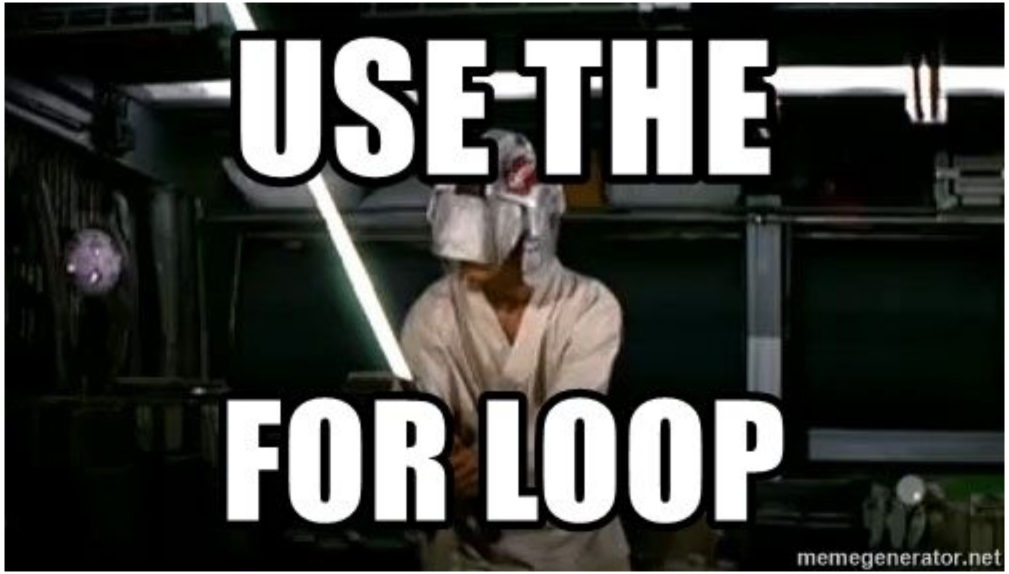
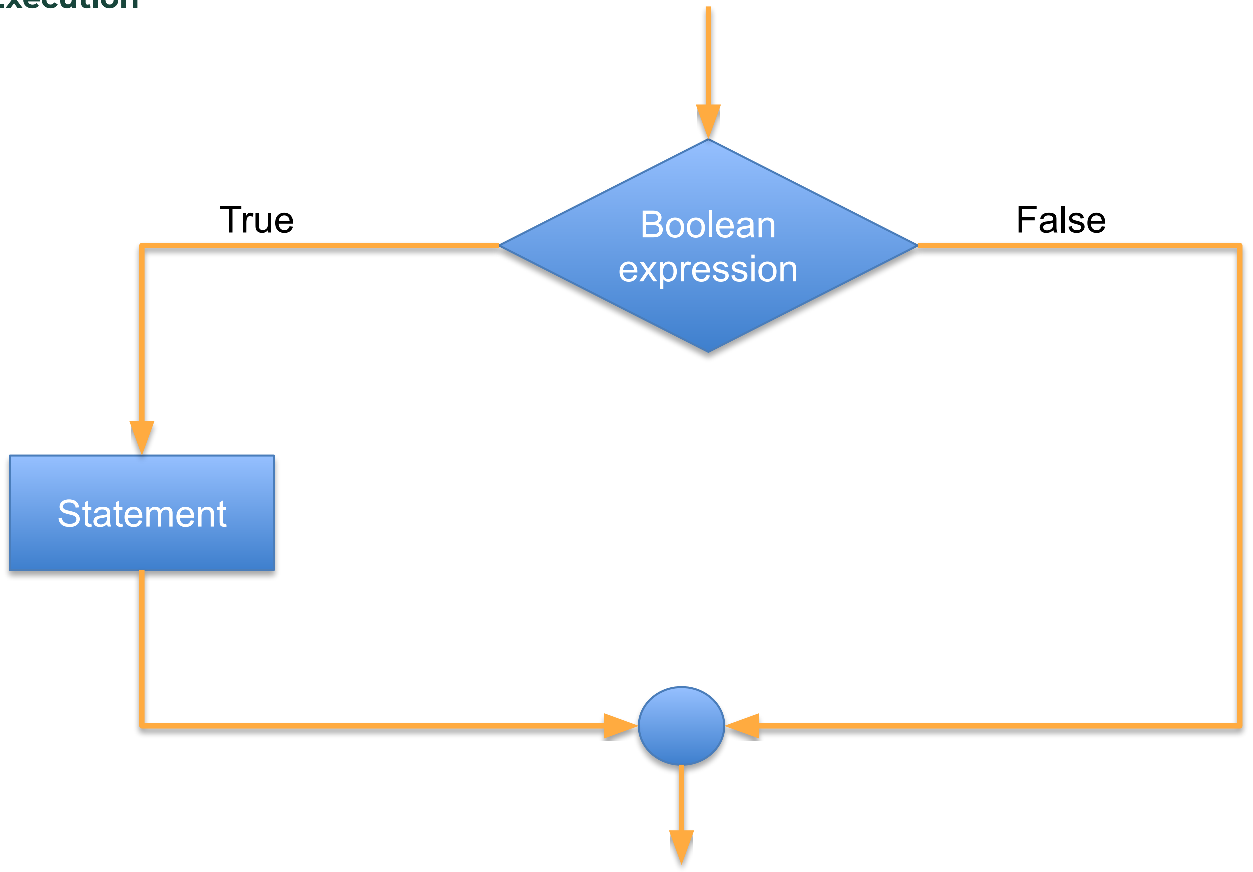
Control Conditionals



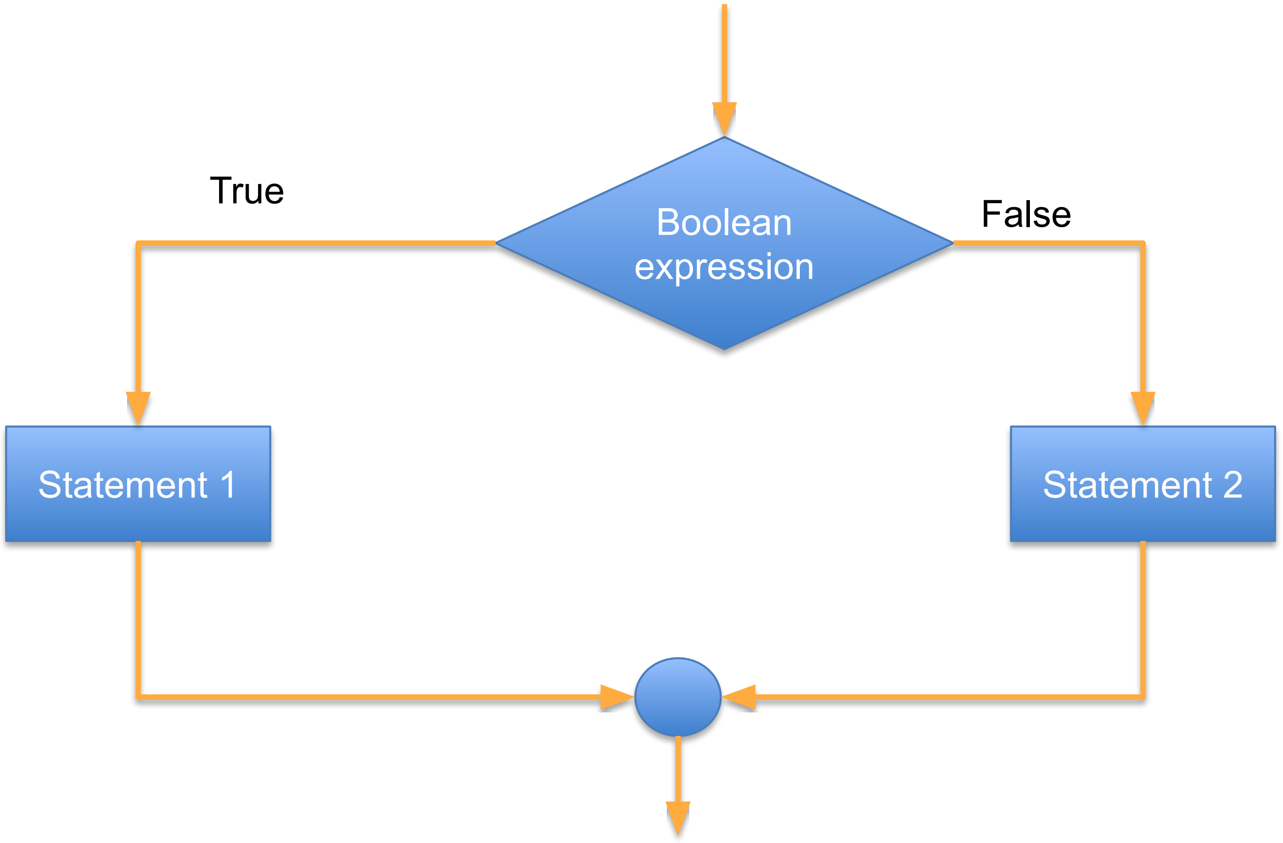
Selective Execution



v1, if alone

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

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
  - Not restricted to selection
- Syntax: set off by brackets {}

Blocks in Selection

```
if (boolean1) {
    statement1;
    statement2;
} else {
    statementA;
    statementB;
}
```

v3, Chain of ifs

```
if (boolean_expr1)
   statement_1;
else if (boolean_expr2)
   statement_2;
else if (boolean_expr3)
   statement_3;
else
   statement_last;
```

- Evaluate Booleans in order
  - If false, go on to the next
- First Boolean that evaluates to true has its statement (or block) run
  - Skip the rest of the `if`s
- If no true Boolean, run the `else`

Nested ifs

```
if (boolean_1) {
    if (boolean_1_1) {
        statement;
    } else if (boolean_1_2) {
        statement;
    } else {
        statement;
    }
} // of boolean_1 if
```

- Formatting is helpful
- It is not required
    - Makes it 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;
```

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

Wrong indentation. `else` goes with the most recent `if` in the code.

What is the output?

```
// Example 2.9
#include<iostream>
int main() {
 int x = -4;
 if (x > 0)
 if (x < 10)
   std::cout << "Orange" << std::endl;
 else
   std::cout << "Yellow" << std::endl;
}
```

- Orange
- Yellow
- (No Output)
- I don't know

Repetition

While Loops

For Loops

Do-While Loops

Three Loops

- `while`
  - top-tested loop (pre-test)
- `for`
  - counting loop
  - forever-sentinel
- `do`
  - bottom-tested loop (post-test)

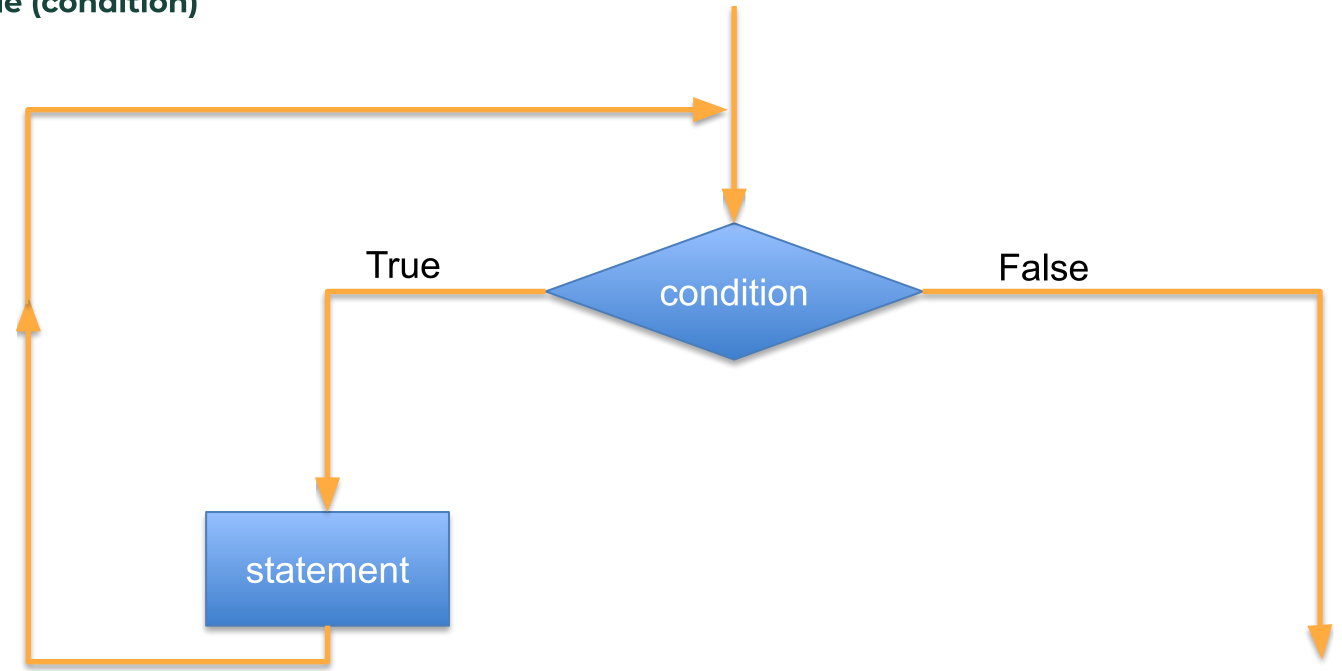
The while loop

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

Test to start
and after every
repetition

Execute
each
repetition

while (condition)



Similar to the if statement

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

= vs ==

- `count = 1` always returns 1 (true)
- Possible solution, reverse: `1 == count` is OK, `1 = count` is illegal

```
int count = 0
while (count != 0) // never
    cout << “Hi Mom”;
while (1 = count) // won’t compile
    count = 0
```

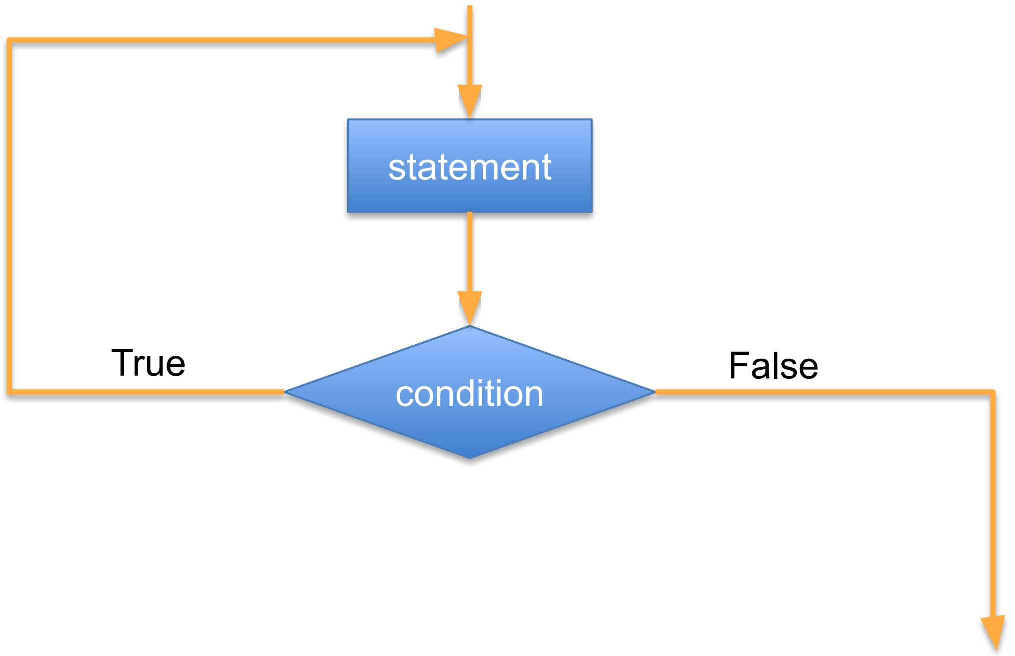
How to count using while

- First, outside the loop, initialize the counter
- Test for the counter’s value in the condition
- Do the body of the loop
- Something in the body should change the value of the counter

More counting

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

do-while (condition)



Bottom-tested loop: do

- Bottom-tested (post-test)
- One trip through the loop is guaranteed
  - i.e. statement is executed *at least* once

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

```cpp
// Example 3.2
#include<iostream>
using std::cout; using std::endl; using std::cin;


int main (){

 // single statement while
 long cnt=0;
 while(cnt < 5)
   ++cnt;
 cout << cnt << endl;
  // while block, count down
 cnt = 5;
 while (cnt > 0){
   cout << cnt <<", ";
   --cnt;
 } // of while
 cout << endl << cnt << endl;

 /* basic while structure
    - set up start condition, outside loop
    - measure condition in Boolean
    - change condition in loop
```

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

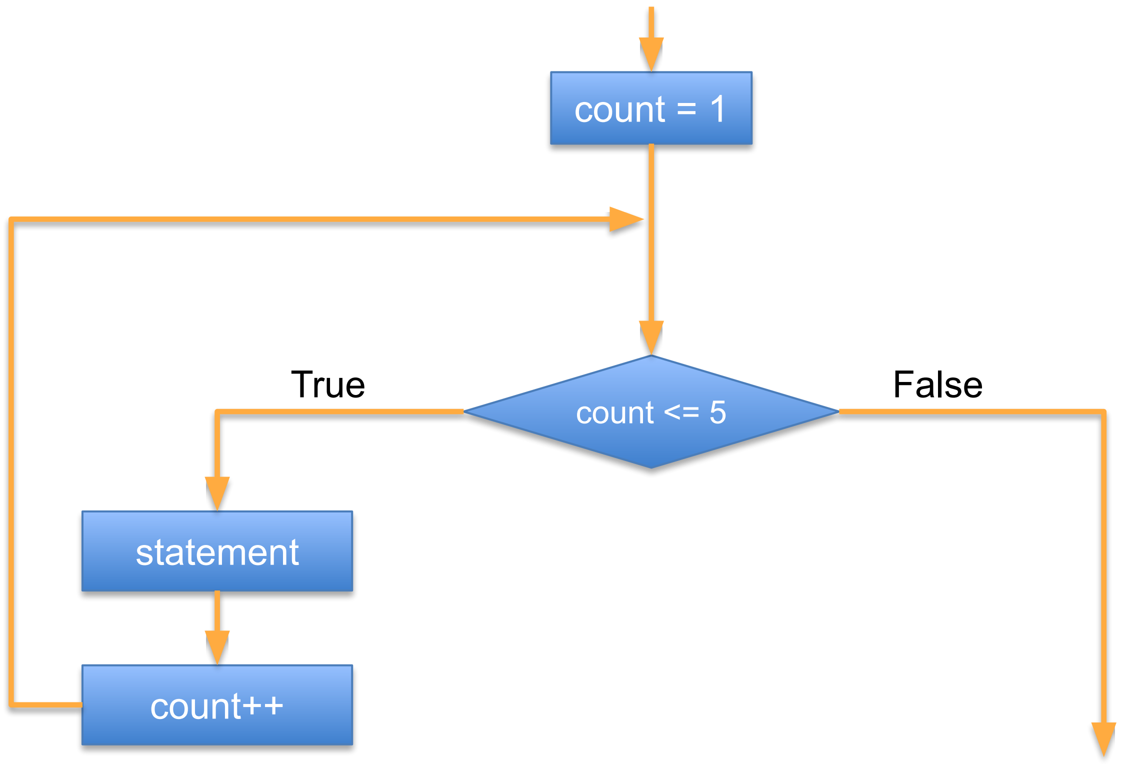
- Three parts to a `for` loop (just like the `while`)
  - Set the initial value for the counter
  - Set the condition for the counter
  - Set how the counter changes each time through the loop

For loop

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

semicolons
required!

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



Comments

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

Which of the following is the least common loop syntax?

- While Loop
- Do-While Loop
- For Loop
- I don't know

Top-tested equivalence

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

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

Declaration inside for

- C++ allows you to declare variables inside the `for` loop
  - If declared *inside* the `for` loop, it is only available *inside* the loop
  - The *scope* of the variable is the statement / block of the loop

Local for variable

```
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

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

- No init
- No condition
- No change per iteration

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” << 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
- Second `j` up by 5

- Loop starts with i=10 and j=20
- i and j both increment by 5 each iteration
- Loop ends when i * j > 500

```cpp
// Example 3.3
#include<iostream>
using std::cout;using std::endl;using std::cin;

int main () {
 // basic for loop
 int i, j;
 for(i = 0; i < 5; ++i)
   cout << i << ", ";
 cout << endl;

 // equivalent while loop
 i = 0;
 while (i < 0){
   cout << i << ", ";
   ++i;
 }
 cout << endl;

 // for with block
 long cnt = 0;
 for(i = 5; i >= 0; --i){
   cout << i << ", ";
```

Non-local exit (break and continue)

- The structure of iteration helps us, as readers, understand clearly when iteration continues and when it ends
- Non-local exits can be important, but beware that they can make the code very difficult to read

Break and Continue

- `break`
  - Exit the nearest enclosing loop struct (`for`, `while`, etc.)
  - If nested, exit to the 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
  - Don’t break out of an `if` block!
- Can `goto` which requires a label (don't ever use this!)

```
…
goto jmp
…
jmp:
```

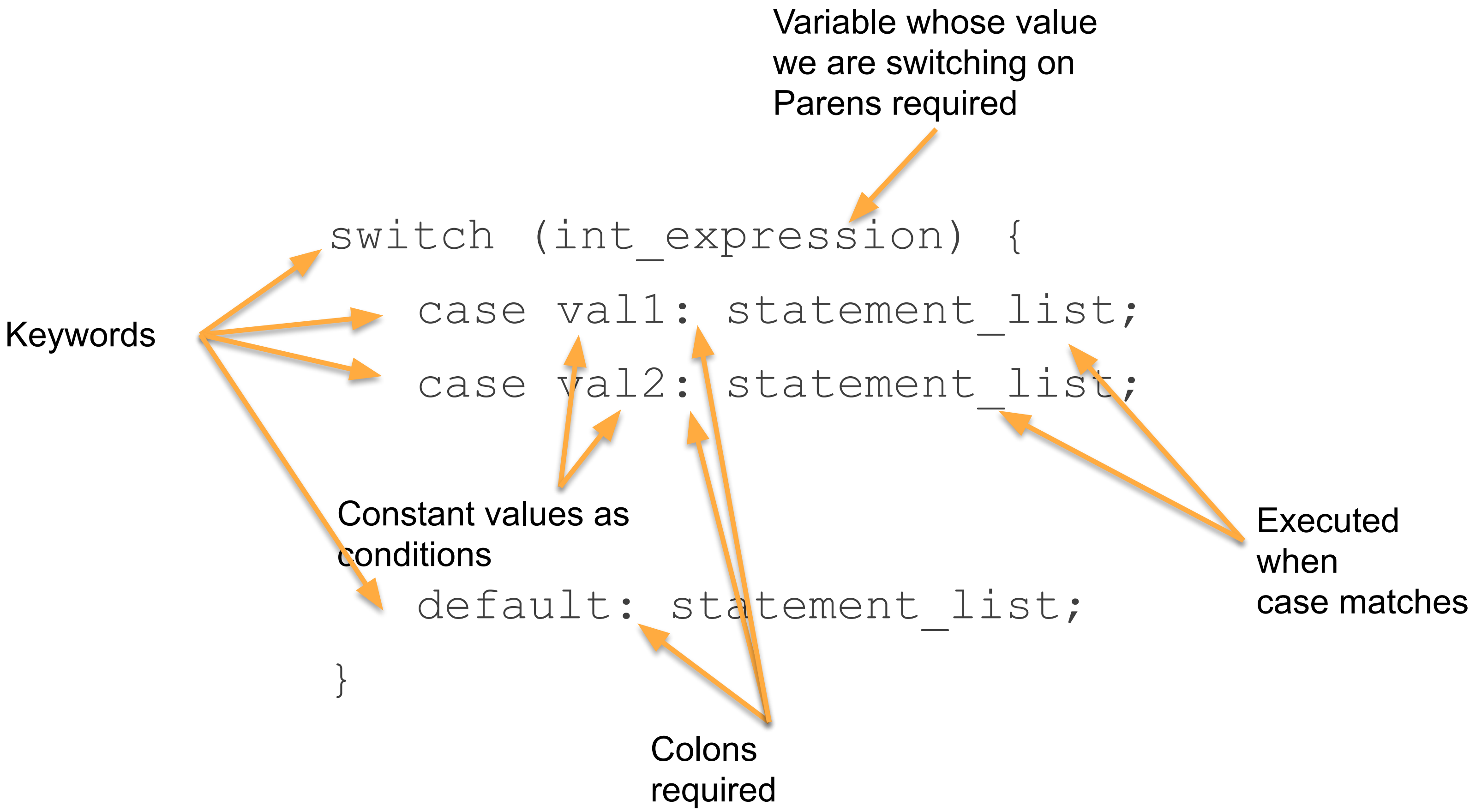
```cpp
// Example 3.4
#include<iostream>
using std::cout; using std::cin; using std::endl;
using std::fixed;
#include<iomanip>
using std::setprecision;
/*
 wfp, 7/23/13
 wfp, updated 1/19/15
 update 1/13/17
 basic break, continue
*/

int main () {

 // sum up the numbers, end at 0
 long sum = 0;
 long num;
 cout << "Enter a number. Ends when 0 entered"<<endl;
 cout << "Number:";
 cin >> num;
 for(;;){     //infinite for
   sum += num;
```

```
// Example 3.5
#include <iostream>
using std::cout; using std::endl; using std::cin;
using std::noskipws;
#include <iomanip>
using std::setw;

/*
 wfp, 9/8/13
 wfp, updated 1/19/15
 Purpose:  Classify each character in an input stream.
*/

int main(){
   char C;                  // Input character
   long char_count  = 0,    // Number of characters in input stream
    line_count   = 0,    // Number of newlines
    blank_count  = 0,    // Number of blanks
    digit_count  = 0,    // Number of digits
    letter_count = 0,    // Number of letters
    other_count  = 0;    // Number of other characters

   cin >> noskipws;
```

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;
}
```



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

   default: statement_list;
}
```

Variable whose value we are switching on Parens required
Keywords
Constant values as conditions
Executed when case matches
Colons required

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_list`s until `break` or `return`
3. If no `case` is true do the default
    1. `default` is optional, do nothing if nothing is true and no default

```
// Example 3.6
#include <iostream>
using std::cout; using std::endl; using std::cin;
#include <iomanip>
using std::noskipws; using std::setw;

/*
 wfp, 7/23/13
 wfp updated 1/19/15
 basic switch
 Purpose:  Count digits in the input stream
*/

int main()
{
   char C;                  // Input character
   long char_count   = 0,     // Number of characters in input stream
    line_count   = 0,     // Number of newlines
    white_count  = 0,     // Number of whitespace characters
    digit_count  = 0,     // Number of digits
    other_count  = 0;     // Number of other characters

   cin >> noskipws;
```

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…

Ternary operator

- An if statement does not return a value
  - Sometimes we want exactly that
- Enter the ternary operator
  - Book calls it the conditional operator

Boolean conditional

Required elements

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

- If the boolean returns true, return the result of expr1, else return result of expr2

- Similar to the following if but with a return.

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

- but with a return of the appropriate expr.

Example

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