Chapter 8 – Loose Ends

We need to cover some loose ends that there wasn't enough room for in the other chapters.

We start with math operators:

```
Remainder:          dividend % divisor;
Pre-increment:      ++variable;
Post-increment:     variable++;
Pre-decrement:      --variable;
Post-decrement:     variable--;
```

The pre- and post- are identical unless used with other operators. Here are some examples:

```
int d1 = 17 % 4;    // d1 equals 1; positive remainder for positive dividend
int d2 = -17 % 4;   // d2 equals -1; negative remainder for negative dividend
int d3 = -17 % -4;  // d3 equals -1 still; sign of divisor has no effect
int x = 10;
int y = x++ * 5;    // y now equals 50; x gets incremented after it is used
x = 10;
y = ++x * 5;        // y now equals 55; x gets incremented before it is used
// in both cases, x equals 11 afterwards
```

When you change integers to floats or vice-versa, you have to be aware of the effects:

```
double result1 = 10/3;
       // answer is 3.0 because it is an integer divide before converting
double result2 = (double)10/3;
       // answer is 3.33333 because 10 is converted to 10.0 before divide
int result3 = (int)(11.0/3.0);
       // answer is 3 because it is truncated, not rounded up
int result4 = (int)(-11.0/3.0);
       // answer is -3 because it is truncated, not rounded down
```

Exercise 8.1: Recall that floating point values have limited precision. In this exercise, we will test what operations will give more or less precision. Modify Exercise_8_1. Multiply or divide "startValue" by powers of 10 until you get a reasonable value for the "product answer" and record the value of "startValue". Continue until you get a reasonable value for "addition answer" and record the value of "startValue". If there are no reasonable values, make a guess as to why.

Exercise 8.2: Modify Exercise_8_1 again. Change all the *float*s to *double*s. Remove the "f" from "startValue" and "increment". Multiply or divide "startValue" by powers of 10 until you stop getting a reasonable value for the "addition answer" and record the value of "startValue". Continue until you stop getting a reasonable value for "product answer" and record the value of "startValue". If all the values are reasonable, make a guess as to why.

Ternary operator is simply a compact *if..else* statement. Unlike the *if..else* statement, it is meant to be written on a single line. Here is the syntax:

```
condition ? true statement : false statement;
```

Here is an example of how to use it:

```
int max = a >= b ? a : b;  // this finds the larger value between a and b
int max2;                  // using the if..else requires more code
if (a >= b)
       max2 = a;
else
       max2 = b;
```

However a better use is with prints or strings because the ternary operator is compact:

```
printf("Student A has %s\n", (grade >= 50 ? "passed" : "failed"));
printf("Student B has ");        // the ternary is shorter and easier to read
if (grade >= 50)
       printf("passed\n");
else
       printf("failed\n");
```

Alternately, you can do:

```
printf("Student A has ");
grade >= 50 ? printf("passed\n") : printf("failed\n");
```

Exercise 8.3: Create a program that asks at least 3 true/false questions. Use the ternary operator to output whether the user is correct or not. Remember to fully test your program.

You will notice that we have used '*break*' statements in previous code. This statement will break out of the inner most loop (if they are nested); meaning the loop will terminate regardless of the stop condition. This works for the *for*.., *while*.., or *do..while* loops. Sometimes you will want to execute only half of a loop block; rather than using an indented *if*.. statement, it is cleaner to use the *continue* statement. You will see an example of this in Exercise_8_4.

Switch statements are a replacement for cascading *if..elseif* statements where the variable is the same and it is an equality test. Depending on the test values, the compiler may optimize the code better than the cascading *if..elseif*. A typical use is for state machines, for example:

```
enum GAME_STATE {PLAYING, DELAYED_PENALTY, DELAYED_OFFSIDE, DELAYED_ICING,
POWER_PLAY, FACEOFF, TIMEOUT};
GAME_STATE gameState;
switch(gameState) {
       case DELAYED_PENALTY:
       case DELAYED_OFFSIDE:
       case DELAYED_ICING:
              checkWhistle();     // want to continue with justPlay()
       case PLAYING:
              justPlay();
              break;
       case POWER_PLAY:
              powerPlay();
              break;
       case FACEOFF:
              dropPuck();
              break;
       case TIMEOUT:
              checkTimeoutOver();
              break;
```

```
        default:                    // handle any unexpected states
                break;              // usually when new states are added
}
```

Notice the use of the same logic for multiple cases, eg. the "delayed" cases. So in addition to ending loops, a *break* is normally used at the end of a case, however it is not required. If the *break* is left out, the code will continue to the case below, eg. the delayed cases again. A source of unexpected errors is that some compilers need to have an individual case enclosed in a block statement if any local variables are declared; this to avoid local variables that exist in some cases, but not others and the case statement falls through rather than breaks. It is good practice to explicitly comment code that intentionally falls through so that other programmers don't think it's an error. The above code can be done with the cascading (or nested) *if..elseif*:

```
If (gameState == DELAYED_PENALTY || gameState == DELAYED_OFFSIDE ||
DELAYED_ICING) {
        checkWhistle();
        justPlay();
} else if (gameState == PLAYING) {
        justPlay();
} else if (gameState == POWER_PLAY) {
        powerPlay();
} else if (gameState == FACEOFF) {
        dropPuck();
} else if (gameState == TIMEOUT) {
        checkTimeoutOver();
} else {
}
```

Once you see the switch statement more often, you will probably prefer it over the cascading *if..elseif*.

> **Exercise 8.4: Your task is to create a simple calculator. Use Exercise_8_4 as a template if you need. It should perform add, subtract, multiply, divide, and clear. Add more functions for bonus marks. You need to be careful with divide.**

Notice that in the Exercise_8_4 template that the string libraries are used. We will now start using them instead of using our own. These are more powerful and contain many more functions.

> **Exercise 8.5: Please include a link to the string library documentation for your language. Insert it as a comment at the top of your Exercise_8_4 file. Also name 2 string functions that were not part of your exercises; then briefly explain what they do and what the parameters represent (in your own words).**