# The Assignment Statement

## Use

To assign a value to a variable. Another way to think about this is that you are storing a value in a particular memory location, and the name of the variable, or its *identifier*, is a human-friendly way of indicating *which* location in memory. When you specify the name, the computer does the work of figuring out where in memory the current value of the variable with that name is stored.

## Form

```
variable = expression;
```

where `variable` is a valid C++ identifier, the assignment operator (`=`) is an equal sign, and `expression` is either:

- a constant (e.g. `4, 2.89, 'r'`)
- another variable to which a value has previously been assigned or read in from user input, or
- a formula to be evaluated containing constants and/or variables, e.g.

```
(score / maxScore) * 100
basePrice - discount + (basePrice * SALES_TAX_RATE)
```

where the type of the value of the expression on the righthand side is compatible with the type of the variable on the lefthand side. Note that the assignment statement must end with a semi-colon.

## Action

Evaluates the expression on the righthand side and assigns the resulting value to the variable on the lefthand side. Note that a variable may appear on both the lefthand side and righthand side of an assignment statement. When this occurs, the original value of the variable is used to evaluate the righthand side expression and the resulting value is then assigned to that variable, destroying the original value of the variable. For example, in the second of the following two statements:

```
int counter = 0;
counter = counter + 1;
```

the original value of `counter`, which is `0`, is used to evaluate the expression on the righthand side. The resulting value of `1` is then assigned to `counter`. (Notice that, as the first of these statements indicates, it is possible to declare a variable and assign it a value, or *initialize* it, in a single statement.

**NOTE:** An assignment statement of the form:

```
indexOfLargest = i;
```

stores the *current value* of `i` in the variable `indexOfLargest`. It does *not* mean that `indexOfLargest` will have the same value as `i` from now on.

## Examples

Suppose I have the following constants and variables:

```
const int MAX_SCORE = 90;
const double NORMAL_TEMP = 98.6;
const char MAX_LETTER_GRADE = 'A';
const int MAX_NUM_SCORES = 3;

double thisPercentGrade;
       temperature,
       degreesOfFever,
       percentAboveNormal;

char grade;

int baseScores[MAX_NUM_SCORES];  // an array holding 3 integers
int extraCredit[MAX_NUM_SCORES]; // another array holding 3 integers
int thisBaseScore,
    thisExtraCredit,
    thisTotalScore;
```

Then the following assignment statements are legal:

```
temperature = 101.5;
degreesOfFever = temperature - NORMAL_TEMP;
percentAboveNormal = (degreesOfFever / NORMAL_TEMP) * 100;

grade = MAX_LETTER_GRADE;

baseScores[0] = 83;
extraCredit[0] = 2;
thisBaseScore = baseScores[0];
thisExtraCredit = extraCredit[0];
thisTotalScore = thisBaseScore + thisExtraCredit;
thisPercentGrade = (thisTotalScore * 100.0) / MAX_SCORE;
```

Notice a few things:

- I can declare a bunch of variables of the same type by separating their identifiers with commas.
- I assigned values only to the first items in the `baseScores` and `extraCredit` arrays; the other values in these arrays are garbage at this point. Normally, I would have a loop that would either initialize each item in the array to a particular initial value or would get a value from the user for each item in the array.
- Since both `totalScore` and `MAX_SCORE` are integers, I had to multiply one of them by a real-valued amount to turn it into a double, so that when `percentGrade` was calculated, real division was performed instead of integer divison. Since I knew I wanted to multiply the result of the division by 100 to get a percent, it made sense to multiply `thisTotalScore` by `100.0` before I did the division. Note that this would have been unnecessary if I had declared `MAX_SCORE` to be a double:

```
const double MAX_SCORE = 90.0;
```