Expressions

Arithmetic expressions in C++ are based on normal algebra and so will look quite familiar.

Nevertheless, there are important differences. For instance, computers implement different arithmetic for integers and doubles.

**Double Arithmetic**

We characterize C++ operators as being

1. Unary (one operand): + –
2. Binary (two operands) + – * /

where * and / signify respectively multiplication and division

In double arithmetic, all operands are double and the result is always double.

```cpp
int main(){
    double x = 2.4;
    double y;
    y = x*x + 2.0 * x + 1.5;
    cout << "y is " << y << " when x is " << x << 'n';
    return 0;
}
```

Notice the similarity between the C++ equation and the algebraic equation it represents.

**Integer Arithmetic**

The integer operators are

1. Unary (one operand): + –
2. Binary (two operands) + – * /

where * and / signify respectively multiplication and division

In integer arithmetic, all operands are int and the result is always int.

```cpp
int main(){
    int i = 5;
    int j = 3;
}
```

The results are what you would expect until you get to division.

In normal arithmetic 5/3 would be 1.6. However, the result of an integer operation is always an integer.

So why not 2?

C++ int arithmetic doesn't round. Instead it gives us two int division operators.

1. / gives us the integer part.
2. % gives us the remainder

**Conversions**

Ints and doubles are different types. Computers can

- do double arithmetic
- do int arithmetic

They can't do mixed arithmetic. Instead, they convert from one type to the other.

```cpp
int main(){
    double x = 2.4;
    double y;
    y = x*x + 2 * x + 1;
    cout << "y is " << y << " when x is " << x << 'n';
    return 0;
}
```

In the example evaluation of the term 2 * x requires an implicit conversion.

The 2 is automatically converted to a double yielding 2.0 * x and then a double multiply is called.
A programmer can also force a conversion explicitly by doing a type cast

```cpp
int y = 2 * (int) x;
```

Here the operator `(int)` is an int type cast applied to the double variable `x` coercing it to an int.

This is known as a downcast because precision is lost.

```cpp
int main()
{
    double x = 3.7;
    int i;
    i = 3 * (int) x;
    return 0;
}
```

### Rounding

When a double is converted to an int, it is not rounded, it is truncated.

The fractional part is discarded.

This is consistent with the integer / operator’s behaviour.

Here’s how you round positive nos.

```cpp
int main()
{
    double x = 3.4999;
    double y = 3.6;
    cout << "Force " << x << " to int: " << (int)x << endl;
    cout << "Force " << y << " to int: " << (int)y << endl;
    cout << "Round " << x << " to int: " << (int)(x + .5) << endl;
    cout << "Round " << y << " to int: " << (int)(y + .5) << endl;
    return 0;
}
```

The technique has to be amended for negative nos. We’ll show you how later.

### The Assignment Operator

The assignment operator is for storing a value in a variable:

```cpp
x = expression;
```

The value of the expression on the right is computed and stored in the variable specified on the left (`x`).

### It is not an equal sign!

expression may contain `x`—the ‘old’ value is used:

```cpp
x = x + 1;  // Increase the value in x by 1.
```

Always think of the = as a replacement operator

```cpp
x <- x+1;
```

One way to think of this is that

1. On the right side of the equation we are reading the current value of x
2. The assignment operator causes the new value to be written into the bin specified on the left (which is again x).

### The Order of Evaluation

The order of evaluation in compound expressions is determined by

1. Parenthesis (`()`
2. Precedence
Expressions

<table>
<thead>
<tr>
<th>unary</th>
<th>* /</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>``, <code>-</code></td>
<td>``, <code>/</code></td>
<td><code>-</code>, <code>+</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Highest (evaluated first)</th>
<th>Lowest (evaluated last)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>5 + 6 * 2</code></td>
<td><code>10 / (2 * 5)</code></td>
</tr>
<tr>
<td><code>(5 + 6) * 2</code></td>
<td><code>10 / 2 * 5</code></td>
</tr>
<tr>
<td><code>5 + -6 * 2</code></td>
<td><code>5.0 / 2</code></td>
</tr>
<tr>
<td><code>5.0 / 2.0</code></td>
<td><code>5 / 2</code></td>
</tr>
</tbody>
</table>

```cpp
int main() {
    cout << "5 + 6 * 2 = " << 5 + 6 * 2 << endl;
    cout << "(5 + 6) * 2 = " << (5 + 6) * 2 << endl;
    cout << "10 / (2 * 5) = " << 10 / (2 * 5) << endl;
    cout << "5 + -6 * 2 = " << 5 + -6 * 2 << endl;
    cout << "5.0 / 2.0 = " << 5.0 / 2.0 << endl;
    cout << "5 / 2 = " << 5 / 2 << endl;
    return 0;
}
```

This page last updated on Friday, January 16, 2004