More on Loops

The while loop

General syntax:

```
// specify a starting condition
while (condition) {
    // code to repeat until the condition is false
    // modify the condition somehow
}
```

Loops (generally) share three common features:

- a starting condition
- a condition check that occurs before every iteration of the loop
- some way to modify the variables used in the condition (increment a counting variable, change the value of a flag variable on a certain event, etc.)

The while loop

General syntax:

```
// initializer
while (condition) {
    // code to repeat until the condition is false
    // modifier
}
```

Loops (generally) share three common features:

- initializer: define the starting condition
- condition: a condition check that occurs before every iteration of the loop
- modifier: a statement that modifies the variables used in the condition

The for loop

General syntax:

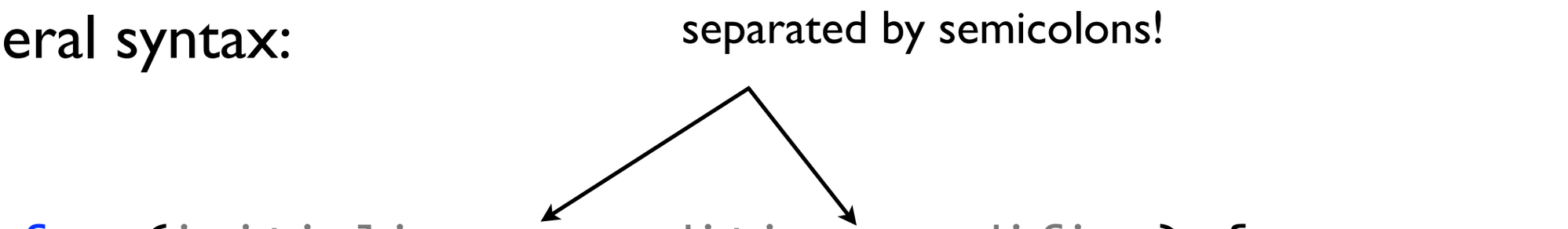
```
for (initializer; condition; modifier) {
    // code to repeat until the condition is false
}
```

The for loop integrates these three components into its structure!

- initializer: define the starting condition
- condition: a condition check that occurs before every iteration of the loop
- modifier: a statement that gets executed *after each iteration* to modify some variable

The for loop

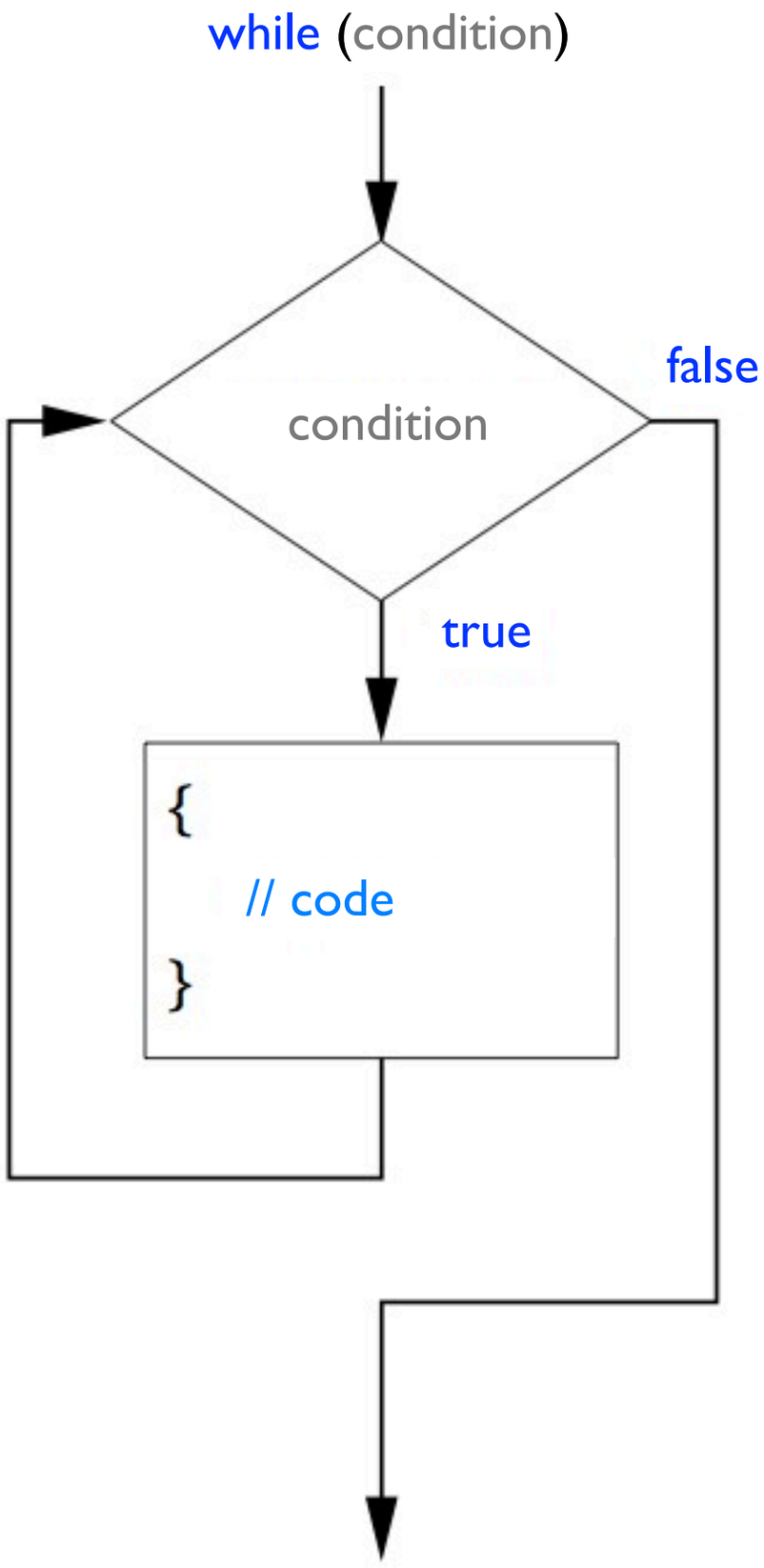
General syntax:



The for loop integrates these three components into its structure!

- each of the three components is separated from the others by a semicolon
- you cannot use commas here... *they have to be semicolons*--it's a rule.

The while loop

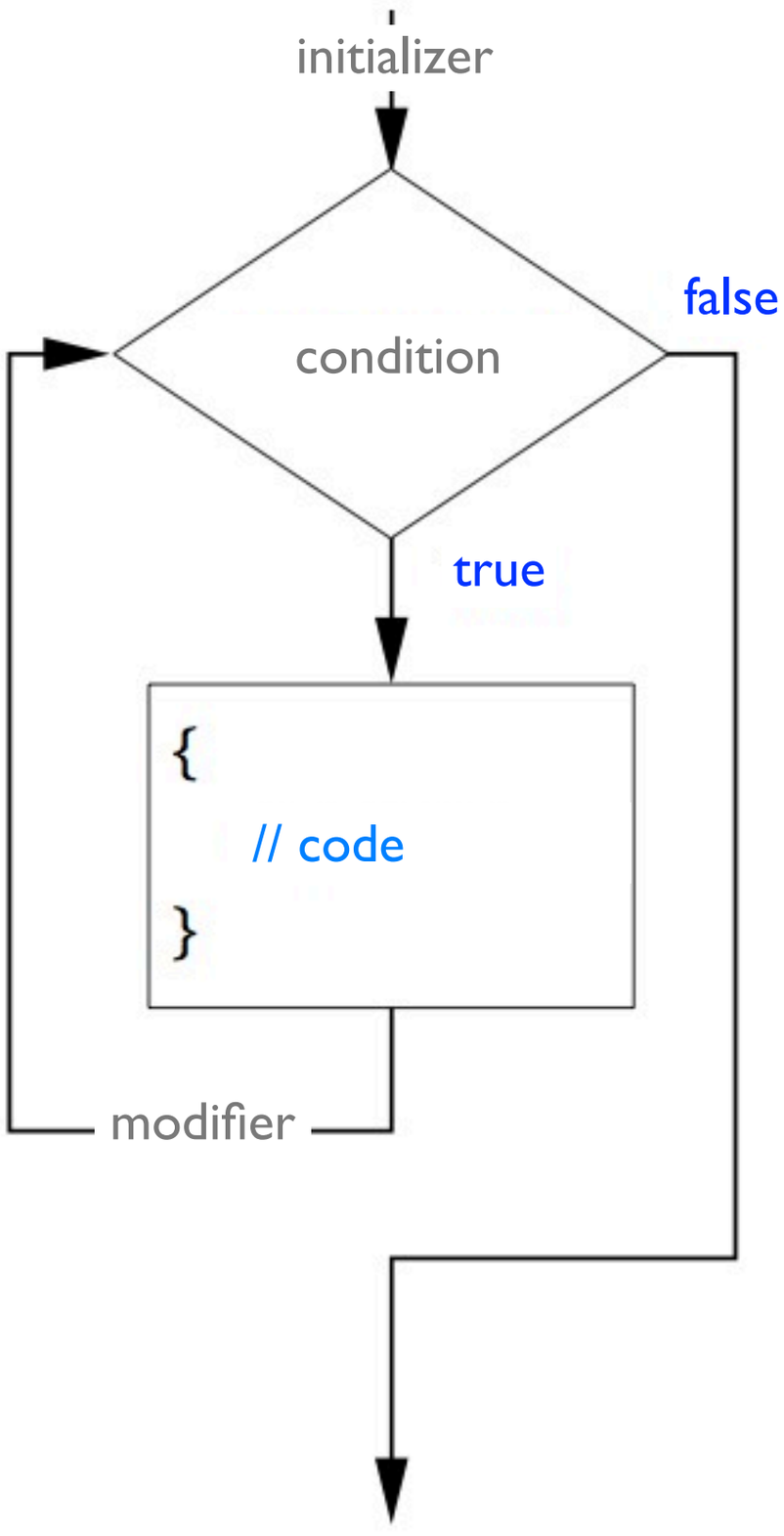


For each iteration of the while loop:

- the condition gets evaluated
- if true, the block of code associated with the while loop gets executed
  - and the loop proceeds to the next iteration
- if false, the loop is finished

The for loop

for (initializer; condition; modifier)



At the start of the for loop:

- initializer sets up the variables to use

For each iteration of the for loop:

- the condition gets evaluated
- if true, the block of code associated with the for loop gets executed
  - afterwards, modifier gets executed
  - and the loop proceeds to the next iteration
- if false, the loop is finished

The for loop

With a little voodoo magic, a for loop acts just like a while loop:

```
// initializer
for ( ; condition; ) {
    // code to repeat until the condition is false
    // modifier
}
```

The initializer, condition, and modifier are separated by semicolons

- you can omit any or all of the three components (but you still have to have the semicolons)
- with only the condition, a for loop acts just like a while loop!

Great… So why two different loops?

Each loop was designed with different patterns in mind.

Counted loop using while:

```
int counter = 1;
while (counter <= 10) {
    cout << “Behold, the number ” << counter << “!” << endl;
    counter++;
}
```

Counted loop using for:

```
for (int n = 1; n <= 10; n++) {
    cout << “Behold, the number ” << n << “!” << endl;
}
```

Great… So why two different loops?

Each loop was designed with different patterns in mind.

Conditional loop using while:

```
char letter = '\0';
while (letter < 'A' || letter > 'Z') {
    cout << "Please enter an uppercase letter: ";
    cin >> letter;
}
```

Conditional loop using for:

```
for (char letter = '\0'; letter < 'A' || letter > 'Z'; cin >> letter) {
    cout << "Please enter an uppercase letter: ";
}
```

Great… So why two different loops?

Each loop was designed with different patterns in mind.

Then there's the matter of personal preference. =)

- use what you like!

Should now be obvious that...

ANY for loop can be converted to a while loop, and vice-versa!

So, convert this to a for loop:

```
int i = 1, f = i;
while (i <= 10) {
    f *= i++;
}
```

Here's one possible conversion (bwahaha!):

```
int i = 1, f = i;
for ( ; i <= 10; ) {
    f *= i++;
}
```

Should now be obvious that...

ANY for loop can be converted to a while loop, and vice-versa!

So, convert this to a for loop:

```
int i = 1, f = i;
while (i <= 10) {
    f *= i++;
}
```

Here's another:

```
for (int i = 1, f = i; i <= 10; ) {
    f *= i++;
}
```

Should now be obvious that...

ANY for loop can be converted to a while loop, and vice-versa!

So, convert this to a for loop:

```
int i = 1, f = i;
while (i <= 10) {
    f *= i++;
}
```

And another (less readable, but valid):

```
for (int i = 1, f = i; i <= 10; f *= i++) {
    // yawn
}
```

Practice AGAIN!

(don't hate me… at least I'm not using an ACTUAL drill)

(you know, to drill the concepts into your head?)

(was that too much of a leap? sorry…)

Fat-Finger Dialing

If the President (mistakenly) called you and asked you to save the Western world by writing a loop to print out each of the letters of the alphabet, could you do it (again)???

He liked your last loops, but he'd really like a for loop this time...

```
// using a char variable as if it were an int (it is!)
for (char letter = 'A'; letter <= 'Z'; letter++) {
    cout << letter;
}

// casting to a char instead
for (int i = 0; i < 26; i++) {
    cout << char('A' + i);
}
```

Bad Timing

In a TOTALLY DIFFERENT alternate future, Earth is still under threat from the Buggers, and Ender is on vacation… Scientists have recently discovered that Buggers are allergic to sequences of perfect squares (1, 4, 9, 16, etc…).

Make the Buggers slightly uncomfortable by writing a FOR loop that prints the squares of the first 100 positive integers!

```
// in ascending order
for (int i = 1; i <= 100; i++) {
    cout << i*i << endl;
}
// in descending order
for (int i = 100; i; i--) {
    cout << i*i << endl;
}
```

Not every situation is life-threatening...

Pretend you're really bored again (don't worry, I know you're **DEFINITELY NOT BORED** this time).

Initialize variables x to 1 and y to 1,000,000,000. Write a FOR loop that multiplies x by 2 and divides y by 3 until x is bigger than y. After the loop ends, print out the number of iterations required.

```
// i has to be declared here, not in the for loop! Why?
int i = 0;


for (int x = 1, y = 1E9; x <= y; i++) {
    x *= 2;
    y /= 3;
}
cout << "x > y after " << i << " iterations" << endl;
```

Variable Scope

Variables only exist in certain *scope*s or contexts

- Each block of code creates a new *scope* `{ /* new scope */ }`
- Nested blocks inherit the variables of their parent scopes
- Variables declared in a nested scope are NOT available to their parent scopes
- Variables must be declared BEFORE they are referenced.
- This is an important concept for understanding C++, especially when we talk about functions!

Some terminology:

- "global" scope: available everywhere in your program (after the declaration)
- "function" scope: available only within the given function (e.g. `main()`)
- "block" scope: available only in the given block and blocks nested within it

Variable Scope

Example (doesn't work):

```
if (single) {
    // will only be available in the if clause!
    string relation_status = “single”;
} else {
    // will only be available in the else clause!
    string relation_status = “complicated”;
}

// relation_status is NOT available here!
cout << “Your relationship status: ” << relation_status << endl;
```

Variable Scope

Example (this *does* work):

```
// declare the variable in the outer scope
string relation_status;


if (single) {
    // relation_status was declared in a parent scope
    relation_status = “single”;
} else {
    // relation_status was declared in a parent scope
    relation_status = “complicated”;
}


// Hooray! This works like expected (same scope)
cout << “Your relationship status: ” << relation_status << endl;
```

Variable Scope

```
1  using namespace std;
2
3  // GLOBAL SCOPE
4  int x;
5
6  int main() {
7
8      // MAIN FUNCTION SCOPE
9      int y = x; // x is visible in this scope
10
11     if (1) {
12         // FIRST IF-STATEMENT SCOPE
13         int z = y; // y is visible in this scope
14
15         if (1) {
16             // SECOND IF-STATEMENT SCOPE
17             int a = z; // z is visible in this scope
18         }
19
20         // a DOES NOT exist here!
21         z = a;
22     }
23
24     return 0;
25 }
26
```

Scope, you say?

Variables only exist in their relevant scopes

- those declared in the initializer of a for loop only exist within that for loop.
- because the variable `i` is referenced outside of the for loop, it cannot be declared in the for-loop initializer!

Example code:

```
// i has to be declared here, not in the for loop! Why?
int i = 0;


for (int x = 1, y = 1E9; x <= y; i++) {
    x *= 2;
    y /= 3;
}
cout << “x > y after ” << i << “ iterations” << endl;
```

i is used outside the loop!

break keyword

The break statement immediately jumps to the first statement outside of the current (innermost) loop

- basically exits the current loop

General Syntax:

```
break; // that's all there is to it!
```

You typically use break inside a selection statement:

```
if (end_of_loop_condition) {
    break;
}
```

break keyword

The break statement immediately jumps to the first statement outside of the current (innermost) loop

- basically exits the current loop

What do you see when this loop runs?

```
for (int i = 0; i < 100; i++) {
    cout << “i is now: ”; // how many times do we see this?

    if (i > 10) {
        break;
    }

    cout << i; // and how many times do we see this?
}
```

Nested Loops

Loops can occur within other loops.

```
for (int i = 0; i < 8; i++) {
    // this loop is repeated every time through the outer loop
    for (int j = 1; j <= 8; j++) {
        cout << '*';
    }
    cout << endl;
}
```

Common uses:

- creating 2-dimensional data / output
- sorting
- test questions ^_^

Nested Loops

Loops can occur within other loops.

How many question marks get printed?

```
for (int x = 0; x < 5; x++) {
    for (int y = x; y >= 1; y--) {
        cout << '?';
    }
}
```