MININT, -13 + MININT)
* adding MAXINT-1 to a number (12 + (MAXINT-1), -13 + (MAXINT-1))
* adding MININT+1 to a number (12 + (MININT+1), -13 + (MININT+1))

If we add these all up, we now have a total of 18 different tests. If the program can add all of these correctly, we have a lot of confidence that the software works as expected. This is a huge reduction in the number of tests compared to exhaustive testing but provides the same level of confidence that the software works.

Now, let's consider the case of concatenating two strings. We have written our own concatenation function and want to test it to make sure it works before we use it in our programs. Take a look at the program:

void concat(char dest[], const char src[])  
 {  
 int dp = 0, sp = 0;  
 if (dest != NULL)  
 {  
 for (dp = 0; dest[dp] != '\0'; dp++);  
 if (src != NULL)  
 {  
 for (sp = 0; src[sp] != '\0'; sp++)  
 {  
 dest[dp++] = src[sp];  
 }  
 dest[dp] = '\0';  
 }  
 }  
 }

This code is designed to concatenate the src string onto the end of the dest string and work in every situation. To test this we need to:

* check the edge conditions for empty strings
  + concat("", ""),
  + concat("", "a"),
  + concat("", "abc"),
  + concat ("a", ""),
  + concat("abc", "")
* concatenate routine values with a single character in one string
  + concat("a", "a")
  + concat("a", "abc")
  + concat("abc", "x")
* routine longer strings that do not exceed the limit of the allocated array size of 16
  + concat("abcdefg", "hijk"),
  + concat("abcdefg", "hijklmn")
* a string that is the longest possible
  + concat("012345678901234", "x")
* a string that is longer than the allocated storage to prove it will fail
  + concat("012345678901234", "xy")

Those tests cover all of the black box cases. Reading the code, we also see that it can handle either of the parameters being NULL. These are special cases that need to be tested. If the handling of NULL had not been in the specification for the funtion, we would only find it by inspecting the code and it would be a white box test. To do this test, we add:

* {concat(NULL, "a"),
* concat(tmp, NULL) }.

In the actual tests I created, I ended up with 15 tests for string concatenation. This gives high confidence the function works and is much shorter than exhaustive testing.

## White Box Testing[​](https://software-testing.sdds.ca/B-DataSelection2/data-selection-1#white-box-testing)

The use of the NULL values in the example above is an example of white box testing. White box tests are those that are not stated in the requirements of the software but are found by reading the code. Often, programmers write code to cover unusual situations that were not placed in the requirements. In the case above, the programmer handled the cases of NULL being passed but this was likely not in the original requirements. Test cases for these situations can only be found by reading the code.

White box testing describes tests which are added to the test suite after examining the code. They are called white box because the code is no longer treated as a black box but needs to be opened up to see how it works inside. In the next example, we will also see some test cases that could be missed unless the code was examined carefully.

#define \_CRT\_SECURE\_NO\_WARNINGS  
#include <stdio.h>  
#include <string.h>  
  
#define MAX\_WORD\_LEN 20  
  
  
  
int split(char line[], char words[][MAX\_WORD\_LEN + 1], int maxWords)  
{  
 char ch;  
 char buf[MAX\_WORD\_LEN + 1] = { 0 };  
 int lp = 0, bp = 0, wp = 0, result = 0;  
  
 ch = line[lp];  
 while( ch != '\0' && result >= 0)  
 {  
 while (ch != '\0' && (ch == ' ' || ch == '\t')) ch = line[++lp];  
  
 bp = 0;  
 while (ch != '\0' && !(ch == ' ' || ch == '\t'))  
 {  
 buf[bp++] = ch;  
 ch = line[++lp];  
 }  
 if (bp > 0)  
 {  
 buf[bp] = '\0';  
 if (wp >= maxWords)  
 {  
 result = -1;  
 }  
 else  
 {  
 strcpy(words[wp++], buf);  
 }  
 }  
 }  
 result = (result < 0) ? result : wp;  
 return result;  
}  
  
int main(void)  
{  
 char line[] = { " The quick\t brown fox " };  
 char words[10][MAX\_WORD\_LEN + 1] = { 0 };  
 int nwords, i;  
  
 nwords = split(line, words, 10);  
  
 for (i = 0; i < nwords; i++)  
 {  
 printf("%s\n", words[i]);  
 }  
  
 return 0;  
}

This code splits a line of test into words and then returns the words as an array of strings. The function returns either a -1 if the maximum number of words has been exceeded or the number of words that were placed in the array.

The test string tests for:

* blanks at the begging and ending of the string,
* single blanks between words,
* multiple blanks between words,
* a combination of blanks and tabs between words.

This is not comprehensive testing but does provide some confidence that the function works correctly. When we read the code, we will find that:

* there is a situation where it does not store a word into the word list if the variable **bp** is zero. **bp** is the counter for the number of letters in the word and we do not save the word unless there are some letters in it. While this makes sense, it raises the question of how we can have a word with no letters in it. After giving this some thought, you realize this can only happen if the last letter in a word is immediately followed by the string terminator. This means, to test for this case, we need to include a test where there is no space or tab after the last word in the string.
* Examining the code further, we see that it checks if the number of words in the words array is less than the maximum size of the array before it adds the word into the array. To make sure this code works correctly, we need to add a test which will exceed the side of the array to make sure it handles this situation correctly.

White box testing adds new test cases that would often be missed if the code had not been examined. White box tests usually test the situations that rarely happen. Without these tests, we would have no idea whether this code works and might find that it fails in a production system when the rare condition actually happens.

## Unit Testing

Unit testing is the practice of testing code in small, isolated units, typically corresponding to functions or methods. The primary goal of unit testing is to verify that each unit of code behaves as expected. To accomplish this, unit tests provide specific inputs to the unit under test and compare the actual output to the expected output. If the actual and expected outputs match, the test passes; otherwise, it fails.

Unit testing can be done manually by writing code to invoke functions and validate their outputs. However, it's more common to use a unit test framework, which streamlines the testing process. A unit test framework requires test cases to be written in a specific format and automatically executes them, recording the results. These frameworks often provide an interface for displaying test results, making it easy to identify which tests have passed or failed.

In the previous chapter, we discussed the selection of test data for unit tests, emphasizing the importance of careful data selection. Once test data is chosen, it can be organized into a series of unit tests.

Unit tests are typically grouped into test suites, with each suite focusing on testing a specific function or method. Within a test suite, various tests are designed to cover different scenarios and inputs.

Assertions play a central role in unit testing. They are statements that compare the expected value with the actual value produced by running a function. When the comparison is successful, the test passes. Assertions often allow an associated error message to be displayed in case of a failure, providing information about where the test failed.

There are various unit testing frameworks available, but they all adhere to the fundamental principles outlined here. In the next section, we will explore the unit testing framework provided by Microsoft Visual Studio in more detail.

## Testing Coverage

Testing coverage is a crucial aspect of software testing that ensures all parts of your code are exercised, both the parts that have been executed and those that haven't. This practice helps identify untested code, which may contain hidden bugs. By analyzing code coverage, you can design tests to cover previously untested areas.

To perform testing coverage, you typically use compiler-specific flags or options to instruct the compiler to insert additional code for counting the number of times each line of your code is executed. Here, we'll demonstrate testing coverage using GCC (GNU Compiler Collection) since Visual Studio's coverage analysis tool is available only in the enterprise version of the tool.

To illustrate this, let's take an example program that inserts one string into another:

c

#include <stdio.h>

#include <string.h>

#define STR\_SIZE 20

void insert(char dest[], int posn, char src[])

{

int dlen = strlen(dest);

int slen = 0, i;

if (posn < dlen)

{

slen = strlen(src);

for (i = dlen; i >= posn; i--)

{

dest[i + slen] = dest[i];

}

for (i = 0; i < slen; i++)

{

dest[posn + i] = src[i];

}

}

else

{

strcat(dest, src);

}

}

int main(void)

{

char dest[STR\_SIZE] = {"The fox"};

insert(dest, 4, "red ");

printf("%s\n", dest);

return 0;

}

To enable code coverage analysis with GCC, you need to compile the code with specific flags:

sh

gcc -Wall -fprofile-arcs -ftest-coverage -o insert insert.c

These flags instruct the compiler to include code for tracking line execution counts.

After compiling, you must run the program once to generate data files containing execution information. These files are later used for coverage analysis. To prepare the data for human-readable inspection, you can use the gcov command:

sh

gcov insert.c

This command generates a file named insert.c.gcov. In this file, each line of the original source code is accompanied by execution count information. Lines marked with "-:" were not executed and were not part of the test coverage. Lines marked with hash signs (#) were executable but remained unexecuted. These unexecuted lines are the ones you should focus on when identifying untested code.

By analyzing coverage reports like this, you can pinpoint areas of your code that lack adequate testing. Once identified, you can design additional tests to improve coverage and increase confidence in the reliability of your software.

**Debugging**

Debugging is indeed a crucial part of the software development process. It involves identifying and fixing issues or bugs in your code to ensure that your program runs correctly. Here's a summary of the debugging techniques and common types of errors you mentioned:

**Debugging Techniques**:

1. **Print Statements**: Inserting print statements in your code is a common technique to debug issues. You print out variable values, messages, or markers to see where your program is and what values it's using at various points in the execution.
2. **Logging**: Creating log files can be helpful for capturing the output of multiple print statements. Log files allow you to track program behavior and variable values over time.
3. **Interactive Debuggers**: Debugging tools and IDEs often provide interactive debuggers that allow you to pause your program's execution, inspect variable values, step through code line by line, and set breakpoints to analyze its behavior.

**Types of Errors**:

1. **Syntactic Errors**: These occur during compilation when your code doesn't adhere to the rules of the programming language. Syntactic errors generate compiler warnings or errors. Warnings should not be ignored as they can indicate potential issues.
2. **Runtime Errors (Semantic Errors)**: These errors happen when your program compiles successfully but doesn't produce the expected results during execution. Semantic errors can be challenging to find and often require careful analysis of your code's logic.
3. **Warnings**: Compiler warnings are messages that indicate unusual or potentially problematic code constructs. Ignoring warnings can lead to future semantic errors, so it's essential to address them.
4. **Link Errors**: Linker errors occur when the linker cannot find a definition for a function or variable you've referenced in your code. These errors usually involve issues with linking object files or libraries.
5. **Segmentation Faults**: Segmentation faults (segfaults) occur when your program attempts to access memory it's not allowed to access, such as accessing an array element beyond its bounds or dereferencing a null or invalid pointer.
6. **Bus Errors**: Bus errors are similar to segmentation faults but often involve trying to access memory locations that don't exist or are not accessible. They can also result from dereferencing a pointer that points to invalid memory.
7. **Random Behavior**: Random program behavior can be caused by uninitialized variables. When your program uses variables without initializing them first, it may produce unpredictable results.
8. **Program Executes the Wrong Code**: If your program seems to be executing the wrong code, it might be due to pointer issues, bad memory access, or unexpected jumps in your code. Tracking down such issues can be challenging.
9. **Variable Values Change Unexpectedly**: If you notice that a variable's value changes unexpectedly, it might result from memory corruption, buffer overflows, or pointer issues.
10. **Works on One Compiler but Fails on Another**: Differences in compiler behavior, memory layout, or padding between variables can lead to programs behaving differently on various compilers or operating systems.
11. **Program Stops Abruptly**: Programs can stop abruptly if they encounter exceptions or signals. Ignoring exceptions or signals can lead to unexpected program termination.
12. **Memory Errors**: Memory-related errors include stack overflows, heap exhaustion, memory leaks, and accessing deallocated memory. These errors can lead to program crashes or erratic behavior.
13. **Numeric Problems**: Incorrect arithmetic calculations can lead to unexpected results, such as losing fractional parts of numbers, arithmetic overflow, division by zero, or NaN (Not-a-Number) results.
14. **Non-Zero Return Code**: When a program returns a non-zero exit code, it indicates that something unexpected or erroneous happened during execution. Identifying the cause of non-zero return codes requires careful investigation.

Debugging can indeed be a challenging and time-consuming process, but with experience and the right tools, you can become more efficient at identifying and fixing bugs in your code. It's crucial to approach debugging systematically, isolating issues step by step until you pinpoint the root cause and apply the necessary fixes.