Is *n* prime?

Consider the following problem:

> Create a loop that prints all prime numbers less than or equal to 10000.

The main loop

Immediately we now that we must test numbers up to 10000, and this can be done with a for-loop where the loop variable ends with 10000. We know the smallest prime number is 2, so we can initialize the loop variable with 2. In general, we ask "Is *n* prime?", so it's not that bad an idea to use the variable name *n* in this case:

```
for ( int n{2}; n <= 10000; ++n ) {
    // If 'n' is prime, print it.
}
```

We will need to check if *n* is prime, so what we will likely do is declare a Boolean-valued local variable where we initially assume that *n* is prime, and then, after checking that it is, we check this variable again and print out *n* if it is indeed prime:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    // If we determine 'n' is not prime,
    //      set 'n_is_prime' to false.

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Checking if *n* is prime

The integer *n* is not prime if it is divisible by any other integer greater than 2 and less than or equal to *n*. Thus, we must check if *n* is divisible by all integers from 2 all the way up to, but not including *n*. This is another for-loop:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; i < n; ++i ) {
        // Check if 'n' is divisible by 'i'
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

The integer *n* is divisible by *i* if when we divide *n* by *i*, there is zero remainder. The remainder operator in C++ is %. If we determine that n % i is ever zero, then we can set `n_is_prime` to false:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; i < n; ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

First case of doing less work

Note that as soon as we set n_is_prime to false, there is no longer any need to keep looping: for example, if we have determined *n* is divisible by 17, then why check 18, 19 or any other value? Thus, looking at the condition of the inner loop, we keep iterating so long as i < n; however, now we realize that we want to keep iterating so long as both i < n and we still think that *n* may possibly be prime. We still think *n* may possibly be prime if n_is_prime is set to true, so we want to loop as long as (i < n) && (n_is_prime == true)

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; (i < n) && (n_is_prime == true); ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Some of you may be aware we don't have to explicitly compare n_is_prime with true, as in this case, it already has the value true, so we may write clearer code as follows:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; (i < n) && n_is_prime; ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Note that this is only clearer because we chose an intelligent variable name for the local variable we are calling `n_is_prime`. Imagine what this code would look like if someone just used the variable name b for a "B"oolean varaiable?

```
for ( int n{2}; n <= 10000; ++n ) {
    bool b{ true };

    for ( int i{2}; (i < n) && b; ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Good variable names clarify and simply your code.

Now, some of you may object: "What, you're using `n` and `i` as looping variables? Aren't those no different from `b`? True, and some of you may be able to come up with better variable names (any suggestions?), but it is common for the variable names `i`, `j`, `k`, `m`, and `n` to represent integers. Do not use `u`, `v`, `w`, `x`, `y`, or `z` to represent integers. These are usually reserved for representing real or complex numbers.

Which order of conditions?

Now, let's consider the condition `(i < n) && n_is_prime`. A logical AND operation is true if both operands are true, so this is equivalent to `n_is_prime && (i < n)`, so does it matter which one we use? At this point, you may remember something about *short-circuit evaluation* of logical operators. If the first is false, then regardless of the second condition, the entire expression will be false, so C++ does not even try to run the second; however, if the first is true, then we must always check the second.

In this case, one operand requires us to make a comparison (more work) while the other is just checking the value of a local variable (less work). We really want to do the simpler one first, for if that is false, we're done: we don't need to perform the comparison. Thus, a better solution is:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; n_is_prime && (i < n); ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Doing even less work

This is, however, a lot of extra work, because all divisors of *n* must be less than or equal to the square root of *n*, so there is no point in the inner loop going beyond the square root of *n*:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; n_is_prime && (i <= sqrt( n )); ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

The square root of an integer, however, may no longer be an integer, and actually, if the square root of an integer is not an integer, it must be irrational. Thus, we must somehow make 100% sure that the test `i <= sqrt( n )` works correctly even if `sqrt( n )` is irrational and may have an error. This is where we need to apply some thought: if $i < \sqrt{n}$ then $i^2 < n$. There are no irrational numbers in this second expression, so let's use that instead:

```
for ( int n{2}; n <= 10000; ++n ) {
    bool n_is_prime{ true };

    for ( int i{2}; n_is_prime && (i*i <= n); ++i ) {
        // 'n' is divisible by 'i' if (n % i) has zero remainder
        if ( (n % i) == 0 ) {
            n_is_prime = false;
        }
    }

    if ( n_is_prime ) {
        std::cout << n << std::endl;
    }
}
```

Summary

Thus, we have a reasonably efficient algorithm for finding all prime numbers less than or equal to 10000. We could do better; for example, we really only need to try to divide *n* by all prime numbers less than or equal to the square root of *n*, but this would require additional memory (we will introduce arrays later). We could also use the Sieve of Eratosthenes, but that, too, requires an array. Thus, given the tools we have so far, this is a good solution. We will add one small simplification in an appendix.

Appendix: Only check 2 and odd numbers

How would you change this code so that it only tries two and odd numbers? Our solution is on the next page, but try it yourself.

```
        std::cout << 2 << std::endl; // 2 is prime...

        // No other even number is prime
        for ( int n{3}; n <= 10000; n += 2 ) {
            bool n_is_prime{ true };

            if ( (n % 2) == 0 ) {
                n_is_prime = false;
            } else {
                for ( int i{3}; n_is_prime && (i*i <= n); i += 2 ) {
                    // 'n' is divisible by 'i' if (n % i) has zero remainder
                    if ( (n % i) == 0 ) {
                        n_is_prime = false;
                    }
                }
            }

            if ( n_is_prime ) {
                std::cout << n << std::endl;
            }
  }
```