# *COMP 161 - Lecture Notes - - Loops and Common Loop Patterns*

## *Spring 2014*

In these notes we begin exploring C++ LOOP control structures.

### *Predicates*

Functions[1] that return a *bool* value are called PREDICATE or boolean-valued functions. Boolean expressions can always be replaced by predicate functions, and since boolean expressions are an integral part of loop structures, we'll stop and talk about a clean, concise style of writing predicates.

[1] procedures sans side-effects

In many cases, predicates can, and should, be written without the use of conditional statements. Let's say we need a predicate *isEven* which takes an *int* type and returns true if it is even and false otherwise. We could do this:

```
bool isEven(int n){
  if( n % 2 == 0 ){
    return true;
  }
  else{
    return false;
  }
}
```

However, notice that the *boolean expression n % 2 == 0* takes on exactly the value we want to return. So, a better implementation is to return the value of that expression.

```
bool isEven(int n){

  return n % 2 == 0 ;

}
```

Maybe you tested to see if it was odd? That's OK, you can always use ! to negate a boolean value. This implementation,

```
bool isEven(int n){
  if( n % 2 == 1 ){
    return false;
  }
  else{
    return true;
  }
```

```
}
```

becomes,

```
bool isEven(int n){

  return !(n % 2 == 1) ;

}
```

## *Basic Loop Types*

C++ offers three flavors of loops. We'll begin with the two most basic forms and then introduce the third when it fits a lopp pattern.

### *do while loops*

The first loop we'll look put to use is for when you need to repeat some code *at least one time*. It has the following form:

```
do{

// loop body

}while( /* continuation test */    );
```

The */* continuation test */* expression must be a boolean expression, function, or value. The do while loop will repeat the code between its curly braces[2] as long as the continuation test is *true*. The test is carried out after each execution of the loop body. Thus, the do while loop will execute the loop body at least once.

[2] the *loop body*

### *while loops*

By moving the continuation test to before the execution of the loop body, it is possible to write loops that will never actually execute the loop body because the continuation test fails[3] the first time it is evaluated. This kind of "*zero or more* repetitions" loops can be carried out by the C++ *while* loop.

[3] evaluates to false

```
while( /* continuation test */ ){

// loop body
```

```
}
```

## *Some Loop Patterns*

The two loop constructs we've seen are powerful enough to express most of the patterns of repetition you'll encounter in the course of designing a program. Let's explore a few common patterns.

### *Input Validation Loop*

Our "guess a number" game from previous notes requires that the user enter a number between 1 and 10. If the user were to enter something like 15, then the game shouldn't proceed. This is a case where we need to validate the user input and then repeat the input sequence until the input is valid. For this we can use a do while loop as we need to carry out the input sequence at least once.

```
do{
  // prompt the user for input
  // get user input
  if( /* input is invalid test */ ){
     // tell the user it's invalid and why if possible
  }

}while( /* input is invalid test*/ );
```

Realistically, it's possible the user entered some kind of gibberish that not only produced invalid input but caused the read operation to fail. Thankfully, checking for failed input is relatively easy. The input stream for the standard input, *std::cin*, is in fact an input stream object[4]. The input stream class provides several useful methods for detecting and dealing with failed read operators.

[4] http://www.cplusplus.com/reference/ios/ios/

```
do{
  // prompt the user for input
  // get user input

  // check for failure and validity
  if( std::cin.fail() ){
   // tell the user the input failed and if possible why
   // set input to invalid value to force looping

std::cin.clear(); //reset the stream for
    std::cin.ignore(INT_MAX); //ignore anything left in the stream
```

```
  }
  else if( /* input is invalid test */ ){
     // tell the user it's invalid and if possible why
  }

}while( /* input is invalid test */ );
```

Three *ios* methods are used on *std::cin* in this loop: *fail*, *clear*, and *ignore*. They're listed under state flag functions in the *ios* class documentation.

### *Polling Loop*

Polling loops continue as until a particular state is reached. Just about any loop can be called a polling loop. For example, validation loops poll the system to see if valid input was was or was not read. A generalized polling loop[5] might use a specific variable, called a FLAG to track the state that drives the loop. Here we show a while loop version.

[5] lacking a more descriptive name

```
bool continue = true;
while ( continue ){

  // looped code


  if( /* poll test */){
    continue = false;
  }
}
```

Here we see the loop polling the system at the end of every loop. If the desired state is detected, then the *continue* flag is flipped and the loop will terminate.

### *Counted Loops*

Sometimes repetition isn't driven by some larger system state, but simply needs to occur a specific number of times. For this we use COUNTED LOOPS. These loops effectively count off the number of repetitions. To accomplish this we need a counter variable. Let's start with a simple loop for ten repetitions by counting though 0 to 9[6];

[6] the reason for starting at 0 will be clear as we move forward

```
int counter(0);
while( counter < 10 ){
  // looped code;
```

```
  counter++;
}
```

This loop carries out a classic, general loop pattern:

```
// initialize state s
while( /* test s for continuation condition */ ){

  // loop code

  //update s
}
```

This pattern is based off three parts: pre-loop initialization of state, looping based on state condition, state update at the end of the loop. This pattern is exactly what the third C++ loop statement is meant to do. This loop is called the *for* loop. Here's the general pattern and the count to 10 loop as a for loop.

```
for(/*initialize s*/ ; /* test s */ ; /*update s*/ ){

 // looped code

}

for( int counter(0); counter < 10 ; counter++){
  //looped code
}
```

Notice the for loops lets you move all the administrative stuff[7] to one line and frees up the loop body for the code logic you're trying to repeat.

[7] code that drives the loop

Another use of counted loops is for working with indexed collections[8] or counting through a specific sequence of numbers[9]. We call these for-each loops because the counting is just a vehicle by which we can get at each element of the collection or sequence. So, if you need to access every character in a string of 10,000 characters, then you can count through the interval of indexes $[0, 10000)$. The difference here is intension; we want each character and use each integer index to get it. Pure counted loops only differ in that the goal is to count of numbers[10].

[8] like strings,vectors, and arrays

[9] as opposed to a specific number of numbers

[10] The distinction is a a minor one, but worth thinking about

Let's say $s$ is a string. The following counts through all of its index values and accesses every entry in $s$.

```
for( int i(0) ; i < s.length() ; i++){
  ... s[i] ... //or s.at(i)
}
```

The variable name *i* is short for "index" and is the traditional name for this situation. Maybe you wanted to go from end to beginning, not beginning to end?

```
for( int i(s.length()-1) ; i >= 0 ; i--){
  ... s[i] ... //or s.at(i)
}
```

What if we only wanted the odd numbers between some odd number *a* and some odd number *b*?

```
for( int i(a); i <= b; i+=2) {
  //...i...
}
```

How about the first 10 powers of 2?

```
for( int i(1); i<=1024 l i*=2 ){
  //...i...
}
```

The key to these last examples is that counting doesn't always happen in increments of 1, sometimes we count in twos, sometimes threes, sometimes we multiply instead of add. In all cases, the *for* loop makes for concise statements of counting in code.