Computer Science I

Loops

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

1. Introduction
2. For loops
3. While loops
4. Other Loops & Issues
5. Pitfalls
6. Exercises

Part I: Introduction

Motivation & Introduction to Loop Control Structures

Motivation

- Need a way to *repeatedly* execute a block of code

Motivation

- Need a way to *repeatedly* execute a block of code
- Process data: apply an operation to each piece of data

Motivation

- Need a way to *repeatedly* execute a block of code
- Process data: apply an operation to each piece of data
- Repeat an operation until some condition is satisfied

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied
- While some boolean expression or condition evaluates to true, the loop will continue to execute

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied
- While some boolean expression or condition evaluates to true, the loop will continue to execute
- Once the condition is no longer satisfied, the loop *terminates* its execution

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied
- While some boolean expression or condition evaluates to true, the loop will continue to execute
- Once the condition is no longer satisfied, the loop *terminates* its execution
- Each time a loop executes is referred to as an *iteration*

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied
- While some boolean expression or condition evaluates to true, the loop will continue to execute
- Once the condition is no longer satisfied, the loop *terminates* its execution
- Each time a loop executes is referred to as an *iteration*
- Different types of loops

Loops

- A *loop* allows us to *repeatedly* execute a block of code while some *condition* is satisfied
- While some boolean expression or condition evaluates to true, the loop will continue to execute
- Once the condition is no longer satisfied, the loop *terminates* its execution
- Each time a loop executes is referred to as an *iteration*
- Different types of loops
- Components of a loop

Loop Components

A loop has several main components:

1. An initialization statement – a statement that indicates how the loop *begins*

Loop Components

A loop has several main components:

1. An initialization statement – a statement that indicates how the loop *begins*
2. A continuation condition – a boolean statement that specifies whether the loop should continue executing

Loop Components

A loop has several main components:

1. An initialization statement – a statement that indicates how the loop *begins*
2. A continuation condition – a boolean statement that specifies whether the loop should continue executing
3. An iteration statement – a statement that makes progress toward the termination of the loop

Loop Components

A loop has several main components:

1. An initialization statement – a statement that indicates how the loop *begins*
2. A continuation condition – a boolean statement that specifies whether the loop should continue executing
3. An iteration statement – a statement that makes progress toward the termination of the loop
4. Loop Body – the block of code that gets executed for each *iteration*

Loop Components

A loop has several main components:

1. An initialization statement – a statement that indicates how the loop *begins*
2. A continuation condition – a boolean statement that specifies whether the loop should continue executing
3. An iteration statement – a statement that makes progress toward the termination of the loop
4. Loop Body – the block of code that gets executed for each *iteration*

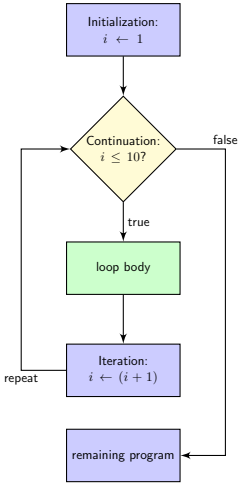
Termination Condition – negation of the continuation condition

Simple Example

Printout numbers 1 through 10:

1. Initialize a variable i to 1
2. While the variable i's value is less than or equal to 10...
3. Print i
4. Increment i by adding 1 to it
5. Go to step 2

Flow Diagram



Part II: For Loops

For Loop

- A a *for loop* uses the keyword `for`

For Loop

- A a *for loop* uses the keyword `for`
- All three elements: initialization, continuation and increment are

For Loop

- A a *for loop* uses the keyword `for`
- All three elements: initialization, continuation and increment are
  - placed on the same line

For Loop

- A a *for loop* uses the keyword `for`
- All three elements: initialization, continuation and increment are
  - placed on the same line
  - enclosed within parentheses

For Loop

- A a *for loop* uses the keyword `for`
- All three elements: initialization, continuation and increment are
  - placed on the same line
  - enclosed within parentheses
- Loop body is denoted using curly brackets

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop
- New syntax: `i++`

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop
- New syntax: `i++`
- Loop body denoted with curly brackets

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop
- New syntax: `i++`
- Loop body denoted with curly brackets
- Correct usage of semicolons

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop
- New syntax: `i++`
- Loop body denoted with curly brackets
- Correct usage of semicolons
- Style elements

For Loop


```
1  int i;
2  for(i=1; i<=10; i++) {
3    printf("i = %d\n", i);
4  }
```

- Initialization, executed only once before the loop begins
- Continuation condition, evaluated before each iteration
- Iteration executed at the end of each loop
- New syntax: `i++`
- Loop body denoted with curly brackets
- Correct usage of semicolons
- Style elements
- Demonstration

Increment Operators

Several common “short-hand” operators for increment/decrement:

Increment Operators

Several common "short-hand" operators for increment/decrement:

- `i++` adds 1 to the variable `i`

Increment Operators

Several common "short-hand" operators for increment/decrement:

- `i++` adds 1 to the variable `i`
- `i--` subtracts 1 from the variable `i`

Increment Operators

Several common "short-hand" operators for increment/decrement:

- `i++` adds 1 to the variable `i`
- `i--` subtracts 1 from the variable `i`
- "Equivalent" to `i = i + 1` and `i = i - 1`

Increment Operators

Several common "short-hand" operators for increment/decrement:

- `i++` adds 1 to the variable `i`
- `i--` subtracts 1 from the variable `i`
- "Equivalent" to `i = i + 1` and `i = i - 1`
- Honorable mention: prefix versions `++i` and `--i`

More Increment Operators

You can also use compound increment operators:

More Increment Operators

You can also use compound increment operators:

- `a += 10;` adds 10 to the variable `a`

More Increment Operators

You can also use compound increment operators:

- `a += 10;` adds 10 to the variable `a`
- `a -= 5;` subtracts 5 from the variable `a`

More Increment Operators

You can also use compound increment operators:

- `a += 10;` adds 10 to the variable `a`
- `a -= 5;` subtracts 5 from the variable `a`
- `a *= 2;` multiplies `a` by 2

More Increment Operators

You can also use compound increment operators:

- `a += 10;` adds 10 to the variable `a`
- `a -= 5;` subtracts 5 from the variable `a`
- `a *= 2;` multiplies `a` by 2
- `a /= 3;` divides `a` by 3

More Examples


```
1  //print numbers 10, 20, 30, ... 100
2  int i;
3  for(i=10; i<=100; i+=10) {
4    printf("%d\n", i);
5  }
6  printf("%d\n", i);
7
8  int n = 10;
9  int sum = 0; //without initialization, no default value
10 for(i=1; i<n; i++) {
11   //sum = sum + i;
12   sum += i;
13 }
14 printf("sum of integers 1..%d = %d\n", n, sum);
```

Part III: While Loops

While Loops

- A a *while loop* uses the keyword `while`

While Loops

- A a *while loop* uses the keyword `while`
- Three elements (initialization, continuation, increment) are not on the same line

While Loops

- A a *while loop* uses the keyword `while`
- Three elements (initialization, continuation, increment) are not on the same line
- Same behavior: continuation condition is checked at the start of the loop

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    printf("i = %d\n", i);
4    i++;
5  }
```

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    printf("i = %d\n", i);
4    i++;
5  }
```

- Initialization is before and outside the loop structure

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    printf("i = %d\n", i);
4    i++;
5  }
```

- Initialization is before and outside the loop structure
- `while` statement contains only the continuation condition

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    printf("i = %d\n", i);
4    i++;
5  }
```

- Initialization is before and outside the loop structure
- `while` statement contains only the continuation condition
- Increment is done *inside* the loop body

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    printf("i = %d\n", i);
4    i++;
5  }
```

- Initialization is before and outside the loop structure
- `while` statement contains only the continuation condition
- Increment is done *inside* the loop body
- Take care: order inside the loop matters

While Loops


```
1  int i = 1;
2  while(i <= 10) {
3    i++;
4    printf("i = %d\n", i);
5  }
```

Why Multiple Loops?

Why do we have multiple loop control structures?

Why Multiple Loops?

Why do we have multiple loop control structures?

- Any for loop can be rewritten as a while loop and vice versa

Why Multiple Loops?

Why do we have multiple loop control structures?

- Any for loop can be rewritten as a while loop and vice versa
- Syntactic sugar, flexibility, variety

Why Multiple Loops?

Why do we have multiple loop control structures?

- Any for loop can be rewritten as a while loop and vice versa
- Syntactic sugar, flexibility, variety
- Generally, the situation/context will inform how "natural" each loop is

Why Multiple Loops?

Why do we have multiple loop control structures?

- Any for loop can be rewritten as a while loop and vice versa
- Syntactic sugar, flexibility, variety
- Generally, the situation/context will inform how "natural" each loop is
- Use for loops when the number of iterations is known up front

Why Multiple Loops?

Why do we have multiple loop control structures?

- Any for loop can be rewritten as a while loop and vice versa
- Syntactic sugar, flexibility, variety
- Generally, the situation/context will inform how "natural" each loop is
- Use for loops when the number of iterations is known up front
- Use while loops when you don't know how many iterations will be executed/they may vary depending on the variable values

While Loop Scenario

Example: write a loop (or loops) to *normalize* a number:

$$89237.49283 \rightarrow 8.923749283 \times 10^4$$

$$321.321 \rightarrow 3.21321 \times 10^2$$

$$0.00432 \rightarrow 4.32 \times 10^{-3}$$

While Loop Scenario


```
1  double x = 89237.49283;
2  int counter = 0;
3  while(x > 10) {
4    x /= 10.0;
5    counter++;
6  }
7  printf("x = %f * 10^%d\n", x, counter);
8  while(x < 1) {
9    x *= 10.0;
10   counter--;
11 }
12 printf("x = %f * 10^%d\n", x, counter);
13 return 0;
```

Part IV: Other Issues & Types of Loops

Loop Variable Scoping

- For loop examples declared the iteration variable before the loop

```
1  int i;
2  for(i=0; i<=10; i++) {
3    ...
4  }
```

Loop Variable Scoping

- For loop examples declared the iteration variable before the loop

```
1  int i;
2  for(i=0; i<=10; i++) {
3    ...
4  }
```

- The *scope* of `i` lasted after the loop

Loop Variable Scoping

- Many modern languages and newer versions of C (C99) allow a loop-scoped variable declaration:

```
1  for(int i=0; i<=10; i++) {
2    ...
3  }
```

Loop Variable Scoping

- Many modern languages and newer versions of C (C99) allow a loop-scoped variable declaration:

```
1  for(int i=0; i<=10; i++) {
2    ...
3  }
```

- Scope of `i` is limited to the loop

Loop Variable Scoping

- Many modern languages and newer versions of C (C99) allow a loop-scoped variable declaration:

```
1  for(int i=0; i<=10; i++) {
2    ...
3  }
```

- Scope of `i` is limited to the loop
- If you use this modern style, you must compile with:
  `gcc -std=c99` or
  `c99`

Flag-Controlled Loops

- Instead of a continuation condition, we could use a boolean “flag” variable

Flag-Controlled Loops

- Instead of a continuation condition, we could use a boolean "flag" variable
- Most commonly used with while loops

```
1  int flag = 1;
2  while(flag) {
3    ...
4    if(<some complex logic>) {
5      flag = 0;
6    }
7  }
```

Flag-Controlled Loops

- Instead of a continuation condition, we could use a boolean "flag" variable
- Most commonly used with while loops

```
1  int flag = 1;
2  while(flag) {
3    ...
4    if(<some complex logic>) {
5      flag = 0;
6    }
7  }
```

- Sometimes, people use `break` to break out instead:

```
1  while(1) {
2    ...
3    if(<some complex logic>) {
4      break;
5    }
6  }
```

Do-While Loops

- Do-while loops check the continuation condition at the *end* of the loop

Do-While Loops

- Do-while loops check the continuation condition at the *end* of the loop
- Consequence: they always execute *at least once*

Do-While Loops

- Do-while loops check the continuation condition at the *end* of the loop
- Consequence: they always execute *at least once*
- Example:

```
1  int i = 0;
2  do {
3    i++;
4    print("i = %d\n", i);
5  } while(i < 10);
```

For Each Loops

- Many languages support “for each” loops

For Each Loops

- Many languages support “for each” loops
- Used to iterate over “collections” (arrays, sets, lists, etc.)

For Each Loops

- Many languages support “for each” loops
- Used to iterate over “collections” (arrays, sets, lists, etc.)
- Not supported in C

Nesting Loops

- Loops can be written inside of other loops

Nesting Loops

- Loops can be written inside of other loops
- Called “nesting” or *nested loops*

Nesting Loops

- Loops can be written inside of other loops
- Called "nesting" or *nested loops*
- Very common when iterating over matrices/tables of data

Nesting Loops

- Loops can be written inside of other loops
- Called "nesting" or *nested loops*
- Very common when iterating over matrices/tables of data
- For each row, then for each column in the row

Nesting Loops

- Loops can be written inside of other loops
- Called "nesting" or *nested loops*
- Very common when iterating over matrices/tables of data
- For each row, then for each column in the row
- Example:

```
1  int i, j;
2  for(i=1; i<=10; i++) {
3    for(j=1; j<=10; j++) {
4      printf("%d\n", (i+j));
5    }
6  }
```

Part V: Pitfalls
Common Errors & Misconceptions

Consider the following code:

```
1  int i = 1;
2  while(i <= 10) {
3    printf("%d\n", i);
4  }
```

Pitfall
Improper Increment

Consider the following code:

```
1  int i = 1;
2  while(i <= 10) {
3    printf("%d\n", i);
4  }
```

- Results in an infinite loop

Pitfall
Improper Increment

Consider the following code:

```
1  int i = 1;
2  while(i <= 10) {
3    printf("%d\n", i);
4  }
```

- Results in an infinite loop
- There is no increment/progress toward the termination condition

Pitfall
Improper Increment

Consider the following code:

```
1  int i = 1;
2  while(i <= 10) {
3    printf("%d\n", i);
4  }
```

- Results in an infinite loop
- There is no increment/progress toward the termination condition
- To kill a command line program: control-C

Pitfall
Misplaced Semicolon

Consider the following code:

```
1  int i = 1;
2  while(i <= 10); {
3    printf("%d\n", i);
4    i++;
5  }
```

Pitfall
Misplaced Semicolon

Consider the following code:

```
1  int i = 1;
2  while(i <= 10); {
3    printf("%d\n", i);
4    i++;
5  }
```

- Results in an infinite loop

Pitfall
Misplaced Semicolon

Consider the following code:

```
1  int i = 1;
2  while(i <= 10); {
3    printf("%d\n", i);
4    i++;
5  }
```

- Results in an infinite loop
- Extraneous semicolon, similar to conditional pitfall

Pitfall
Misplaced Semicolon

Consider the following code:

```
1  int i = 1;
2  while(i <= 10); {
3    printf("%d\n", i);
4    i++;
5  }
```

- Results in an infinite loop
- Extraneous semicolon, similar to conditional pitfall
- Empty loop body

Consider the following code:

```
1  int i = 1;
2  while(i <= 10)
3    printf("%d\n", i);
4    i++;
```

Pitfall
Bad Coding Style

Consider the following code:

```
1  int i = 1;
2  while(i <= 10)
3    printf("%d\n", i);
4    i++;
```

- Results in an infinite loop

Pitfall
Bad Coding Style

Consider the following code:

```
1  int i = 1;
2  while(i <= 10)
3    printf("%d\n", i);
4    i++;
```

- Results in an infinite loop
- Missing brackets means loop body is *only the next line*

Consider the following code:

```
1  int i = 1;
2  while(i <= 10)
3    printf("%d\n", i);
4    i++;
```

- Results in an infinite loop
- Missing brackets means loop body is *only the next line*
- Always include curly brackets even if they are not necessary!

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully
- Loops may be off-by-one iteration (start or end)

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully
- Loops may be off-by-one iteration (start or end)
- Zune Bug: December 31st, 2008

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully
- Loops may be off-by-one iteration (start or end)
- Zune Bug: December 31st, 2008
- Thousands of Zunes froze for 24 hours

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully
- Loops may be off-by-one iteration (start or end)
- Zune Bug: December 31st, 2008
- Thousands of Zunes froze for 24 hours
- 2008 was a leap year: 366 days

Pitfall
Off-By-One Errors

- Initialization/continuation must be considered carefully
- Loops may be off-by-one iteration (start or end)
- Zune Bug: December 31st, 2008
- Thousands of Zunes froze for 24 hours
- 2008 was a leap year: 366 days
- An embedded module in the Zune contained the following (actual) code

Zune Bug

What happened?


```
1  while(days > 365) {
2    if(IsLeapYear(year)) {
3      if(days > 366) {
4        days -= 366;
5        year += 1;
6      }
7    } else {
8      days -= 365;
9      year += 1;
10   }
11 }
```

Part VI: Exercises

Exercise

Write code snippets to do the following.

1. Print a list of even integers 0 to n, one to a line
2. The same list, but delimited by commas
3. A list of integers divisible by 3 between 10 and 100 (print a total as well)
4. Prints all positive powers of two, $1, 2, 4, 8, \ldots, 2^{30}$
5. Prints all even integers 2 thru 200 on 10 different lines (10 numbers per line) in reverse order
6. Prints the following pattern of numbers (hint: use some value of $i + 10j$):

```
11, 21, 31, 41, ..., 91, 101
12, 22, 32, 42, ..., 92, 102
...
20, 30, 40, 50, ..., 100, 110
```

Exercise

Write a FizzBuzz program: print numbers from 1 to 100, but for numbers that are multiples of 3 print “Fizz” instead. For numbers that are multiples of 5, print “Buzz” instead. For numbers that are multiples of both 3 and 5, print “FizzBuzz”

Exercise

Implement a program to use the Babylonian method to compute a square root of a number a using the series,

$$x_{n+1} = \frac{1}{2} \cdot \left(x_n + \frac{a}{x_n} \right), \quad x_0 = 1$$

Exercise

Example: compute $\sqrt{2}$ $(a = 2)$

Exercise

Example: compute $\sqrt{2}$ $(a = 2)$

$$x_1 = \frac{1}{2} \cdot \left(x_0 + \frac{a}{x_0} \right)$$

Exercise

Example: compute $\sqrt{2}$ $(a = 2)$

$$x_1 = \frac{1}{2} \cdot \left(x_0 + \frac{a}{x_0} \right)$$

$$x_1 = \frac{1}{2} \cdot \left(1 + \frac{2}{1} \right) = 1.5$$

Exercise

Example: compute $\sqrt{2}$ $(a = 2)$

$$x_1 = \frac{1}{2} \cdot \left(x_0 + \frac{a}{x_0} \right)$$

$$x_1 = \frac{1}{2} \cdot \left(1 + \frac{2}{1} \right) = 1.5$$

$$x_2 = \frac{1}{2} \cdot \left(x_1 + \frac{a}{x_1} \right)$$

Exercise

Example: compute $\sqrt{2}$ $(a = 2)$

$$x_1 = \frac{1}{2} \cdot \left(x_0 + \frac{a}{x_0} \right)$$

$$x_1 = \frac{1}{2} \cdot \left(1 + \frac{2}{1} \right) = 1.5$$

$$x_2 = \frac{1}{2} \cdot \left(x_1 + \frac{a}{x_1} \right)$$

$$x_2 = \frac{1}{2} \cdot \left(1.5 + \frac{2}{1.5} \right) = 1.41666$$

Exercise

Would you accept a job with the following conditions?

- You only get paid a dollar a month.
- However, each month your pay doubles.
- Your contract lasts 2 years.

Write a program to project your earnings.

Exercise

DogeCoin is 'sploding! It increases 20% in value every week! Suppose we start with 1000 DogeCoin which is initially worth \$10 (1 DogeCoin = 1 cent). Let's take the following strategy: at the end of each week, we sell half our gains, and keep the rest growing. Then, at the end of the year we'll report how much cash we have, how much DogeCoin we have, and how much that is worth in total USD.