Looping (I)

Outline

- Motivation

1. `while` Statement (Syntax)
2. How to Use Loops
3. Using Repetition to Solve Problems
4. Tips on Planning to Write a Loop
5. A More Complicated Example
6. Infinite Loop

Motivation

- **`if-else` (branching)** statements allows some statements to be executed zero or one times.

- **Loop statements** allows some tasks to be executed repeatedly zero or more times.

- In our previous lectures, we have seen many cases in which we need to repeat certain actions again and again.

Example #1 (Motivating Example)

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   scanf("%d", &list[0]);
5   scanf("%d", &list[1]);
6   scanf("%d", &list[2]);
7   scanf("%d", &list[3]);
8
9   // Print the input values in reverse order
10  printf("You have entered (in reverse): ");
11  printf("%d %d %d %d\n", list[3], list[2], list[1], list[0]);
```

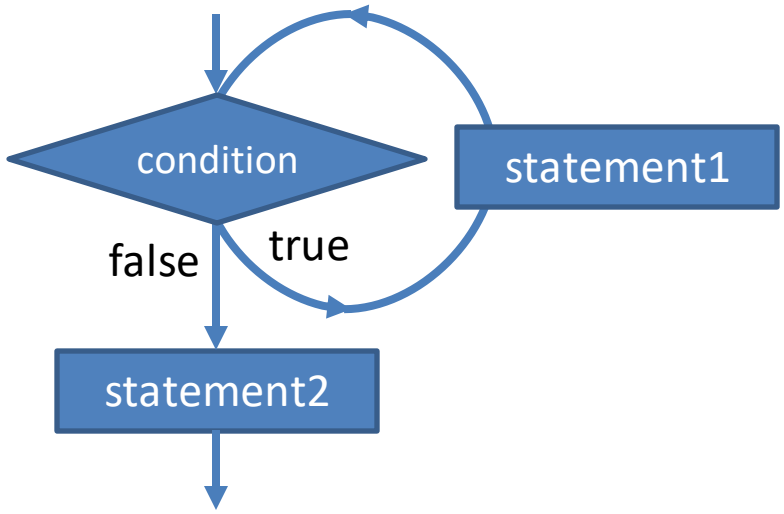
Only the array index is different in each line

```
Enter 4 #'s: 5 12 6 1110↵
You have entered (in reverse): 1110 6 12 5
```

What if we want to enter 40 numbers?! Isn't that very silly to copy and paste a very similar statement 40 times?

1. `while` Statement (*Syntax*)

```
while (condition)
    statement1;
statement2;
```

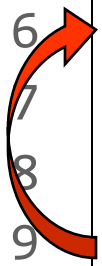


- Repeatedly execute **statement1** as long as **condition** is ***true***.

- When **condition** is ***false***, execute **statement2**.

1.1. while Statement (Example #1)

```
1   int i;
2
3   i = 1;
4
5   // A simple loop that iterates 5 times
6   while (i <= 5) {
7       printf("%d\n", i);
8       i++; // i=i+1;
9   }
10
11  printf("Lastly, i = %d\n", i);
12
```



```
1
2
3
4
5
Lastly, i = 6
```

1.2. Key Components of a Loop

```
1   int i;
2
3   i = 1;
4
5   // A simple loop that iterates 5 times
6   while (i <= 5) {
7       printf("%d\n", i);
8       i++;
9   }
10
11                                  ;
12
```

4. "Loop variable" initialization

Assign a value to the variable that is used in the loop condition to make the loop condition ***true*** initially.

1. Loop condition

- When this condition is true, the loop body is executed.
- Usually controlled by a variable.

2. Loop body

Statements to be repeated.

3. Change of loop condition

To stop the loop, we need to make the loop condition ***false***. This can usually be done by changing the loop variable.
You should never omit this!

Rewriting Example #1

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   scanf("%d", &list[0]);
5   scanf("%d", &list[1]);
6   scanf("%d", &list[2]);
7   scanf("%d", &list[3]);
8
9   // Print the input values in reverse order
10  printf("You have entered (in reverse): ");
11  printf("%d %d %d %d\n", list[3], list[2], list[1], list[0]);
```

We can use a `while` loop to help us!

We want to repeat the statement for multiple times, starting with index **0**, then **1**, then **2**, then **3**

Rewriting Example #1

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   int i = 0;
5
6
7
8
9
10
11
```

Let's rewrite our program.
We first create a variable **`i`** to store the index of the array **`list`**.
We said we would start from 0, so we set **`i = 0`**.

This is our *loop variable initialization*.

Rewriting Example #1

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   int i = 0;
5   while (i <= 3) {
6
7
8   }
9
10
11
```

Let's create the loop and the *loop condition*.

We said we want to end when **`i == 3`**,
so we should loop as long as **`i`** is less than or equal to **`3`**.

Notice how we have used the braces { } to mark the loop body first, which is currently empty.

Rewriting Example #1

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   int i = 0;
5   while (i <= 3) {
6       scanf("%d",&list[i]);
7       i++;
8   }
9
10  printf("You have entered (in
11  printf("%d %d %d %d\n", list
```

Now we will fill up the *loop body*!

We get an integer from user, and then put it into the **i**th slot in the array.

Remember to increase **i** by **1** every time you loop! *The change of looping condition is a must!*

In this loop, we hope to generate i==**0**, i==**1**, i==**2**, i==**3** in this order.

After the loop ends, C will continue to execute the statements ahead.

Rewriting Example #1

```
1   int list[4];
2
3   printf("Enter 4 #'s: ");
4   int i = 0;
5   while (i <= 3) {
6       scanf("%d",&list[i]);
7       i++;
8   }
9
10  printf("You have entered (i
11  printf("%d %d %d %d\n", lis
```

If you execute the program, this is the order of execution:

First time executing the loop body:

```
scanf("%d",&list[i]); // i == 0
i++;                  // i == 1
```

Second time:

```
scanf("%d",&list[i]); // i == 1
i++;                  // i == 2
```

Third time:

```
scanf("%d",&list[i]); // i == 2
i++;                  // i == 3
```

Forth time:

```
scanf("%d",&list[i]); // i == 3
i++;                  // i == 4
```

When returning to line 5, the loop will stop because **i** is NOT **<= 3**

➔ loop condition became false

Loop ended:

```
printf("Loop ended!\n");
```

Now the four **scanf()** lines work the same as in the original Example #1, and you can expect the same behaviour as before.

2. How to Use Loops

- Loops can help us in many different daily computing tasks.
- As a beginner, you will need to at least know two basic looping cases:
  - Basic Case #1: Repeat a task a finite number of times (as in Example #1)
  - Basic Case #2: Repeat a task indefinitely until a condition is met

2.1. Basic Case #1: Finite Repetition

Extending from Example #1, we can also use another loop to print out the contents of the array; we will do it 4 times, which is finite.

```
1   int list[4];
2   printf("Enter 4 #'s: ");
3   int i = 0;
4   while (i <= 3) {
5       scanf("%d",&list[i]);
6       i++;
7   }
8   printf("You have entered (in rev
9   int j = 3;      // let's print in reverse
10  while (j >= 0) {
11      printf("%d ",list[j]);
12      j--;
13  }
14  printf("\n");
```

Are you able to identify the:

- loop variable initialization,
- looping condition,
- loop condition change, and
- loop body?

If you write down the repeated **`printf()`**, you will see why you can again expect the same behaviour as the original Example #1

2.1. Basic Case #1: Finite Repetition

i, **j**, **k** are commonly used as loop variables, and are often referred to as ***counters*** in finite loops.

```
1  int list[4];
2  printf("Enter 4 #'s: ");
3  int i = 0;
4  while (i <= 3) {
5      scanf("%d",&list[i]);
6      i++;
7  }
8  printf("You have entered (in reverse):
9  int j = 3;     // let's print in revers
10 while (j >= 0) {
11     printf("%d ",list[j]);
12     j--;
13 }
14 printf("\n");
```

Loop #1

Loop #2

Let's have an exercise: Can you dry run and explain to yourself how the loop executes?

You can see there are two loops here. ***Why do we need two loops?***

Can we use only one to achieve the same effect?

2.2. Basic Case #2: Indefinite Repetition

In this example, we will keep asking the user for a number until the input is –1; we do not have a fixed or finite number of repeats in mind.

Our loop condition variable is *NOT* a counter this time: the loop condition variable is the user input.
We conveniently set it to 0 first to make sure that the loop will start.

```
1  int input = 0;
2
3  while (input != -1) {
4      scanf("%d",&input);
5      printf("You have input %d\n",input);
6  }
7
8  printf("Last input is %d, the loop has ended\n", input);
```

How many times will the loop execute? Do you really know?

```
5↵
You have input 5
-1↵
You have input -1
Last input is -1, the loop has ended
```

2.2. Basic Case #2: Indefinite Repetition

- You cannot rewrite the above example without using loops.
  - The user can theoretically go on and input numbers forever, indefinitely (!)
  - This is very common in real life computing applications: users can continue something indefinitely until they want to stop.
- Looping enables us to write programs with a flexible repeating behavior.

3. Using Repetition to Solve Problems

- As long as a solution requires multiple steps that are the same or very similar to each other, we can use looping to help us.

- Let's look at two slightly more complex situations that will be challenging to new learners of looping.

3.1. Using Repetition for Statistics Generation

- When you need to generate statistics and insights from a set of data, you often need to go through each piece of data one by one.
  - E.g., sum, average, maximum, minimum, etc.

- Looping will be our best tool to achieve this, although it does not appear to be a repetitive task from the first sight.

3.1. Using Repetition for Statistics Generation

Here we want to ask for several inputs from the user and print the average value out:

```
1   printf("Enter 4 #'s: ");
2   int i = 0;
3   int input;
4
5   while (i <= 3) {
6       scanf("%d",&input);
7       ...
8       i++;
9   }
10  ...
11  printf("Average of 4 numbers is %.2f\n", average);
12
13
14
```

Let's first start with writing a C program to ask for several inputs from the user.

Here we see the proper loop variable initialization, looping condition, and loop variable change

3.1. Using Repetition for Statistics Generation

```
1   printf("Enter 4 #'s: ");
2   int i = 0;
3   int input;
4   double average, sum = 0.0;
5   while (i <= 3) {
6       scanf("%d",&input);
7       sum = sum + input;
8       i++;
9   }
10  average = sum/4;
11  printf("Average of 4 numbers is %.2f\n", average);
12
```

To calculate the average, we need to add up the input to a sum.
We will want to
1) initialize **sum** to be **0**,
2) add up each input to the **sum**, and finally
3) calculate the **average** from the **sum** after all inputs are considered.

The added statements are in red.

```
Enter 4 #'s: 7 11 45 23↵
Average of 4 number is 21.50
```

Question:
input is **int**, why do we use **double** for sum?

3.1. Using Repetition for Statistics Generation

- To generate statistics, we usually need to set up another variable (NOT the looping variable) to aggregate the data. In the previous example, it is **`sum`**.
  - Such variables should be ***initialized*** BEFORE the loop.
  - Such variables must be ***updated*** INSIDE the loop.
  - Such variables are usually only useful AFTER the loop ends.

- If you put them into the wrong place, the program won't work!

3.1. Using Repetition for Statistics Generation

What will be the result of this modified program? *Hint: It will generally give an unexpected answer!*

```
1   printf("Enter 4 #'s: ");
2   int i = 0;
3   int input;
4   double average, sum = 0.0;
5   while (i <= 3) {
6       scanf("%d",&input);
7       i++;
8   }
9   sum = sum + input;
10  average = sum/4;
11  printf("Average of 4 numbers is %.2f\n", average);
12
13
14
```

3.1. Using Repetition for Statistics Generation

How about this one? *Hint: this is rather a performance issue.*

```
1   printf("Enter 4 #'s: ");
2   int i = 0;
3   int input;
4   double average, sum = 0.0;
5   while (i <= 3) {
6       scanf("%d",&input);
7       sum = sum + input;
8       average = sum/4;  // how many times does this execute?
9       i++;
10  }
11  printf("Average of 4 numbers is %.2f\n", average);
12
13
14
```

3.1. Using Repetition for Statistics Generation

How about this? *Hint: It will generally give a wrong answer as well. Do you see why?*

```
1   printf("Enter 4 #'s: ");
2   int i = 0;
3   int input;
4   double average, sum;
5   while (i <= 3) {
6       sum = 0;
7       scanf("%d",&input);
8       sum = sum + input;
9       i++;
10  }
11  average = sum/4;
12  printf("Average of 4 numbers is %.2f\n", average);
13
14
```

3.2. Using Loop Variables in Computation

- In some cases, we use the loop variable directly in our computations
- In such cases, we do not just care about how many times we are repeating, we also care about how the loop variable change from the initial value to the final, terminating value

- We have already seen it in action in Example #1

3.2. Using Loop Variables in Computation

```
1  int list[4];
2
3  printf("Enter 4 #'s: ");
4  int i = 0;
5  while (i <= 3) {
6      scanf("%d",&list[i]);
7      i++;
8  }
9  int j = 3;     // let's print in rev
10 while (j >= 0) {
11     printf("%d ",list[j]);
12     j--; // j=j-1;
13 }
14 printf("\n");
```

0
➔1
➔2
➔3

3
➔2
➔1
➔0

We have two loops here.
In both loops, we repeat exactly **4** times.

However, in loop #1, the loop variable changes in this way:
0 ➔ 1 ➔ 2 ➔ 3.

In loop #2, we use the loop variable as an index to access the array, and we wish to access in reverse. The loop variable changes in this way:
3 ➔ 2 ➔ 1 ➔ 0

3.2. Using Loop Variables in Computation

```
1  int list[4];
2  printf("Enter 4 #'s: ");
3  int i = 0;
4  while (i <= 3) {
5      scanf("%d",&list[i]);
6      i++;
7  }
8  printf("You have entered (in revers
9  int j = 0;
10 while (j <= 3) {
11     printf("%d ",list[j]);
12     j++;
13 }
14 printf("\n");
```

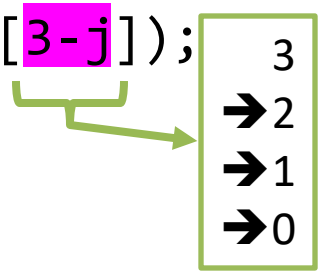
0
➔1
➔2
➔3

If we alter our loop variable in loop #2 to change in this way:
0 ➔ 1 ➔ 2 ➔ 3,
then <u>we are still repeating **4** times</u>, BUT we will NOT be printing the list in a reverse order.

<u>How the loop variable changes</u> IS IMPORTANT in here because we use the loop variable directly in our calculation (array index).

3.2. Using Loop Variables in Computation

```
1  int list[4];
2  printf("Enter 4 #'s: ");
3  int i = 0;
4  while (i <= 3) {
5      scanf("%d",&list[i]);
6      i++;
7  }
8  printf("You have entered (in revers
9  int j = 0;
10 while (j <= 3) {
11     printf("%d ",list[3-j]);
12     j++;
13 }
14 printf("\n");
```



Note that we can print in reverse order again, if we use `3-j` instead of `j` to access the array.

Do you understand why? Can you dry-run the loop and see?

Some people prefer the complexity to be with the loop variable preparation, yet some prefer it with the computation later.

3.2. Using Loop Variables in Computation

Another example of using loop variable directly in computation, in which we add from 1 to N, where N is a positive integer from user.

```
1   int N;
2
3   scanf("%d", &N);
4
5   int i = 1;    // our loop variable
6   int sum = 0;
7   while (i <= N) {
8       sum = sum + i;
9       i++;
10  }
11
12  printf("1 + .. + %d = %d\n",N,sum);
13
14
```

```
5↵
1 + .. + 5 = 15
```

4. Tips on Planning to Write a Loop

- Before you write a loop, please make sure you figure out:
  - Before the Loop
    What should be done before the loop?
    - You almost always need to initialize the looping variable(s)
    - You may need to initialize the variables that persists through your repetition
  - Inside the Loop
    What should be done repeatedly?
    And how should the loop variable change?
  - After the Loop
    What should be done after all repeats are finished?

4.1. Tips: Debugging a Loop

- Your loop will be wrong if you are confused with what you should do Before/Inside/After the loop

- When you write a loop or debug a loop, you have at least these TWO things to check:
  - ***Structurally***
    - It should have loop variable initialization, looping condition and loop condition update
  - ***Logically***
    - You should make sure statements before/inside/after the loop are in the right place

5. A More Complicated Example

- Given the following task:
  - **Step 1**. Ask the user for a number.
  - **Step 2**. If the input is *not zero*, add the new input to the sum all previous inputs. Then, go back to **Step 1**.
  - **Step 3**. If the input is zero, print the sum of all inputs and terminate the program.

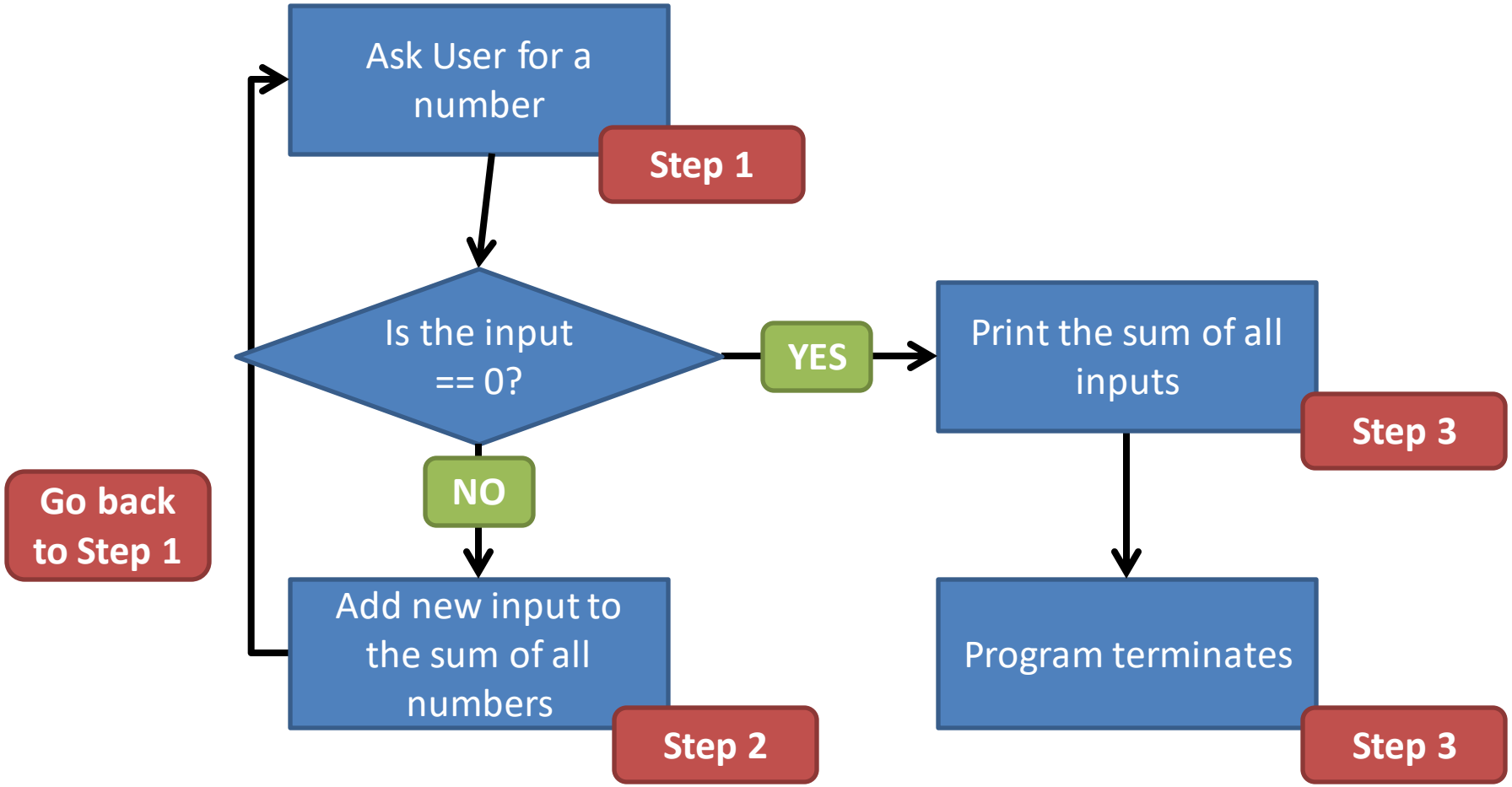
- How can we transform it into a while-loop program?
- First: What basic loop case is involved?
  - Basic Case #2 – Indefinite repetition
  - It also involves statistics generation

5. A More Complicated Example

- Let's plan for the looping
- **Before the Loop**:
  What do we need to do ONCE before entering loop?
  - We need to set a *loop condition variable* that indicates whether the user has input zero
  - We also need to *initialize a variable* to store the sum
- **Inside the Loop**:
  What needs to be repeated?
  - We will ask for input from user
  - We need to check if it is zero
  - We need to *add input to sum*
- **After the Loop**:
  What needs to be done AFTER everything is finished?
  - Print out the sum, of course!

5. A More Complicated Example

- Transforming Steps 1 – 3 to a flow chart:



5. A More Complicated Example

```
1   int input, sum = 0; // To store input value and their sum
2   int getZero = 0;    // For controlling the loop;
3                       // 1 => stop loop; 0 => continue iterate
4
5   while (getZero == 0) {
6       printf("Input: ");
7       scanf("%d", &input);
8
9       if (input == 0)
10          getZero = 1;
11      else
12          sum = sum + input;
13  }
14
15  printf("Sum = %d\n", sum);
```

```
Input: 1↵
Input: 3↵
Input: 5↵
Input: 7↵
Input: 0↵
Sum = 16
```

5. A More Complicated Example

```
1  int input, sum = 0; // To store input value and their sum
2  int getZero = 0;    // For controlling th
3                      // 1 => stop loop; 0
4
5  while (getZero == 0) {
6      printf("Input: ");
7      scanf("%d", &input);
8
9      if (input == 0)
10         getZero = 1;
11     else
12         sum = sum + input;
13 }
14
15 printf("Sum = %d\n", sum);
```

Loop variable initialization

Loop condition

Loop condition update (conditionally)

Again, all components of a loop are included.

5. A More Complicated Example

```
1   int input, sum = 0; // To store input value and their sum
2   int getZero = 0;    // For controlling th
3                       // 1 => stop loop; 0
4
5   while (getZero == 0) {
6       printf("Input: ");
7       scanf("%d", &input);
8
9       if (input == 0)
10          getZero = 1;
11      else
12          sum = sum + input;
13  }
14
15  printf("Sum = %d\n", sum);
```

Before the loop

Inside the loop

And the C program matches our previous planning!

After the loop

6. Infinite Loop

- A loop that never stops. e.g.,

```
while (1)
    printf("Hello!\n");
```

- Usually introduced by mistakes.

- What could happen when a program runs into an infinite loop?

6.1. Common Mistakes that Result in Infinite Loops

- A condition that is always true:

```
while (a > -10 || a < 10)  {
…
}
```

- Failing to update/modify the value of the loop variable in the loop:

```
i = 0;
while (i <= 5)  {
   printf("i = %d\n", i);
}
```

  - In this example, `i` is always `0`.

6.1. Common Mistakes that Result in Infinite Loops

- Using **=** instead of **==** as equality operator

```
while (a = 1) {
  ...
}
```

  - Variable **a** is assigned 1 and the whole expression is evaluated to 1, and 1 means ***true***.

- Placing '**;**' after the condition of a **while** loop

```
while (a != 0);
{
   …
}
```

; here represents an *empty statement*. That is

```
while (a != 0);
```

is interpreted the same as

```
while (a != 0) {
}
```

Summary

- Syntax of `while` loops
- Using repetition to solve problems
- How to plan for a loop
- Common mistakes and infinite loops

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 3 Structured Program Development in C
  - Sections 3.7 – 3.9
- Chapter 4 C Program Control
  - Sections 4.1 – 4.3