1 C's Operator Shorthands

1.1 The General Shorthand Forms

Operators are often used in programs in the following ways:

```
i = i + 1;
j = j * 2;
k = k - 5;
```

In C, there is a way for expressing such operations in a more shorthand way. The following three operations are perfectly equivalent to the three operations above:

```
i += 1;
j *= 2;
k -= 5;
```

In other words, for all binary operators, the following two forms are equivalent:

```
a = a <op> b;
a <op> = b;
```

1.2 The Special Increment and Decrement Operators

Apart from the above shorthands, there are the special increment and decrement operators. If the one mostly used operation on a variable should be named, it is probably adding one to that variable. For this purpose, C has the increment operator, ++. For example,

```
i++;
```

increments the value of the variable `i` by one. This is called the *post-increment* operator. The following, however, also increments the value of `i` by one:

```
++i;
```

In this form, ++ is called the *pre-increment* operator. In this bare use, both forms are equivalent. The difference is apparent when the operators are used in the middle of an expression. If the pre-increment form is used, the value of the variable is updated *before* the expression is evaluated. But, if the post-increment form is used, the value of the variable is updated only *after* the expression is fully evaluated.

There is a matching pair of *pre-decrement* and *post-decrement* operators, which both use −−. They decrease the value of the variable they act on by one, and the meaning of the pre- and post- forms are symmetric to that of the increment operator.

The following example demonstrates the difference between the pre- and post-forms of these operators:

```
int a = 1, b, c;

b = a++;
c = ++a;
```

After this program fragment is executed, `b` will have the value 1, while `c` will have the value 3.

2 Automatic Type Conversions

In many cases, an operator will act on two operands of different types, or a value of one type will be assigned to a variable of a different type. In these cases, an automatic type conversion will occur.

One case is when an integer type is converted to another integer type. If a smaller type is being converted to a larger integer type, there is no problem, the variable will keep its value without problems. However, when a longer integer type gets assigned to a shorter integer type, if the value will not fit, the result is the value with the higher-order bits lost. This is usually garbage, and should be carefully avoided.

When one floating-point type gets converted to another floating-point type, if the converted type is larger, there is no problem. The other way around, there is loss of precision, and possibly overflow, in which case the result will have the value of infinity.

When converting from an integer type to a floating-point type, there *may* loss of precision. The other way around, the number loses its fractional part, and the result may be garbage if it will not fit into the given variable.

In an assignment, the value will always be converted to the type of the variable. One should really be careful about loss of precision and garbage production with such automatic conversions.

For binary operators, there are standard promotion rules. First, all integer types smaller than a regular `int` will be converted to `int`s. Then, if there is a mismacth between the two operands, the shorter type will be "promoted" to the longer type and the operation will be carried out. The folating-point types are considered to be "longer" than the integers, so the promotion goes in the direction of the arrows below:

$$\texttt{int} \to \texttt{long} \to \texttt{float} \to \texttt{double} \to \texttt{long double}$$

3 Type Casts

Aside from automatic conversions, it is possible to change the type of an expression momentarily during evaluation. In order to do that, you place the type name in a pair of parentheses, and place that in front of the expression whose type you want to modify. Here is an example:

```
int a, b;
float c;

a = 1;
b = 2;

c = a/b;
```

When this fragment is executed, the variable `c` will have the value of 0.000, probably not what you expected. This is because both operands are integers, in which case integer division is carried out. If we want the "expected" answer, here is one solution:

```
int a, b;
float c;

a = 1;
```

```
b = 2;

c = (float)a/(float)b;
```

In this case, the variable `c` will have the value of 0.500, as you would expect. This is because we converted both `a` and `b` to `floats` before performing the divison.

There are two points: First, the type cast does not change the type of the variable here. The variables `a` and `b` are still integers. Second, we could have just used a cast on *one* of the variables, since the other one would automatically be promoted according to the given rules.

4 Types of Constants

Constants are the numbers that you type into a C program such as 12, -1222, or 124.03. But what are the types of these numbers?

For integers, the default rule is that the type of a constant is the smallest type it will fit in. But, it is possible to modify this default behavior by appending a letter to the end of the number. The letter `U` appended to an integer constant will make the type of that integer `unsigned`, while an `L` appended to the same will make it a `long`.

For floating-point types the rule is different. Anything you type which contains a decimal point is a `double` by default. Again, it is possible to modify this behavior by appending letters to the end of the numeric constant. In this case, the letter `F` causes the constant to be a `float`, while an `L` makes the floating point constant a `long double`.

5 The Conditional Operator

There is one ternary (three-argument) operator in C, which is the conditional `(?)` operator. The general form of this operator is as below:

```
<condition> ? <value-if-true> : <value-if-false>
```

If the condition is true, the value of the expression is the first value after the operator. If the condition is false, the value is the second value after the operator. In other words, the following two program fragments are equivalent:

```
if (a > 0) {
  b = 5;
} else {
  b = 7;
}

b = a > 0 ? 5 : 7;
```

This operator is normally used as a shorthand notation in a single line. It should not be abused, since expressions containing the conditional operator are hard to follow.

6 Exercises

6.1 Exercise 1

```
#include <stdio.h>

int main(void)
{
  int numerator=1;
  int denominator=1;
  int limit;
  int i;

  printf("Please enter the numerator: ");
  scanf("%d", &numerator);
  printf("Please enter the denominator: ");
  scanf("%d", &denominator);

  printf("%d/%d", numerator, denominator);

  limit = numerator > denominator ? denominator : numerator;

  for(i=2; i<=limit; i++) {
    if (numerator % i == 0 && denominator % i == 0) {
      numerator /= i;
      denominator /= i;
    }
  }

  printf(" = %d/%d\n", numerator, denominator);

  return 0;
}
```

The intended use of the above program is to reduce a fraction to its lowest terms. Type it in to your computer and run it. Check that the output is correct for, say (5, 10).

Now, for each of the following numerator-denominator pairs, run the program once more: (3, 6); (30, 42); (324, 567); (256, 512). Note for each pair whether or not the code correctly reduces the fraction to its lowest terms.

Find out why the program fails in some cases, and does not fail in others. Fix the program.

6.2 Exercise 2

An algorithm (or, a game) is defined by the following rules:

1. Start with a positive integer.
2. If the number is 1, stop.
3. If it is divisible by 2, divide it by 2. Go back to step 2.
4. If not, multiply the number by 3 and add 1. Go back to step 2.

As an example, assume we start with 3. Here is the sequence one obtains:

3 10 5 16 8 4 2 1

In this case, it took 7 steps to end. How many steps it takes to reach 1 depends on the number we start with. For instance, if the "seed" is a power of 2, the number of the steps will be very few (compared to similarly sized numbers).

The following code is written to do just that, count the number of steps, and print the numbers.

```
#include <stdio.h>

int main(void) {
  int number=3;
  int steps=0;

  printf("Please enter the seed: ");
  scanf("%d", &number);

  while (number != 1) {
    printf("%d ", number);
    steps++;

    if (number % 2 == 0) {
      number /= 2;
    } else {
      number = 3*number + 1;
    }
  }

  printf("1\n");
  printf("Steps: %d\n", steps);

  return 0;
}
```

Type the program in. Run the program for various values of the initial number. Then, modify the program to find which integer less than 1000 takes the most number of steps to reach 1.