3.1 — Syntax and semantic errors

Software errors are prevalent. It's easy to make them, and it's hard to find them. In this chapter, we'll explore topics related to the finding and removal of bugs within our C++ programs, including learning how to use the integrated debugger that is part of our IDE.

Although debugging tools and techniques aren't part of the C++ standard, learning to find and remove bugs in the programs you write is an extremely important part of being a successful programmer. Therefore, we'll spend a bit of time covering such topics, so that as the programs you write become more complex, your ability to diagnose and remedy issues advances at a similar pace.

If you have experience debugging programs in another programming language, much of this will be familiar to you.

Syntax and semantic errors

Programming can be challenging, and C++ is somewhat of a quirky language. Put those two together, and there are a lot of ways to make mistakes. Errors generally fall into one of two categories: syntax errors, and semantic errors (logic errors).

A **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. This includes errors such as missing semicolons, using undeclared variables, mismatched parentheses or braces, etc… For example, the following program contains quite a few syntax errors:

```
#include <iostream>

int main()
{
    std::cout < "Hi there"; << x << '\n'; // invalid operator (<), extraneous
semicolon, undeclared variable (x)
    return 0 // missing semicolon at end of statement
}
```

Fortunately, the compiler will generally catch syntax errors and generate warnings or errors, so you easily identify and fix the problem. Then it's just a matter of compiling again until you get rid of all the errors.

Once your program is compiling correctly, getting it to actually produce the result(s) you want can be tricky. A **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended.

Sometimes these will cause your program to crash, such as in the case of division by zero:

```
#include <iostream>

int main()
{
    int a { 10 };
    int b { 0 };
    std::cout << a << " / " << b << " = " << a / b << '\n'; // division by 0 is
undefined in mathematics
    return 0;
}
```

More often these will just produce the wrong value or behavior:

```
#include <iostream>

int main()
{
    int x; // no initializer provided
    std::cout << x << '\n'; // Use of uninitialized variable leads to undefined
result

    return 0;
}
```

or

```
#include <iostream>

int add(int x, int y) // this function is supposed to perform addition
{
    return x - y; // but it doesn't due to the wrong operator being used
}

int main()
{
    std::cout << "5 + 3 = " << add(5, 3) << '\n'; // should produce 8, but produces 2

    return 0;
}
```

or

```
#include <iostream>

int main()
{
    return 0; // function returns here

    std::cout << "Hello, world!\n"; // so this never executes
}
```

Modern compilers have been getting better at detecting certain types of common semantic errors (e.g. use of an uninitialized variable). However, in most cases, the compiler will not be able to catch most of these types of problems, because the compiler is designed to enforce grammar, not intent.

In the above example, the errors are fairly easy to spot. But in most non-trivial programs, semantic errors are not easy to find by eyeballing the code. This is where debugging techniques can come in handy.

Next lesson
3.2The debugging process
Back to table of contents
Previous lesson
2.xChapter 2 summary and quiz