6.3 — Remainder and Exponentiation

The remainder operator (`operator%`)

The **remainder operator** (also commonly called the **modulo operator** or **modulus operator**) is an operator that returns the remainder after doing an integer division. For example, 7 / 4 = 1 remainder 3. Therefore, 7 % 4 = 3. As another example, 25 / 7 = 3 remainder 4, thus 25 % 7 = 4. The remainder operator only works with integer operands.

This is most useful for testing whether a number is evenly divisible by another number (meaning that after division, there is no remainder): if *x % y* evaluates to 0, then we know that *x* is evenly divisible by *y*.

```
#include <iostream>

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

	std::cout << "Enter another integer: ";
	int y{};
	std::cin >> y;

	std::cout << "The remainder is: " << x % y << '\n';

	if ((x % y) == 0)
		std::cout << x << " is evenly divisible by " << y << '\n';
	else
		std::cout << x << " is not evenly divisible by " << y << '\n';

	return 0;
}
```

Here are a couple runs of this program:

```
Enter an integer: 6
Enter another integer: 3
The remainder is: 0
6 is evenly divisible by 3

Enter an integer: 6
Enter another integer: 4
The remainder is: 2
6 is not evenly divisible by 4
```

Now let's try an example where the second number is bigger than the first:

```
Enter an integer: 2
Enter another integer: 4
The remainder is: 2
2 is not evenly divisible by 4
```

A remainder of 2 might be a little non-obvious at first, but it's simple: 2 / 4 is 0 (using integer division) remainder 2. Whenever the second number is larger than the first, the second number will divide the first 0 times, so the first number will be the remainder.

Remainder with negative numbers

The remainder operator can also work with negative operands. `x % y` always returns results with the sign of *x*.

Running the above program:

```
Enter an integer: -6
Enter another integer: 4
The remainder is: -2
-6 is not evenly divisible by 4

Enter an integer: 6
Enter another integer: -4
The remainder is: 2
6 is not evenly divisible by -4
```

In both cases, you can see the remainder takes the sign of the first operand.

Nomenclature

The C++ standard does not actually give a name to `operator%`. However, the C++20 standard does say, "the binary % operator yields the remainder from the division of the first expression by the second".

Although `operator%` is often called the "modulo" or "modulus" operator, this can be confusing, because modulo in mathematics is often defined in a way that yields a different result to what `operator%` in C++ produces when one (and only one) of the operands is negative.

For example, in mathematics:
-21 modulo 4 = 3
-21 remainder 4 = -1

For this reason, we believe the name "remainder" is a more accurate name for `operator%` than "modulo".

In cases where the first operand can be negative, one must take care to note that the remainder can also be negative. For example, you might think to write a function that returns whether a number is odd like this:

```
bool isOdd(int x)
{
    return (x % 2) == 1; // fails when x is -5
}
```

However, this will fail when x is a negative odd number, such as `-5`, because `-5 % 2` is -1, and -1 != 1.

For this reason, if you're going to compare the result of a remainder operation, it's better to compare against 0, which does not have positive/negative number issues:

```
bool isOdd(int x)
{
    return (x % 2) != 0; // could also write return (x % 2)
}
```

Best practice

Prefer to compare the result of the remainder operator (`operator%`) against 0 if possible.

Where's the exponent operator?

You'll note that the ^ operator (commonly used to denote exponentiation in mathematics) is a *Bitwise XOR* operation in C++ (covered in lesson O.3 -- Bit manipulation with bitwise operators and bit masks). C++ does not include an exponent operator.

To do exponents in C++, #include the <cmath> header, and use the pow() function:

```
#include <cmath>

double x{ std::pow(3.0, 4.0) }; // 3 to the 4th power
```

Note that the parameters (and return value) of function pow() are of type double. Due to rounding errors in floating point numbers, the results of pow() may not be precise (even if you pass it integers or whole numbers).

If you want to do integer exponentiation, you're best off using your own function to do so. The following function implements integer exponentiation (using the non-intuitive "exponentiation by squaring" algorithm for efficiency):

```
#include <cassert> // for assert
#include <cstdint> // for std::int64_t
#include <iostream>

// note: exp must be non-negative
// note: does not perform range/overflow checking, use with caution
constexpr std::int64_t powint(std::int64_t base, int exp)
{
	assert(exp >= 0 && "powint: exp parameter has negative value");

	// Handle 0 case
	if (base == 0)
		return (exp == 0) ? 1 : 0;

	std::int64_t result{ 1 };
	while (exp > 0)
	{
		if (exp & 1)  // if exp is odd
			result *= base;
		exp /= 2;
		base *= base;
	}

	return result;
}

int main()
{
	std::cout << powint(7, 12) << '\n'; // 7 to the 12th power

	return 0;
}
```

Produces:

```
13841287201
```

Don’t worry if you don’t understand how this function works -- you don’t need to understand it in order to call it. The constexpr tag allows the compiler to evaluate this function at compile-time when the arguments are constant expressions.

Related content

We cover asserts in lesson 9.6 -- Assert and static_assert, and constexpr functions in lesson 5.8 -- Constexpr and consteval functions.

Warning

In the vast majority of cases, integer exponentiation will overflow the integral type. This is likely why such a function wasn’t included in the standard library in the first place.

Here is a safer version of the exponentiation function above that checks for overflow:

```
#include <cassert> // for assert
#include <cstdint> // for std::int64_t
#include <iostream>
#include <limits> // for std::numeric_limits

// A safer (but slower) version of powint() that checks for overflow
// note: exp must be non-negative
// Returns std::numeric_limits<std::int64_t>::max() if overflow occurs
constexpr std::int64_t powint_safe(std::int64_t base, int exp)
{
    assert(exp >= 0 && "powint_safe: exp parameter has negative value");

    // Handle 0 case
    if (base == 0)
        return (exp == 0) ? 1 : 0;

    std::int64_t result { 1 };

    // To make the range checks easier, we'll ensure base is positive
    // We'll flip the result at the end if needed
    bool negativeResult{ false };

    if (base < 0)
    {
        base = -base;
        negativeResult = (exp & 1);
    }

    while (exp > 0)
    {
        if (exp & 1) // if exp is odd
        {
            // Check if result will overflow when multiplied by base
            if (result > std::numeric_limits<std::int64_t>::max() / base)
            {
                std::cerr << "powint_safe(): result overflowed\n";
                return std::numeric_limits<std::int64_t>::max();
            }

            result *= base;
        }

        exp /= 2;

        // If we're done, get out here
        if (exp <= 0)
            break;

        // The following only needs to execute if we're going to iterate again

        // Check if base will overflow when multiplied by base
        if (base > std::numeric_limits<std::int64_t>::max() / base)
```

```
        {
            std::cerr << "powint_safe(): base overflowed\n";
            return std::numeric_limits<std::int64_t>::max();
        }

        base *= base;
    }

    if (negativeResult)
        return -result;

    return result;
}

int main()
{
        std::cout << powint_safe(7, 12) << '\n'; // 7 to the 12th power
        std::cout << powint_safe(70, 12) << '\n'; // 70 to the 12th power (will
return the max 64-bit int value)

        return 0;
}
```

Quiz time

Question #1

What does the following expression evaluate to? `6 + 5 * 4 % 3`

Show Solution

Question #2

Write a program that asks the user to input an integer, and tells the user whether the number is even or odd. Write a constexpr function called `isEven()` that returns true if an integer passed to it is even, and false otherwise. Use the remainder operator to test whether the integer parameter is even. Make sure `isEven()` works with both positive and negative numbers.

Show Hint

Your program should match the following output:

```
Enter an integer: 5
5 is odd
```

Show Solution