Loops and Iteration

CS 1044

Iteration

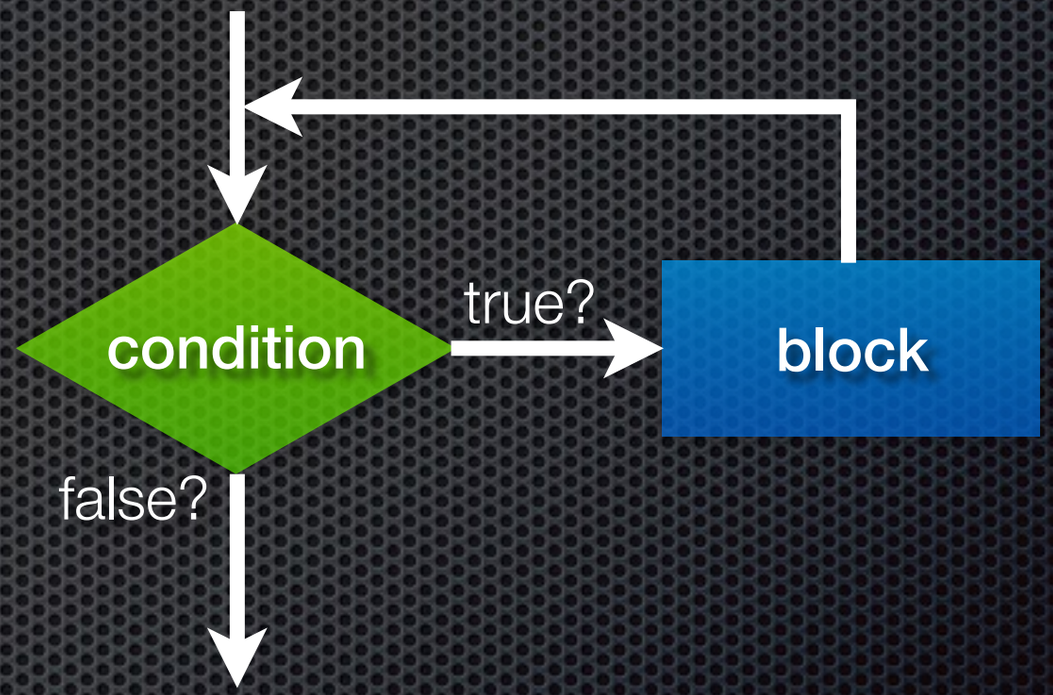
- Third type of control flow
- Remember: Selection says “if a condition is true, do something”
- **Iteration** says “**while** a condition is true, do something over and over **until** it becomes false”
- Just like `if` statements, most iteration is controlled using **Boolean expressions**

Why Do We Need Loops?

- So far, we've only dealt with small amounts of data
- What if we have a larger amount, and all of this data is processed the same way?
- Or, if we just want to repeat the same task without knowing exactly how many times it should run

`while` Loop

```
while (condition)
{
    block;
}
```



`block` is **one or more statements** that will be executed **as long as** *`condition`* is true

Tricky Points about Loops

- If the condition is **not true** when the loop is first reached, the block inside **is not executed at all**
- The condition is only checked at the **beginning of each loop cycle** – if the condition changes inside the block, the rest of the **block still finishes** before the condition is checked again
- The block inside the loop must be able to **change the outcome of the condition** or the loop will never terminate (**infinite loop**)

```
bool cond = false;
// First bullet on the last slide.
while(cond)
{
   // Your code won’t execute.
}
```

```
int x = 0;
// Second example.
while(x < 10)
{
   x++;
   // Code here executes
   // before reevaluating x < 10
}
```

```
int x = 0, y = 0;

// Third example.
while(x < 10)
{
   y++;
   // x never changes, so x < 10
   // will never be false.
   // This is an “infinite loop”.

}
```

Example: Summing Integers

- Let's say we want to sum an **arbitrary number** of integers that read from the user (the keyboard).
- We'll ask the user if they want continue summing numbers **each time the loop executes.**

```
int sum = 0, num = 0;
string answer = “yes”;

while(answer == “yes”)
{
   cout << “Enter a number: “;
   cin >> num;

   sum += num;

   cout << “Go again? (yes/no)? ”;
   cin >> answer;
}
```

`for` Loop

```
for (initializer; condition; updater)
{
    block;
}
```

- *`initializer`* is executed **only once**, before the loop begins (regardless of *`condition`*)
- *`condition`* is the same as it is in a `while` loop
- *`update`* is executed at the end of each pass of the loop, immediately after the *`block`* and before *`condition`* is tested again

```
for (int x = 0; x < 10; x++)
{
   // x takes on a different value
   // for each iteration of the loop.

   // x starts at 0 and goes till 9.
   cout << x << endl;
}

// Using x here will cause an error.
cout << x << endl;
```

`for`/`while` Equivalence

```
for (initializer; condition; updater)
{
    block;
}
```

... is **exactly** the same as...

```
initializer;
while (condition)
{
    block;
    updater;
}
```

Why Use `for` Loops?

- Mostly used for loops that involve **counting**
- We see this pattern very frequently:

```
for (int var = start; var < end; var++)
{
    block;
}
```

- Loop is executed `(end – start)` times:
  `start`, `start + 1`, …, `end – 2`, `end – 1`, DONE

`for` Loop Patterns

- Counting forward

```
for (int var = start; var < end; var++)
{
    block;
}
```

Counts up from `start` to `end`, *excluding* `end`

```
for (int var = start; var <= end; var++)
{
    block;
}
```

Counts up from `start` to `end`, *including* `end`

`for` Loop Patterns

- Counting backward

```
for (int var = end; var >= start; var--)
{
    block;
}
```

Counts down from **`end`** to **`start`**,
including both

Finer Loop Control

- `break` statement: Use inside a loop to exit it **immediately**
- `continue` statement: Use inside a loop to **skip the rest of the block** and immediately start a new cycle
- If used in nested loops, **only the innermost loop** is exited

```
int y = 0;

// Break statement example.
for(int x = 0; x < 10; x++)
{
    y++;
    break;

    // Nothing here is executed.
    cout << y;
}
```

Intentionally Infinite Loops

```
while (true)
{
    block;
}
```

```
for (;;)
{
    block;
}
```

- You may encounter both – they are exactly the same
- The `while` version is probably clearer
- The block inside the loop **must** have a statement that will transfer control out the loop, such as `break`

Nested Loops

- Just like every other control structure, you can **nest** loops if you want

- Common application: Nesting `for` loops to process every part of a **2-dimensional** structure, like a grid

Nested Loops

```
for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {
        // do something with x and y...
    }
}
```

- The inner loop makes a complete run-through each time the outer loop runs