Programming Notes 6: Loops & Recursion

As usual, more examples of the notes are in the example program.

1. Looping

The basic concept of looping is very simple—a loop repeats the same code many times. All loops must have a few critical pieces in order to run correctly: an initialization, where starting conditions are set up; a condition to continue, which decides when the loop will stop; and finally an updating condition that specifies what will change each time the looped code is run. If your condition to continue is always met, the loop will continue to run for the life of your program—this is called an infinite loop, and is a bug, as your program cannot progress if it is stuck in one. While all loops have this basic functionality, and hence can be used interchangeably, there are several types of loops that are more suitable for different situations.

2. While loops

The simplest type of loop is the while loop, where it repeats its block of code (funnily enough) while a Boolean expression is true. The syntax is also quite simple: type the keyword "while," and then your Boolean terms in parenthesis. Hence, when your program gets to the loop, it will test your condition, and if it is true, it will run the looped code, and when it gets to the end of that, it will test your condition again, and so on. Because this loop only does one thing, you have to implement the other two parts of the loop yourself: before your loop you must perform the initialization, and within the loop you must specify your updating condition. While loops are usually used for event triggered loops, meaning loops where the updating condition is invalidated when something specific happens in the code, for example counting through a string or displaying a menu system, whereas the for loop (subtopic 4) is more suitable for loops where you know exactly how many times you want the loop to run. Example:

```
char selection;                              -Initialize the loop
bool continueProgram = true;

while(continueProgram) {                -Specify continue condition
  cout << "Continue (y/n): "
  cin >> selection;
  if(selection == n)
    continueProgram = false;          -Update continue condition
}
```

3. Do while loops

The second type of loop is almost exactly the same as the while loop, except in a do while loop the loop code will always be run at least once. Basically, your program will go through the loop once under any condition, and then it will check your continue condition. If it is true, your program will go back to the start of the loop, and so on. The syntax is also very similar, except that you start with the keyword "do," and put the "while" statement at the end of the code block, followed by a semicolon. Do/while loops are especially useful in things like menus, where you know it should be displayed at least once. Example:

```
do {
  do things, this will always happen at least once
  do other things
  update your condition
}
while(condition);
```

4. For loops

The third type of loop is my personal favorite (I use it for most things), the for loop. As mentioned previously, the “for” loop does essentially the same things as a while loop, but it does it in a much more compact manner. The ‘classic’ use of the “for” loop is to count from one value to another, which it excels at, but it can be used for many other things as well. A for loop has five parts to the definition: first, of course, type the keyword “for,” followed by parenthesis. Now, there are three things you have to include in these parenthesis. First, there is the initializing statement. This first statement will be run once when the program first gets to the loop, and never again. Generally, here you want to set up variables or counters that will be used in the loop. Then, add a semicolon to end the first statement and start the second one. The second statement describes the continue condition of the loop, and must be a Boolean expression. Finally, add another semicolon and the third statement, which is the updating condition. This statement will be run each time your program gets to the end of the loop. Then, it will check the continue condition—not before. Remember that. This statement usually includes some sort of numerical manipulation of a counter variable, such as adding one to it each time. Furthermore, you can add multiple operations to the first and third statements by simply adding a comma between each one. Lastly, as with other control flow code, if you only have one statement in your loop, you can omit the curly brackets. Examples:

```
for(int i = 0; i < 20; i++) {
  cout << “Loop is on number “ << i << endl;
}

for(pizza = 0, steve = 10; pizza <= steve; pizza++, steve--)
   cout << “Pizza is “ << steve – pizza << “ less than steve.” << endl;
```

You always need to pay close attention to where you start and end your loop, as improper use of greater than/less than/or equal to, and starting on values like 0 or 1 can lead to what are called “off-by-one” errors, where your loop will run one too many or one two few times. Be VERY careful of these errors, as they can be hard to find. For example, the first example would loop between values i = 9 and i = 19, not anything else. On the other hand, if the conditional used a less than or equal to instead of just a less than, it would loop between i = 0 and i = 20.

5. Nesting loops

Just like you can nest conditional statements, you can nest loops. It works just as you'd think—the outer loop runs the smaller loop several times, and each time the inner loop is run, it runs its code however many times. This will be especially useful when we get to topics such as multidimensional arrays. In this example, the numbers 0-1, 0-2, etc will display, then 1-1, 1-2, etc, etc:

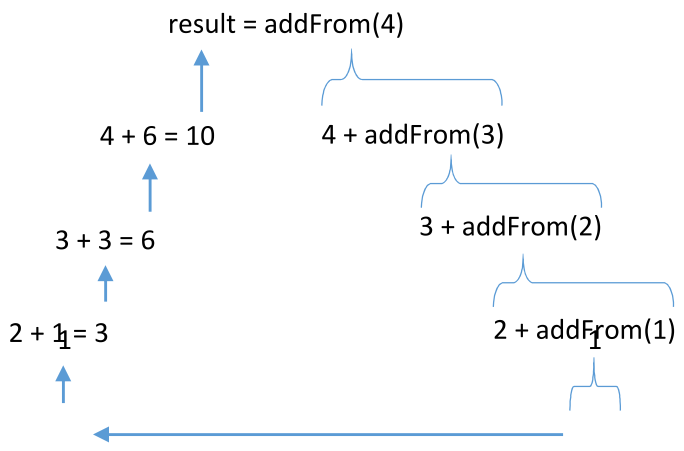
```
for(int i = 0; i < 10; i++) {          -This will run the following loop and cout 10 times.
   for(int j = 0; j < 10; j++)         -Note that this will only run the next statement
     cout << i << "-" << j << " ";     10 times, as there are no brackets
  cout << endl;
}
```

6. Recursion

Finally, there is the last type of looping—looping without loops. You've probably heard of it before, and it's recursion. Recursion is the process of a function calling itself. There are more complicated forms of recursion, but what we'll cover for now are tail-recursive functions, meaning that they must have some sort of conditional to end the recursion process, and if that is not triggered, it will call itself in the tail of the function with some sort of modified parameters. Example:

```
int addFrom(int x) {

  if(x == 1)

    return 1;

  else

    return x + addFrom(x – 1);

}
```

This function will take a value, and if that value is equal to one, it will simply return 1. However, if the value is not one, it will return the value plus the return of itself, called with one less than x. This process will continue until the value of x – 1 is equal to 1, in which case the final call will return 1, and the return values will cascade back up, adding each returned value to the last. It will end up with x + (x – 1) + (x – 2) + … + 1, or basically the number plus each integer lower than it until 1. This may seem a bit hard to wrap your head around, try to think of it visually:


result = addFrom(4)
4 + 6 = 10
4 + addFrom(3)
3 + 3 = 6
3 + addFrom(2)
2 + 1 = 3
2 + addFrom(1)
1
1