

# More if-statements



# If... if... else!

When in doubt, use parenthesis and blocks!  
(Some people like to put the first brace after the if, others on a new line)

What happens if  
you have an  
if if else?

(See: ifIfElse.cpp)


```
if(true) {  
    // code here  
}
```

```
if(true)  
{  
    // code here  
}
```

# Short-circuit evaluation

Short-circuit evaluation is when you have a complex bool expression (&& or ||) but you don't need to compute all parts.

```
if(false && 7/0 == 2) {  
    cout << "Will I crash?\n";  
}
```



If this is false, then it will not check next

(See: shortCircuit.cpp)

# Short-circuit evaluation

Simple cases of short-circuit:

When you have a bunch of ORs

```
if( expression || exp || exp || exp )
```

Once it finds any true expression,  
if statement will be true

When you have a bunch of ANDs

```
if( expression && exp && exp && exp )
```

Once it finds any false expression,  
if statement will be false

# Complex expressions

Be careful when negating, that you follow De Morgan's Law:

`bool` a, b;

$!(a \text{ OR } b)$  is equivalent to  $(!a) \text{ AND } (!b)$

$!(a \text{ AND } b)$  is equivalent to  $(!a) \text{ OR } (!b)$

“Neither rainy or sunny” means

“Both not rain and not sunny”

# Nested if statements

You can have as many if statements inside each other as you want.

```
if (teacherAwake)
{
    if (studentAwake)
    {
        if (classWellPrepared)
        {
            learning = true;
        }
    }
}
```

# Nested if statements

From a truth table perspective, nested loops are similar to AND

The previous if code is equivalent to:

```
if(teacherAwake && studentAwake && classWellPrepared)
{
    learning = true;
}
```

However, sometimes you want to do other code between these evaluations

# Nested if statements



(See: `bridgeOfDeath.cpp`)



# Scope

Where a variable is visible is called its scope

Typically variables only live inside the block (denoted with matching { and } )

A variable lives until the block is closed, so inner blocks can see everything from the block it was created inside

# Scope

```
5  int main()  
6  {  
7      int x;  
8      // can use x here  
9      {  
10         int y;  
11         // can use x or y here  
12     }  
13     // can use x here  
14     return 0;  
15 }
```

(See: scope.cpp)

# Multiway if/else

This is a special format if you put an if statement after an else.

This second “if statement” only is tested when the first “if statement” is not true

(See: grades.cpp)

# Switch

A switch statement checks to see if a variable has a specific value.

```
switch( controllingVariable)
{
    case 2:
    case 4:
        cout << "controllingVariable is either 2 or 4" << endl;
        break;
    case 3:
        cout << "controllingVariable is 3\n";
        break;
    default;
        cout << "controllingVariable is not 2, 3 or 4...\n";
        break;
}
```

Controlling Variable

Case label

Break statement

# Switch

If the value of the controlling variable is found in a case label, all code until a break statement is ran (or the switch ends)

Switch statements only test equality with case labels (not greater or less than)

(See: `switch.cpp`)

# Switch

Switch statements can be written as multiway if/else statements.

Could use just “if statements” but “else if” shows only one of these will run

(See: `switchToIf.cpp`)

# Conditional operator

We will not use in this class, but if you use other people's code you will encounter

Shorthand for an if-else statement

(boolean) ? [if true] : [if false]

Example:

```
max = (x > y) ? x : y;
```

(See: max.cpp)

# if/else vs loops

if/else statements makes code inside only sometimes run

Loops make code inside run more than once

Both use boolean expressions to determine if the code inside is run



# while loop

A while loop tests a **bool** expression and will run until that expression is **false**

```
while (i < 10)
{
    // looped code
    // variable i should change in here
}
```

← **bool** exp., no ;

(See: whileLoop.cpp)

# while loop

The **bool** expression is tested when first entering the while loop

And!

When the end of the loop code is reached (the } to close the loop)

```
int i = 0;
while (i < 5) {
    cout << "Looping, i = " << i << "\n";
    i++;
}
cout << "Outside the loop, i = " << i << "\n";
```

# while loop

It can be helpful to manually work out what loops are doing and how variables change in each loop iteration

This will build an insight into how loops work and will be beneficial when working with more complicated loops

# while loop

3 parts to any (good) loop:

- Test variable initialized

```
i=0;
```

- **bool** expression

```
while (i < 10)
```

- Test variable updated inside loop

```
i++;
```

# for loop

A for loop is a compacted version of the while loop (the 3 important parts are together)

for loops are used normally when iterating over a sequence of numbers (i.e. 1, 2, 3, 4)

```
for (int i=0; i < 3; i++)
```

Initialization

boolean expression

Update

(See: forLoop.cpp)

# do-while loop

A do-while loop is similar to a normal while loop, **except** the **bool** expression is only tested at the end of the loop (not at the start)

```
8   cout << "How many times do you want to run the loop?\n";
9   cin >> i; // what happens if i is less than 1?
10  do {
11      cout << "Looping, i = " << i << "\n";
12      i--;
13  } while (i > 0); ← Note semicolon!
14  cout << "Outside the loop, i = " << i << "\n";
```

(See: doWhile.cpp)

# do-while loop

Q: Why would I ever want a do-while loop?

A: When the first time the variable is set is inside the loop.

You can initialize the variable correctly and use a normal while loop, but this makes the logic harder

# Loops

99 bottles of beer on the wall, 99 bottles of beer!  
Take one down, pass it around, 98 bottles of beer on the wall!

98 bottles of beer on the wall, 98 bottles of beer!  
Take one down, pass it around, 97 bottles of beer on the wall!

97 bottles of beer on the wall, 97 bottles of beer!  
Take one down, pass it around, 96 bottles of beer on the wall!

...

Write a program to output the above song  
(See 99beer.cpp)



# continue

There are two commands that help control loops:

continue tells the loop to start over again

break stops the loop



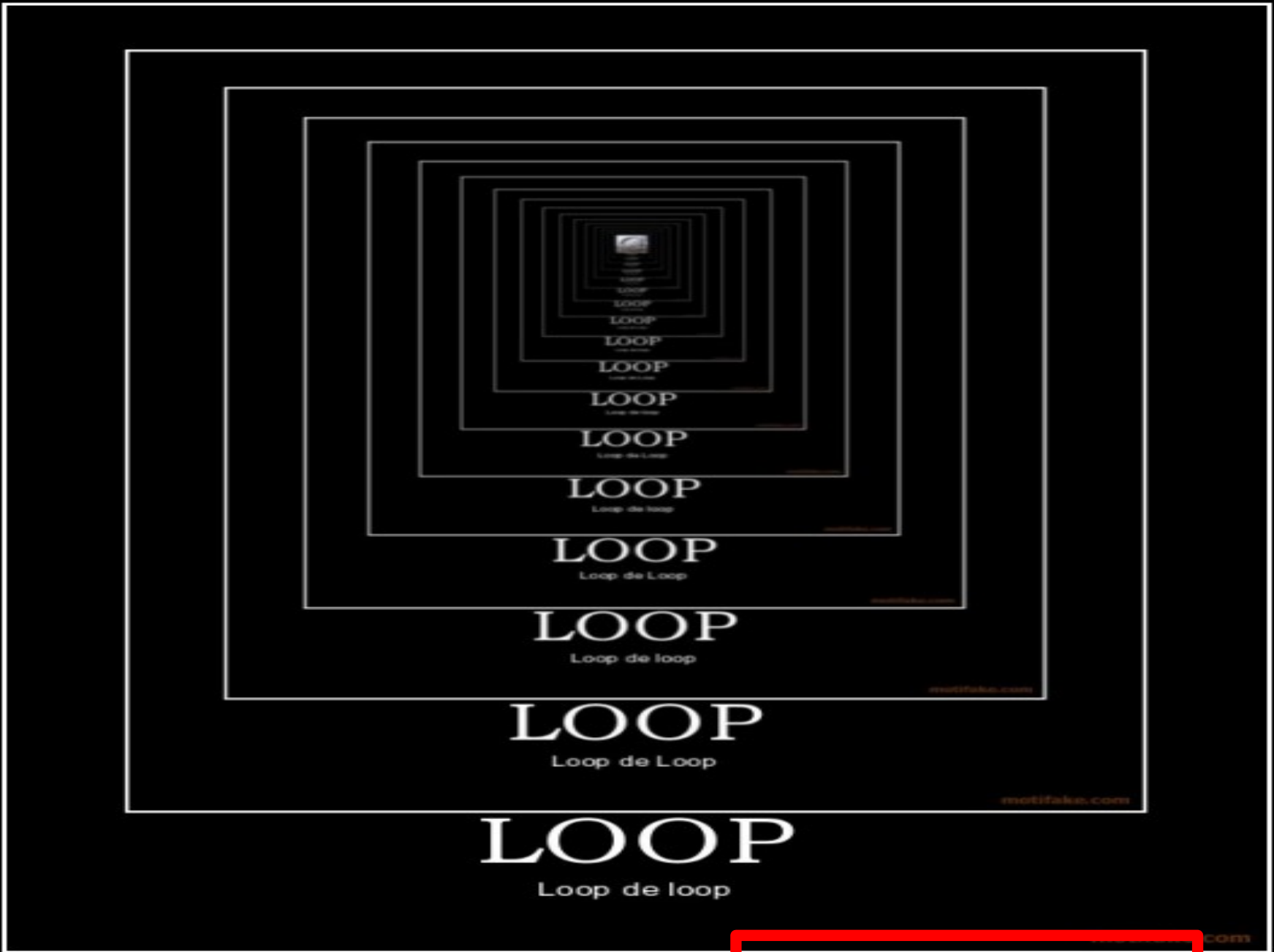
# continue

continue command can be issued to start at the next iteration of a loop

```
for (i = 0; i < 10; i++)  
{  
    // code will run everytime  
  
    if (doSkip)  
    {  
        continue;  
    }  
  
    // code will not run  
    // if doSkip is true  
}
```

doSkip  
true

(See: continue.cpp)



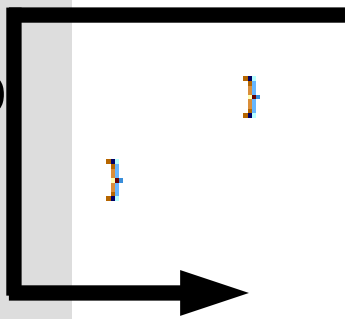
C-C-COMBO **BREAKER**

# break

break will exit the current loop

```
for (i = 0; i < 10; i++)  
{  
    // code  
  
    if (doSkip)  
    {  
        break;  
    }  
}  
  
// outside loop code
```

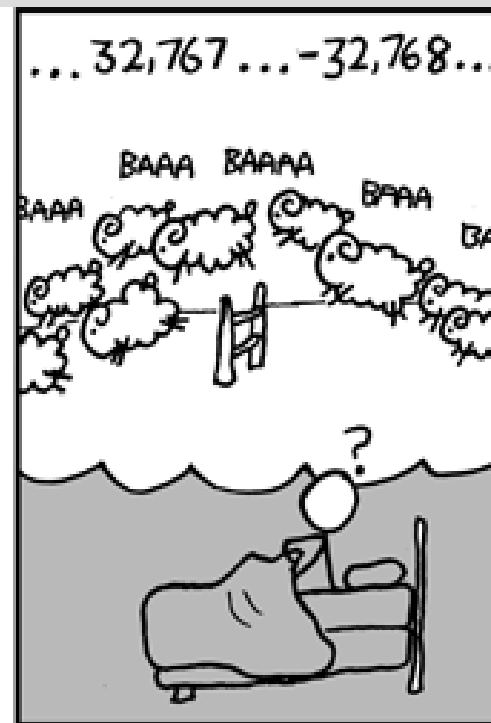
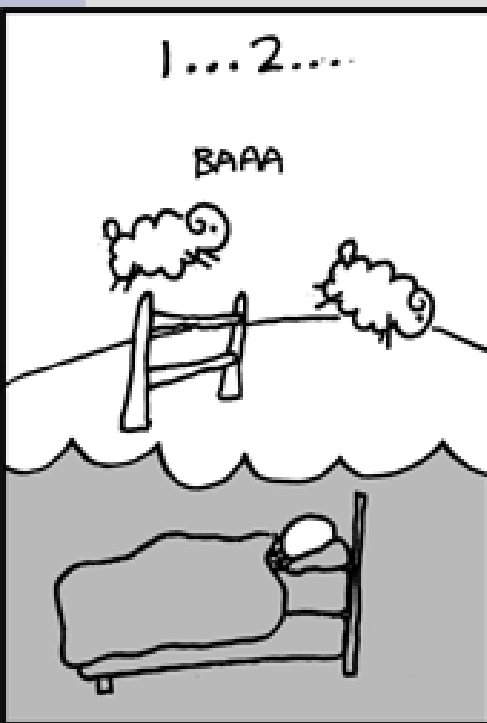
doSkip  
true



(See: break.cpp)

# Infinite loops

(See: countingSheep.cpp)



# while loop



<https://www.youtube.com/watch?v=7-Nl4JFDLOU>

# Loops to sum

Loops allow you to decide how many times a piece of code should run on the fly (i.e. at run time, not compile time)

You can either directly prompt the user how many times or make a special value to “exit” on

(See: `sumLoop.cpp`)

# Debugging

When your program is not working, it is often helpful to add cout commands to find out what is going on

Normally displaying the value of your variables will help you solve the issue

Find up until the point where it works, then show all the values and see what is different than you expected