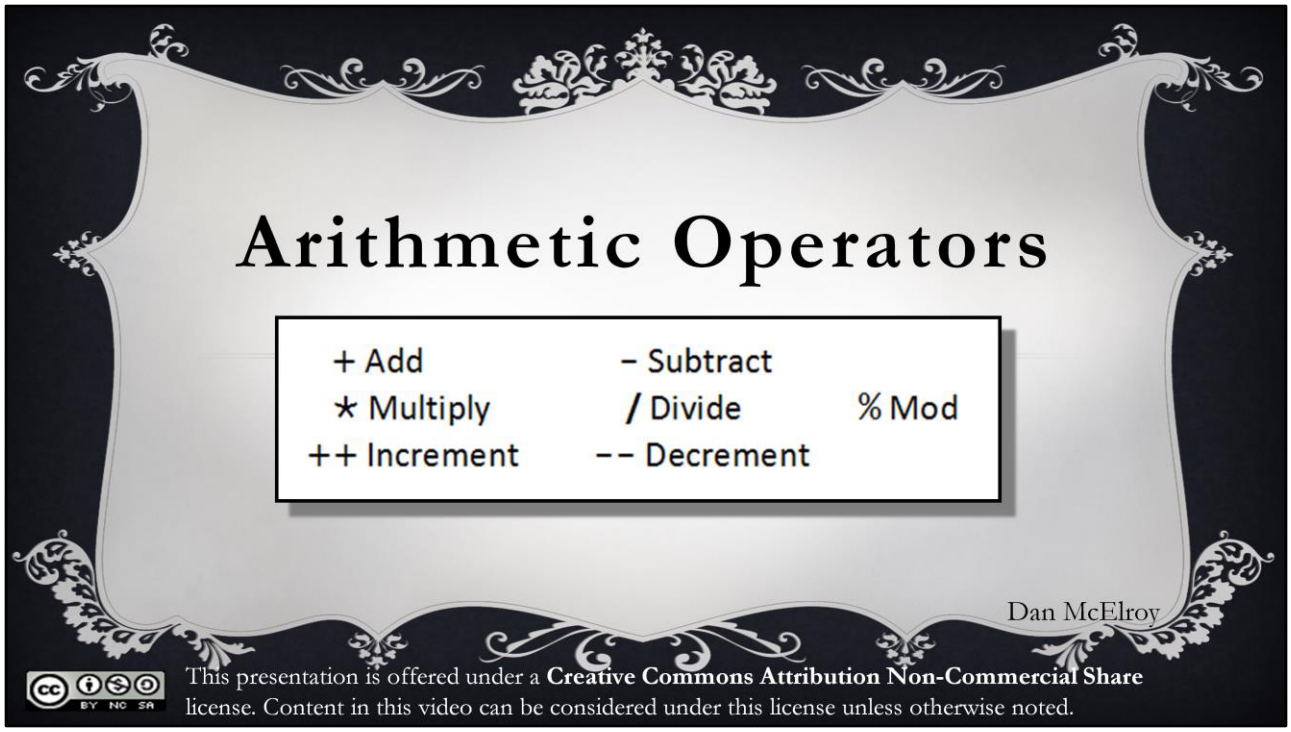
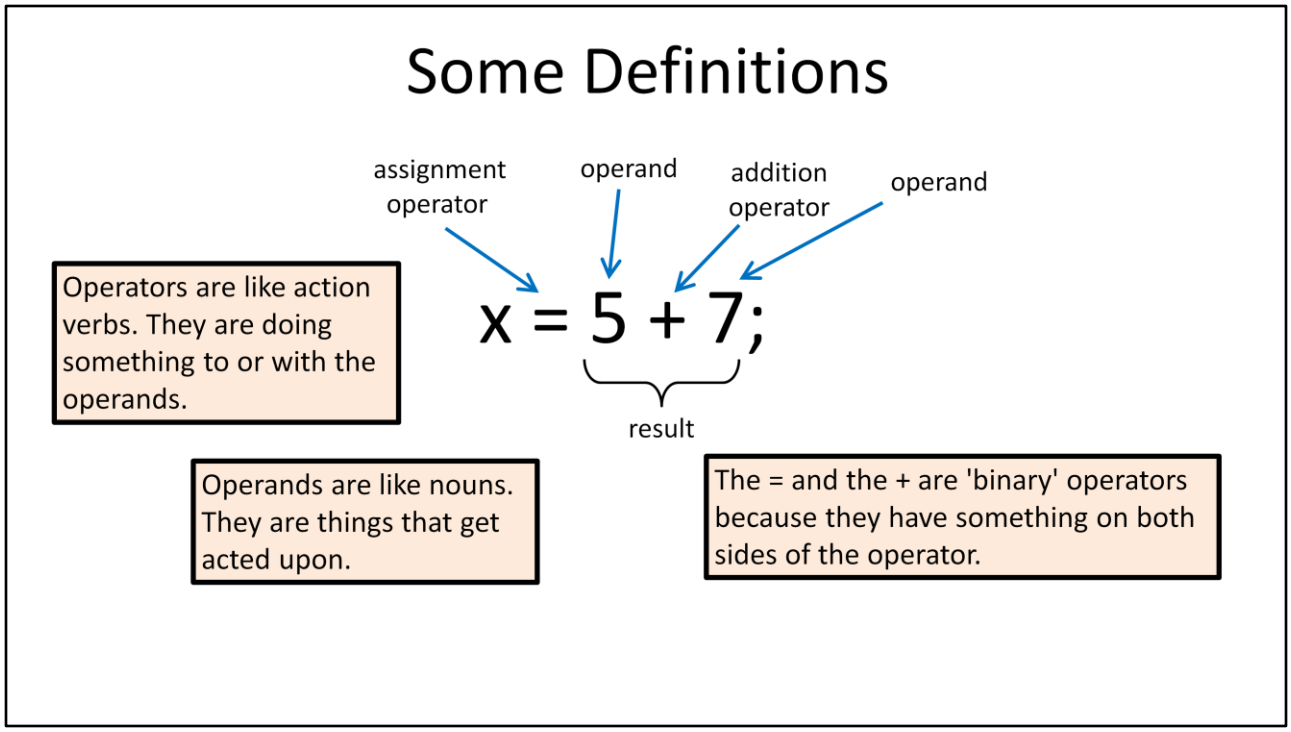


Hello programmers! This discussion covers everything you need to know about arithmetic operators and an introduction to precedence of operations. It may even cover things you didn't want to know, but as a programmer, you need to know them.

Add Subtract Multiply Divide

Add, Subtract and Multiply are easy to understand if you know basic math. Division has a few tricks that you need to understand.

Add, Subtract and Multiply are easy to understand if you know basic math. Division has a few tricks you need to understand. Also covered is the MOD operator which gives the remainder when performing a division.

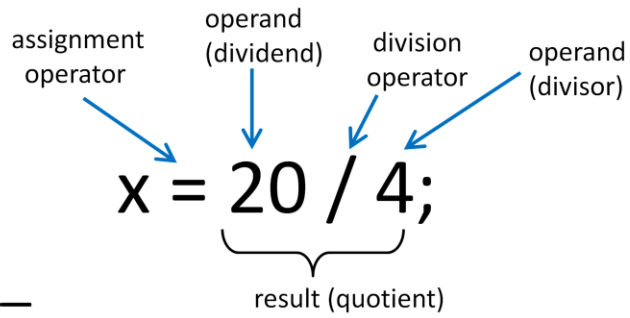


Here are some basic definitions.
Operators are like action verbs. They are doing something to or with the operands.
Operands are like nouns. They are things that get acted upon.
The = and the + are 'binary' operators. The word 'binary' is used here to indicate that there is an operand on both sides of the operator. The word 'binary' here is not to be confused with 'binary numbers' which are base 2 which have only values of 0 or 1.
Although the assignment operator = looks like the math equal-sign, we need to understand that what it does is store whatever is on the right side into a variable that is on the left side of the assignment operator. A variable MUST be placed on the left side to receive data. Not the other way around. x = 7; is OK 7 = x; will not work.

If I write x = 3; The assignment operator = is a binary operator. It requires something on each side of the =.
If I write x = -3; The negative sign - is a unary operator, while the = is a binary operator.

If I write x = 5 - 3; Then the - is the binary subtraction operator. The computation result is 2.
If I write x = 5 + -3; I get the same answer, 2. But this time, it starts with 5 and adds a negative 3. The + is the binary addition operator with an operand on both sides while the - in front of the 3 is a 'unary' operator that creates a negative 3.

Definitions for Division



$$x = 4\overline{)\,20}\;\;5$$

Division uses the terms Dividend, Divisor and Quotient. In programming, the dividend and divisor are both operands because they are being worked on by the operator, which in this case is the / divide symbol.

Mixing Data Types

	Data Types
Integer	byte, short, int, long
Real	float, double

If all of the values are integers, the computation result will be an integer.

If any of the values are real numbers, the computation result will be real.

Let's review mixing data types. Integers data types are byte, short, int and long. Real numbers have the float and double data types. Real numbers are also called floating point numbers.

If all of the data types in a math operation are integers, the result will be an integer.

If all of the data types are floating point, the result will be floating point.

Going from left to right, if we have a floating point value followed by an integer, the integer data type is promoted to a floating point data type during the computation and the result of the is a floating point value. The integer is still an integer in memory and has not changed.

Integer Result

```
int x = 5;
int y = 4;
double z = 3.0;
int a = x * y;          // a becomes a 20
int b = y * z;          // illegal to store a floating into an integer
int c = (int)(y * z);  // OK, y*z was cast into an integer
```

x, y and z are declared and initialized.

```
int x = 5;
int y = 4;
double z = 3.0;
int a = x * y;      // a becomes a 20.
int b = y * z;      // illegal.
```

It is illegal to store a double into an integer because a double uses more memory and can contain digits past the decimal. Depending on how it is configured, some C or C++ compilers will only give a warning, truncate all the digits past the decimal and store the number in the integer variable. Other settings for the C or C++ compiler will give a syntax error and not successfully compile the program. In Java, it is totally illegal to store a floating point data type into an integer, even if there are no digits past the decimal.

```
int c = (int)(y * z);  // OK, y*z was cast into an integer.
```

Here we are telling the compiler to treat y*z as an integer so that it can be stored into c.

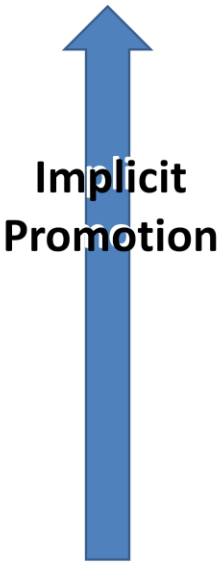
Implicit Data Type Promotion

```
double a = 5.0 * 2.0;     // both real numbers
double b = 5.0 * 2;       // the 2 was converted to real
double c =  5 * 2.0;      // the 5 was converted to real
int d = 5 * 2.0;          // can't store a real into an integer
```

Here are some examples of implicit data type promotion. For example, an integer can be promoted to a double. It is called implicit because the compiler will do it for you automatically.

```
double a = 5.0 * 2.0;          // both real numbers.
double b = 5.0 * 2;            // the 2 was converted to real.
double c =  5 * 2.0;           // the 5 was converted to real.
int d = 5 * 2.0; // can't store a real into an integer.
```

When the compiler itself is doing the data type conversion, it is referred to as IMPLICIT.



Data Type	Bytes	Range	
double	8	+/-1.7e+/-308 (~15 digits)	
float	4	+/-3.4e+/-38 (~7 digits)	
unsigned long	4	0 to 4261967295	not in Java
long	4	-2147483648 to 2147483647	
unsigned int	4	0 to 4261967295	not in Java
int	4	-2147483648 to 2147483647	
short	2	-23768 to 32767	
byte	1	-128 to 127	
char	1 or 2	-128 to 127	depends on compiler
bool	1	true/false	boolean in Java

This chart not only shows the number of bytes in each data type, their numeric range but also which data types can be 'promoted' to another data type. For example, an int can be promoted to a long, float or double but not the other way.

There are a couple of other interesting things to notice in the chart. The int and long are both the same. That is because when C was first being developed, minicomputers were popular and they typically organized their memory in 2 bytes, while bigger computers organized their memory in 4 bytes. A programmer could then specify whether a 2-byte or 4-byte int was desired by specifying either the short or long data type. Now, by default, an int is 4 bytes long.

Two more things to notice is that C++ calls the boolean data type 'bool' while Java calls it 'boolean'. C and C++ have both signed and unsigned integers. Java only has signed integers.

Loss of Precision

Depending on the compiler settings for C and C++, the compiler will ‘promote’ an integer to a double, but will either issue a **Loss of Precision Warning** or an **Error** when converting from a double to an integer. If only a warning is issued, the program will still compile and run, but you may not get the results you were expecting. Java won't automatically convert a floating-point data type to an integer.

```
int Q1 = 27.0 / 4;   // double quotient = 6.75 is
                     // converted to an integer 6 by
                     // the assignment = statement
```

Depending on the compiler settings for C and C++, the compiler will ‘promote’ an integer to a double, but will either issue a **Loss of Precision Warning** or an **Error** when converting from a double to an integer. If only a warning is issued, the program will still compile and run, but you may not get the results you were expecting.

```
int Q1 = 27.0 / 4;   // double quotient = 6.75 is
                     // converted to an integer 6 by
                     // the assignment = statement
```

Casting - Explicit Data Type Conversion

You can tell the compiler to treat one type of data as though it is another data type. For example, a piece of data that is of the double data type can be temporarily treated as though it is an integer. However, the actual data type is not changed.

```
double x = 7.8;
int    y = (int)x;
```

The value in y becomes a 7.

You can tell the compiler to treat one type of data as though it is another data type. For example, a piece of data that is of the double data type can be temporarily treated as though it is an integer. However, the actual data type is not changed. This is called EXPLICIT data type conversion because the programmer is the one that causes the conversion, not the compiler. Casting a double to an integer causes digits past the decimal point to be truncated, they are lost instead of being rounded up.

```
double x = 7.8;
int    y = (int)x;
```

The value in y **becomes** a 7. Even when 7.99999 is converted to an integer, it becomes a 7.
When you display a numeric value, it might be rounded up on the display even though in memory it has its original value. The 7.99999 might be displayed as 8.0 if the output is being formatted to show only one digit past the decimal but it is really 7.99999 in memory.

Casting - Explicit Data Type Conversion

```
double x = 7.5;
int y = (int)x;    // C-style cast good for C, C++ and Java
int y = int(x);    // like a function call
int y = static_cast<int> (x);  // for cool C++ users
```

C++ has several ways to cast from one data type to another.
There is the good old C-style cast. This works for C, C++ and Java.
C++ can use a function-style cast.
C++ can also use a static_cast. This is really easy to spot when you are looking at code.

Overflow

One important thing you need to know about each of the math operators is to make sure that you don't overflow the data type that you are using for the result.

One important thing you need to know about each of the math operators is to make sure that you don't overflow the data type that you are using for the result.

Division

Integer Division	Floating Point Division
6 r 3 4) 27	6.75 4.0) 27.0

```
int  Q1 = 27  / 4;          // integer quotient = 6
int  R1 = 27 % 4;           // integer remainder = 3
double Q2 = 27.0 / 4.0;     // double quotient = 6.75
double Q3 = 27 / 4;         // double quotient = 6.0 because 27/4 is an
                            //    integer division so the quotient is 6
                            //    and is stored as 6.0 into double Q3
```

Division works differently when working with integers and floating point numbers. When dividing integers, the result is an integer quotient and an integer remainder. For example, dividing 27 by 4 gives a quotient of 6 and a remainder of 3, also called modulo. The / divide operator produces the integer quotient and the % modulo operator produces the remainder. It takes two separate operations to get the quotient and remainder. I suppose that the percent-sign % was chosen as the MOD operator because it has a slash as part of the character. Some programming languages spell out MOD as the modulo operator. Watch out. Some fonts have a funky percent sign.

When working with floating point numbers, the quotient is always a floating point number. For example, 27.0 / 4.0 gives a quotient of 6.75 .

Be careful of this one. Dividing 27 by 4 gives an integer quotient of 6 because both operands are integers. Even though the answer is stored into a variable whose data type is double, the integer value 6 is promoted to a 6.0 and Q3 is a 6.0, not a 6.75. You can cast either the 27 or the 4 to a double, then the division results in a 6.75.

```
Q3 = (double)27 / 4;
```

Remember that if one of the operands is a floating point data type, then the other operand is promoted from an integer to a floating point data type during the computation.

The mod operator % with real numbers

Integer Division	Floating Point Division
6 r 3 4) 27	6.75 4.0) 27.0

% can't be used with real numbers in C or C++.

In Java, when the mod operator % is used on real numbers, the result of **A%B** is what is left over after you remove as many copies of **B** as possible from **A**.

Using % with Java
27.0 % 4.0 = 3.00
27.0 % 4.1 = 2.40
27.0 % 4.2 = 1.80
27.0 % 4.3 = 1.20
27.0 % 4.4 = 0.60
27.0 % 4.5 = 0.00
27.0 % 4.6 = 4.00
27.0 % 4.7 = 3.50
27.0 % 4.8 = 3.00
27.0 % 4.9 = 2.50

The mod operator can not be used with real numbers in C or C++. For example, dividing 27.0 by 4.3 gives a quotient of 6.2790698…

The % mod operator can be used in Java. Using the mod operator takes the digits past the decimal 0.2790698… and multiplies them by the divisor 4.3 to get 1.2
Although the mod operator is permitted in Java, I have never found it useful. But at least Java can say it has a mod operator for floating point numbers and C++ doesn't have one.

Dividing by Zero

The program crashes in C or C++ when attempting to divide by zero unless the division is placed inside of a **try** block, which is part of **try…throw…catch** exception handling.

In Java, the result is either `Infinity` or `-Infinity`.

The program crashes in C or C++ when attempting to divide by zero unless the division is placed inside of a **try** block, which is part of **try…throw…catch** exception handling.
In Java, the result is either `Infinity` or `-Infinity`.
Any time you are dividing, especially when dividing by a variable, it is important to place the division inside a try/catch block for exception handling to prevent the program from crashing. I won't cover exception handling here.

Auto Increment
Auto Decrement

The C language introduced the auto-increment and auto-decrement. The auto-increment adds one to a numeric variable. For example:

```
int a = 5;
a++;        // after the statement, a equals 6

int b = 10;
b--;        // after the statement, b equals 9
```

The C language introduced the auto-increment and auto-decrement. The auto-increment adds one to a numeric variable.

For example.

```
int a = 5;
a++;            // after the statement, a equals 6.
```

This produces the same effect as a = a + 1;

```
int b = 10;
b--;            // after the statement, b equals 9.
```

This produces the same effect as b = b - 1;

Pre-Increment
Post-Increment

The increment (or decrement) can be applied either at the beginning of the statement or after everything else in the statement has executed

```
int a = 5;
int x = ++a  +  10;  // a = 6, x = 16;
int b = 5;
int x = b++  +  10;  // b = 6, x = 15
```

The variable b does not get incremented to a 6 until the rest of the statement has executed. The value of b was still a 5 when 10 was added to get a 15. The 15 was stored into x before b was incremented.

The increment (or decrement) can be applied either at the beginning of the statement or after everything else in the statement has executed.

```
int a = 5;
int x = ++a  +  10;          // a = 6, x = 16;
int b = 5;
int x = b++  +  10;          // b = 6, x = 15
```

The variable b does not get incremented to a 6 until the rest of the statement has executed. The value of b was still a 5 when 10 was added to get a 15. The 15 was stored into x before b was incremented.

Arithmetic Assignment Operators

```
int a = 10;
int b = 20;
a = a + b;  // now a = 30

int a = 10;
int b = 20;
a += b;   // now a = 30
```

```
  a += b;
is the same as
  a = a + b;
```

```
  a –= b;
is the same as
  a = a – b;
```

```
  a *= b;
is the same as
  a = a * b;
```

Other arithmetic and logical assignment operators work the same way

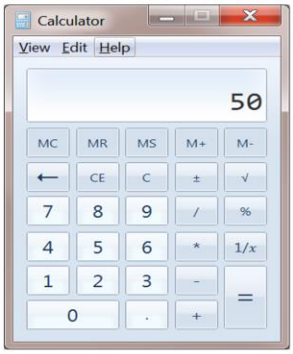
The arithmetic assignment operators create a shortcut way of combining an arithmetic operator with the assignment operator. For example a = a + b; can be shortened to a += b;

I suspect that the arithmetic assignment operators were created because when a program was compiled on the PDP-11 computer it had a similar instruction that could be used instead of two instructions. This would make the program run a little faster.

There is an arithmetic assignment operator for each arithmetic operator. Even the % mod operator has a %= arithmetic assignment operator.

Order of Evaluation

Standard Calculator Left to Right	Scientific Calculator Multiply & Divide before Add & Subtract
2 * 3 + 4 * 5	2 * 3 + 4 * 5
2*3=6	6 + 20
6+4=10	26
10*5=50	



When working with calculators, you can find a different order of evaluation depending on whether it is a 4-function calculator or a scientific calculator. Evaluating 2 * 3 + 4 * 5 is evaluated from left to right without any priority given to multiplication or division over addition. The answer for a standard 4-function calculator ends up being 50.

A scientific calculator gives multiplication and division priority over addition and subtraction. 2 * 3 + 4 * 5 irst evaluates the 2*3 which is 6, and the 4*5 which is 20 and then adds those values to get 26.

The calculator app on Microsoft Windows can select either mode of operation. Priority evaluating operations is called precedence.

Order of Evaluation

Multiply and Divide before Add and Subtract

2 * 3 + 4 * 5

6 + 20

26

Precedence	
( )	Parentheses ( )
* / %	Multiply, Divide, Mod
+ –	Add, Subtract
=	Assignment operators

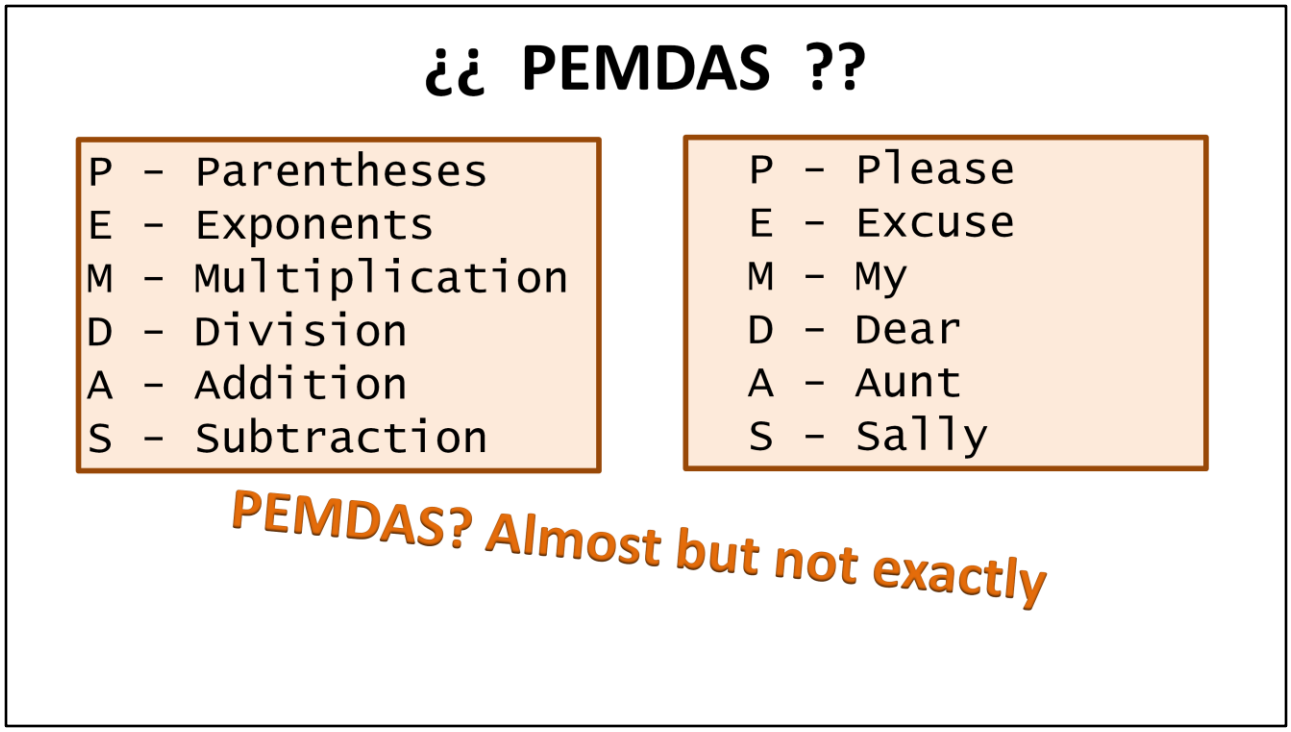
You can change the order of computation by using parentheses

```
int x = 2 * ( 3 + 4 ) * 5 ;   // answer is 70
//      2 * ( 3 + 4 ) * 5 → 2 * 7 * 5 → 70

int x = ( 2 * 3 + 4 ) * 5 ;   // answer is 50
// ( 2 * 3 + 4 ) * 5 → (6+4) * 5 → 20 * 5 → 50
```

You can change the order of evaluation through the use of parentheses. This table shows the beginning of precedence of operators. Math operators are evaluated from left to right.

2 * (3 + 4) * 5 evaluates to 70 because the addition inside the parentheses is performed before the multiplication which is outside the parentheses.



In a math class, you may have been exposed to "Please Excuse My Dear Aunt Sally" or some other version to help remember the acronym PEDMAS as the order of Parentheses, Exponents, Multiplication, Division, Addition and Subtraction. Although it still sounds cool, multiplication and division have the exact same priority and are evaluated from left to right. Addition and subtraction also have the same priority and are evaluated left to right.

If you really like acronyms that use the first letter of each word, maybe you like MATH which stands for Mental Abuse To Humans.

Using the Multiplication Operator

A math book may write

x = 5(2 + 3) where multiplication is assumed

In programming, the * operator is required

x = 5 * (2 + 3);

A math book may write.

x = 5(2 + 3) multiplication is assumed when a number precedes the open parenthesis.

In programming, the * operator is required.

x = 5 * (2 + 3); the * is placed must be placed front of the open parentheses to cause the multiplication to happen.

Parentheses ( ), not { } [ ]

Parentheses are used for two purposes

1) to control the order of operations in an expression

```
x = (5 + 2) * 6;
```

2) to supply parameters to a function, constructor or method

```
a = sqrt(25.0);
```

Curly Braces { }

Define the beginning and end of a block of code or data

Square Brackets [ ]

Used when defining or accessing the contents of an array

NOTE: Visual Basic uses parentheses with arrays

Parentheses, curly-braces and square-brackets are used for totally different things in C, C++ and Java.

Parentheses are used for two purposes.

1) to control the order of operations in an expression.

```
x = (5 + 2) * 6;
```

2) to supply parameters to a function, constructor or method.

```
a = sqrt(25.0);
```

Curly Braces { }

Define the beginning and end of a block of code or data

Square Brackets [ ]

Used when defining or accessing the contents of an array.

NOTE: Visual Basic uses parentheses for array notation.

Parentheses ( ) in Math Expressions

Fifth grade math book expression:

x = { 5 X [6 - 3] + ([7 - 5]) X 2 }

The same expression in C, C++ or Java

```
x = ( 5 * ( 6 - 3 ) + ( ( 7 - 5 ) ) * 2  );  // spaces don't matter
```

I was looking over a fifth-grade math book. They were using curly-braces, square-brackets and parentheses as though they were all parentheses. The different characters helped find the matching pair.

The editors in most IDEs can help you find a matching parentheses, curly-brace or square-bracket by selecting one of the characters. The IDE will then highlight the matching set.

Exponents

C, C++ and Java do not have an operator to calculate an exponent like some languages. The ^ operator is the bit-wise Exclusive-OR operator, not exponent. Use the **pow(x,y)** method from the Math library to raise **x** to the power of **y**.

In C or C++:

```
double a = pow(5.2);   // result = 25.0
```

In Java:

```
double a = Math.pow(5, 2);   // result = 25.0
```

Some languages like Visual Basic use the caret ^ to identify an exponent. C, C++ and Java do not have an operator to calculate an exponent. The ^ operator is the bit-wise Exclusive-OR operator, not exponent. Use the **pow(x,y)** method from the Math library to raise
x to the power of **y**.
In C or C++, make sure you use #include <math.h> for C, or #include <cmath> for C++, then you can use math functions such as double a = pow(5,2); to compute 5 to the 2nd power. The parameters in the pow function are of the double data type. Internally, pow uses logarithms to compute the result of an exponent.
In C or C++:

```
double a = pow(5.2);   // result = 25.0
```

In Java, use:

```
double a = Math.pow(5, 2);   // result = 25.0   .
```

Precedence in Expression Evaluation

Operators are placed in groups for their priority in evaluating an expression. This list is incomplete.

Level	Operator	Description	Associativity
16	[ ] ( )	Array access, parentheses	left to right
14	++ -- + -	Unary pre increment/decrement Unary Plus or minus	right to left
12	* / %	Multiply, divide, mod	left to right
11	+ -	Addition, subtraction	left to right
1	= += -= *= /= %=	Assignment	right to left

Here is a more complete list for precedence of evaluation. It shows angle-brackets, parentheses, math operators and the arithmetic assignment. It also shows the priority level for each operator. You may notice that this table starts at level 16 and skips many levels going down to level 1. There are a lot of other operators that have not been covered yet.

Another thing to notice in the table is whether the operator is processed from left to right, or right to left. For example, x = 5 + 2 * 3; is evaluated as follows:
The highest order here is the multiplication, so 2 * 3 is evaluated to be 6.
Then the next highest is the addition which becomes 5 + 6 and is evaluated to 11.
The assignment = goes right to left. The 11 on the right side of the = is placed in the variable on the left of the =. Some of the older C compilers referred to the variable on the left of the = as the lvalue.

Precedence of Unary + and -

What about **x = -3;** --or-- **x = 5 + -3;**

Level	Operator	Description	Associativity
14	+ −	Unary plus or minus	right to left

The **-** on **x = -3** is a unary operator. It does not have an operand on both sides of the operator. So, **x = -3;** places a negative 3 into x.
Then **x = 5 + -3;** adds 5 plus negative 3. The result is 2.

The unary + is only there because there is a unary minus.

Normally, spaces don't matter. **x = 5 + - 3;** is the same as **x=5+-3;**

This causes an error **x=5++3;** because ++ is a unary increment.
This is OK, **x=5 + +3;** the compiler won't confuse it with ++

What about **x = -3;** --or-- **x = 5 + -3;**

The negative sign - on **x = -3** is a unary operator. It does not have an operand on both sides of the operator. So, **x = -3;** places a negative 3 into x.

Then **x = 5 + -3;** adds 5 plus negative 3. The result is 2.
The unary + is only there because there is a unary minus.
You can put a + in front of a number, but it is not needed.
Normally, spaces don't matter. **x = 5 + - 3;** is the same as **x=5+-3;**
This causes an error **x=5++3;** because ++ is a unary increment.
However, **x=5 + +3;** is OK because the compiler won't confuse + space + with ++ .

Precedence of Multiplication and Division

What about **2 / 3 * 5** --or-- **5 * 2 / 3**

Level	Operator	Description	Associativity
12	* / %	Multiply, divide, mod	left to right

Multiplication does NOT have a higher priority than division. They have the same priority and are evaluated left to right.

5.0 * 2.0 / 3.0 is 3.333

2.0 / 3.0 * 5.0 is also 3.333

5 * 2 / 3 is 3 because 5, 2 and 3 are integers

2 / 3 * 5 is 0 because 2 / 3 is 0

What about **2 / 3 * 5** --or-- **5 * 2 / 3** .

Multiplication does NOT have a higher priority than division. They have the same priority and are evaluated left to right.

5.0 * 2.0 / 3.0 correctly computes to 3.3333 .

2.0 / 3.0 * 5.0 is 3.333 not 1.333 because the operators are evaluated left to right.

5 * 2 / 3 is 3 because 5, 2 and 3 are integers. 10 divided by 3, the quotient is 3. The remainder of 1 is not used.

2 / 3 * 5 is 0 because 2/3 is evaluated first. The quotient is 0 and then multiplied by 5

Looking at the floating point evaluation again, 2.0 / 3.0 * 5.0 is 3.3333, not zero.

Commutative Property

In math, the commutative property states that the answer will be the same when you move things around. For example:

2 + 3 gives the same answer as 3 + 2

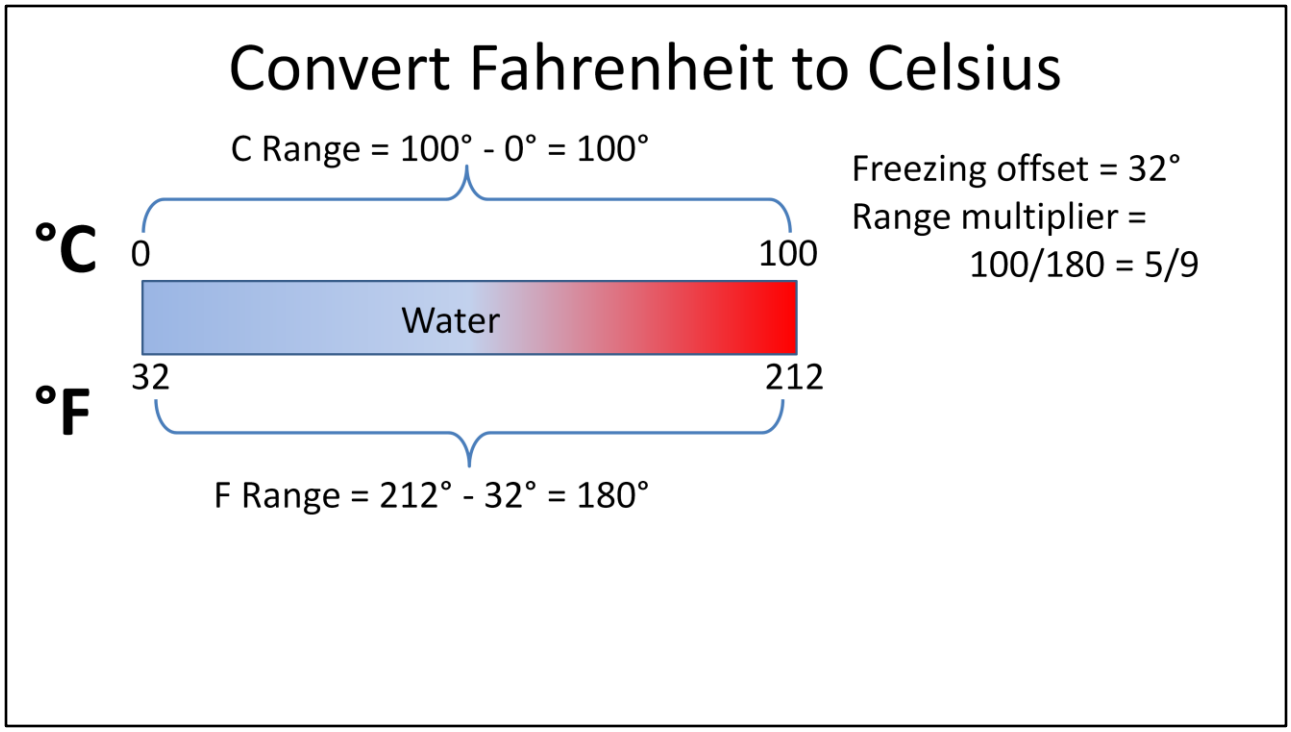
2 * 3 gives the same answer as 3 * 2

In math, the commutative property states that the answer will be the same when you move things around. For example.

2 + 3 gives the same answer as 3 + 2.

2 * 3 gives the same answer as 3 * 2.

Let's see if it always works this way when we write a program, especially when working with integers.



Now lets look at how we convert a temperature expressed in Celsius to Fahrenheit. They don't share a common zero point. Fahrenheit came first. Daniel Fahrenheit chose 0° to be the point at which a solution of salt water froze. 212° just happened to be the number that showed up when water boiled.

Celsius belongs to the metric system and is named after Anders Celsius. Zero degrees is where plain water freezes at sea level, and 100° is where water boils at sea level. Most countries in the world use the metric system.

The freezing point is offset by 32° between Fahrenheit and Celsius. The range between freezing and boiling is 100°C and 180°F. 180/100 reduces to 9/5. The formulas are.
F = C * 9/5 + 32 .
C = (F - 32) * 5/9 .

Although you can easily lookup the formulas for converting between Celsius and Fahrenheit, you should understand how the formula is derived. A similar exercise may be required when taking numbers in a table and trying to determine the position they would be placed on a chart on a computer screen. You would need to compute to location of pixels on the screen by determining the range of numbers, the range of pixels used on the screen, andt the offsets for the starting value of the numbers and starting pixel position on the screen.

Convert Fahrenheit to Celsius

$C = (F - 32)\ 5/9$

$C = 5/9\ (F - 32)$

$(F - 32) / 1.8 = C$

$\frac{F - 32}{9} * 5 = C$

$F = C * 9/5 + 32$

$F = 9/5 * C + 32$

$C * 1.8 + 32 = F$

$\frac{C * 9}{5} + 32 = F$

Here are several ways to display the conversion formulas. For now, I am only going to be using the first formula that converts Fahrenheit degrees to Celsius.

Convert Fahrenheit to Celsius

```
Celsius = 25
Celsius = 25
```

```
int tempFahrenheit = 78;
int tempCelsius = 0;

tempCelsius = (tempFahrenheit - 32) * 5 / 9;

cout << "Celsius = " << tempCelsius << endl;
printf ("Celsius = %d\n", tempCelsius);
```

Using all integers and setting the value 78 for Fahrenheit, the result is 25°C.
78-32 is 46, multiplied by 5 is 230, an integer divide by 9 gives an integer quotient of 25 .
The cout statement is for C++.
The printf statement is for C. The %d in the format specifier string indicates that an integer is to be displayed.
System.out.printf can be used for Java.

Convert Fahrenheit to Celsius

```
double tempFahrenheit = 78;
double tempCelsius = 0;

tempCelsius = 5 / 9 * (tempFahrenheit - 32);

cout << "Celsius = " << tempCelsius << endl;
printf ("Celsius = %f\n", tempCelsius);
```

```
Celsius = 0
Celsius = 0
```

Watch out for this one. According to the math laws of Commutation, it should not matter if we put the multiplication of 5/9 before or after the parentheses for (F - 32). But since 5 and 9 are both integers, the quotient of dividing 5 by 9 is 0, remainder 5. It is the quotient that is being used, therefore we get zero times (78-32) and tempCelsius ends up being a zero regardless of the value for tempFahrenheit.

Convert Fahrenheit to Celsius

```
double tempFahrenheit = 78;
double tempCelsius = 0;

tempCelsius = 5.0 / 9 * (tempFahrenheit - 32);

cout << "Celsius = " << tempCelsius << endl;
printf ("Celsius = %f\n", tempCelsius);
```

```
Celsius = 25.55556
Celsius = 25.55556
```

We can fix this by either using a floating point value for at least one of the numbers 5 or 9, or we could even cast one of the numbers 5 or 9 to a double. Then we can get the correct answer 25.55556 regardless of whether 5.0/9.0 is multiplied before or after the parentheses.

I hope you found this presentation informative even if it seemed to violate some of the rules of mathmatics. As programmers, it is not our option to agree or disagree with how the innards of a computer work but to understand how it works and the write programs to make it do what we want it to do.

I'll see you around. Until then, Bye.