

# CSCI 135 Software Design and Analysis, C++

## Homework 2

### Solution

Saad Mneimneh  
Visiting Professor  
Hunter College of CUNY

#### Problem 2: Nested loops

The objective of this problem is to explore nested loops, i.e. a loop which has another loop in its body, here's an example:

```
while (cond1) {  
  
    . . .  
  
    while (cond2) {  
  
        . . .  
    }  
  
    . . .  
}
```

The above code shows an example of nested loops. In general, a **while**, **for**, or **do while** may be used for either loops. Of course nesting can have as many levels as we want. For instance, we may have 3 or 4 nested loops.

Here's an example that would produce the multiplication table with two nested loops (try it):

```
for (int i=1; i<10; i=i+1) {  
    for (int j=1; j<10; j=j+1) {  
        if (i*j<10)  
            cout<<' ' <<' ';<br>            cout<<i*j<<' ' <<' ';<br>    }  
    cout<<' '\n';<br>}
```

The first for loop iterates values of  $i$  from 1 to 9. The body of the first for loop contains a second for loop that iterates values of  $j$  from 1 to 9. The body of the second for loop outputs  $i \times j$  in some appropriate format (using an if statement). This is possible since the second loop is contained in the body of the first loop and, therefore,  $i$  is also in the scope of the second loop. Finally, when the second loop ends, the first loop prints a new line.

(a) Write a function called `fact` that computes the factorial of an integer  $n$ , denoted  $n!$ , where

$$n! = \prod_{i=1}^n i = 1 \times 2 \dots \times n$$

The factorial of 0 is 1 by definition (empty product). You will only need a single loop inside the function to do that.

```
int fact(int n) {  
  
    //some declarations  
  
    while ( . . . ) {  
  
        //do the work  
    }  
  
    return . . .  
}
```

**ANSWER:** This will be similar to the problem of adding integers from 1 to  $n$  except that now we have to multiply them:

```
unsigned long int fact(unsigned int n) {  
    unsigned long int prod=1;  
    while (n>0) {  
        prod=prod*n;  
        n=n-1;  
    }  
    return prod;  
}  
  
int main() {  
    cout<<fact(12);  
}
```

The parameter is declared as `unsigned int` to prevent negative arguments. If a negative argument is supplied, its negative representation in memory is interpreted as a positive (this is not absolute value). The return type and result are `unsigned long int` because the factorial of  $n$  grows exponentially with  $n$  and, therefore, the 4 byte representation of an integer will soon overflow. Although according to the C++ requirements, a `long int` may be 4 bytes as well, it is better to write the code this way. With 4 bytes, the result will soon overflow and the function is good to compute up to the factorial of 12.

(b) Write a program that outputs the following Pascal triangle using two nested for loops.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
```

Note that for a given row  $i$  and a given column  $j$ , the value in the triangle

$$P(i, j) = \binom{i}{j} = \frac{i!}{j!(i-j)!}, j \leq i$$

where both rows and columns start at 0.

**ANSWER:**

```
int main() {
    for (int i=0; i<9; i=i+1) {
        for (int j=0; j<=i; j=j+1) {
            int p=fact(i)/fact(j)/fact(i-j);
            if (p<10)
                cout<<' ' <<' ';
            cout<<p<<' ' <<' ';
        }
        cout<<' '\n' <<' ';
    }
}
```

The counter for the inner loop goes from 0 to the value given by the counter for the outer loop (this is reflected in the condition ( $j \leq i$ )). In this way, we ensure that  $j \leq i$  any time.

(c) A better alternative to compute the entries of the triangle is to note that, for  $j > 0$ ,

$$\frac{P(i, j)}{P(i, j-1)} = \frac{i-j+1}{j}$$

Therefore, starting with  $P(i, 0) = 1$ , we can compute the rest of the entries of the  $i^{\text{th}}$  row. Implement this idea using loops.

