# Incorporating Decisions (if) and Repetition (for/while)

## If statement

In many programs, we may not always want to perform the same set of steps. Instead, depending on some decision, we may want to carry out one set of steps or skip them. One simple example is calculating the cost of an item at the grocery store. Some items that are deemed to be essential foods (fresh fruit, for example) are not taxes, but other items that aren't necessary (for example chips), are taxed. Thus, we only want to add tax to the cost of an item if it's a taxed item.

In C++, the if statement is used for running different directions depending on a certain condition. The most simple form of an if statement is:

```
if (boolean expression) {
    stmt1;
    stmt2;
    …
    stmtn;
}
```

The parentheses after the if are required, and inside those parentheses, one can have any Boolean expression, of type bool or type int. (If an integer is in the expression, any non-zero number is treated as true and 0 is treated as false.) In short, if the Boolean expression evaluates to true, then all the statements inside the block (indicated by the curly braces) are executed. If the Boolean expression evaluates to false, then all the statements within the curly braces are skipped. The curly braces are only necessary if there is more than one statement to be placed "inside" the if, but it's good practice to always use them to avoid subtle errors.

Simple Boolean expressions are formed using the conditional operators:

```
>, >=, <, <=, ==, !=
```

To form a valid expression, place expressions to both the left and right of these operators. The meaning of the first four are straightforward. The fifth operator ***checks*** for equality between two expressions and the last is the "not equal to" operator. None of these, on their own, will modify the value of any variable, rather, they will check to see if two expressions satisfy the given operator and if so, return true, otherwise return false.

A single bool variable is also a valid Boolean expression.

Finally, we can create more complicated Boolean expressions with the following Boolean operators:

```
&& (and)
|| (or)
```

The and of two Boolean expressions is true ONLY if both expressions are true. The or of two Boolean expressions is true as long as at least ONE of the two expressions is true.

The if statement comes with optional branches (only the if is required). Here is the full possible syntax:

```
if (bool exp 1) {
    stmts1;
}
else if (bool exp 2) {
    stmts2;
}
…
else if (bool exp k) {
    stmtsk;
}
else {
    stmtslast;
}
```

The way the statement is interpreted is that each Boolean expression, in the order they appear, is evaluated until one evaluates to true. At that point in time, each of the statements inside of that branch are executed, and then command jumps to after the else closes. If none of the Boolean expressions is true, then the statements in the else clause are executed, and continues after it.

In the layout above, it's guaranteed that exactly one set of statements will be executed. If the else is omitted, then either one set of statements OR none of the statements will be executed. This layout of one if statement is DIFFERENT than the following layout:

```
if (bool exp 1) {
    stmts1;
}
if (bool exp 2) {
    stmts2;
}
```

This layout is two separate if statements, one after the other. In this layout, it's possible that BOTH sets of statements, stmts1 and stmts2 get executed, because even if bool exp 1 is true, after that if statement finishes, we simply continue to the next statement, which is the if statement with bool exp 2.

So a simple code snipped using the if might look like this:

```
double cost;
cin >> cost;
bool taxed;
int tax;
cin >> tax;
if (tax == 1)
    cost = cost*1.065
```

In this example, we read in the cost and whether or not the item is taxed (assume 1 means taxed and 0 means not taxed). If the item is taxed, then we reassign cost to be 6.5% higher. (As of this writing, this is the sales tax in Orange County, which is near UCF.)

There are quite a few potential pitfalls with if statements that can occur in C++ due to a few reasons:

1) Inadvertent semicolons
2) Forgetting the use of curly braces when they are necessary
3) Accidental use of an arithmetic expression instead of a Boolean expression (since C++ allows both in the if statement decision)
4) Not using parentheses when they are necessary due to order of operations rules.

Rather than go through a full listing of the rules and pitfalls (this could take 20 pages, no joke), our philosophy here will be to get started quickly and expect students to discover these rules as unintended behavior occurs.

**If Statement and Integer Division Example: FYI**

Here's a link to the problem:

https://open.kattis.com/problems/fyi

The gist of the problem is that we must read in an integer guaranteed to be 7 digits long and extract the first three digits to see if they equal to 555 or not. As previously discussed, dividing an integer by 10 is the same as chopping off its last digit. Here we just want to chop off the last 4 digits. It follows that dividing by 10000 will do the trick. (Notice that if we mod by 10000, that will reveal those last four digits. And more generally, dividing by $10^k$ chops off the last k digits and modding by $10^k$ reveals the last k digits.) Using this idea, and noting that we must output differently depending on whether or not the first 3 digits are 555 or not, here is a solution to fyi:

```
using namespace std;
#include <iostream>

int main() {

    int num;
    cin >> num;

    if (num/10000 == 555) {
        cout << 1 << endl;
    }
    else {
        cout << 0 << endl;
    }

    return 0;
}
```

**If Statement Example: Quadrant Selection**

As you might imagine, any type of statement can be placed inside of an if statement, including another if statement. In the following problem:

https://open.kattis.com/problems/quadrant

the input is a point on the Cartesian plane that does not lie on either the x or y axis, and you must determine which quadrant the point is in. This can be solved with a single if statement with four branches, but the following solution illustrates the idea of splitting up the work so that inside of an if statement, there is another if statement. The outer level if statement will check if x is positive. Then, inside of both the other level if and else, another if-else statement will check whether y is positive or not. In this example, the curly braces for the inner if have been omitted to show that they are not necessary (It turns out that the curly braces for the outer if are also not necessary, but they are left in for clarity.)

```
using namespace std;
#include <iostream>

int main() {
    int x, y;
    cin >> x >> y;
    int res = -1;

    if (x > 0) {
        if (y > 0)
            res = 1;
        else
            res = 4;
    }
    else {
        if (y > 0)
            res = 2;
        else
            res = 3;
    }

    cout << res << endl;
    return 0;
}
```

**For Loop**

The true power of computers is that they are much faster than humans and can do many steps (often repeating the same pattern of instructions) many, many times. The most simple way to accomplish repetition in C++ (or in most other programming languages) is the for loop. Here is the basic syntax of the for loop:

```
for (init stmt; bool exp; inc stmt) {
    stmts;
}
stmtA;
```

The way this works is as follows:

1. First the init stmt (initialization statement) is executed.
2. Then the Boolean expression is evaluated.
3. If the expression is true, then the statements are run. Otherwise, go to the first statement after the loop finishes (Go to step 6.)
4. After the statements are executed, then evaluate the increment statement (inc stmt).
5. Go back to step 2.
6. Continue after the loop to statement A (stmtA).

In practice, most for loops look similar to this:

```
for (int i=0; i<n; i++) {
    // stmts
}
```

Here, we have a loop index variable, i, initially set to 0. Assuming that n is an integer variable storing a positive number previously assigned in the program, initially this Boolean expression is true. Then the statements run for the first time, while the variable i is set to 0. Next, we run i++, which is simply shorthand for i = i+1. In short, this statement increments the value of i by 1. Then we check the Boolean expression again. If it is true, then we run the statements again. This continues until the Boolean expression is false, which should happen when i is set to n. This has the effect or running the statements exactly n times (unless i is modified inside of the loop).

Although C++ allows it, it's a good idea **never to modify** the value of the loop index inside of the loop, so that the loop index is an accurate gauge of how many times the loop has run. In nearly all situations, there are alternate ways to get the desired behavior, instead of changing the loop index variable inside the loop. Even though the loop index variable should not be changed inside of the loop, it is typical to use its value inside of the loop, so that the action performed in a single set of the statements can be based on "which iteration" of the loop is running.

**For Loop Example: N Sum**

One of the most basic things we can do with a loop is add up several numbers, which is the task in the following problem:

https://open.kattis.com/problems/nsum

The key to solving this problem is to use an accumulator variable, which stores up the current total added, as we read through the numbers. Here is a sample solution:

```
using namespace std;
#include <iostream>

int main() {

    int n, num, total = 0;
    cin >> n;

    for (int i=0; i<n; i++) {
        cin >> num;
        total = total + num;
    }

    cout << total << endl;
    return 0;
}
```

Notice that we have to set our accumulator variable equal to 0 first, the initial total value. Then, each time we read in a new value, we add it to total and store the result back into total. This particular syntax:

```
var = var + expr;
```

where var is an integer or double variable and expr is an arbitrary arithmetic expression of the same type as the variable occurs so frequently that it has the following shorthand:

```
var += expr;
```

Thus, in the program above, we could have written:

```
total += num;
```

and that would have been equivalent to

```
total = total + num;
```

**For Loop Example: Nasty Hacks**

It is common in some problems to have more than one case in the input. In order to "run multiple cases" a loop is required, as it the case in this problem:

https://open.kattis.com/problems/nastyhacks

In this problem, the problem itself boils down to a comparison between two expressions: the profit without advertising and the profit with advertising. We can solve this with an if statement with three branches, since we also have to report when it doesn't matter. But, the problem statement tells us that there will be multiple cases and that we must process them all. Thus, we can use a loop to control processing each case as follows:

```
using namespace std;
#include <iostream>

int main() {
    int nC;
    cin >> nC;

    for (int loop=0; loop<nC; loop++) {
        int regProf, adProf, adCost;
        cin >> regProf >> adProf >> adCost;

        if (regProf > adProf-adCost)
            cout << "do not advertise" << endl;
        else if (adProf-adCost > regProf)
            cout << "advertise" << endl;
        else
            cout << "does not matter" << endl;
    }

    return 0;
}
```

**Nested For Loop Example: Pot**

Here is the problem statement:

https://open.kattis.com/problems/pot

In this problem, you are given some numbers of input and for each input number you have to split it into two parts, the first being the base of the calculation and the second being the exponent. The split is such that the exponent is always a single digit: the least significant digit. For example, the number 212 should be split into 21 and 2 and then we should calculate $21^2$. Notice that exponentiation is repeated multiplication, so we can use a loop to calculate a power. But, recall that the problem asks us to add up several terms listed in this fashion, so an outer/main loop is necessary to get through all of the input numbers.

When designing a program to solve a problem with multiple pieces, it's important to see if the logic for each of the pieces can be solved separately, and then integrated together.

For this program, we'll have an outer loop that reads through each of the numbers. Inside of that loop, we must do the following, in this order:

1. Separate the input number into two pieces.
2. Set up an accumulator for the exponent calculation at 1.
3. Use a loop to calculate the exponent via repeated multiplication.
4. Add in this result into our overall accumulator.

Now that there's a plan in place, let's look at the program:

```
using namespace std;
#include <iostream>

int main() {

    int n, total = 0;
    cin >> n;

    for (int i=0; i<n; i++) {

        int value, base, exp, term = 1;
        cin >> value;
        base = value/10;
        exp = value%10;

        for (int j=0; j<exp; j++)
            term *= base;

        total += term;
    }

    cout << total << endl;
    return 0;
}
```

Notice that we have two accumulators in this code. total is keeping track of the overall total and is initially set to zero, since this accumulator is additive, and the inner accumulator, term, which is initially set to 1, since this accumulator is multiplicative. Almost all of the code inside the loop is dedicated to calculating the base raised to the exponent. While writing this, one can focus on the task at hand. Once term is set to the right value, then it can be added into total.