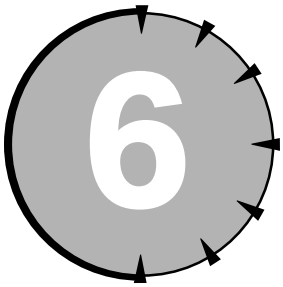
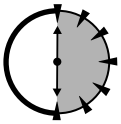
SESSION



Mathematical Operations

Session Checklist

- ✔ Using the C++ mathematical operators
- ✔ Identifying expressions
- ✔ Increasing clarity with "special" mathematical operators



30 Min. To Go

The Conversion and Average programs made use of simple mathematical operators such as addition, multiplication, and division. I have used these operators without describing them because they are largely intuitive. This session describes the set of mathematical operators.

The mathematical operators are listed in Table 6-1. The first column lists the precedence of the operator with those operators at the top of the table having higher precedence than those below.

Table 6-1
C++ Mathematical Operators

Operator	Meaning
+ (unary)	effectively does nothing
- (unary)	returns the negative of its argument
++ (unary)	increment
-- (unary)	decrement
*	multiplication
/	division
%	modulo
+	addition
–	subtraction
=, *=,%=,+=,-=	assignment types

Each of these operators is addressed in the following sections.

Arithmetic Operators

The multiplication, division, modulo, addition, and subtraction are the operators used to perform conventional arithmetic. Each of these operators has the conventional meaning that you studied in grammar school with the possible exception of modulo.

The modulo operator is similar to what my teachers called the remainder after division. For example, 4 goes into 15 three times with a remainder of 3. Expressed in C++ terms 15 modulo 4 is 3.

Because programmers are always trying to impress nonprogrammers with the simplest things, C++ programmers define modulo as follows:

```
IntValue % IntDivisor
```

is equal to

```
IntValue - (IntValue / IntDivisor) * IntDivisor
```

Let's try this out on our earlier example:

```
15 % 4 is equal to 15 - (15/4) * 4
                   15 - 3 * 4
                   15 - 12
                   3
```

Because modulo depends on the round off error inherent in integers, modulo is not defined for floating-point numbers.

Expressions

The most common type of statement in C++ is the expression. An *expression* is a statement that has a value. All expressions also have a type.

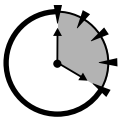
For example, a statement involving any of the mathematical operators is an expression because all of these operators return a value. Expressions can be complex or extremely simple. In fact, the statement "1" is an expression. There are five expressions in the following statement:

```
z = x * y + w;
1. x * y + w
2. x * y
3. x
4. y
5. w
```

An unusual aspect of C++ is that an expression is a complete statement. Thus, the following is a legal C++ statement:

```
1;
```

All expressions have a type. As we have already noted, the type of the expression 1 is `int`. In an assignment, the type of the expression to the right of the assignment is always the same as the type of the variable to the left — if it is not, C++ makes the necessary conversions.



20 Min. To Go

Operator Precedence

Each of the C++ operators has a property which determines the order that operators are evaluated in compound expressions (expressions with more than one operator). This property is known as precedence.

The expression

```
x/100 + 32
```

divides *x* by 100 before adding 32. In a given expression, C++ performs multiplication and division before addition or subtraction.We say that multiplication and division have higher *precedence* than addition and subtraction.

What if the programmer wanted to divide *x* by 100 plus 32? The programmer can bundle expressions together using parentheses as follows:

```
x/(100 + 32)
```

This has the same effect as dividing *x* by 132. The expression within the parentheses is evaluated first. This allows the programmer to override the precedence of individual operators.

The original expression

```
x / 100 + 32
```

is identical to the expression:

```
(x/100) + 32
```

The precedence of each of the operators described in Table 6-1 is shown in Table 6-2.

Table 6-2
C++ Mathematical Operators Including Their Precedence

Precedence	Operator	Meaning
1	+ (unary)	effectively does nothing
1	- (unary)	returns the negative of its argument
2	++ (unary)	increment
2	-- (unary)	decrement

Precedence	Operator	Meaning
3	*	multiplication
3	/	division
3	%	modulo
4	+	addition
4	–	subtraction
5	=, *=,%=,+=,-=	assignment types

Operators of the same precedence are evaluated from left to right. Thus the expression

```
x / 10 / 2
```

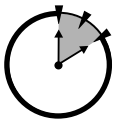
is the same as

```
(x / 10) / 2
```

Multiple levels of parentheses are evaluated from the inside out. In the following expression:

```
(y / (2 + 3)) / x
```

the variable y is divided by 5 and the result divided by x.



10 Min. To Go

Unary Operators

Unary operators are those operators that take a single argument.

A binary operator is an operator that takes two arguments. For example, consider `a + b`. In C++ jargon, the arguments to the addition operator are the expression to the left and the expression on the right.

The unary mathematical operators are +, -, ++, and --.

The minus operator changes the sign of its argument. Positive numbers become negative and vice versa. The plus operator does not change the sign of its argument. Effectively the plus operator has no effect at all.

The ++ and the -- operators increment and decrement their arguments by one.

Why a Separate Increment Operator?

The authors of C++ noted that programmers add 1 more than any other constant. As a convenience factor, a special "add 1" instruction was added to the language.

In addition, most computer processors have an increment instruction that is faster than the addition instruction. When C++ was created, saving a few instructions was a big deal considering what the state of development of microprocessors was.

No matter why the increment and decrement operators were created, you will see in Session 7 that they get a lot more use than you might think.

The increment and decrement operators are limited to non-floating-point variables.

The increment and decrement operators are peculiar in that both come in two flavors: a prefix version and a postfix version.

The prefix version of increment is written ++x while the postfix appears as x++.

Consider the increment operator (the decrement is exactly analogous).

Suppose that the variable n has the value 5. Both ++n and n++ increment n to the value 6. The difference between the two is that the value of ++n in an expression is 6 while the value of n++ is 5. The following example demonstrates this:

```
// declare three integer variables
int n1, n2, n3;

// the value of both n1 and n2 is 6
n1 = 5;
n2 = ++n1;

// the value of n1 is 6 but the value of n3 is 5
```

```
n1 = 5;
n3 = n1++;
```

Thus, n2 is given the value of n1 *after* n1 has been incremented using the preincrement operator, while n3 gets the vale of n1 *before* it is incremented using the postincrement operator.

Assignment Operators

The assignment operators are binary operators that change the value of their left-hand argument.

The simple assignment operator, '=', is an absolute necessity in any programming language. This operator stores the value of the right-hand argument in the left-hand argument. The other assignment operators, however, appear to be someone's whim.

The creators of C++ noticed that assignments often follow the form:

```
variable = variable # constant
```

where '#' is some binary operator. Thus, to increment an integer operator by 2 the programmer might write:

```
nVariable = nVariable + 2;
```

This says, add 2 to the value of nVariable and store the results in nVariable.

It is common to see the same variable on both the right-hand and left-hand sides of an assignment.

Because the same variable appeared to be showing up on both sides of the '=' sign, they decided to add the operator to the assignment operator. All of the binary operators have an assignment version. Thus, the assignment above could be written:

Done!

```
nVariable += 2;
```

Once again, this says add 2 to the value of nVariable.

Other than assignment itself, these assignment operators are not used that often. In certain cases, however, they can actually make the resulting program easier to read.

REVIEW

The mathematical operators are used more than any other operators in C++ programs. This is hardly surprising: C++ programs are always converting temperatures back and forth from Celsius to Fahrenheit and myriad other operations that require the ability to add, subtract, and count.

- All expressions have a value and a type.
- The order of evaluation of operators within an expression is normally determined by the precedence of the operators; however, this order can be overridden through the use of parentheses.
- For many of the most common expressions, C++ provides shortcut operators. The most common is the `i++` operator in place of `i = i + 1`.

QUIZ YOURSELF

1. What is the value of `9 % 4`? (See "Arithmetic Operators.")
2. Is `9 % 4` an expression? (See "Expressions.")
3. If `n` = 4, what is the value of `n + 10 / 2`? Why is it 9 and not 7? (See "Operator Precedence.")
4. If `n` = 4, what is the difference between `++n` and `n++`? (See "Unary Operators.")
5. How would I write `n = n + 2` using the += operator? (See "Assignment Operators.")