

Nebraska  
Lincoln

CSCE150A

Computer Science & Engineering 150A  
Problem Solving Using Computers

Lecture 05 - Loops

Stephen Scott  
(Adapted from Christopher M. Bourke)

Fall 2009

1 / 1

## Notes

---

---

---

---

---

---

---

Nebraska  
Lincoln

CSCE150A

Chapter 5

5.1 Repetition in Programs

5.2 Counting Loops and the While Statement

5.3 Computing a Sum or a Product in a Loop

5.4 The `for` Statement

5.5 Conditional Loops

5.6 Loop Design

5.7 Nested Loops

5.8 Do While Statement and Flag-Controlled Loops

5.10 How to Debug and Test

5.11 Common Programming Errors

2 / 1

## Notes

---

---

---

---

---

---

---

Nebraska  
Lincoln

CSCE150A

Repetition in Programs

Just as the ability to make decisions (*if-else* selection statements) is an important programming tool, so too is the ability to specify the repetition of a group of operations.

When solving a general problem, it is sometimes helpful to write a solution to a specific case. Once this is done, ask yourself:

- Were there any steps that I repeated? If so, which ones?
- Do I know how many times I will have to repeat the steps?
- If not, how did I know how long to keep repeating the steps?

3 / 1

## Notes

---

---

---

---

---

---

---

## Counting Loops

CSCE150A

A **counter-controlled loop** (or **counting loop**) is a loop whose repetition is managed by a loop control variable whose value represents a count. Also called a *while* loop.

```
1 Set counter to an initial value of 0
2 while counter < someFinalValue do
3     Block of program code
4     Increase counter by 1
5 end
```

Algorithm 1: Counter-Controlled Loop

4 / 1

## Notes

---

---

---

---

---

---

---

## The C While Loop

CSCE150A

This *while* loop computes and displays the gross pay for seven employees. The loop body is a compound statement (between brackets) The **loop repetition condition** controls the *while* loop.

```
1 int count_emp = 0;      // Set counter to 0
2 while (count_emp < 7)   // If count_emp < 7, do stmts
3 {
4     printf("Hours> ");
5     scanf("%d",&hours);
6     printf("Rate> ");
7     scanf("%lf",&rate);
8     pay = hours * rate;
9     printf("Pay is $%.2f\n", pay);
10    count_emp = count_emp + 1; /* Increment count_emp */
11 }
12 printf("\nAll employees processed\n");
```

5 / 1

## Notes

---

---

---

---

---

---

---

## While Loop Syntax

CSCE150A

Syntax of the *while* Statement:

- Initialize the loop control variable
  - Without initialization, the loop control variable value is meaningless.
- Test the loop control variable before the start of each loop repetition
- Update the loop control variable during the iteration
  - Ensures that the program progresses to the final goal

```
1 count = 1;
2 while(count <= 10)
3 {
4     printf("Count = %d\n",count);
5     count = count + 1;
6 }
```

6 / 1

## Notes

---

---

---

---

---

---

---

## Common Programming Errors

CSCE150A

- Skipping crucial steps could lead to an *infinite loop*
- Common error: forgetting to increment your loop control variable
- Syntax error: misplaced semicolons

```
1 count = 1;
2 while(count <= 10); ← WRONG
3 {
4     printf("Count = %d\n", count);
5     count = count + 1;
6 }
```

7 / 1

## Notes

---

---

---

---

---

---

---

## General While Loops

CSCE150A

Best to generalize code whenever possible

```
1 int numEmployees, count_emp=0;
2 printf("How many employees> ");
3 scanf("%d", &numEmployees);
4 while(count_emp < numEmployees)
5 {
6     . . .
7     count_emp = count_emp + 1;
8 }
```

Using `numEmployees` instead of the constant 7 allows our code to be more general.

8 / 1

## Notes

---

---

---

---

---

---

---

## While Loop Exercise

CSCE150A

## Exercise

Write a while loop to compute the sum of natural numbers 1 to 100:

$$\sum_{i=1}^{100} i = 1 + 2 + \dots + 100$$

Generalize the loop so that the sum from 1 to any  $n$  can be computed.

Steps to design:

- Identify and define a loop control variable.
- Write the syntax for the loop control structure
- Fill in the code used within the loop to compute the sum

9 / 1

## Notes

---

---

---

---

---

---

---

## While Loop Exercise

Answer

CSCE150A

```
1 int sum = 0;
2 int i = 1; /* our loop control variable */
3 while (i <= 100)
4 {
5     sum = sum + i;
6     i = i + 1;
7 }
8 printf("Sum is %d\n", sum);
```

10 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## While Loop Exercise

Answer: Generalized

CSCE150A

```
1 int sum = 0;
2 int n = 100; /* general variable, may be
3              * changed or read from input */
4 int i = 1; /* our loop control variable */
5 while (i <= n)
6 {
7     sum = sum + i;
8     i = i + 1;
9 }
10 printf("Sum 1 to %d is %d\n", n, sum);
```

11 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## While Loop Example II

CSCE150A

Instead of the sum of integers 1 to  $n$ , compute the product:

$$\prod_{i=1}^{100} i = 1 \times 2 \times \dots \times 100$$

What changes need to be made?

- Variable names?
- Initialized variable value?
- Operators?

Note: this is the *factorial* function,

$$n! = \prod_{i=1}^n i$$

12 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## While Loop Example II

Answer

CSCE150A

```
1 int product = 1;
2 int n = 100; /* general variable, may be
3             * changed or read from input */
4 int i = 1; /* our loop control variable */
5 while (i <= n)
6 {
7     product = product * i;
8     i = i + 1;
9 }
10 printf("Product 1 to %d is %d\n", n, product);
```

13 / 1

## Notes

---

---

---

---

---

---

---

## Program Failed

CSCE150A

Run the previous program: it gives an answer of 0—why?

- Debug your code: use a `printf` statement in the loop to see what intermediate values are computed:  
`printf("i = %3d product = %d\n", i, product);`
- Check the answers with a calculator
- For what  $i$  does this program fail?

14 / 1

## Notes

---

---

---

---

---

---

---

## Overflow

CSCE150A

- We got the wrong answer for  $i = 13$ ,  
$$13! = 6,227,020,800$$
- We used a 32-bit integer to store `product`
- Maximum representable value is  $2^{31} = 2,147,483,648$
- When a number is too large (or too small!!) to be represented by its type, *overflow* occurs (or *underflow*)
- More sophisticated solutions are available, but outside this course's scope

15 / 1

## Notes

---

---

---

---

---

---

---

## Compound Assignment Operators

CSCE150A

- Expressions such as `variable = variable op expression;` (where `op` is a C operator such as `+`, `-`, `*`, `/`, `%`) occur frequently
- C provides several syntax shortcuts
- `x = x + 1;` and `x += 1;` are "equivalent"
- Can do this with other operators (see table)

| Expression              | Shortcut             |
|-------------------------|----------------------|
| <code>x = x + 1;</code> | <code>x += 1;</code> |
| <code>x = x - 1;</code> | <code>x -= 1;</code> |
| <code>x = x * 5;</code> | <code>x *= 5;</code> |
| <code>x = x / 2;</code> | <code>x /= 2;</code> |

Table: Compound Assignment Operators

16 / 1

Navigation icons

### Notes

---

---

---

---

---

---

---

---

## Compound Assignment Operators

Example Revisited

CSCE150A

```
1 int product = 1;
2 int n = 100; /* general variable, may be
3              * changed or read from input */
4 int i = 1; /* our loop control variable */
5 while (i <= n)
6 {
7     product *= i;
8     i += 1;
9 }
10 printf("Product 1 to %d is %d\n", n, product);
```

17 / 1

Navigation icons

### Notes

---

---

---

---

---

---

---

---

## For Loops

CSCE150A

- Program Style
- Increment and Decrement Operators
- Increment and Decrement Other Than 1

18 / 1

Navigation icons

### Notes

---

---

---

---

---

---

---

---

## For Loops

CSCE150A

- Any repetition can be implemented using a while loop
- Another way to construct a counting loop is to use a *for loop*
- C provides **for** statements as another form for implementing loops.
- As before we need to initialize, test, and update the loop control variable.
- The syntax for a **for** statement is more rigid: it designates a specific place for the initialization, testing, and update components

19 / 1

## Notes

---

---

---

---

---

---

---

## For Loop Example

CSCE150A

Computing the sum using a for-loop:

```
1 int sum = 0;
2 int n = 100;
3 int i;
4 for(i = 0; i <= n; i++)
5 {
6     sum += i;
7 }
```

- Advantages: more readable, more predictable
- Easier to debug
- **Pitfall**: note the placement of semicolons!

20 / 1

## Notes

---

---

---

---

---

---

---

## Increment Operators

CSCE150A

- New syntax: **i++**
- Known as a (postfix) increment
- “Equivalent” to **i = i + 1**
- Also available: (postfix) decrement: **i--** (“equivalent” to **i = i - 1**)

21 / 1

## Notes

---

---

---

---

---

---

---

## Program Style

CSCE150A

For clarity, the book usually places each expression of the `for` heading on a separate line. If all three expressions are very short, however, they will be placed on one line.

The body of the `for` loop is indented just as the `if` statement.

22 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Increment and Decrement Operators

CSCE150A

The counting loops that we have seen have all included assignment expressions of the form

- `counter = counter + 1`
- `counter++`
- `counter += 1`

This will add 1 to the variable `counter`.

Using `--` will subtract one from the counter.

23 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Increment and Decrement Other Than 1

CSCE150A

We can use the “shortcut” compound assignment operators with values other than 1

- **Increment operations:** `sum = sum + x` or `sum += x`, will take the value of `sum`, add `x` to it, and then assign the new value to `sum`
- **Decrement operations:** `temp = temp - x` or `temp -= x`, will take the value of `temp`, subtract `x` from it and then assign the new value to `temp`

24 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---



## Increment and Decrement Other Than 1

Example

CSCE150A

```
1  /* increment by 10 */
2  int x = 10;
3  int i;
4  for(i=0; i<100; i+=x)
5      printf("i = %d\n", i);
6
7  /* decrement by 5 */
8  int y = 5;
9  for(i=25; i>=0; i-=y)
10     printf("i = %d\n", i);
```

25 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Conditional Loops

CSCE150A

- The exact number of loop repetitions we need to run for a loop will not always be known before loop execution begins.

Initialization step? Test? Update action?

26 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Exercise

CSCE150A

### Exercise

Create a program that prompts the user for a value  $x$  and multiplies it by the previous value of  $x$ , storing the result in  $x$ , until the user enters a 0.

27 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Exercise Pseudocode

CSCE150A

```

1 Set  $x$  to an initial value of 1
2 Prompt the user for a value input
3 while input is not zero do
4     Set  $x$  to  $x$  multiplied by input
5     Prompt the user for a new input value
6 end
    
```

Algorithm 2: Prompt Product Loop

28 / 1

Navigation icons

## Notes

---

---

---

---

---

---

---

## Exercise Translated to C

CSCE150A

```

1 int x = 1;
2 int value;
3 printf("Enter a value (0 to quit)> ");
4 scanf("%d", &value);
5 while(value != 0)
6 {
7     x = x * value;
8     printf("Enter a value (0 to quit)> ");
9     scanf("%d", &value);
10 }
11 printf("The product is %d", value);
    
```

29 / 1

Navigation icons

## Notes

---

---

---

---

---

---

---

## Loop Design

CSCE150A

To this point, we have been analyzing the actions a loop performs.  
Now, we also want to design our own loops:

- Sentinel-Controlled Loops
- Using a **for** Statement to Implement a Sentinel Loop

30 / 1

Navigation icons

## Notes

---

---

---

---

---

---

---

## Sentinel-Controlled Loops

CSCE150A

- Often we don't know how many data items the loop should process when it begins execution.
- **Sentinel-Controlled Loops** continue to read data until a unique data value is read, called the *sentinel value*.
- The sentinel value should be a value that could not normally occur as data.
- Reading the sentinel value signals the program to stop reading and processing new data and exit the loop.
- Example: Product of a list of numbers, with  $-1$  stopping the loop.

31 / 1

## Notes

---

---

---

---

---

---

---

## Sentinel-Controlled Loops

CSCE150A

```
1 Get a line of data
2 while Sentinel value is not encountered do
3     Process the data
4     Get another line of data
5 end
```

Algorithm 3: Product Loop using a Sentinel

32 / 1

## Notes

---

---

---

---

---

---

---

## Implementing a Sentinel Loop

CSCE150A

Because the `for` statement combines the initialization, test, and update in once place, some programmers prefer to use it to implement sentinel-controlled loops.

```
1 int sentinelValue = -1;
2 int score = 0
3 printf("Enter first score (%d to quit)> ", sentinelValue);
4 for(scanf("%d", &score);
5     score != sentinelValue;
6     scanf("%d", &score))
7 {
8     sum += score;
9     printf("Enter next score (%d to quit)> ", sentinelValue);
10 }
```

33 / 1

## Notes

---

---

---

---

---

---

---

## Implementing a Sentinel Loop

CSCE150A

- `scanf("%d",&score);` ← Initialization: read the first score
- `score != sentinelValue;` ← Terminating condition (test)
- `scanf("%d",&score)` ← Update: read another score

34 / 1

## Notes

---

---

---

---

---

---

---

## Nested Loops

CSCE150A

Like `if` statements, loops can also be nested.

- Nested loops consist of an outer loop with or more inner loops.
- Each time the outer loop is repeated, the inner loops are reentered.
- The inner loop control expressions are reevaluated, and all required iterations are performed.

35 / 1

## Notes

---

---

---

---

---

---

---

## Example

CSCE150A

```
1 int i, j;
2 for(i=1; i<=10; i++)
3 {
4     for(j=1; j<=10; j++)
5     {
6         if(j<i)
7             printf("+");
8         else
9             printf("*");
10    }
11    printf("\n");
12 }
```

36 / 1

## Notes

---

---

---

---

---

---

---

## Example - Output

CSCE150A

```
1 *****
2 +*****
3 +*****
4 +*****
5 +*****
6 +*****
7 +*****
8 +*****
9 +*****
10 +*****
```

37 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## The do-while Statement and Flag-Controlled Loops

CSCE150A

- `do-while` statement
- flag-controlled loops

38 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Do-While Statement

CSCE150A

- The `for` statement and the `while` statement evaluate conditions *before* the first execution of the loop body.
- In most cases, this pretest is desirable;
  - Prevents the loop from executing when there are no data items to process
  - Prevents execution when the initial value of the loop control variable is outside the expected range.
- Situations involving interactive input, when we know that a loop must execute *at least* one time, often use a `do-while` loop.

39 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Do While Example

CSCE150A

```
1 char letter_choice;
2 do
3 {
4     printf("Enter a letter from A through E> ");
5     scanf("%c", &letter_choice);
6 } while (letter_choice < 'A' || letter_choice > 'E');
```

40 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Do While

CSCE150A

- Loop begins with **do**
- Ends with **while**
- **Careful:** Conditional expression *does* end with a semicolon!
- Conditional is checked at the end of each loop (versus the beginning)

41 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Flag-Controlled Loops

CSCE150A

- Sometimes a loop repetition condition becomes so complex that placing the full expression in its usual spot is awkward.
- In many cases, the condition may be simplified by using a *flag*.
  - A **flag** is a type `int` variable used to represent whether or not a certain event has occurred.
  - A flag has one of two values: 1 (`true`) and 0 (`false`).

42 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Flag Example

CSCE150A

```
1 char letter_choice;
2 int isDone = 0;
3 while(!isDone)
4 {
5     printf("Enter a letter from A through E> ");
6     scanf("%c", &letter_choice);
7     isDone = (letter_choice >= 'A' && letter_choice <= 'E');
8 }
```

43 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## How to Debug and Test Programs

CSCE150A

- Debugging using `printf`
- Off-by-One Loop Errors
- Testing

44 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Debugging using `printf`

CSCE150A

- Use several `printf` statements to output the values of your variables to make sure they have the correct value in them as your program executes.
- It is often helpful to print out the value of your loop control variable to make sure you are incrementing it and will not enter an infinite loop.

45 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

---

## Off-by-One Loop Errors

CSCE150A

**Loop boundaries** - the initial and final values of the loop control variable.

- A fairly common logic error in programs with loops is a loop that executes one more time or one less time than required.
  - If a sentinel-controlled loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.
- If a loop performs a counting operation, make sure that the initial and final values of the loop control variable are correct and that the loop repetition condition is right.
  - The sum of 1...100, is not  
`for(i = 1; i < 100; i++) sum += i;`
  - Instead, `i <= 100` should be used.

46 / 1

## Notes

---

---

---

---

---

---

---

## Testing

CSCE150A

After all errors have been corrected and the program appears to execute as expected, the program should be tested thoroughly to make sure it works.

For a simple program, make enough test runs to verify that the program works properly for representative samples of all possible data combinations.

47 / 1

## Notes

---

---

---

---

---

---

---

## Common Programming Errors I

CSCE150A

- `if` and `while` statements can be confused, since they have similar appearance.
- Remember to initialize loop control variable as to prevent infinite loops.
- Infinite loops are bad: kill your program using control-C
- Remember to use brackets { ... } around the code of the loop statements.

48 / 1

## Notes

---

---

---

---

---

---

---



## Common Programming Errors II

CSCE150A

- Be careful about the loop conditions; if we only want positive results, then testing `while (result != 0)` would not work since the result might become negative without ever being 0.
- A `do-while` loop always executes once and *then* tests the condition.

49 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Real World Example

CSCE150A

- Zune Bug: December 31st, 2008
- 2008 was a leap year: 366 days
- Thousands of Zunes froze for 24 hours
- An embedded module in the Zune contained the following (actual) code

50 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Real World Example

What happened?

CSCE150A

```

1  while (days > 365)
2  {
3      if (IsLeapYear(year))
4      {
5          if (days > 366)
6          {
7              days -= 366;
8              year += 1;
9          }
10     }
11     else
12     {
13         days -= 365;
14         year += 1;
15     }
16 }
```

51 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## What happened?

CSCE150A

- If `days` (number of days since January 1, 1980) is 366 and `year = 1980`, then `IsLeapYear(year)` is true, but the nested `if` is not executed, so `days` is not ever changed and an infinite loop occurs
- This will happen on December 31 of any leap year
- <http://bit-player.org/2009/the-zune-bug>

52 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Other Examples

CSCE150A

- September 30, 1999: \$125 million Mars orbiter crashes due to lack of converting pounds of thrust to Newtons
- September 26, 1983: Stanislav Petrov averts nuclear war
  - Russian missile early detection system alerts Petrov of incoming US missiles; he correctly assumes that it's a bug since it's a new system and the size of the supposed attack is small
- Wired Article: History's Worst Software Bugs (<http://www.wired.com/software/coolapps/news/2005/11/69355>)

53 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---

## Questions

CSCE150A

Questions?

54 / 1

◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶ ◀ ▶

### Notes

---

---

---

---

---

---

---