3.x — Chapter 3 summary and quiz

Chapter Review

A **syntax error** is an error that occurs when you write a statement that is not valid according to the grammar of the C++ language. The compiler will catch these.

A **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended.

The process of finding and removing errors from a program is called **debugging**.

We can use a five step process to approach debugging:

1. Find the root cause.
2. Understand the problem.
3. Determine a fix.
4. Repair the issue.
5. Retest.

Finding an error is usually the hardest part of debugging.

Static analysis tools are tools that analyze your code and look for semantic issues that may indicate problems with your code.

Being able to reliably reproduce an issue is the first and most important step in debugging.

There are a number of tactics we can use to help find issues:

- Commenting out code.
- Using output statements to validate your code flow.
- Printing values.

When using print statements to debug, use *std::cerr* instead of *std::cout*. But even better, avoid debugging via print statements.

A **log file** is a file that records events that occur in a program. The process of writing information to a log file is called **logging**.

The process of restructuring your code without changing how it behaves is called **refactoring**. This is typically done to make your program more organized, modular, or performant.

Unit testing is a software testing method by which small units of source code are tested to determine whether they are correct.

Defensive programming is a technique whereby the programmer tries to anticipate all of the ways the software could be misused. These misuses can often be detected and mitigated.

All of the information tracked in a program (variable values, which functions have been called, the current point of execution) is part of the **program state**.

A **debugger** is a tool that allows the programmer to control how a program executes and examine the program state while the program is running. An **integrated debugger** is a debugger that integrates into the code editor.

Stepping is the name for a set of related debugging features that allow you to step through our code statement by statement.

Step into executes the next statement in the normal execution path of the program, and then pauses execution. If the statement contains a function call, *step into* causes the program to jump to the top of the function being called.

Step over executes the next statement in the normal execution path of the program, and then pauses execution. If the statement contains a function call, *step over* executes the function and returns control to you after the function has been executed.

Step out executes all remaining code in the function currently being executed and then returns control to you when the function has returned.

Run to cursor executes the program until execution reaches the statement selected by your mouse cursor.

Continue runs the program, until the program terminates or a breakpoint is hit. **Start** is the same as continue, just from the beginning of the program.

A **breakpoint** is a special marker that tells the debugger to stop execution of the program when the breakpoint is reached.

The **set next statement** command allows us to change the point of execution to some other statement (sometimes informally called jumping). This can be used to jump the point of execution forwards and skip some code that would otherwise execute, or backwards and have something that already executed run again.

Watching a variable allows you to inspect the value of a variable while the program is executing in debug mode. The **watch window** allows you to examine the value of variables or expressions.

The **call stack** is a list of all the active functions that have been executed to get to the current point of execution. The **call stack window** is a debugger window that shows the call stack.

Quiz time

Question #1

The following program is supposed to add two numbers, but doesn’t work correctly.

Use the integrated debugger to step through this program and watch the value of x. Based on the information you learn, fix the following program:

```
#include <iostream>

int readNumber(int x)
{
        std::cout << "Please enter a number: ";
        std::cin >> x;
        return x;
}

void writeAnswer(int x)
{
        std::cout << "The sum is: " << x << '\n';
}

int main()
{
        int x {};
        readNumber(x);
        x = x + readNumber(x);
        writeAnswer(x);

        return 0;
}
```

Show Solution

Question #2

The following program is supposed to divide two numbers, but doesn’t work correctly.

Use the integrated debugger to step through this program. For inputs, enter 8 and 4. Based on the information you learn, fix the following program:

```
#include <iostream>

int readNumber()
{
        std::cout << "Please enter a number: ";
        int x {};
        std::cin >> x;
        return x;
}

void writeAnswer(int x)
{
        std::cout << "The quotient is: " << x << '\n';
}

int main()
{
        int x{ };
        int y{ };
        x = readNumber();
        x = readNumber();
        writeAnswer(x/y);

        return 0;
}
```

Show Solution

Question #3

What does the call stack look like in the following program when the point of execution is on line 4? Only the function names are needed for this exercise, not the line numbers indicating the point of return.

```
#include <iostream>

void d()
{ // here
}

void c()
{
}

void b()
{
	c();
	d();
}

void a()
{
	b();
}

int main()
{
	a();

	return 0;
}
```

Show Solution

Author’s note

It’s hard to find good examples of simple programs that have non-obvious issues to debug, given the limited material covered so far. Any readers have any suggestions?