COMP 1633: Intro to CS II

Loop loop loopy loops 

Charlotte Curtis

January 29, 2024

Where we left off

- Boolean expressions
- `if-else` statements
- Some C++ specific boolean behaviour
- All the git lab chaos

> *Textbook Sections 2.4, 3.1-3.2*

```
if (x % 3 == 0)
    cout << "Fizz";
if (x % 5 == 0)
    cout << "Buzz";
```

Today's topics

- `while` and `for` loops
- Event controlled vs counted loops
- Some useful sentinels
- The last lecture of "review", and the last thing needed for assignment 1

> *Textbook Sections 3.3-3.4*

Review: Loop design decisions

Think about:

1. What **statements** do you want to repeat?
2. What **variable** (the LCV) should control the loop?
3. What **condition** should cause the loop to terminate? Then, invert it
4. What should the **initial conditions** of the loop be?
5. How should the LCV be updated?

Complete `while` loop example

```
int x = 1;                  // Initialization
while (x <= 100) {          // Condition
    if (x % 3 == 0)
        cout << "Fizz";
    if (x % 5 == 0)
        cout << "Buzz";
    cout << "\n";
    x++;                    // Update
}
```

- Forgetting to update the LCV leads to an **infinite loop**
- Initializing with the wrong value can lead to the loop never executing

Not a complete FizzBuzz implementation - what is this missing?

`for` loops - a bit more different

```python
for i in range(10):
    # code to execute
```

```java
for (int i = 0; i < 10; i++) {
    // code to execute
}
```

- Notice the semicolons! Inside the parentheses, there are three **statements**:
    i. Initialization
    ii. Condition
    iii. Update
- The LCV is declared inside the loop, and only exists inside the loop
- BUT this isn't actually mandatory - it's a good idea though

FizzBuzz as a `for` loop

Since FizzBuzz is counting from 1 to 100, it's a good candidate for a `for` loop:

```
for (int x = 1; x <= 100; x++) {
    if (x % 3 == 0)
        cout << "Fizz";
    if (x % 5 == 0)
        cout << "Buzz";
    cout << "\n";
}
```

- `for` loops help protect you from forgetting to initialize or update the LCV
- More readable for counted scenarios, as all 3 steps are in one place
- BUT you ~~can't~~ shouldn't use a `for` loop for **event controlled** repetition

Review: Compound conditions

Say you want to roll a pair of dice until you get a 12 OR you reach 5 rolls. Which of the following is the correct condition?

A. `roll != 12 || n_rolls < 5`

B. `roll != 12 && n_rolls < 5`

C. `roll == 12 || n_rolls >= 5`

D. `roll == 12 && n_rolls >= 5`

E. `roll == 12 || n_rolls < 5`

```
int roll = roll_dice();
int n_rolls = 1;

while (<condition>) {
    roll = roll_dice();
    n_rolls++;
}
```

De Morgan's Laws

- To determine the loop condition, it's often easier to think of when you want it to **stop** rather than when you want it to **continue**
  - "Stop when we get a 12 or reach 5 rolls"
  - "Stop when the user presses `q`"
- **Inverting** compound conditions can be tricky, but De Morgan's laws can help
  - `!(A && B) == !A || !B`
  - `!(A || B) == !A && !B`
- You can also just use the `!(stop condition)` syntax if it makes more sense

`while` loops vs `for` loops

- `for` loops allow you to keep your LCV in the local scope
- Otherwise, they're basically the same thing!

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

```
for (int i = 0; i < 10; i++) {
    cout << i << endl;
}
```

- You can do some really weird things with `for` loops (but please don't)

```
for ( ; ; ) cout << "I'm a loop in one line!" << endl;
```

`for` loop conventions

You really really should stick to the syntax of:

```
for (initialization; condition; update) {
    // loop body
}
```

- The initialization is only run once, at the start of the loop
- The condition is checked before a new iteration
- The update is run at the **end** of each loop body

> *C++ is highly flexible, and that power means it's your job to understand exactly what you want to have happen.*

Why `while`?

If `for` loops are just syntactic sugar for `while` loops, why do we have both?

- `while` loops are a good choice for **event-controlled** loops
  - You don't know how many times it'll run
  - The end of the loop is triggered by some kind of an event
- This includes **sentinel** loops

```
while user provides input
    keep on processing
```

Recall: Sentinel loops

- A **sentinel** is a specific value that is only used to signal the end of the data
- The sentinel is typically:
  - The same **data type** as the data
  - Added to the end of a stream of data to indicate the end
  - **Excluded** from processing
- Example: `.` at the end of a sentence

> *Write a function that reads a sentence character by character and counts the vowels, stopping when it reaches a period.*

Remember this pattern?

The loop we just wrote is an example of a **sentinel**, but it's also an example of which common loop pattern?

A. Counted loop

B. Accumulator

C. Summation

D. Variable-controlled loop

E. Fruit loop

End of input: a useful sentinel

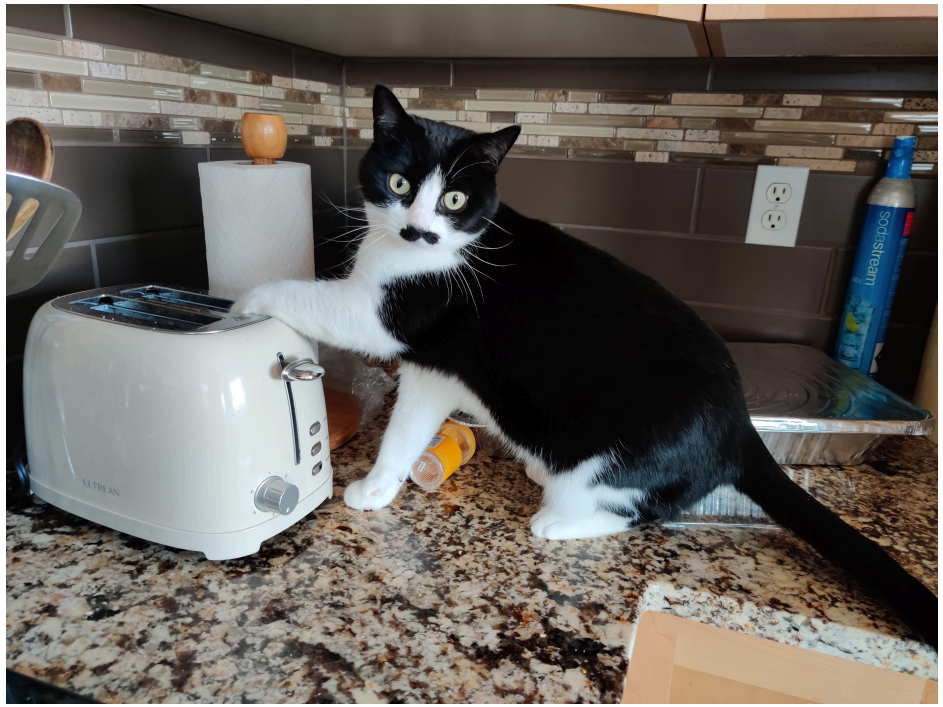
- Often we want to keep reading input until the end of file is reached
- This is so common that C++ provides a special sentinel for it: `eof()`
- For a given **input stream** the syntax is `stream_name.eof()`

```
while (!cin.eof()) {
    // read input
}
```

- This is a **member function** (aka "method") of the `istream` class that returns:
  - `true` if the end of file has been reached
  - `false` otherwise
- `eof()` only return `true` **after** an attempt to read past the end of file

Example using `eof()`

Modify the vowel-counting program to use `eof()` instead of a period as a sentinel.



Why a cat picture? The slide felt kind of empty.

More `eof()` considerations

- The internet will tell you that `eof()` as the loop condition is **always bad**
- This is because of the following (incorrect) code:

```
while (!cin.eof()) {
    cin >> x;
    // do something with x
}
```

- This code will always **repeat** the last value of `x`!
- Again, the LCV update should always be at the **end** of the loop body, necessitating a **priming read** before the loop

Alternatives to `eof()`

- The `>>` operator will evaluate to `false` if it fails to read a value
- This means we can put the read **inside** the `while` condition:

```
int x;
cin >> x;
while (!cin.eof()) {
    //do something with x
    cin >> x;
}
```

```
int x;
while (cin >> x) {
    //do something with x
}
```

- This is a **common pattern** for reading input in C++, though it might be more confusing than using `eof()`

Controversial loop topics

- `break` and `continue` are statements that interrupt the flow of the loop
  - `break` exits the loop immediately
  - `continue` skips the rest of the loop body and goes back to the top
- In general, these can make the flow of the program harder to follow
- For this course, **do not use them**
- **Definitely** don't use `goto`. From the textbook:

> *"Labels are a remnant from the C language and are used with `goto` statements. Their use is generally shunned because they can result in logic that is difficult to follow"*

Re-writing a loop with `break`

This is an example of actual code I've had submitted for assignments, often with a ChatGPT attribution:

```
int x, y;
while (true) {
    cin >> x >> y;
    if (eof())
        break;
    cout << x + y << endl; // actually more complex, but you get the idea
}
```

> *How would you re-write this loop without using `break`?*

Summary of loop types

- **Counted** loops: you know how many times you want to repeat
  - Prefer a `for` loop for readability and and less chance of errors
- **Event-controlled** loops: you don't know how many times you want to repeat
  - Prefer a `while` loop to signal that the loop is event-controlled
  - A **sentinel** is an example of an event-controlled loop
- `do-while` loops: run at least once, but less common than `while` loops
- **Avoid** `break`, `continue`, `return` in a loop, and `goto`!
  - anything that interrupts the flow of control makes things harder to follow

Getting fancy: nested loops

- Just like `if` statements, loops can be nested inside each other
- This gets a little brain-melty, but is quite useful

> *Challenge: write a function that takes an integer `n` and displays the times table up to $n \times n$*

Arrays preview

- Remember the `list` type in Python?

```
cities = ["Calgary", "Vancouver", "Toronto"]
current_temp = [15, 18, 20]
```

- It's possible, but **not a good idea**, to have mixed data types

```
city_and_current_temp = ["Calgary", 15]
```

- **Arrays** in C++ are kind of like lists, but the data types **must be the same**
- We'll start by looking at "C-style" arrays

C-style arrays

- C-style arrays are a **fixed size** (length) collection of elements of the same type
- When an array is declared, memory is allocated all at once
- An array is **not** a separate data type! The general form of the declaration is:

```
data_type variable_name[array_size];
```

- For example:

```
double current_temp[3];
```

- The array size must be a **constant** (not a variable)

Working with arrays

- Like Python, arrays can be indexed using `[]` with the index starting at 0

```
current_temp[0] = 15;
current_temp[1] = 18;
current_temp[2] = 20;
```

- Also like Python, this provides read/write access to the **array element**

```
for (int i = 0; i < 3; i++) {
  cout << "The current temperature is: " << current_temp[i] << endl;
}
```

- Finally, arrays can be **initialized** when they are declared

```
double current_temp[3] = {15, 18, 20};
```

Coming up next

- Loop lab
- Lecture: Arrays
- Assignment 1: Due February 9, 2024 (Next Friday)

> *Textbook Chapter 7*