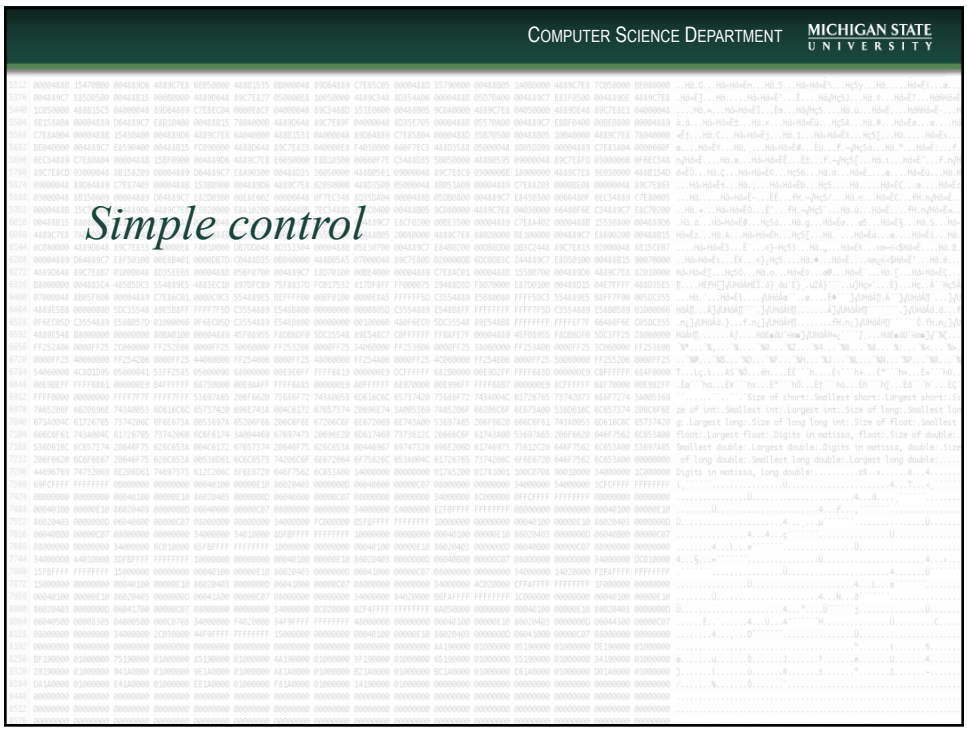
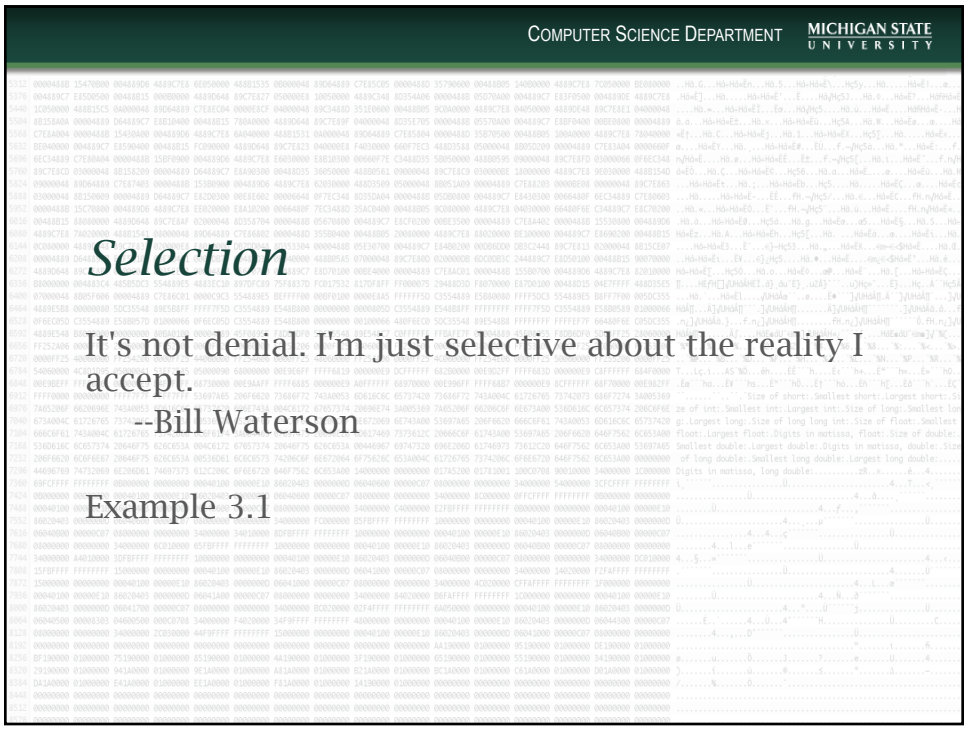
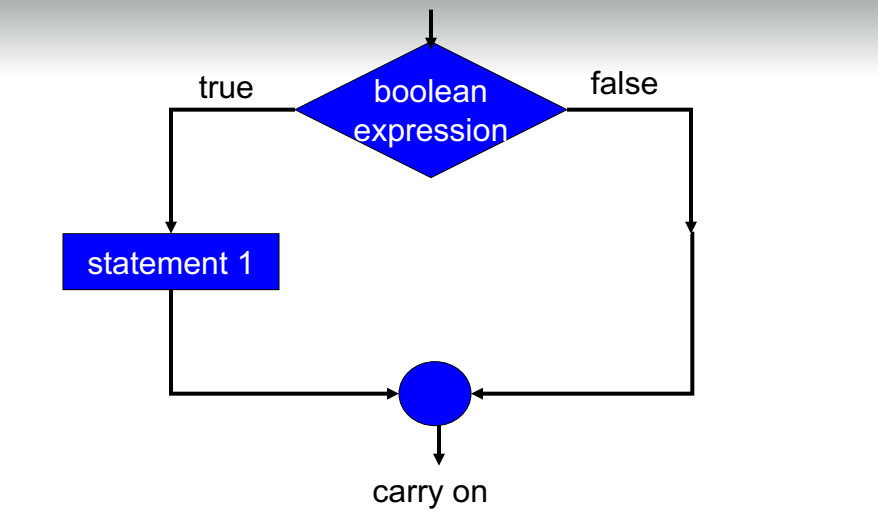

COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
Simple control


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
Selection
It's not denial. I'm just selective about the reality I accept.
--Bill Waterson
Example 3.1

Selective Execution



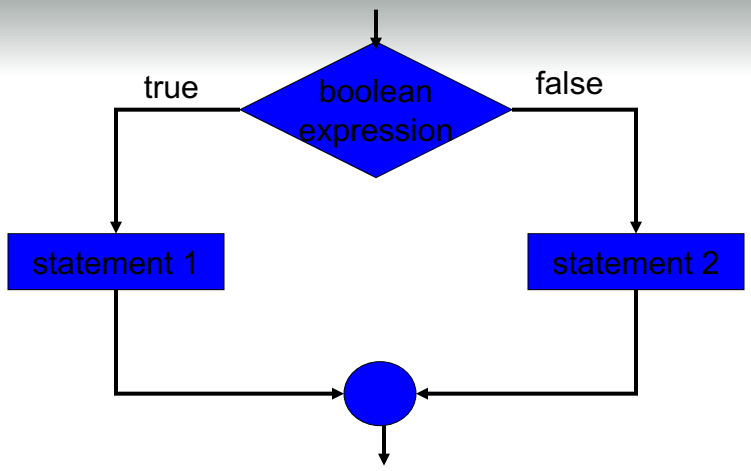
v1, if alone

```
if (boolean expression)
   statement;
```

semi after statement

parens required

two alternatives



v2 optional else

```
if (boolean expression)
   statement_1;
else
   statement_2;
```

Blocks

- A sequence of statements treated like a single statement.
- A block of statements can go wherever any single statement can go, i.e. not restricted to selection.
- Syntax: set off by brackets, e.g. `{ }`

Blocks in Selection

a block

```
if (boolean1){
   statement1;
   statement2;
      ⋮     //many as you like
}
else {
   statementA;
   statementB;
      ⋮     //many as you like
}
```

blue part of `if` block
red part of `else` block

semis for each statement, ***none*** for block end

v3, chain of ifs

```
if (boolean_expr1)
  statement_1;
else if (boolean_expr2)
  statement_2;
else if (boolean_expr 3)
  statement_3;
```
… many as you like
```
else
  statement_last;
```


Results

```
if (boolean_expr1)
   statement_1;
else if (boolean_expr2)
   statement_2;
else if (boolean_expr 3)

   statement_3;
```
… many as you like
```
else
   statement_last;
```

This is an `if` structure

Evaluate Booleans in order
- if false, go on to the next

First Boolean that evaluates to true has its statement (or its block) run.
- skip the rest of the `if`

If no true Boolean, run the `else`

```
if (boolean_1){
   … statements …
   if (boolean_1_1){
      … statements…
   }
   else if (boolean_1_2){
      … statements…
   }
   else{
      … statements…
   }
} // of boolean_1 if
```

formatting is helpful.
it is not required but
it makes clear where blocks begin and end

use comments if it is ugly

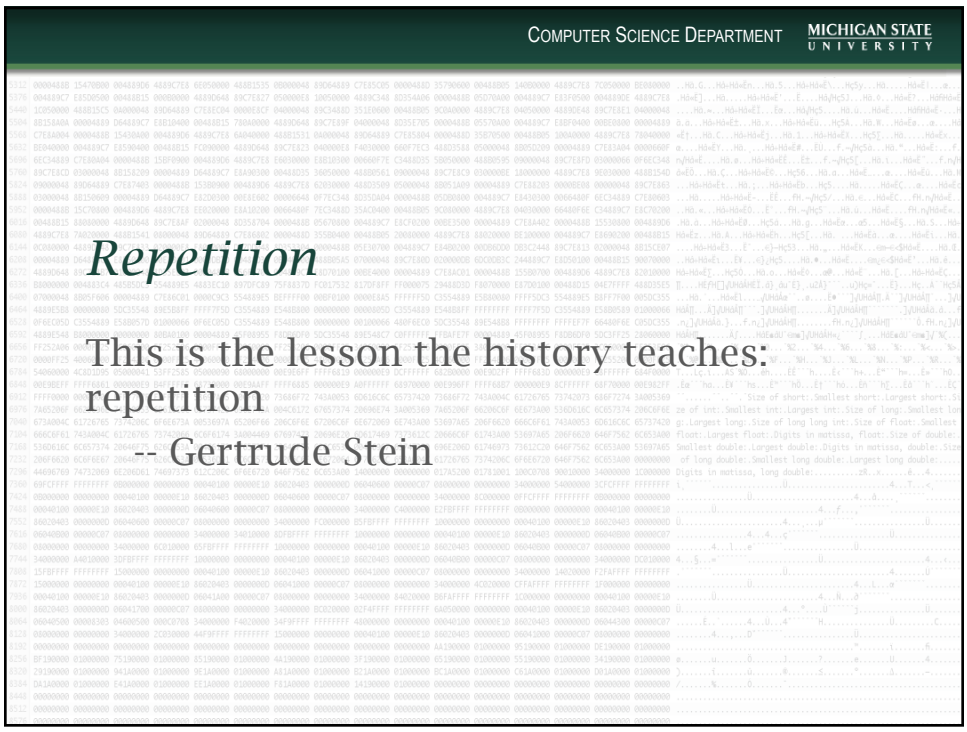
Dangling else problem

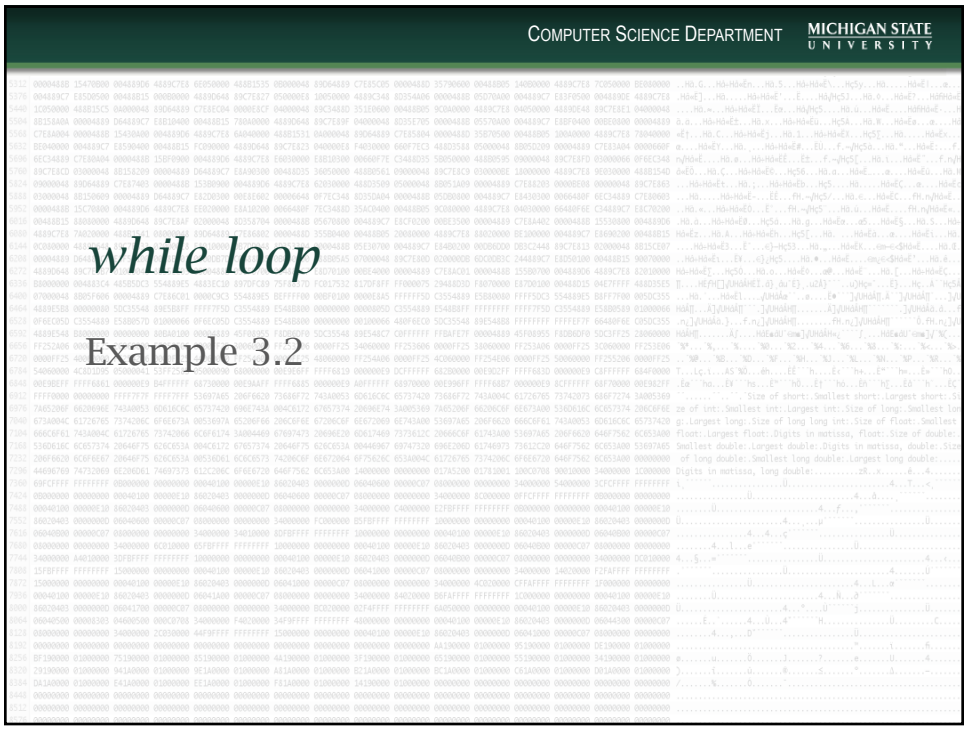
```
if (boolean_1)
   if (boolean_1_1)
      statement_1_1;
else
   statement_2;
```

wrong indentation. `else` goes with the last `if` in the code

this `else` is part of the boolean_1_1 `if`

```
if (boolean_1)
   if (boolean_1_1)
      statement_1_1;
   else
      statement_2;
```


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
Repetition
This is the lesson the history teaches: repetition
-- Gertrude Stein


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
while loop
Example 3.2

Three Loops

`while`

- top-tested loop (pretest)

`for`

- counting loop
- forever-sentinel

`do`

- bottom-tested loop (posttest)

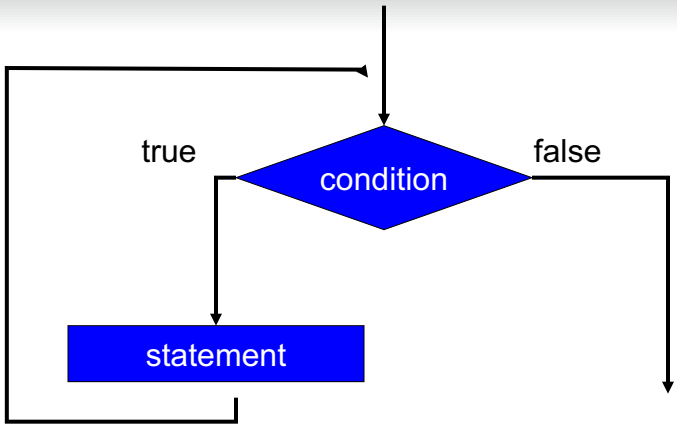
The `while` loop

```
while (expression)
  statement;
```

execute each repetition

- test to start
- test after every repetition

while(condition)



Similar to the if statement

- Check the Boolean condition
- If true, execute the statement/block

Repeat the above until the Boolean is false

Forever loops and never loops

Because the conditional can be "always true" or "always false", you can get a loop that runs forever or never runs

```
int count=0;
while(count ==0) // forever
  cout << "Hi Mom";
while (count=1)//insidious error!!!
  count = 0;
```


small help on confusion, = vs ==

`count = 1` always returns 1 (true)

Possible solution, reverse: `1 == count` is OK, `1 = count` is illegal

```
int count=0;
while(count !=0) // forever
  cout << "Hi Mom";
while (1 = count)//won't compile!
  count = 0;
```

How to count using while

- first, outside the loop, initialize the counter
- test for the counter's value in the boolean
- do the body of the loop
- last thing in the body should change the value of the counter!

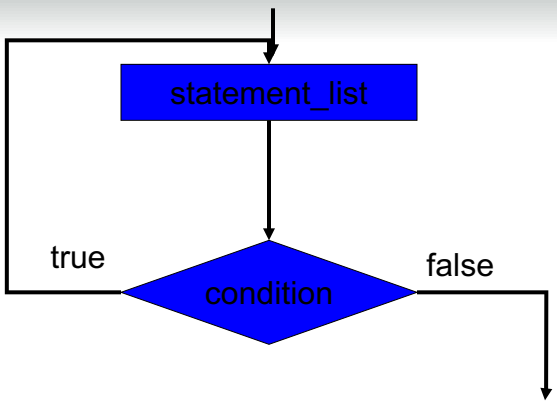
More counting

```
int counter = 0;          //init count
while (counter < 10){     // test count
  cout << "hi mom";
  cout << "Counter is: "
       << counter<<endl;
  counter=counter+1;      //change count
}
```

do ... while

Bottom tested loop

do … while(condition)



Bottom-tested Loop: do

Bottom-tested (posttest)

One trip through loop is guaranteed, i.e. `statement_list` is executed *at least once.*

```
do
  statement;
while (expression);
```

loop body

semi required here

test after every body execution

for loop

Example 3.3

For loop

- `while` loop is pretty general. Anything that can be done using repetition can be done with a `while` loop
- because counting is so common, there is a specialized construct called a `for` loop.
- `for` loop makes it easy to setup a counting loop

Three parts

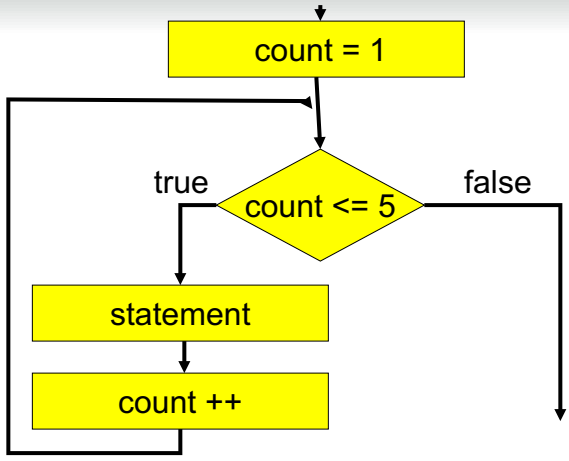
Three parts to a *for* loop (just like the while):

- set the initial value for the counter
- set the condition for the counter
- set how the counter changes each time through the loop

semicolons required

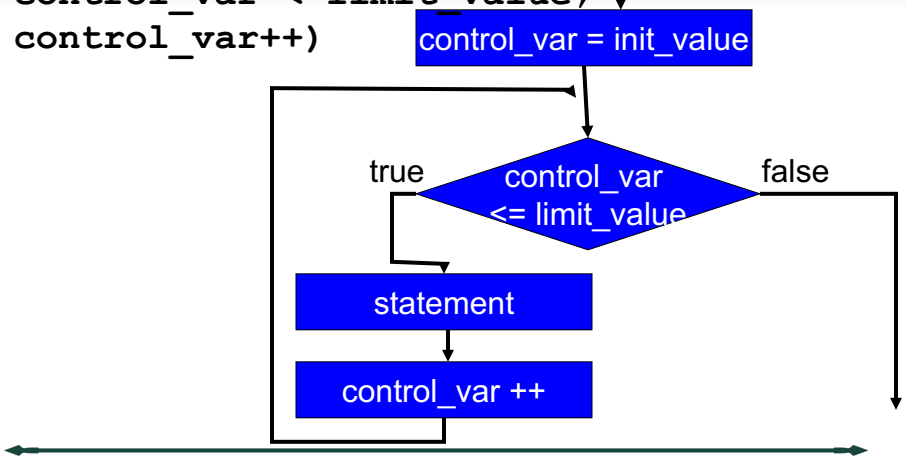
```
for(initialize; condition; change)
    statement;
```


for(count=1; count<=5; count++) statement



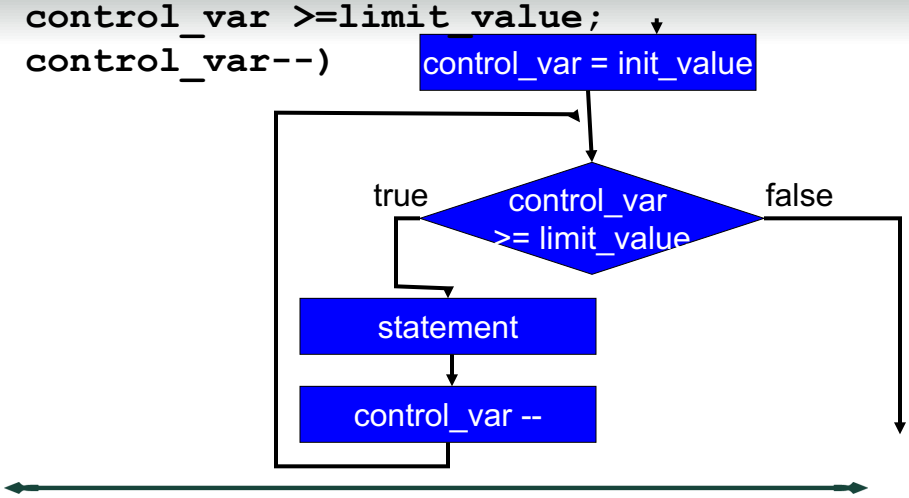
Ascending for :<=,++

```
for (control_var=init_value;
   control_var <=limit_value;
   control_var++)
```



Descending for: >=,--

```
for (control_var=init_value;
   control_var >=limit_value;
   control_var--)
```



Comments

- It is generally considered poor programming practice to alter `control_var` or `limit_var` within the body of the `for` loop.
- The components of the `for` statement can be a arbitrary statements, e.g. the loop condition may be a function call.

Top-tested Equivalence

The following loop

```
for(x=init; x<=limit; x++)
   statement;
```

is equivalent to

```
x=init;
while (x<=limit){
   statement;
   x++;
}
```

declare var inside for

C++ allows you to declare your variable(s) inside the `for` loop

- difference is that, if declared *inside* the `for` loop, than that variable is only available `inside` the loop
  - the *scope* of the variable is the statement/block of the loop

local for var

```
int i = 100;
for(int i=10; i>0; i--)
  cout << i;
cout << i;
```

this `i` is local scope to the loop. prints 10 to 1

this `i` is global scope prints 100

three fields are optional

- no init
- no condition
- no change per iteration

```
int val = 10;
 for(;;){
   if (val <= 0)
     break;
   cout << "Infinite break val:"
        <<val<<endl;
   val-=3;
 }
```

forever loop
need break to get out

comma operator

the comma operator, usually found inside one of the `for` loop fields, is used to perform a *sequence* of operations in that field.

- comma guarantees execution order, left to right

comma example

```
for(int i = 10, j=20; i*j<500; i+=5,j+=5)
    cout <<"Values are i:"<<i<<", "<<j<<endl;
```

two *local* vars in the `for` loop
- both `int`
  - only one declare type allowed
- both initialized
  - `i` first, then `j`

two changes every iteration
- first i up by 5
- then j up by 5

Loop starts with 10→i and 20→j, i and j increment by 5 each iteration, loop ends when i*j > 500

non local exit

Example 3.4

Non local exit

The structure of iteration helps us, as readers, understand clearly when iteration continues and when it ends

While non-local exits can be important, beware that they make the code very difficult to read.

break & continue

`break`

- exit the *nearest enclosing* loop struct (for, while, etc.)
  - if nested, exit to enclosing control

`continue`

- stop the present iteration of the loop, start the next

breaks are for loops/switches

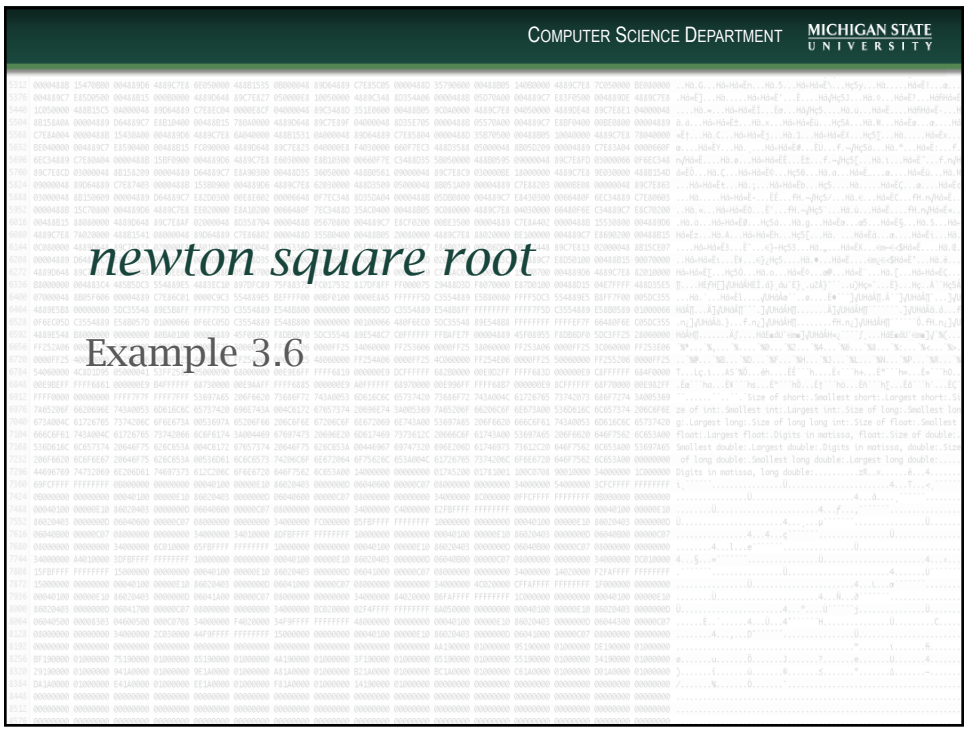
The `break` statement is for loops and the (upcoming) switch statement. Doesn't break out of an if block!

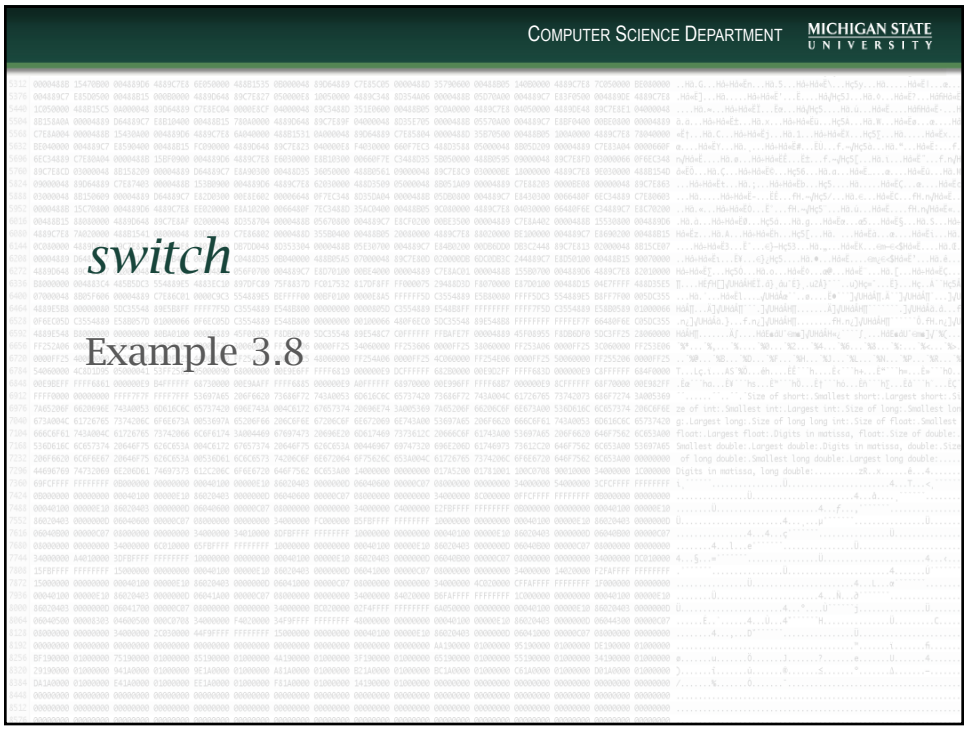
Can `goto` which requires a label.

```
...
goto jmp
...
jmp:
```


count chars

Ex 3.5

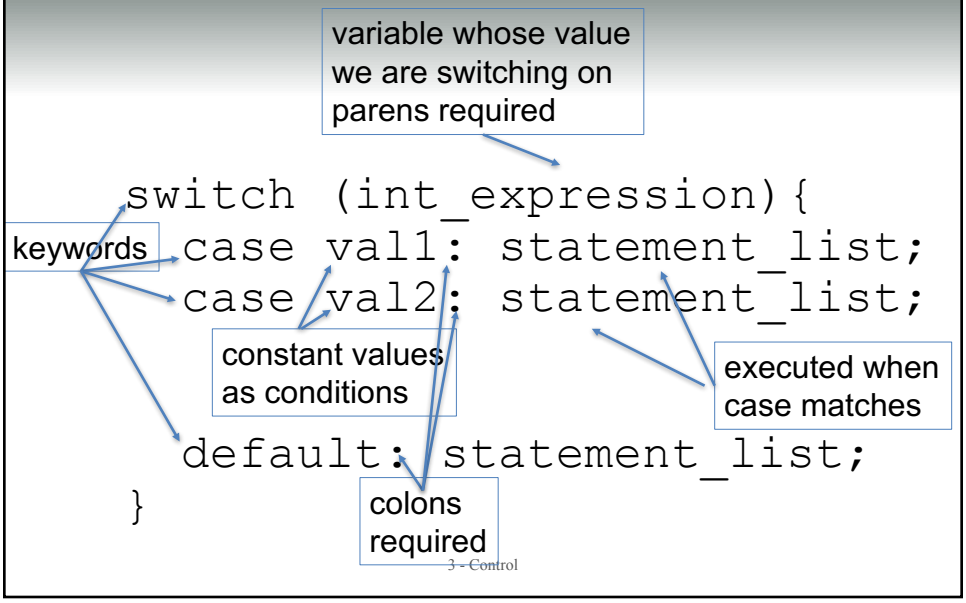

COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
newton square root
Example 3.6


COMPUTER SCIENCE DEPARTMENT
MICHIGAN STATE UNIVERSITY
switch
Example 3.8

Switch Statement

A less general substitute for the multibranch `if`. It is used for selecting among ***discrete values*** (int-ish), i.e. not continuous values.

```
switch (int_expression){
  case val1: statement_list;
  case val2: statement_list;
  …
  default: statement_list;
}
```

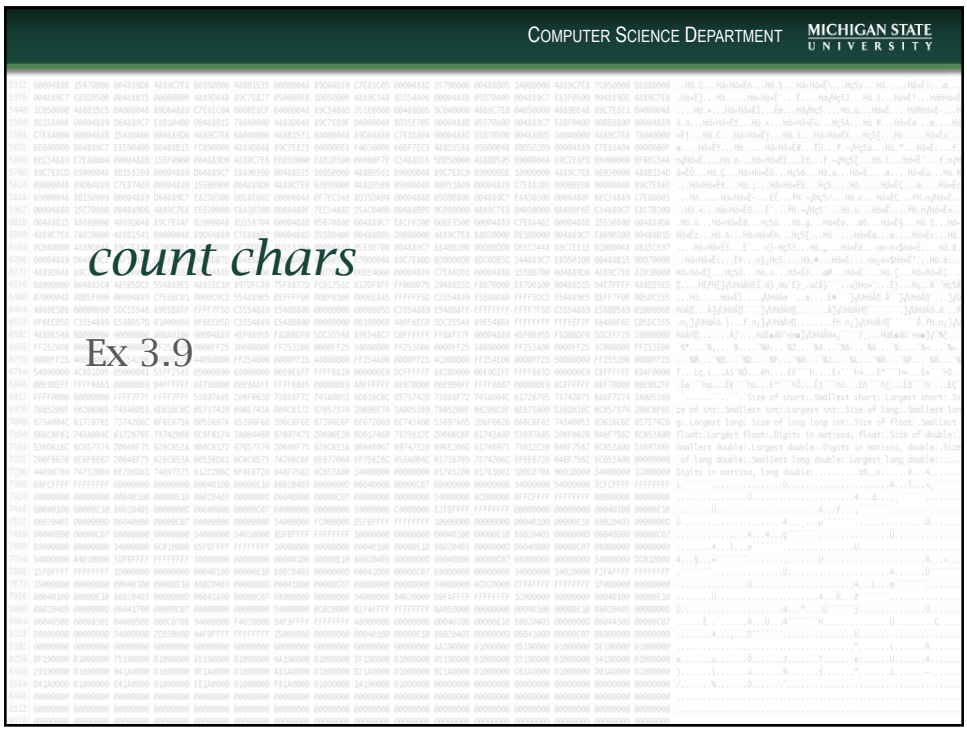
Behavior

1. The `int_expression` is evaluated.
2. If the value is in a `case`, execution begins at that `statement_list`
   1. continues through subsequent `statement_lists` until: `break`, `return`, or end of switch.
3. if no `case` is true, do the default
   1. `default` is optional, do nothing if nothing true and no default

The problem with `break`

You get "fall-through" behavior if you do not put a break at the end of every case group.

Easily forgotten! It's a feature, not a bug, unless you forget!



count chars

Ex 3.9

ternary operator

An if expression does not return a value, but there are times when you would like exactly that: a conditional that returns a value.

That's the ternary operator

- book calls it the conditional operator

Boolean conditional

required elements

```
conditional ? expr1 : expr2;
```

If the Boolean returns true, return result of expr1, else return result of expr2

Similar to the following `if` but with a return.

```
if (cond) then expr1 else expr2;
```

but with a **return** of the appropriate expr.

Examples

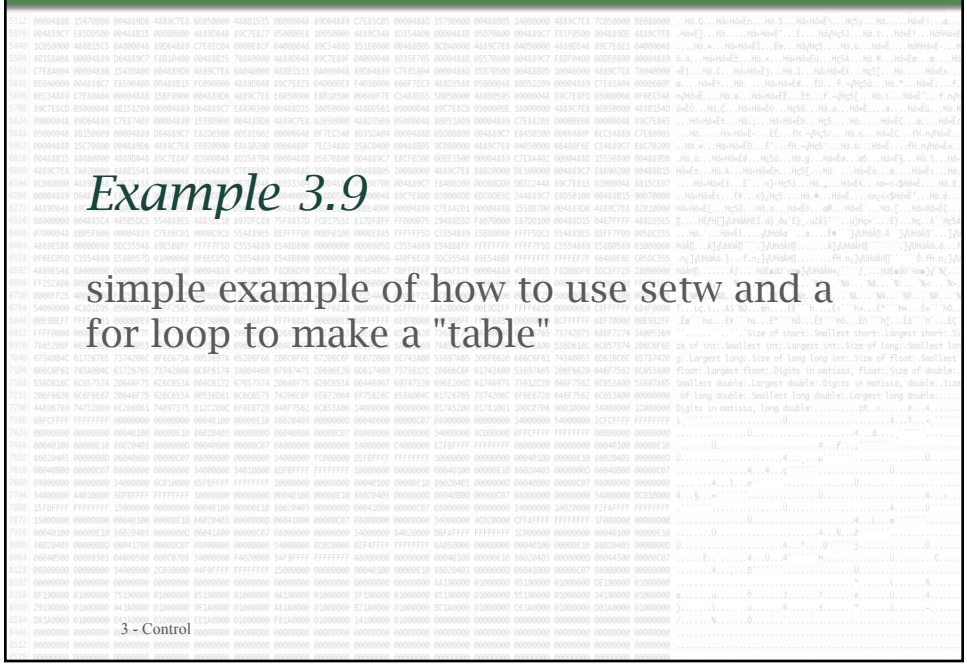
```
int abs_val, val;
cin >> val;
if (val < 0)
   abs_val = -val;
else
   abs_val = val;
```

```
int abs_val, val;
cin >> val;
abs_val = (val < 0) ? -val : val;
```

```
int parity, val;
cin >> val;
if (val % 2)
   parity = 0;
else
   parity = 1;
```

```
int parity, val;
cin >> val;
parity = (val % 2) ? 0 : 1;
```

```
cout << "give me a file name";
cin >> name
std::ostream &sout = name.empty() ?
                     std::cout :
                     ofstream(name);
```

Example 3.9

simple example of how to use setw and a for loop to make a "table"