The for Loop

Class 16

Common Errors in Loop

Re-declaring a variable inside the loop body:

```
int sum=0;
int count=0;

while(count<5)
{       count ++ ;
        // this is replacing the sum declared
        // on the outer scope
        int sum = sum  + count ;
}       // scope of inner sum ends here on each loop iteration

cout<<" " << sum; // output would be 0
```

Sentinel Loop

```
int value = 1;

while(value !=0) // sentinel condition
{       cout<<"\nEnter a value: (0 to exit)";
        cin>>value;
}
```

- The loop continues as long as the sentinel condition is true

- In this example the sentinel condition is value ==0

Interactive Loop

```
int value = 0;
char moredata = 'y';
while(moredata == 'y') // interaction condition
{       cout<<"\nEnter a value: ";
        cin>>value;
        cin.ignore();

        cout<<"\nDo you want more data? [y/n]: ";
        cin>>moredata;
}
```

- On each iteration of the loop, we ask the user whether they want to continue one more time. Depending on user's response the loop would make iteration one more time or stop

The do-while Loop

- the second loop construct of C++ is the do-while loop
- its form is below — note the semicolon!

```
do
{
   statement;
   statement;
   ...
} while (expression);
```

- this is a posttest loop
- its Boolean expression is tested after the loop body executes
- it is guaranteed that the loop body will execute at least once
- look at program count_accumulate_do_while.cpp, which is the previous program converted to use a do-while loop

The do-while Loop

```
int a = 0;
while(a>0)
{
        a = a - 1; }
cout<<a<<endl;
```

```
int a = 0;
do
{       a = a - 1;
}while(a>0);
cout<<a<<endl;
```

Regardless of whether the loop condition is true or false

the do .. while loop will execute at least once

In certain cases, this behavior is advantageous

In the above, for the while loop, the output would be 0 as the pretest is false

In case of do..while loop the loop body was executed and stopped after the postest, the ouput would be -1

Controlling a Loop With a Flag

```
1   bool done = false;
2
3   while (!done)
4   {
5     cout << "Enter a plan: ";
6     char plan;
7     cin >> plan;
8
9     if (plan == 'A')
10       // stuff for plan A ...
11    else if (plan == 'B')
12       // stuff for plan B ...
13    else if (plan == 'C')
14       // stuff for plan C ...
15    else
16    {
17      done = true;
18    }
19  }
```

- it is extremely common to control a while loop with a Boolean flag
- the flag is initialized to false
- when the loop exit condition is recognized, the flag is set to true

Input Validation

- a common use for a while or a do-while loop is input validation

```
int act_score;
// this program validates the input entered in the program
// the input value should be greater than 0 and less than 37


cout<<"\nEnter act score: ";
cin>> act_score;
while (not (act_score>0 && act_score<37))
{
    cout<<"\nInvalid value; enter act score: ";
    cin>> act_score;

}

// now act_score is valid, so use it
```

The for Loop

- C++'s third looping construct

```
for (initialization; test; update)
{
  statement;
  statement;
  ...
}
```

- example

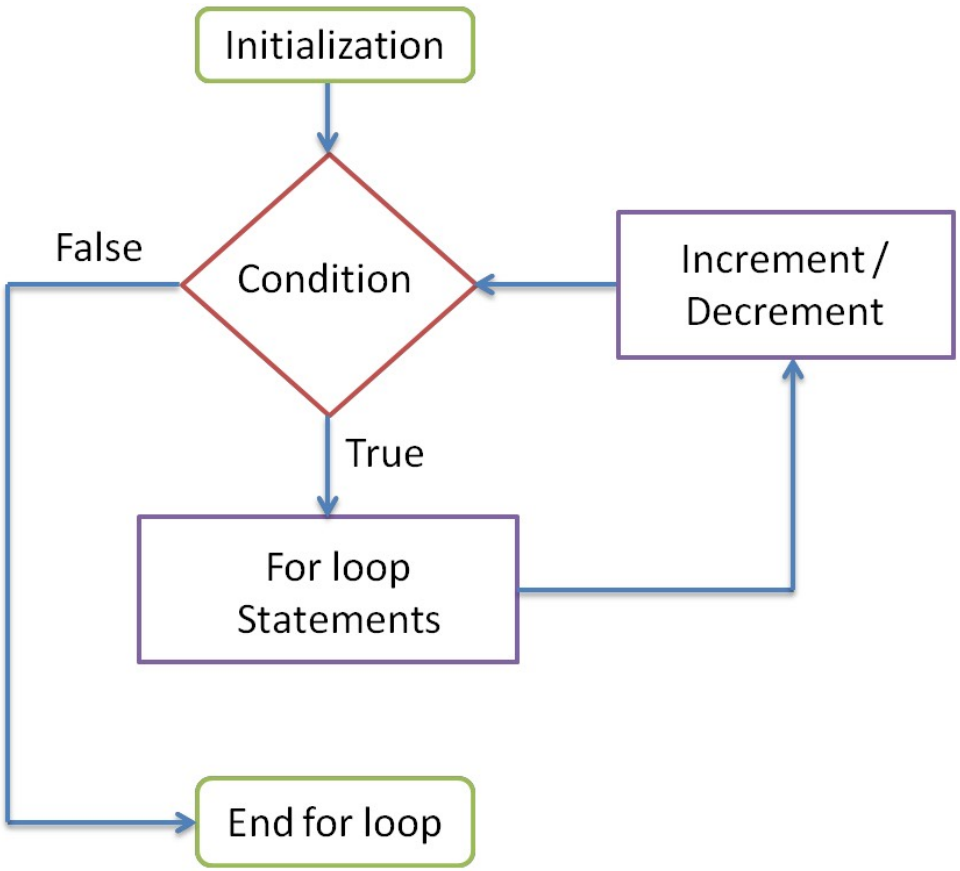
```
for (unsigned count = 0; count < 5; count++)
{
  cout << "Hello" << endl;
}
```

The for Loop

```
for (unsigned count = 0; count < 5; count++)
{
   cout << "Hello" << endl;
}
```

- semicolons separate initialization, test, and update
- there is no semicolon after update
- the scope of a variable declared in the initialization part is inside the curly braces of the loop body block

The for Loop


Initialization
Condition
False
Increment / Decrement
True
For loop Statements
End for loop

The for Loop

```
for (unsigned count = 0; count < 5; count++)
{
   cout << "Hello" << endl;
}
```

- semicolons separate initialization, test, and update
- there is no semicolon after update
- the scope of a variable declared in the initialization part is

  inside the curly braces of the loop body block

- the initialization statement is only done once, the first step
- the test statement is done every time
  - right after the initialization the first time
  - right after the update on subsequent iterations
  - this makes the for loop a pretest loop construct
- the update statement is not done the first time, but is done before the test on every subsequent iteration

Things to Avoid

1. declare the loop control variable before the loop header

```
unsigned count;
for (count = 0; count < 5; count++)
{ ...
```

- unless you absolutely need count to exist after the loop ends, you should declare count in the loop header like this:

```
for (unsigned count = 0; count < 5; count++)
{ ...
```

Things to Avoid

2. missing parts of the loop header

```
unsigned count = 0;
for ( ; count < 5; )
{
   ...
   count++;
}
```

- this is legal, but even worse than #1
- in old code you'll sometimes see: `for ( ; ; )` ugh!

Things to Avoid

3. multiple statements in the loop header

```
for (unsigned count = 0; count < 5; count++, foobar--)
{
   ...
```

- this is legal, but you should never do it
- the only thing the loop header should do is control the loop
- since foobar is not involved in controlling the loop, modifying it should be done in the loop body, not in the loop header

Things to Avoid

4. modifying the loop control variable in the loop body

```
for (unsigned count = 0; count < 5; count++)
{
  count += 2;
  ...
```

- never do this!
- the entire control of the loop should reside in the header

Things to Avoid

5. using a floating point value to control a for loop

```
for (double count = 0; count < 5.0; count++)
{
   ...
```

- doubles should never be used for counting purposes
- while loops, which are not conceptually count-controlled loops, can use doubles as loop control variables

Other Step Sizes

- the most common step size of for loops is positive one

```
for (unsigned count = 0; count < 5; count++)
```

- but negative one is also common

```
for (unsigned count = 5; count > 0; count--)
```

- and it's easy to count by 2's or other increments

```
for (unsigned count = 0; count < 10; count += 2)
```

Number of Loop Iterations

```
const int START_VALUE = 5;
const int STOP_VALUE = 8;
for (int control = START_VALUE; control < STOP_VALUE; control++)
```

```
for (int control = START_VALUE; control <= STOP_VALUE; control++)
```

- if a for loop acts by incrementing and the test condition is less than, the number of loop iterations will be $\texttt{STOP\_VALUE} - \texttt{START\_VALUE}$
- if a for loop acts by incrementing and the test condition is less than or equal to, the number of loop iterations will be $\texttt{STOP\_VALUE} - \texttt{START\_VALUE} + 1$

Number of Loop Iterations

```
const unsigned START_VALUE = 2;
const unsigned STOP_VALUE = 21;

for(int control =START_VALUE; control<STOP_VALUE; control += 2)
```

- if a for loop acts by incrementing or decrementing by a value different than 1
- the number of loop iterations depends on whether the step size is an even multiple of the range size
- and whether the condition includes equal to or not
- typically need to trace by hand and run some test cases to check

Infinite Loops

- be careful not to make an infinite loop!

```
for (unsigned count = 1; count != 10; count += 2)
```

- this is an infinite loop because count is going up by two on the odd numbers and will never be equal 10

```
for (unsigned count = 10; count >= 0; count--)
```

- this is tricky
- this keeps going until count is negative
- but an unsigned cannot be negative — it wraps around to a huge number and keeps going
- this is an infinite loop

For Loop Accumulator

- last class we saw an example of a while loop being used to count the number of and accumulate the sum of ACT scores
- this was an appropriate use of the while loop
- it was impossible to know when the loop started how many ACT scores the user was going to enter

For Loop Accumulator

- last class we saw an example of a while loop being used to count the number of and accumulate the sum of ACT scores
- this was an appropriate use of the while loop
- it was impossible to know when the loop started how many ACT scores the user was going to enter

- a slightly different version of the same idea makes it appropriate to use a for loop
- see accumulate_for.cpp
- when the loop starts, we know exactly how many scores the user will enter