

Introduction to Programming (CS 101)

Spring 2024



Lecture 4:

while loops, break/continue, do while, scope

Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade

Recap-I (if and logical operators)

What is the output from the following piece of code?

 1 1 -1

 1 2 -1

 2 1 -1

 2 2 -1

```
#include <simplecpp>
main_program{
```

```
    int i = 1, j = 1, k = -1;
```

```
    if(!i || j && k)
```

```
        i += 1;
```

```
    else
```

```
        j += 1;
```

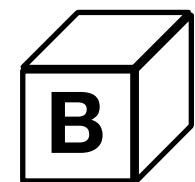
```
    cout << i << " " << j << " " << k;
```

```
}
```

Recap-II (switch statement)

What is the output from the following piece of code?

 1 1 0

 0 2 0

 1 2 1

 1 2 0

```
#include <simplecpp>
main_program{

    int i = 0, j = 1, k = -1;

    switch(!i * j - k) {
        case 2:
        case 1:
            j += 1; k += 1;
        case 0:
            i += 1;
            break;
        default:
            k += 1;
    }

    cout << i << " " << j << " " << k;
}
```

Recap-III (ternary operator)

What is the output from the following piece of code?

☐ A 0

☒ B 1

☐ C -1

```
#include <simplecpp>
main_program{

    int i = 0, j = 0;

    cout << (i > j ? i-1 : j+1) << endl;
}
```



An aside: `nan` and `inf`

CS 101, 2025

NaN (Not a Number) vs. inf (Infinity)

- **nan**: Short for *Not a Number*; cannot be defined or represented
- Examples where **nan** appears:
 - Log of a non-positive number
 - Square root of -1
- **inf**: Short for *Infinity*; numbers that are too large (in absolute value)
- Examples where **inf** appears:
 - Divide (non-zero) number by zero
 - *Overflow*: When a number exceeds the maximum representable floating-point number
- Note both these quantities relate to floating point numbers



while statement

CS 101, 2025

Compute average of scores

- *Requirement:* Read as input a sequence of student's scores (0 to 100) and print its average
 - Number of students is not known beforehand
 - Assume that at least one positive score will be given
 - Treat a negative number as a signal to end the sequence
- Example:
 - Input: 80,20,-5 Output: 50
- Implement using **repeat**?
 - **repeat** repeats fixed number of times and we do not know the number of students
- New looping constructs (while, do while, for) that naturally support such requirements

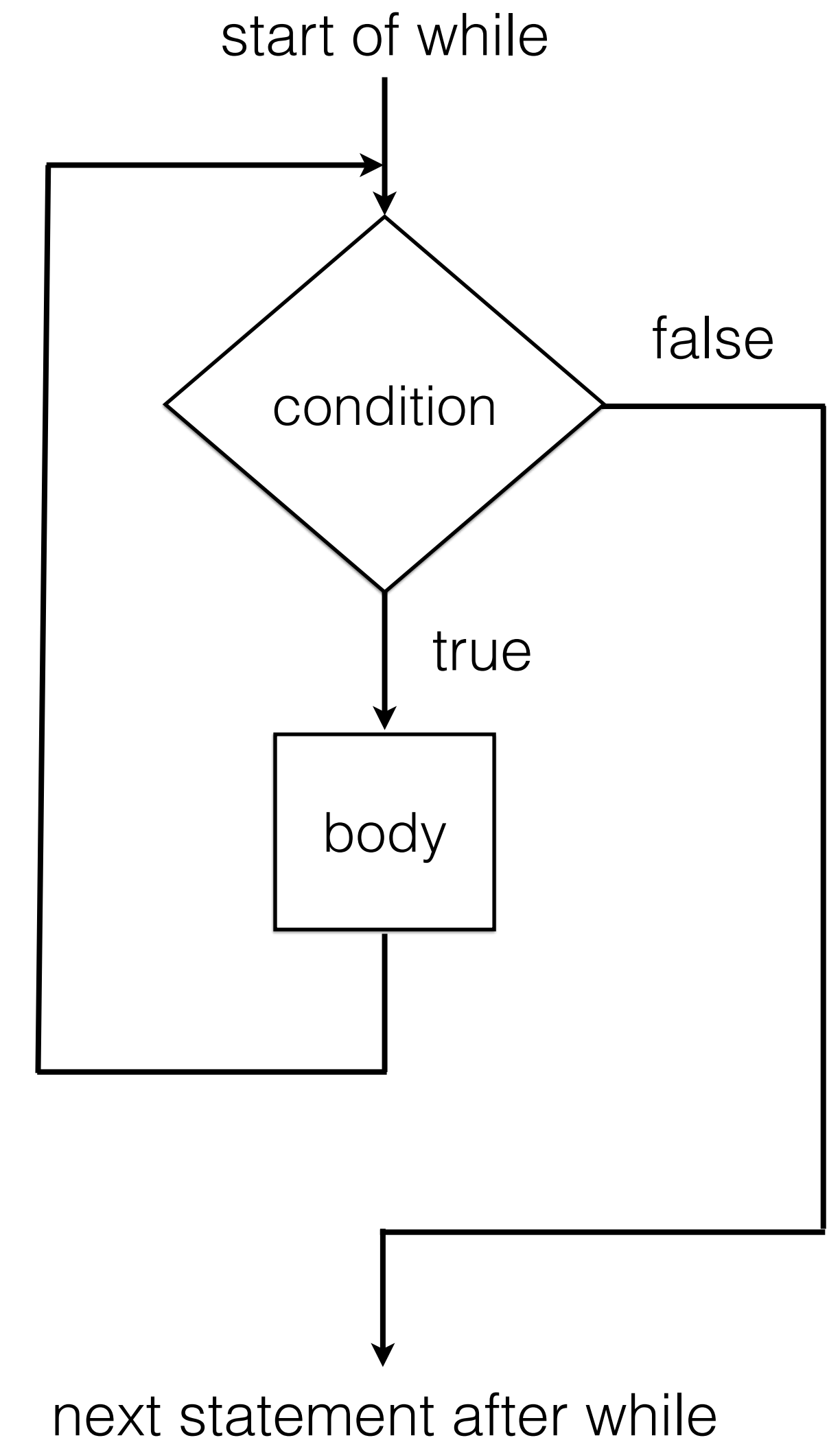
while statement

- Syntax:

```
while(condition) {  
    body  
}
```

- Semantics:

1. Evaluate condition
2. If condition evaluates to **true**, then **body** is executed
3. If condition evaluates to **false**, then skip the **while** block and move to the statement following **while**
4. Go back to step 1 and repeat



while statement (I)

- What does this program do?

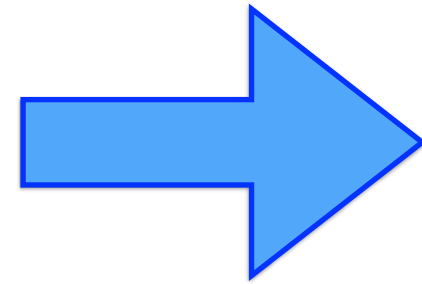
```
#include <simplecpp>
```

```
main_program {  
    int i = 0;  
    while(i <= 10) {  
        i = i + 2;  
        cout << i << endl;  
    }  
}
```

2
4
6
8
10
12

output

equivalent to



```
#include <simplecpp>
```

```
main_program {  
    int i = 0;  
    while(i <= 10) {  
        cout << (i+=2) << endl;  
    }  
}
```

while statement (II)

- What does this program do?

```
#include <simplecpp>
```

```
main_program {  
    int i = 0;  
    while(false) {  
        i = i + 2;  
        cout << i << endl;  
    }  
}
```

What if this was **true**?

Infinite loop!

Nothing will be printed

output

- The **while** condition must eventually become **false**, otherwise the program will never halt.

Code to average student's scores

Demo in class and code shared on Moodle



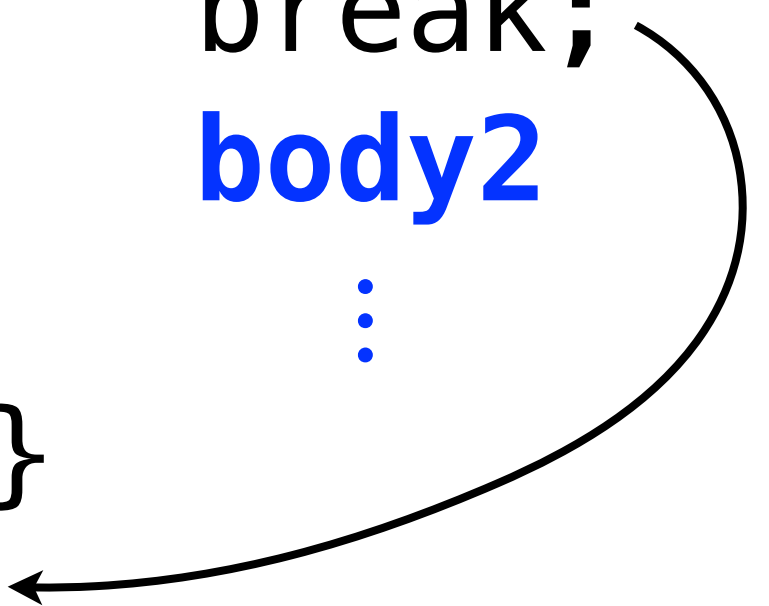
break/continue statements

CS 101, 2025

break statement within while loop

- Syntax of **break** within a **while** loop:

```
while(condition) {  
    body1  
    break;  
    body2  
    :  
}
```

A curved arrow originates from the right side of the 'break;' statement and points to the closing curly brace '}' of the while loop, indicating that the loop is terminated and control moves to the next statement after the loop.

- Semantics:
 - if *condition* is satisfied, **body1** is executed and when control reaches **break**, the execution of the **while** statement is terminated.
 - That is, **body2** is not executed if **break** appears right before it.
 - Execution continues from the next statement following the **while** block.

break in code to average student's scores

- Consider `break` in the following piece of code that implements averaging student's scores:

```
main_program {  
    float next, sum = 0;  
    int count = 0;  
    while(true) {  
        cin >> next;  
        if(next < 0) break;  
        sum += next;  
        count += 1;  
    }  
    cout << sum/count << endl;  
}
```

if `next < 0`, then the while loop execution terminates

Execution continues from the statement after `while`, i.e., `cout << ...`

break in code to average student's scores

- Consider **break** in the following piece of code that implements averaging student's scores:

```
main_program {  
    float next, sum = 0;  
    int count = 0;  
    while(true) {  
        cin >> next;  
        if(next < 0) break;  
        sum += next;  
        count += 1;  
    }  
    cout << sum/count << endl;  
}
```

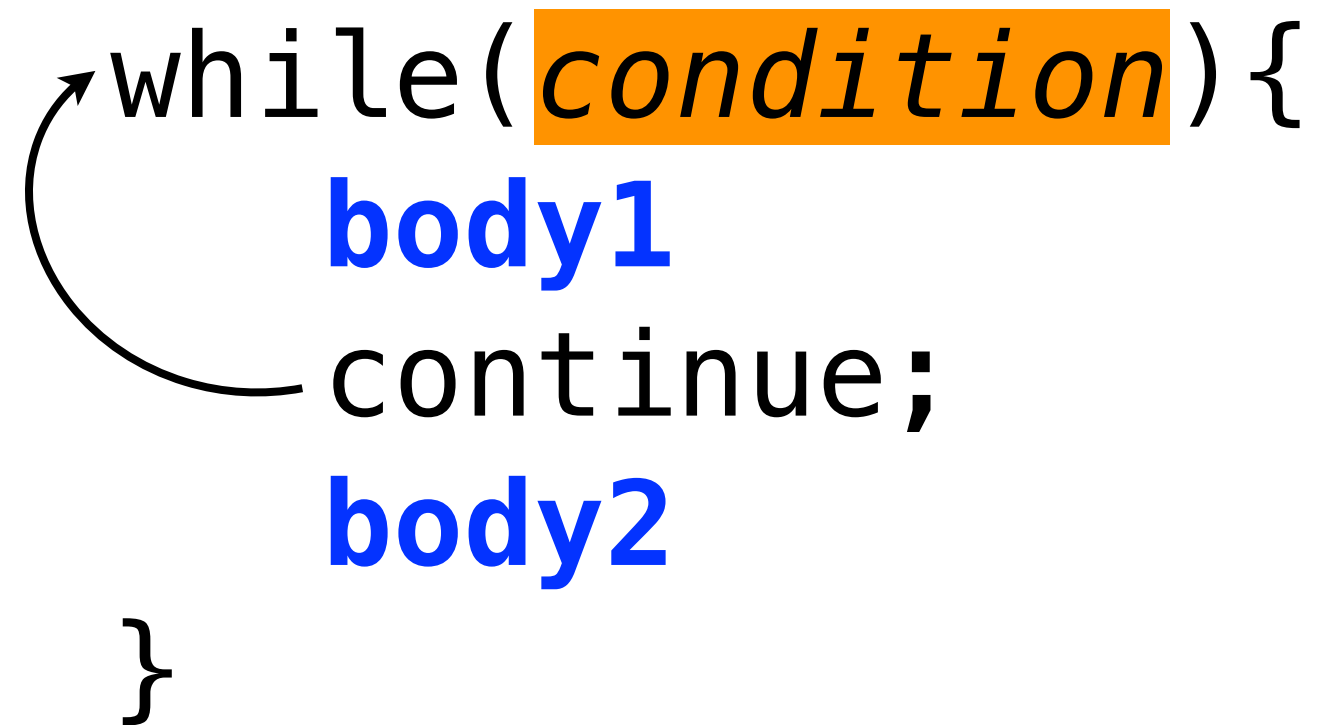
Note how **break** is written. Since **{}** is omitted, the single statement after **if** can appear on the same line.

- If **break** appears inside a **while** which is itself nested inside another **while**, then the inner **while** statement is terminated

continue statement within while loop

- Syntax of `continue` within a `while` loop:

```
while(condition) {  
    body1  
    continue;  
    body2  
}
```

A curved arrow originates from the `continue;` statement and points back to the `while(condition)` line, illustrating the jump to the start of the next iteration.

- Semantics:
 - if *condition* is satisfied, **body1** is executed and when control reaches `continue`, it goes to the `while` loop for the next iteration.
 - **body2** i.e., statements from `continue` to the end of the loop are skipped.

Averaging student's scores, with a constraint

- Ignore if a score > 100 , and move on to the next score in the input

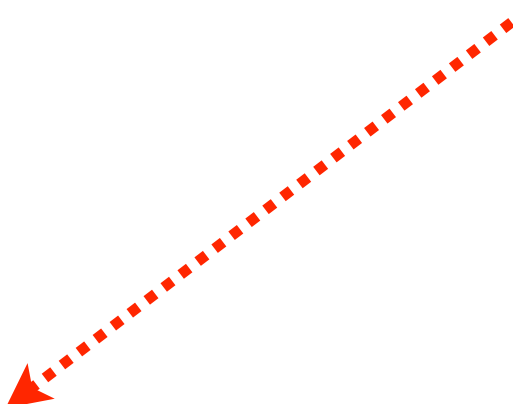
```
main_program {  
    float next, sum = 0;  
    int count = 0;  
    while(true) {  
        cin >> next;  
        if(next < 0) break;  
        sum += next;  
        count += 1;  
    }  
    cout << sum/count << endl;  
}
```

Averaging student's scores, with a constraint

- Ignore if a score > 100 , and move on to the next score in the input

```
main_program {  
    float next, sum = 0;  
    int count = 0;  
    while(true) {  
        cin >> next;  
        if(next > 100) continue;  
        if(next < 0) break;  
        sum += next;  
        count += 1;  
    }  
    cout << sum/count << endl;  
}
```

if **next > 100**, then control is transferred to the beginning of the while loop





do while statement

CS 101, 2025

do while statement

- Syntax:

```
do{  
    body  
}while(condition);
```

- Semantics: Equivalent to

```
{ body }  
while(condition) {  
    body  
}
```

- Note: The above equivalence holds only when **body** does not contain a `continue` statement. `continue` is only used within loop bodies.
- Compared to `while`, can avoid one condition evaluation, if it holds anyway in the beginning
- Compared to `while`, `do while` is less commonly used

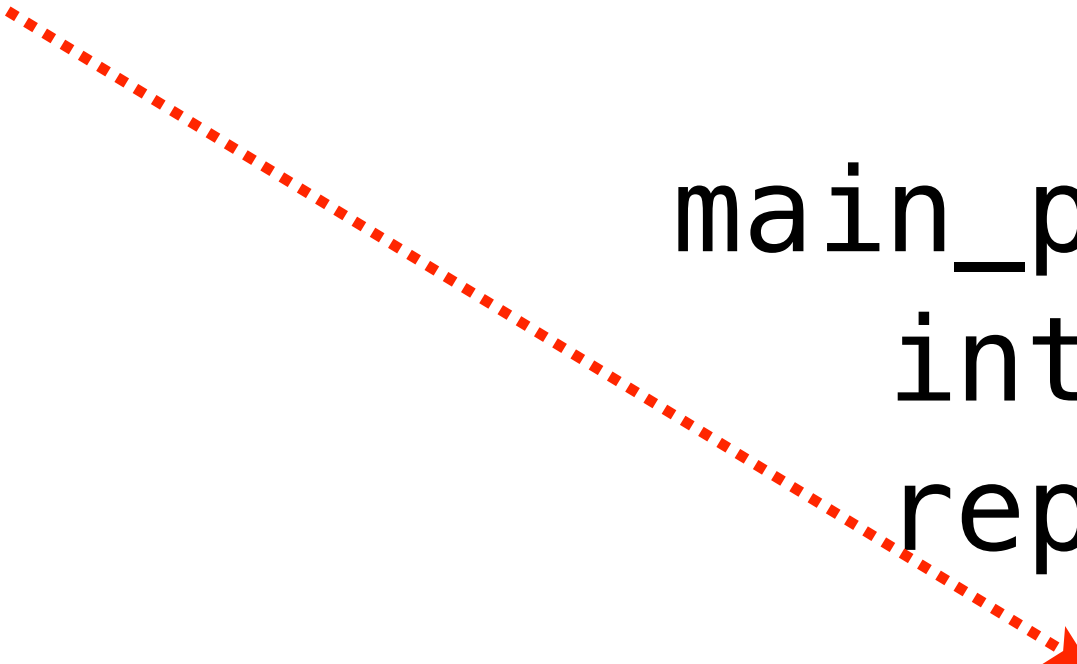


Blocks and scope

CS 101, 2025

Blocks and scope

- Code inside `{}` is referred to as a ***block***
- `repeat`, `if`, etc. typically consists of blocks; one could create them otherwise too by just adding `{}`
- Variables can be declared inside a block



```
main_program {  
    int sum = 0;  
    repeat(10) {  
        int term;  
        cin >> term;  
        sum += term;  
    }  
    cout << sum << endl;  
}
```

How definitions in a block execute

- A variable is defined/created every time control reaches the definition.
- All variables defined in a block are destroyed every time control reaches the end of the block.
- “Creating” a variable is only notional; the compiler simply starts using that region of memory from then on.
- Likewise “destroying” a variable is notional.

Scope and shadowing

- Variables defined outside a block can be used inside the block, if no variable of the same name is defined inside the block.
- If a variable of the same name is defined, then from the point of definition to the end of the block, the newly defined variable gets used.
- The new variable is said to “**shadow**” the old variable.
- The region of the program where a variable defined in a particular definition can be used is said to be the “**scope**” of the definition.

```
main_program {  
    int i = 3;  
    cout << i << endl;  
    {  
        cout << i << endl;  
        int i = 8;  
        cout << i << endl;  
    }  
    cout << i << endl;  
}
```

output

3
3
8
3



Next class: Looping construct "for"
CS 101, 2025