3.x — Chapter 3 summary and quiz

BY ALEX ON FEBRUARY 1ST, 2019 | LAST MODIFIED BY ALEX ON FEBRUARY 17TH, 2019

Quick Summary

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, 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 what it actually does is called **refactoring**.

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.

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

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;
}

int main()
{
    int x{ 0 };
    readNumber(x);
    x = x + readNumber(x);
    writeAnswer(x);

    return 0;
}
```

Show Solution

Question #2

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;
}

int main()
{
    int x{ 0 };
    int y{ 0 };
    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();

```

```
26      return 0;
27  }
```

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?

4.1 -- Introduction to fundamental data types

Index

3.10 -- Finding issues before they become problems

C++ TUTORIAL | PRINT THIS POST

45 comments to 3.x — Chapter 3 summary and quiz

Apaulture
January 29, 2020 at 12:52 pm · Reply

I think it's a great opportunity for learners to try and fix example #2, not just semantic errors, but also the naming of variables. The pass by value x/y can be confused with writeAnswer's parameter int x.

alex
January 16, 2020 at 1:33 pm · Reply

Im trying to do the first quiz, debugging the code? everytime i try to press debug the program just gives me 1 failed and nothing happens, whats going on here?

i can debug fine on code that has no errors but 1 error and i cant debug, so whats the point of debugging wtf lol, im confused, what am i missing?

alex
January 17, 2020 at 4:27 am · Reply

nevermind i figured out what was causing the problem xd

Wiktor
February 9, 2020 at 2:51 am · Reply

I bet you had some additional code files (leftover from previous lessons xd) with "void writeAsnwer" already defined. Been there xd.

Giang
December 28, 2019 at 7:29 pm · Reply

Thank for these lessons. But I still can't distinguish the main differences between Run to cursor and Breakpoint

Humam
January 1, 2020 at 1:57 am · Reply

The difference is simple.

You have the following program:

```
1  void programActive()
2  {
3     std::cout << "Hello!";
4  }
5
6  int main()
7  {
8     programActive();
9     programActive();
10    programActive();
11    return 0;
12 }
```

If you use "Run To Cursor" on Line 3 of my example code above, the program will execute until it reaches Line 3 for the first time after it has been invoked by my first function call on Line 8, then I will have regained control in debug mode. If I chose to "Continue", the rest of the program will execute ~NON STOP~ until the program ends (reaches return 0;).

However, if I set my "Breakpoint" on Line 3, when I execute the program, it will give me control back at Line 3 ~EACH TIME~ execution reaches the breakpoint I placed!
So for the first time it is invoked at Line 8, I will regain control once it reaches Line 3. (The breakpoint we set there)
If I chose to "Continue", the second time it is invoked at Line 9, I will again regain control once it reaches

Line 3. (The breakpoint we already have there)
If I chose to "Continue", the third time it is invoked at Line 10, I will regain control once it reaches Line 3. (The breakpoint we already have there)
If I chose to "Continue", the program will end at return 0 as we want.

So:

Breakpoints will continue to stop the point of execution at the point it is placed at, until we remove the breakpoint itself!

Even better, we can use multiple breakpoints in our program! Unlike "Run To Cursor" which can only be used once every time we want to use it.

Hope that makes sense! :)

giang
January 1, 2020 at 9:45 pm · Reply

Really simple but very understandable and make sense. thank you so much!!

HolzstockG
November 26, 2019 at 3:13 am · Reply

I did the first one a little bit easier by using expression inside of curly braces of "x" variable

```
int readNumber()
{
    std::cout << "Please enter a number: ";

    int x{};
    std::cin >> x;

    return x;
}

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

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

    return 0;
}
```

chai
November 30, 2019 at 7:53 am · Reply

you don't want to get into a habit of this. The problem is the machine does't know which function call is executed first. Might not be a problem now but may be later on. Not a good practice.

Apaulture

January 29, 2020 at 12:48 pm · Reply

When Alex mentioned the order of execution, I believe it was regarding the order of arguments passed to the called function. In this case, it looks valid. The left readNumber() gets called then the right readNumber() gets called. Nevertheless, I would still avoid this as it is less readable than

```
1 int x{};
2 x = readNumber() + readNumber();
```

and it is good practice to initialize the variable with a literal or zero-initialize if the value is expected to be replaced.

nascardriver
January 30, 2020 at 2:00 am · Reply

It's unspecified which `readNumber` gets called first.
If you know the value of a variable at its declaration, you should initialize it. Only initialize with empty braces if you don't know the value.

```
1 // We don't care about the order in which the @readNumbers are called
2 int x{ readNumber() + readNumber() };
3
4 // If we care about the order
5 int a{ readNumber() };
6 int b{ readNumber() };
7 int x{ a + b };
```

Jake
November 5, 2019 at 4:29 pm · Reply

Hi Alex. Another wonderment at not being properly stopped, but this time at the run level.

In question 2, line 22, we are obviously supposed to run into the zero-divide error. However, if I run without debugging, or run it in a CMD window, all I get is the standard Windows error box "Program Quiz-3.2 has stopped working". The only time I see the "Unhandled Exception" box is if I try to step over the

```
1 writeAnswer(x / y);
```

line.

Why won't is kick me out the way it would in Unix? OK, maybe not the crash dump but at least a zero-divide error croak message! (I am trying to provoke the bouncer. I'm SO sociopathic today! :-)

nascardriver
November 6, 2019 at 4:17 am · Reply

Division by 0 produces undefined behavior. For integers, that's usually a crash. `double` sometimes has fancy behavior, like NaN.

Jake
November 5, 2019 at 3:55 pm · Reply

Question, Alex. (Yuh think? :-)

Since we defined the readNumber() function as returning an int, why didn't I get a compile error, or at least a warning, at line 13, where we are discarding the return value and treating it like a void function? Of course I

saw this by eyeball but even with the "treat warnings like errors" setting (as well as level 4), I got no error.

Thanks for this tutorial!

nascardriver
November 6, 2019 at 4:10 am · Reply

Hi Jake,

it's perfectly legal to discard the return value of a function. For `readNumber`, it doesn't make sense to discard it, because the whole point of the function is to produce the value that it returns.
There are functions that return values we might not care about, because their purpose is something else and the return value may or may not be relevant.
When you write a function like `readNumber`, where discarding the return value is considered to be a mistake, mark it as `[[nodiscard]]`.

```
[[nodiscard]] int readNumber()
{
  // ...
}
```

`[[nodiscard]]` tells the compiler that you want a warning to be printed if the return value is discarded.

Jake
November 6, 2019 at 11:54 am · Reply

WOW! In all the C++ books I have, I found one mention of [[nodiscard]] in the Stroustrup book "A Tour of C++
Second Edition", page 216, without a description of what that does. I'm using the community version of Visual Studio 19 and when I added [[nodiscard]] that to the code I got this error message:

```
Warning C5051:  attribute 'nodiscard' requires at least '/std:c++17'; ignored
```

On the other hand, I do have a complete Cygwin environment and when I compiled it with that release of g++ I did get the expected error message:

```
$ g++ Quiz-03.2.cpp
Quiz-03.2.cpp: In function ‘int main()’:
Quiz-03.2.cpp:24:13: warning: ignoring return value of ‘int readNumber()’, declare
   readNumber();
   ~~~~~~~~~~^~
Quiz-03.2.cpp:6:19: note: declared here
[[nodiscard]] int readNumber()
```

But it still produces an executable. And when I ran that executable with the bad code I got the error I had originally expected:

```
$ ./a.exe
Please enter a number:8
Please enter a number:4
Floating point exception (core dumped)
```

Floating point?? No mention of zero-divide? OK, not exactly the expected error message.

This has been VERY instructive, although I'm not totally ready to start using the Cygwin environment for this tutorial.

nascardriver

November 7, 2019 at 1:41 am · Reply

`[[nodicard]]` asks the compiler to produce a warning, not an error. If you want an error, you need to treat Wunused-result as an error

```
-Werror=unused-result
```

VS should support `[[nodiscard]]` too, you just didn't set up your project correctly. In you project's settings, enable the highest standard support that's available (Should be C++17). `[[nodiscard]]` was added in C++17.

Mike
September 16, 2019 at 1:09 pm · Reply

For Quiz #2, I copied and pasted your sample code twice now, and it compiles and executes, but as soon as I press enter after entering the second number, it crashes every time with a Windows error that my test.exe has stopped working. One of my options is "Debug the Program" which I thought was fitting considering this chapter. I tried it, but it seems way to advanced for me.

Any suggestions as to why this is happening? BTW: I'm using Visual Studio 2019.

The Exception code is c0000094, if that helps. I just googled it. Appears it may be related to dividing by zero. But why, I entered 8 and 4 as per your instructions?

nascardriver
September 16, 2019 at 10:41 pm · Reply

It's supposed to crash. The purpose of the quiz is that you figure out why. You can solve it by reading the code too, but you won't learn how to use the debugger that way.

Mike
September 17, 2019 at 6:51 am · Reply

Ha! too funny! I guess I just wasn't expecting a full blown crash of the program at this point, though it makes perfect sense. Can't wait to try to debug it now!

Sorry for wasting your time on this one.

Jose
August 1, 2019 at 4:33 pm · Reply

Hi, question #3 shows the functions sorted this way:

```
void d()
void c()
void b()
void a()
int main ()
```

but if I had to write that code I would have sorted them this way:

```
void a()
void b()
void c()
void d()
int main ()
```

Is there any rule/best practice about sorting the functions called from main()?

nascardriver
August 2, 2019 at 12:19 am · Reply

If you ordered them like that, you'd need forward declarations for `b`, `c`, and `d`.
Order your functions in a way that requires as few forward declarations as possible.
If 2 functions are related and you think it's better to have them right after the other, you can use a forward declaration.

Jose
August 2, 2019 at 9:54 am · Reply

Makes sense, thank you :)

Joe Das
June 28, 2019 at 10:28 am · Reply

Outstanding Tutorial. You are really helping me get my old coding chops back so quickly, and your discussion points about good working practices are invaluable.

You were asking about nasty problems in debugging: An example which will quickly aggravate coders but which does happen is when a variable is misspelled because it is declared twice by similar names and the wrong variable is used in a function. This would tie back into your recommendations about variable naming. This can be made even nastier with confusing pre-processor statements >:D.

Alex
June 28, 2019 at 3:16 pm · Reply

> An example which will quickly aggravate coders but which does happen is when a variable is misspelled because it is declared twice by similar names and the wrong variable is used in a function

Agreed -- that's pretty similar to quiz question #2, no?

Lawrence
May 26, 2019 at 3:28 pm · Reply

Why can't I add enter a value when I am stepping in during debugging?

Alex
May 29, 2019 at 4:59 pm · Reply

You can, you just need to step forward to the point where the program is actually wait for your input.

NotJoeRogan
May 14, 2019 at 5:14 pm · Reply

Hello Alex or Nas,

Let's say that I enter a breakpoint at line 7 of the program in the first question, and for the number I enter 4. Why is it that when I hover my cursor over all of the x variables throughout the entire program, they show a value of 4?

Regards,
NotJoeRogan

Alex
May 17, 2019 at 12:27 pm · Reply

When you hover over a variable, the debugger will show you the _current_ value for that variable, not the value the variable might have wherever you are hovering. This can be misleading the variable had a different value at the point you're hovering

To do otherwise would require the debugger to remember what values every variable had at every point in the program, which would be significantly more complex, and potentially misleading in other ways (you might think you're seeing the current value when you aren't).

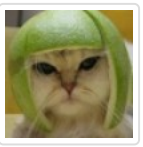
Maura
April 9, 2019 at 7:40 am · Reply

Please better explain on Question #3 why the answer for the call stack is
d
b
a
main

and not instead
d
c
b
a
main

Jason
April 9, 2019 at 11:47 am · Reply

Because when the stack reaches function b, it begins it execution.
First it reaches the call to function c.
At this point the stack is c/b/a/main.
However c does not make any calls and fully resolves itself.
At which it returns to the next object in the stack b, and moves on to the call to function d.
Thus the stack is d/b/a/main



Alex
April 9, 2019 at 11:50 am · Reply

main calls a()
a calls b()
b calls c()
c terminates
b calls d()
At this point we've hit our breakpoint.

Working backwards, the functions still in memory are:
d
b
a
main

c isn't in the list because it terminated and was removed from the call stack before d was called.

John Doe
March 11, 2019 at 11:18 am · Reply

What about leveraging your example of function argument order of evaluation issues in chapter 2.3? It would be a weird scenario but would force the user to step through while debugging and see the flow issues in their stack. Testing in visual studio and gcc, this processed right to left, but clang did left to right.

```
#include <iostream>

int f(int a, int b) {
    std::cout << "a: " << a << "\tb: " << b << "\n";
    return 0;
}

int a() {
    std::cout << "in a\n";
    return 1;
}

int b() {
    std::cout << "in b\n";
    return (42);
}

int main(void) {
    std::cout << "This program should go 'in a' first, then 'in b'\n";
    f(a(), b());
    return 0;
}
```

Alex
March 11, 2019 at 10:19 pm · Reply

Good thought! However, the program would only manifest the problem on some machines and not others, making it a hard one to debug on machines that didn't exhibit the issue.

Louis Cloete
February 15, 2019 at 2:40 pm · Reply

Debugging exercise ideas: use "/n" for a newline somewhere instead of "\n" and watch the ensuing chaos when /n prints on the screen... ;-p

Do something like this:

```
#include <iostream>

int x;
x = x + readNumber();
std::cout << "x = " << x << '\n'; // Uninitialised variable causes undefined behaviour
```

[This next idea I borrow from my Delphi textbook. You will have to wait till after you've done loops, strings and arrays for this one.]

```
#include <iostream>
#include <string>

// Prints string vertically
void prtStrVertical(std::string &s)
{
    for (int i { 0 }; i < s.length(); ++i)
        char c { s.at(i) };
        std::cout << c << '\n';
}
```

The output line only executes once, because of missing braces around the intended loop body. The function prints only the last character of the string.

nascardriver
February 16, 2019 at 4:27 am · Reply

Hi!

> Uninitialised variable causes undefined behaviour
The value of @x is undefined, behavior is well defined.

> next idea
@s should be const and @i should be an @std::size_t (Or cast s.length()).
This code wouldn't compile, because line 9 cannot see @c.
I like this one. Even though compilers print a warning, it's a common mistake.

Louis Cloete
February 16, 2019 at 6:39 am · Reply

About prtStrVertical():

In Pascal, you must define all local variables for a function or procedure (Pascal's name for a group of statements that doesn't return a value) at the top of the function, so the scoping issue doesn't appear. To work around, either declare char c before the loop, or modify the function like this:

```
#include <iostream>
#include <string>

// Prints string vertically
void prtStrVertical(std::string &s)
{
    for (int i { 0 }; i < s.length(); ++i)
        std::cout << s.at(i);
        std::cout << '\n';
}
```

The modification will cause a different, but related, bug.

Alex
February 16, 2019 at 11:46 am · Reply

Thanks for the thoughts! The '/n' idea should be easy to diagnose by inspection. I'm trying to find things that are not quite as obvious to inspection, to validate the usefulness of having an integrated debugger.

There are a ton of fun debugging problems once we do loops -- but alas, that chapter is a bit far off from this lesson. I didn't want to wait that long to introduce basic debugging topics.

Louis Cloete
February 15, 2019 at 2:07 pm · Reply

Question #1 solution:

readNumber():

```
int readNumber()
{
    std::cout << "Please enter a number:";
    int x {}; // Declare as close as possible to first use and init
    std::cin >> x;
    return x;
}
```

main():

l. 18:

```
int x { readNumber() }; // Uniform init
```

Question #2:

The problem could've been avoided if the program didn't declare and zero init variables, just to assign different values to them in the first place. A better solution would be to init x and y with the result of separate calls to readNumber(). I think the solution should reflect that. I.e. replace Lines 18 - 21 in the solution with:

```
int x { readNumber() };
int y { readNumber() };
```

Alex
February 16, 2019 at 11:42 am · Reply

Thanks for the suggestions. Both have been incorporated.

Louis Cloete
February 16, 2019 at 11:56 am · Reply

readNumber() is now different in all three places it occurs: Solution 1, Question 2 and Solution 2. While you are at it, maybe make them identical.

Anthony
February 5, 2019 at 6:30 pm · Reply

For the note at the end, you could maybe add "debugging practice revisited" after some chapters since the basics have been covered. Would allow for some more intricate issues to solve and extra reinforcement for practicing good debugging habits.

Alex
February 7, 2019 at 6:27 am · Reply

Good idea. I'll revisit this shortly.

Ethan Smith
February 3, 2019 at 6:45 pm · Reply

"The process of finding and removing errors from a programming is called debugging" i think you may have made a grammatical mistake there.

Alex
February 4, 2019 at 8:57 pm · Reply

Indeed. Thanks for pointing that out.