Expressions

Expressions are simply statements that can be evaluated to calculate some value. For instance, 5 + 2 \* 8 is an expression that evaluates to 21, and speed \* time + 5 is an expression that evaluates to 5 more than speed \* time.

There is no limit to how complex an expression can be. However, it is a good idea to break up a long expression into multiple steps when doing so would improve its readability or ease of use.

An expression consists of at least one **operand** (value) and one or more **operators** (symbols). In the first example above, the numbers 5, 2, and 8 are the operands, and the symbols + and \* are the operators.

There are three types of operands:

1. Variables, which are values represented by identifiers (such as speed and time in the example above) where the value can be changed.
2. Constants, which are values represented by identifiers where the value cannot be changed, such as e and PI.
3. Literals, which are values not represented by an identifier, such as 5, ‘A’, and “hello world”.

# Operators

## Assignment Operators

The assignment operator sets the value of the operand on the left to whatever value the expression on the right evaluates to.

|  |  |
| --- | --- |
| **Expression** | **Result** |
| int foo = 3; | after this statement, foo equals 3 |
| int bar = 4 + 9; | after this statement, bar equals 13 |
| foo = bar \* 3 + 14; | after this statement, foo equals 53 |

It is very common to modify the value of a variable with some simple arithmetic, for instance wanting to increase the value of a variable by five. Instead of having to write this out like foo = foo + 5, there are **compound assignment** operators that give us a shortcut.

|  |  |
| --- | --- |
| **Assignment Expression** | **Compound Assignment Expression** |
| foo = foo + 5 | foo += 5 |
| foo = foo – 5 | foo -= 5 |
| foo = foo \* 5 | foo \*= 5 |
| foo = foo / 5 | foo /= 5 |

## Increment and Decrement

Increasing or reducing an integral variable by one is so common that there is also a special operator to make this easier. These operators only work with integral types, so you can’t use this method on a float or double variable.

|  |  |
| --- | --- |
| **Expression** | **Result** |
| int foo = 2; | foo is now 2 |
| foo++; | foo is now 3 |
| ++foo; | foo is now 4 |
| foo--; | foo is now 3 |
| --foo; | foo is now 2 |

As you can see, there are two versions of the increment and decrement operators, one that goes in front of the variable and one that goes behind it. The one before the variable is a **prefix** operator, and it changes the value of the variable before it is evaluated in the expression. The one after the variable is a **postfix** operator, and it changes the value of the variable after the expression is evaluated.

|  |  |
| --- | --- |
| **Code** | **Result** |
| int a = 6;  int b;  int c; | a is 6  b is undefined  c is undefined |
| b = a++; | a is 7  b is 6  c is undefined |
| c = ++b; | a is 7  b is 7  c is 7 |

## Arithmetic Operators

These are the simple mathematical operators that we use all the time. They are used to calculate a value that will either be used in an expression or assigned to a variable.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| + | Addition | int foo = 2 + 8; |
| - | Subtraction | int foo = 5 – 1; |
| \* | Multiplication | int foo = 3 \* 9; |
| / | Division | int foo = 15 / 5; |

## Comparison Operators

Comparison operators, as their name suggestions, compare two different values. They result in either true or false. (In Java, these will be the boolean variable type, while in C the values will actually be integers that will be 0 for false and 1 for true.)

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Result is True** | **Result is False** |
| < | Less than | 5 < 8 | 8 < 5 |
| > | Greater than | 9 > 4 | 4 > 9 |
| == | Equal to | 2 == 2 | 2 == 3 |
| <= | Less than or equal to | 2 <= 3, 3 <= 3 | 4 <= 3 |
| >= | Greater than or equal to | 4 >= 3, 3 >= 3 | 2 >= 3 |
| != | Not equal to | 7 != 5 | 7 != 7 |

## Logical Operators

Logical operators are used to combine two or more comparison operators. For example, if we want to know whether a variable is both less than ten and does not equal zero, or whether it is either less than five or greater than fifteen. They can be used to combine any number of comparison operators: if we wanted to, we could check for a variable which is not ten or eleven, is less than twenty, and is greater than five or is equal to fifteen. Or something even more complicated.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Result is True** |
| && | AND – Both comparison operators must be true | 2 < 3 && 3 < 4 |
| || | OR – One of the comparison operators must be true | 2 < 3 || 2 > 3 |
| ! | NOT – The comparison operator result will be flipped | !(2 > 3) |

## Operator Precedence

All operators have a certain precedence, which controls the order in which an expression is evaluated. For instance, multiplication or division operations are always calculated before addition and subtraction.

While it can be useful to know the precedence of various operators, there is nothing wrong with simply using parentheses to explicitly control the order of your expressions. This is more obvious and readable for anyone looking at your code.

# Exercises

1. What is the value of c? *[ 13 ]*

int a = 5;

int b = 7;

int c = ++a + b++;

2. What is the value of c? *[ 34 – the pre-increment on b is evaluated before anything else ]*

int a = 4;

int b = 9;

int c = (a \* b) - ++b + a;

3. What is the value of d? *[ 13 ]*

int a = 4;

int b = 10;

int c = b / a + ++a – b++;

int d = c + a + b;

4. What is the value of d? *[ 3000 ]*

int a = 10;

int b = 2;

int c = a / b + --a \* ++b;

int d = c / b \* c \* ++a;