9.3 — Common semantic errors in C++

In lesson 3.1 -- Syntax and semantic errors, we covered `syntax errors`, which occur when you write code that is not valid according to the grammar of the C++ language. The compiler will notify you of such errors, so they are trivial to catch, and usually straightforward to fix.

We also covered `semantic errors`, which occur when you write code that does not do what you intended. The compiler generally will not catch semantic errors (though in some cases, smart compilers may be able to generate a warning).

Semantic errors can cause most of the same symptoms of `undefined behavior`, such as causing the program to produce the wrong results, causing erratic behavior, corrupting program data, causing the program to crash -- or they may not have any impact at all.

When writing programs, it is almost inevitable that you will make semantic errors. You will probably notice some of these just by using the program: for example, if you were writing a maze game, and your character was able to walk through walls. Testing your program (9.1 -- Introduction to testing your code) can also help surface semantic errors.

But there's one other thing that can help -- and that's knowing which type of semantic errors are most common, so you can spend a little more time ensuring things are right in those cases.

In this lesson, we'll cover a bunch of the most common types of semantic errors that occur in C++ (most of which have to do with flow control in some way).

Conditional logic errors

One of the most common types of semantic errors is a conditional logic error. A **conditional logic error** occurs when the programmer incorrectly codes the logic of a conditional statement or loop condition. Here is a simple example:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    if (x >= 5) // oops, we used operator>= instead of operator>
        std::cout << x << " is greater than 5\n";

    return 0;
}
```

Here's a run of the program that exhibits the conditional logic error:

```
Enter an integer: 5
5 is greater than 5
```

When the user enters `5`, the conditional expression `x >= 5` evaluates to `true`, so the associated statement is executed.

Here's another example, using a for loop:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int x{};
    std::cin >> x;

    // oops, we used operator> instead of operator<
    for (int count{ 1 }; count > x; ++count)
    {
        std::cout << count << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

This program is supposed to print all of the numbers between 1 and the number the user entered. But here's what it actually does:

```
Enter an integer: 5
```

It didn't print anything. This happens because on entrance to the for loop, `count > x` is `false`, so the loop never iterates at all.

Infinite loops

In lesson 8.8 -- Introduction to loops and while statements, we covered infinite loops, and showed this example:

```
#include <iostream>

int main()
{
    int count{ 1 };
    while (count <= 10) // this condition will never be false
    {
        std::cout << count << ' '; // so this line will repeatedly execute
    }

    std::cout << '\n'; // this line will never execute

    return 0; // this line will never execute
}
```

In this case, we forgot to increment `count`, so the loop condition will never be false, and the loop will continue to print:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

… until the user shuts down the program.

Here's another example that teachers love asking as a quiz question. What's wrong with the following code?

```
#include <iostream>

int main()
{
    for (unsigned int count{ 5 }; count >= 0; --count)
    {
        if (count == 0)
            std::cout << "blastoff! ";
        else
          std::cout << count << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

This program is supposed to print `5 4 3 2 1 blastoff!`, which it does, but it doesn't stop there. In actuality, it prints:

```
5 4 3 2 1 blastoff! 4294967295 4294967294 4294967293 4294967292 4294967291
```

and then just keeps decrementing. The program will never terminate, because `count >= 0` can never be `false` when `count` is an unsigned integer.

Off-by-one errors

An **off-by-one** error is an error that occurs when a loop executes one too many or one too few times. Here's an example that we covered in lesson 8.10 -- For statements:

```
#include <iostream>

int main()
{
    for (int count{ 1 }; count < 5; ++count)
    {
        std::cout << count << ' ';
    }

    std::cout << '\n';

    return 0;
}
```

The programmer intended for this code to print `1 2 3 4 5`. However, the wrong relational operator was used (`<` instead of `<=`), so the loop executes one fewer times than intended, printing `1 2 3 4`.

Incorrect operator precedence

From lesson 6.7 -- Logical operators, the following program makes an operator precedence mistake:

```
#include <iostream>

int main()
{
    int x{ 5 };
    int y{ 7 };

    if (!x > y) // oops: operator precedence issue
        std::cout << x << " is not greater than " << y << '\n';
    else
        std::cout << x << " is greater than " << y << '\n';

    return 0;
}
```

Because `logical NOT` has higher precedence than `operator>`, the conditional evaluates as if it was written `(!x) > y`, which isn't what the programmer intended.

As a result, this program prints:

```
5 is greater than 7
```

This can also happen when mixing Logical OR and Logical AND in the same expression (Logical AND takes precedence over Logical OR). Use explicit parenthesization to avoid these kinds of errors.

Precision issues with floating point types

The following floating point variable doesn't have enough precision to store the entire number:

```
#include <iostream>

int main()
{
    float f{ 0.123456789f };
    std::cout << f << '\n';

    return 0;
}
```

Because of this lack of precision, the number is rounded slightly:

```
0.123457
```

In lesson 6.6 -- Relational operators and floating point comparisons, we talked about how using `operator==` and `operator!=` can be problematic with floating point numbers due to small rounding errors (as well as what to do about it). Here's an example:

```
#include <iostream>

int main()
{
    double d{ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 }; // should
sum to 1.0

    if (d == 1.0)
        std::cout << "equal\n";
    else
        std::cout << "not equal\n";

    return 0;
}
```

This program prints:

```
not equal
```

The more arithmetic you do with a floating point number, the more it will accumulate small rounding errors.

Integer division

In the following example, we mean to do a floating point division, but because both operands are integers, we end up doing an integer division instead:

```
#include <iostream>

int main()
{
    int x{ 5 };
    int y{ 3 };

    std::cout << x << " divided by " << y << " is: " << x / y << '\n'; // integer
division

    return 0;
}
```

This prints:

```
5 divided by 3 is: 1
```

In lesson 6.2 -- Arithmetic operators, we showed that we can use static_cast to convert one of the integral operands to a floating point value in order to do floating point division.

Accidental null statements

In lesson 8.3 -- Common if statement problems, we covered `null statements`, which are statements that do nothing.

In the below program, we only want to blow up the world if we have the user's permission:

```
#include <iostream>

void blowUpWorld()
{
    std::cout << "Kaboom!\n";
}

int main()
{
    std::cout << "Should we blow up the world again? (y/n): ";
    char c{};
    std::cin >> c;

    if (c == 'y');     // accidental null statement here
        blowUpWorld(); // so this will always execute since it's not part of the if-
statement

    return 0;
}
```

However, because of an accidental `null statement`, the function call to `blowUpWorld()` is always executed, so we blow it up regardless:

```
Should we blow up the world again? (y/n): n
Kaboom!
```

Not using a compound statement when one is required

Another variant of the above program that always blows up the world:

```
#include <iostream>

void blowUpWorld()
{
    std::cout << "Kaboom!\n";
}

int main()
{
    std::cout << "Should we blow up the world again? (y/n): ";
    char c{};
    std::cin >> c;

    if (c == 'y')
        std::cout << "Okay, here we go...\n";
        blowUpWorld(); // Will always execute.  Should be inside compound statement.

    return 0;
}
```

This program prints:

```
Should we blow up the world again? (y/n): n
Kaboom!
```

A `dangling else` (covered in lesson 8.3 -- Common if statement problems) also falls into this category.

Using assignment instead of equality inside a conditional

Because the assignment operator (`=`) and equality operator (`==`) are similar, we may intend to use equality but accidentally use assignment instead:

```
#include <iostream>

void blowUpWorld()
{
    std::cout << "Kaboom!\n";
}

int main()
{
    std::cout << "Should we blow up the world again? (y/n): ";
    char c{};
    std::cin >> c;

    if (c = 'y') // uses assignment operatpr instead of equality operator
        blowUpWorld();

    return 0;
}
```

This program prints:

```
Should we blow up the world again? (y/n): n
Kaboom!
```

The assignment operator returns its left operand. `c = 'y'` executes first, which assigns `y` to `c` and returns `c`. Then `if (c)` is evaluated. Since `c` is now non-zero, it is implicitly converted to `bool` value `true`, and the statement associated with the if-statement executes.

Because assignment inside a conditional is almost never intended, modern compilers will often warn when they encounter this. However, if you aren't in the habit of resolving all your warnings, such warnings can easily get lost.

What else?

The above represents a good sample of the most common type of semantic errors new C++ programmers tend to make, but there are plenty more. Readers, if you have any additional ones that you think are common pitfalls, leave a note in the comments.