while Loop Outline

1. while Loop Outline
2. Repetition/
   Looping
3. The while Loop
4. while Loop Flowchart
5. while Loop Example
6. while Loop Example’s Flowchart
7. How Many Times Will the Body of a while Loop Be Executed?
8. An Infinite Loop
9. What Kind of Statements Can Go Inside a while Loop?
10. Jargon: Compound Statement
11. Another while Loop Example
12. Another while Loop Example (continued)
13. Yet Another while Loop Example
14. States and Traces
15. Tracing the Loop

Repetition

Repetition means performing the same set of statements over and over. The most common way to perform repetition is via looping.

Looping

A loop is a sequence of statements to be executed, in order, over and over, until some condition is reached.

C has a loop construct known as the while loop:

```c
while ( condition ) {
    statement1;
    statement2;
    ... 
}
```

Here, the condition is a Boolean expression completely enclosed in parentheses, just like in an if block.

The sequence of statements between the while statement’s block open and block close is known as the loop body.
The while Loop

while ( condition ) {
    statement1;
    statement2;
    ...
}

When a while statement is encountered, the following sequence of events occurs, in order:

1. The condition (a Boolean expression completely enclosed in parentheses) is evaluated, resulting in a value of either true (1) or false (0).
2. If the condition evaluates to false (0), then the statements inside the loop body are skipped, and control is passed to the statement immediately after the while loop’s block close.
3. If the condition evaluates to true (1), then the statements inside the loop body are executed in sequence.
4. When the while loop’s block close is encountered, the program jumps back up to the associated while statement and starts over with Step 1.

A while loop is SIMILAR to an if block, EXCEPT that:

- UNLIKE an if block, when a while loop gets to its block close, it jumps back up to the associated while statement;
- UNLIKE an if block, a while loop can have only one clause, which is analogous to the if clause; a while loop cannot have anything analogous to an else if clause or to an else clause.
while Loop Example

```c
#include <stdio.h>

int main ()
{
    /* main */
    const int no_chickens = 0;
    int number_of_chickens;

    printf("How many chickens did you think of?\n");
    scanf("%d", &number_of_chickens);
    while (number_of_chickens < no_chickens) {
        printf("ERROR: you can't think of negative chickens!\n");
        printf("So really, how many chickens did you think of?\n");
        scanf("%d", &number_of_chickens);
    }
    printf("The number of chickens is valid.\n");
    return 0;
}
/* main */
```

% cat chickenidiot_while.c
#include <stdio.h>

int main ()
{
    /* main */
    const int no_chickens = 0;
    int number_of_chickens;

    printf("How many chickens did you think of?\n");
    scanf("%d", &number_of_chickens);
    while (number_of_chickens < no_chickens) {
        printf("ERROR: you can't think of negative chickens!\n");
        printf("So really, how many chickens did you think of?\n");
        scanf("%d", &number_of_chickens);
    }
    printf("The number of chickens is valid.\n");
    return 0;
}
/* main */
```
How Many Times Will the Body of a while Loop Be Executed?

Recall:

```c
while (condition) {
    statement1
    statement2
    ...
}
```

1. The condition (a Boolean expression completely enclosed in parentheses) is evaluated, resulting in a value of either true (1) or false (0).
2. If the condition evaluates to false (0), then the statements inside the loop body are skipped, and control is passed to the statement immediately after the block close of the while loop.
3. If the condition evaluates to true (1), then the statements inside the loop body are executed in sequence.
4. When the block close of the while statement is encountered, the program jumps back up to the associated while statement and starts over with Step 1.

Suppose that, the first time that the while statement is encountered, its condition evaluates to false (0).

In that case, the loop body will be skipped completely.

So, it may be the case that the loop body is not executed at all, or it may be executed once, or twice, or many times.

---

An Infinite Loop

An infinite loop is a loop that never reaches its termination condition.

For example:

```c
#include <stdio.h>
int main ()
{ /* main */
    const int my_number = 5;
    int your_number;
    printf("Enter an integer:\n");
    scanf("%d", &your_number);
    printf("I had %d.\n", my_number);
    while (your_number < my_number) {
        printf("Your number is less than mine!\n");
    } /* while (your_number < my_number) */
    return 0;
} /* main */
```

% cat infiniteloop.c
% include <stdio.h>
int main ()
{ /* main */
    const int my_number = 5;
    int your_number;
    printf("Enter an integer:\n");
    scanf("%d", &your_number);
    printf("I had %d.\n", my_number);
    while (your_number < my_number) {
        printf("Your number is less than mine!\n");
    } /* while (your_number < my_number) */
    return 0;
} /* main */

% gcc -o infiniteloop infiniteloop.c
% infiniteloop
Enter an integer:
4
I had 5.
% infiniteloop
Enter an integer:
5
I had 5.
% infiniteloop
Enter an integer:
4
I had 5.
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
Your number is less than mine!
...

On most Unix systems, including roosevelt, lincoln and kennedy, you can quit out of a program that is currently executing by typing 

Control-C
What Kind of Statements Can Go Inside a \texttt{while} Loop?

Between the \texttt{while} statement’s block open and block close, there can be \texttt{any} kind of \texttt{executable} statements, and any number of them. For example:

- \texttt{printf} statements;
- \texttt{scanf} statements;
- assignment statements;
- \texttt{exit} statements;
- if blocks;
- while loops.

There are other kinds of executable statements that can occur inside a \texttt{while} loop, some of which we’ll learn later in the semester.

In the event that the \texttt{while} condition evaluates to true (1), these statements will be executed one by one, in the order in which they appear in the \texttt{while} loop.

Notice that a \texttt{while} loop \texttt{CANNOT} contain declaration statements, because the \texttt{while} statement is an executable statement, and \texttt{ALL} declarations \texttt{MUST} come before \texttt{ANY} executable statements.

Jargon: Compound Statement

A \textit{compound statement} is a sequence of statements with a well-defined beginning and a well-defined end that are executed, in order, under certain circumstances.

Examples of compound statements include \texttt{if} blocks and \texttt{while} loops.

Although an \texttt{if} block is actually a sequence of statements, we can think of it as a single “super” statement in some contexts. The same is true of a \texttt{while} loop.

Compound statements are sometimes called \texttt{blocks}. Thus, we speak of an \texttt{if} block. A \texttt{while} loop is also a kind of block.
Another while Loop Example

```c
#include <stdio.h>
int main ()
{ /* main */
    const int minimum_number = 1;
    const int maximum_number = 100;
    const int my_number = 32;
    const int very_close_distance = 1;
    const int negative_distance = -1;
    int your_number, your_distance;
    int your_last_distance = negative_distance;
    char correct_number_hasnt_been_input = 1;
    printf("I'm thinking of a number between %d and %d.
", minimum_number, maximum_number);
    while (correct_number_hasnt_been_input)
    { /* while (correct_number_hasnt_been_input) */
        printf("What number am I thinking of?
");
        scanf("%d", &your_number);
        if ((your_number < minimum_number) || (your_number > maximum_number))
            printf("Hey! That's not between %d and %d!
", minimum_number, maximum_number);
        else if (your_number == my_number)
            printf("That's amazing!
");
        else
        { /* else */
            your_distance = abs(your_number - my_number);
            if (your_distance == very_close_distance)
                printf("You're incredibly hot!
");
            else if (your_distance < your_last_distance)
                printf("You're getting warmer ....
");
            else if (your_distance > your_last_distance)
                printf("Ouch! You're getting colder.
");
            else
                printf("Uh oh. You made no progress.
");
        }
        printf("Good for you!
");
    } /* while (correct_number_hasnt_been_input) */
    return 0;
} /* main */
```

Another while Loop Example (continued)

```bash
% gcc -o warmercolder warmercolder.c
% warmercolder
I'm thinking of a number between 1 and 100. What number am I thinking of?
0
Hey! That's not between 1 and 100! I'll pretend you didn't say that. What number am I thinking of?
101
Hey! That's not between 1 and 100! I'll pretend you didn't say that. What number am I thinking of?
50
Not bad for your first try. What number am I thinking of?
40
You're getting warmer .... What number am I thinking of?
60
Ouch! You're getting colder. What number am I thinking of?
30
You're getting warmer .... What number am I thinking of?
35
Ouch! You're getting colder. What number am I thinking of?
33
You're incredibly hot! What number am I thinking of?
31
You're incredibly hot! What number am I thinking of?
32
That's amazing! Good for you!
```

11

12
Yet Another while Loop Example

```c
#include <stdio.h>
int main ()
{
    int initial_value, final_value;
    int count;
    int sum;

    printf("What value would you like to start counting at?\n");
    scanf("%d", &initial_value);
    printf("What value would you like to stop counting at, which must be greater than or equal to %d?\n", initial_value);
    scanf("%d", &final_value);
    if (final_value < initial_value)
    {
        printf("ERROR: the final value %d is less than the initial value %d.\n", final_value, initial_value);
        exit(-1);
    }
    sum = 0;
    count = initial_value;
    while (count <= final_value)
    {
        sum = sum + count;
        count = count + 1;
    }
    printf("The sum of the integers from %d through %d is %d.\n", initial_value, final_value, sum);
    return 0;
}
```

States and Traces

The *state* of a program is the set of values of all of its variables at a given moment during execution; that is, it's a snapshot of the memory that's being used.

A *trace* of a program is a listing of the state of the program after each statement is executed.

Suppose that, in the previous example program, the user input 1 for `initial_value` and 5 for `final_value`.

Let's examine the program fragment around the loop.

```c
sum = 0;
count = initial_value;
while (count <= final_value) {
    sum = sum + count;
    count = count + 1;
}
```

If we number these statements, we have:

1. `sum = 0;`
2. `count = initial_value;`
3. `while (count <= final_value) {
    sum = sum + count;
    count = count + 1;
} /* while (count <= final_value)`

If we number the trace:

1. `sum = 0;`
2. `count = initial_value;`  
3. `while (count <= final_value) {`
4. `sum = sum + count;`
5. `count = count + 1;`
6. `}` /* while (count <= final_value)`

Tracing helps us to examine the behavior of a piece of code, and so it sometimes can be useful in debugging.
Tracing the Loop

```java
1    sum = 0;
2    count = initial_value;
3 while (count <= final_value) {
4        sum = sum + count;
5        count = count + 1;
6    } // while (count <= final_value) */
```

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Stmt #</th>
<th>Value of sum</th>
<th>Value of count</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>1</td>
<td>garbage</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>Condition evaluates to true (1)</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>new sum = old sum + count = 0 + 1 = 1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>new count = old count + 1 = 1 + 1 = 2</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Condition evaluates to true (1)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>new sum = old sum + count = 1 + 2 = 3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>new count = old count + 1 = 2 + 1 = 3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Condition evaluates to true (1)</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>new sum = old sum + count = 3 + 3 = 6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>new count = old count + 1 = 3 + 1 = 4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>Condition evaluates to true (1)</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10</td>
<td>4</td>
<td>new sum = old sum + count = 6 + 4 = 10</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>new count = old count + 1 = 4 + 1 = 5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>10</td>
<td>5</td>
<td>Condition evaluates to true (1)</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>15</td>
<td>5</td>
<td>new sum = old sum + count = 10 + 5 = 15</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
<td>6</td>
<td>new count = old count + 1 = 5 + 1 = 6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>15</td>
<td>6</td>
<td>jump back up to stmt #3 to start iteration #2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>15</td>
<td>6</td>
<td>Condition evaluates to false (0), loop exited</td>
</tr>
</tbody>
</table>