String Streams & Assertions

CS 150 – C++ Programming I

Lecture 13

Introducing String Streams

- Instead of writing output to the screen or to a file, or reading input from the keyboard or a file, a string stream reads and writes data to and from `string` objects
- Include the header: `<sstream>`
- For writing create a new output string stream

  -
    ```
    ostringstream out;   // empty, no data yet
    ```
- Then, write any kind of data to the stream

  -
    ```
    out.put('Q');
    out << "ED. " << 2018 << endl;
    ```

Using String Streams

- After writing, make a copy of the *string* you've written to
  - `string s = out.str();   // when finished`

- String stream classes are used when you need to mix numeric and text formatting
  - `String ans = "The answer is: " + 42; // Java ONLY`

  -
    ```
    out << "The answer is: " << 42; // C++
    string ans = out.str();
    ```

- You may reuse the same stream object again
  - `out.str("");   // fresh string buffer`

Applying Output String Streams

- Exercise: Write a function taking a double monetary value and returning a dollar formatted C++ string.
- You use the function like this:

```
double amt = 1234.0;
cout << toDollars(amt) << endl;
```

- Input 1234 should produce "$ 1,234.00"

String Stream Input

- In Java, you can parse a *String* by using a *Scanner*

```
– Scanner in = new Scanner("Mar 17 2022");
  String month = in.next();      // "Mar"
  int day = in.nextInt();        // 17
```

- In C++, use an input string stream to do something similar

```
– istringstream in("Jan 1, 2018");
– string month;
  in >> month;
  int day, year; char comma;
  in >> day >> comma >> year;
```

Applying Input String Streams

- Complete the `list()` function which takes a string containing a file-name and a pair of numbers, and then prints only the input lines falling between those lines. The `main()` function uses it like this:

```
list("alice.txt 40 50")
```

- Prints lines `40-50` in `alice.txt`. Assume that lines start at `1` and that you include both line `40` and `50`. Return `true` if successful, otherwise `false`.

Assumptions & Preconditions

- Often functions make assumptions about their inputs
  - These are called a function's precondition
- What is assumed about `n` in `cout << sqrt(n)`?
  - We assume that it is a positive number
- The `stoi()` function converts a `string` to an `int`
  - What would we assume about `s` when calling `stoi(s)`?
  - That `s` contains something like `"125"` and NOT `"one"`
- At a minimum, document your assumptions about inputs
  - *`@pre n should be >= 0 // sqrt`*

Assumptions & Postconditions

- A postcondition is what we assume will be true when the function has completed
  - May include external side effects (global variables, etc)
    - `cout.put(65); // 'A' sent to standard output`
  - Should include what the function is assumed to return
- Document these as well using these DOXYGEN tags
  - `@post status is true if number read correctly`
  - `@exception throws std::out_of_range when num is out of range`

Precondition Violations

- Five things you can do when given inappropriate input
  - 1. Fail "safely": aka defensive programming
    - eg. have `stoi("one")` return `0`;
    - Problem? makes it very hard to find errors in your code
  - 2. You can terminate the program with an error message
  - 3. You can return an error code which the user can check
    - Or, you can set an error state (eg. `cin.fail()`)
    - Problem? programmers may (will?) ignore
  - 4. You can throw an exception which can be caught
  - 5. You can do nothing (it's a feature, not a bug!)

How To Handle Your Errors

- Some errors are caused by external circumstances
  - User types in wrong URL, saves to full thumb drive
  - Don't want the program to terminate when that happens
- Other errors, though, are caused by you, the programmer
  - Here, you do want the error to "announce" itself and stop
  - You don't want it to fail silently since you'll never find the bug
- To do this, you use an `assert`
  - `#include <cassert>`
  - `assert(counter > 3);`

Using assert

- Assertions allow you to write self-checking code
  - This is called instrumenting your programs
  - Add code to automatically detect and notify you
  - Sometimes called sanity checks or smoke tests
- Use `assert()` for things that cannot logically happen

  -
    ```
    int sum_between(int lower, int upper) {
       assert(lower <= upper); // cannot happen
    ```
- Remove checks in production code
  - Use `#define NDEBUG` before `#include <cassert>`
  - Alternatively, add flag `-D NDEBUG` when compiling