Working with Strings, the Type `std::queue`

Lecture 10

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

Data Centric vs Client/Server

- In the data centric architecture, both clients and servers modify the same database
  - Firebase apps use a data centric design
- Using dynamic, table-driven logic, as opposed to logic embodied in previously compiled programs
- Using stored procedures that run on database servers, as opposed to greater reliance on logic running in middle-tier application servers
- Using a shared database as the basis for communicating between parallel processes in distributed computing applications
- This is different than a client/server model
  - In client/server, clients go through the server, and the server updates the database

Data Centric Benefits

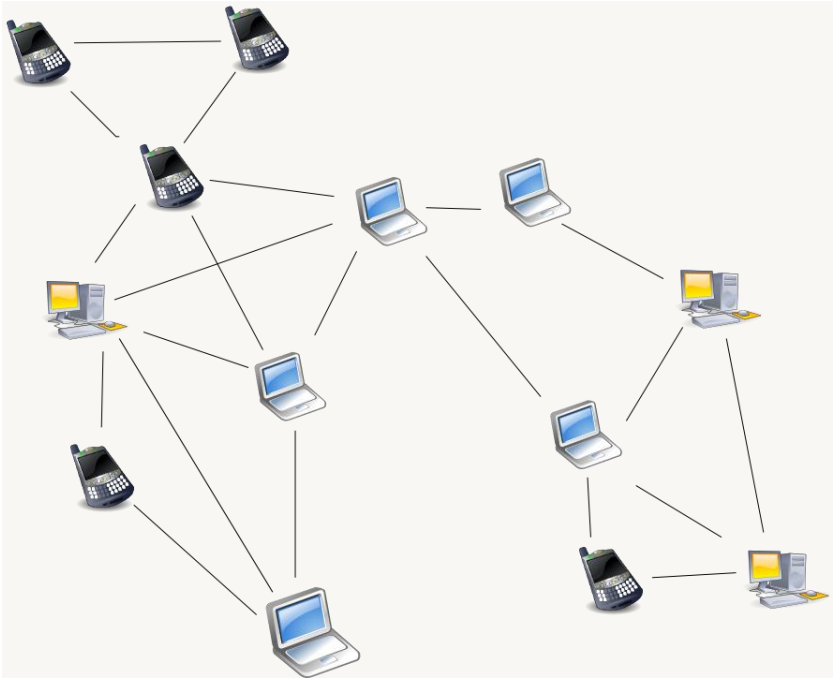
- Lower latency than client/server
- Easy to scale number of servers (major benefit)
- Database handles authentication and security

Data Centric Drawbacks

- Some operations are too complicated to allow clients to manage
- Breaks encapsulation
  - adds coupling between client and data
- Uglier communication interface
  - Servers pull requests from database instead of API
- Single point of failure
  - Attractive target for DDoS attacks

Peer to Peer Concept

- No central server, peers broadcast directly to each other
- Each client maintains private state
- Needs to manage:
  - Synchronization
  - Dropout
  - Security
  - Flooding
- Quite complicated



Peer to Peer Benefits

- Robust: no single point of failure
- Doesn't require central infrastructure
- Useful when communication range is limited (e.g., underwater drones)

Peer to Peer Drawbacks

- Synchronization problems
  - What time is it?
- Consensus problems
  - What is the state?
- Byzantine attacks
  - A node becomes infected and provides wrong information to other peers, creating an unstable network

Peer to Peer Examples

- Cryptocurrency
- Automotive networks
- Internet of Things (IOT)
- PC to PC local Wi-Fi hotspot

The Type `std::string`

Abstract

- In this lecture we look at strings in C++: `std::string`
  - Briefly introduce simple string manipulation
- Create a user defined data type that represents an ‘image’
  - A set of strings, that when printed resemble ASCII art
- Develop a set of algorithms that can be used to manipulate those images
- Briefly introduce `std::queue<T>`, a standard container adaptor modelling – well – queues.

Brief Overview

Framing a Name

- Given the following interaction:

```
Please enter your name: John
```

- We would like to print:

```
**********************
*                    *
*   Hello John!      *
*                    *
**********************
```

Framing a Name

```
// Ask a persons name, greet the person

#include <iostream>
#include <string>

int main()
{
    std::cout << "Please enter your first name: ";        // ask for the persons name
    std::string first_name;                               // define 'first_name'
    std::cin >> first_name;                               // read into 'first_name'

    std::string const greeting = "Hello, " + first_name + "!";       // build the message we intend to write
    std::string const spaces(greeting.size(), ' ');       // build the second and fourth string
    std::string const second = "*   " + spaces + "   *";
    std::string const first(second.size(), '*');          // build the first and fifth lines

    std::cout << first << std::endl;                      // write all
    std::cout << second << std::endl;
    std::cout << "*   " + greeting + "   *" << std::endl;
    std::cout << second << std::endl;
    std::cout << first << std::endl;
    return 0;
}
```

Framing a Name

- Variable definition
  - Initialization
    - Important!
  - Use operator + for concatenation
    - Custom operators do not change precedence, associativity, or number of arguments
  - Constness (keyword `const`)
    - Variable will not be changed
    - Requires initialization
    - May allows for more optimizations
    - Expresses intention to the compiler

C++ `std::string`

- Models a sequence of characters
- `std::string` is defined as a class (user defined) type
- Simple operations
  - Member function `size()` returns number of chars
  - `operator[]` to access individual characters
  - C++ strings are mutable
- Operators on strings
  - `operator=` assigns, makes new copy
  - Compare with relational operators: <, <=, ==, >=, >
    - Lexicographical ordering
  - `operator+` concatenates (associative, but not commutative)
- It's a `TotallyOrdered` type

Framing a Name

- Initializing using special constructor

```
std::string spaces(greeting.size(), ' ');
```

  - Number of characters and character literal (`' '`)
  - Quoting similar to string literals (`'\t'`, etc.)

Exercise: write a function to create a `std::string` consisting of `N` equal characters using the `power` algorithm.

Explain why `power` can be used.

C++ std::string

```cpp
#include <iostream>                          // std::cout, std::endl
#include <string>                            // std::string

int main()
{
    std::string s, t = "hello";

    s = t;                                   // s == "hello"
    std::cout << "Length: " << s.size() << std::endl;

    t[0] = 'j';                              // t == "jello"
    s = s + ' ';                             // s == "hello "
    s += t;

    std::cout << s << std::endl;             // prints: 'hello jello'
    return 0;
}
```

C++ std::string

```cpp
#include <iostream>     // std::cout, std::endl
#include <string>       // std::string
#include <cctype>       // std::toupper
#include <algorithm>    // std::transform

int main()
{
    std::string s;
    s = "csc3380";

    // converts to upper case
    std::transform(std::begin(s), std::end(s), std::begin(s),
        [](char c) { return std::toupper(c); });

    std::cout << s << std::endl;    // prints: CSC3380
    return 0;
}
```

C++ std::string

```cpp
#include <iostream>     // std::cout, std::endl
#include <string>       // std::string
#include <algorithm>    // std::sort

int main()
{
    std::string s;
    s = "hello John";

    // sort all characters of the string
    std::sort(std::begin(s), std::end(s));

    std::cout << s << std::endl;           // prints: ' Jehhllnoo'
    return 0;
}
```

C++ `std::string` member functions

- Invoke member functions as: `str.function(args)`
- Sample member functions:
  - Return index after `pos` of first occurrence or `std::string::npos`

```
int find(char c, int pos = 0);
int find(std::string const& pattern, int pos = 0);
```

  - Return new string, copies `len` characters starting at `pos`

```
std::string substr(int pos, int len);
```

  - Changes string, inserts `txt` at `pos`

```
void insert(int pos, std::string const& txt);
```

  - Changes string, removes `len` characters starting at `pos`

```
void erase(int pos, int len);
```

  - Changes string, removes `len` characters starting at `pos`, inserts `txt`

```
void replace(int pos, int len, std::string const& txt);
```

More String Details

- IO is using `operator>>` and `operator<<`
  - `os << s`:
    - output s to stream os without formatting changes
    - evaluates to `os`
  - `is >> s`:
    - input s from stream is with whitespace handling (stops at whitespace!)
    - evaluates to `is`
- Ways to initialize:

```
std::string hello = "hello";
std::string stars(10, '*');
std::string name;
```

C++ strings vs. C strings

- C++ inherits legacy of old-style C string
  - String literals are actually C strings (e.g. `"C-string literal"`)
  - (pointer to array of null terminated characters)
- Converting C string to C++ string
  - Happens automatically in most cases
  - Can be forced: `std::string("abc")`
- Converting C++ string to C string
  - Using member function `s.c_str()`
  - Critical when using: `printf("%s", s.c_str());`
- Why do we care
  - Some older functionality requires use of C strings
  - C strings are not compatible with concatenation

C++ strings vs. C strings

- Concatenation pitfalls
  - If one operand is a C++ string, all is good

```
std::string str = "Hello ";
str = str + "John";
str = str + '!';
```

  - If both operands are C strings/characters, bad times

```
"abc" + "def";     // won't compile
"abc" + 'd';       // will compile, but does wrong thing
```

  - Can force conversion if needed

```
std::string("abc") + 'd';
```

The `image` Type

Putting Strings Together

- Earlier we wrote a program to frame a string
  - Never created the output in a string
  - Rather printed the parts separately
- So far we generated 'images' on the fly
  - Let's compose the image in memory
- Let's create a user defined type representing a 'picture' (ASCII art), each string is one line
- Now, we will build a program framing such a picture
  - Puts together the whole picture in a data structure before printing it

The image Type

- Let's make it a Regular type:

```
struct image
{
    std::vector<std::string> data;     // image data

    image() = default;                 // default constructor (empty image)
    ~image() = default;                // destructor

    image(image const&) = default;     // copy constructor
    image& operator=(image const&) = default;    // copy assignment

    friend bool operator==(image const& lhs, image const& rhs)
    {
        return lhs.data == rhs.data;
    }
    friend bool operator!=(image const& lhs, image const& rhs)
    {
        return !(lhs == rhs);
    }
};
```

The image Type: Constructors

```
// construct an 'image' from a single line
explicit image(std::string const& line)
  : data(1, line)
{
}

// construct an 'image' from multiple lines
explicit image(std::vector<std::string> const& lines)
  : data(lines)
{
}

// construct a 'space-filled' image of the given size
image(int width, int height)
  : data(height, std::string(width, ' '))
{
}
```

The image Type: Constructors

```
// construct an 'image' from a single line
explicit image(std::string const& line)
  : data(1, line)
{
}

// construct an 'image' from multiple lines
explicit image(std::vector<std::string> const& lines)
  : data(lines)
{
}

// construct an optionally 'space-filled' image of the given size
image(int width, int height, char fill = ' ')
  : data(height, std::string(width, fill))
{
}
```

The image Type

- Add output support:

```
 // print the 'image'
friend std::ostream& operator<<(std::ostream& os, image const& img)
{
    for (auto const& line : img.data)
        os << line << '\n';
    return os;
}
```

Putting Strings Together

- Store all strings in a `std::vector<std::string>` one line each and surround it by a border
- For example, a framed `image` holding “this is an”, “example”, “to illustrate”, “framing” will result in:

```
*****************
* this is an    *
* example       *
* to illustrate *
* framing       *
*****************
```

Putting Strings Together

- Box is rectangular not ragged as single strings
- Find the overall width and height of the image (i.e. the longest string and the number of strings):

```
std::string::size_type image::width() const
{
    std::string::size_type maxlen = 0;
    for (auto const& s: data)
        maxlen = std::max(maxlen, s.size());
    return maxlen;
}

std::string::size_type image::height() const
{
    return data.size();
}
```

- Exercise: use standard algorithm (`accumulate`, or even better, `max_element`)

Framing Pictures

- What should the interface look like?
- Should it be based on member functions or (global) free functions?
  - No clear criteria, what about
    - Member function to apply the framing 'in-place'
    - Free function to create a new (now framed) image
- Let's create it as a global function:
  - Algorithm `frame()` will take an `image` and should return a *new* `image`:

```
image frame(image const& img, int gap = 1)
{
    // extract width of right hand side image
    // create top line and append to result
    // append each line from img to result after adding '*'
    // create bottom line and append to result
}
```

- Exercise: create it as a member function

Framing Pictures

```
image frame(image const& img, int gap = 1)
{
    image ret;
    auto maxlen = img.width();    // find longest string

    // create top line and append to result
    std::string const border(maxlen + 2 + 2 * gap, '*');
    ret.append(border);

    // append each line from v to result after adding '*'
    std::string const spaces(' ', gap);
    for (auto const& s : img.data) {
        ret.append("*" + spaces + s + std::string(maxlen - s.size(), ' ') + spaces + "*");
    }

    ret.append(border);          // 'write' the bottom border
    return ret;
}
```

More Image Functions

- What else can we do to 'images'?
  - Concatenation! Vertical and horizontal
  - Flipping and rotating images
- The `image` type represents an 'ascii' picture
  - Vertical concatenation is simple: just concatenate the two vectors
  - 'Pictures' will line up along their left margin
  - No predefined concatenation of vectors, let's define one:

```
image vcat(image const& top, image const& bottom, int gap = 0);
```

  - Optional gap between images using the default argument
    - If not specified, it will use the default value

Vertical Concatenation

```
// vertical concatenation
image vcat(image const& top, image const& bottom, int gap = 0)
{
    // copy the top picture
    image ret = top;

    for (int i = 0; i < gap; ++i)
        ret.append(std::string());   // insert empty string

    // copy entire bottom picture
    for (auto const& s : bottom)     // <<-- unfortunately doesn't work
        ret.append(s);

    return ret;
}
```

Expose Iterators from `image`

- What is necessary for this to work?

```
for (auto const& l : img) {...}
```

- Type `image` needs to expose `begin` and `end`
  - Those need to return an iterator
- Our underlying vector exposes `begin`, `end`, which in turn expose iterators
  - Simply re-expose those from our `image` type:

```
auto image::begin() const
{
    return data.begin();
}
auto image::end() const
{
    return data.end();
}
```

Vertical Concatenation

```
// vertical concatenation
image vcat(image const& top, image const& bottom, int gap = 0)
{
    // copy the top picture
    image ret = top;

    for (int i = 0; i < gap; ++i)
        ret.append(std::string());   // insert empty string

    // append entire bottom picture, new member function!
    ret.append(bottom);

    return ret;
}
```

Horizontal Concatenation

- For example:

```
this is an **************
example    * this is an *
to         * example    *
illustrate * to         *
framing    * illustrate *
           * framing    *
           **************
```

Horizontal Concatenation

- If left hand side picture is shorter than right hand side, we need padding
  - Otherwise just copy the picture
- Interface similar to vcat:

```
// horizontal concatenation
image hcat(image const& left, image const& right, int gap = 1)
{
    image ret;

    // get width of left, add 1 to leave a space between pictures
    // handle all rows from both pictures, line by line
        // copy a row from the left-hand side, if there is one
        // pad the line to full width of left + 1
        // copy row from the right-hand side, if there is one
        // append overall line to result image

    return ret;
}
```

Horizontal Concatenation

```
// horizontal concatenation
image hcat(image const& left, image const& right, int gap = 1)
{
    image ret;
    auto width1 = left.width() + gap;    // add ‘gap’ to leave space between images
    auto i = 0, j = 0;
    while (i != left.height() || j != right.height()) {
        std::string s;    // construct new string to hold characters from both images

        if (i != left.height()) s = left[i++];       // copy a row from the left-hand side, if there is one

        s += std::string(width1 - s.size() + gap, ' ');    // pad to full width, add gap

        if (j != right.height()) s += right[j++];    // copy a row from the right-hand side, if there is one

        ret.append(s);    // add s to the picture
    }
    return ret;
}
```

Vertically Flip an image

- For example:

```
this is an   flipping
example      illustrate
to           to
illustrate   example
flipping     this is an
```

Vertically Flip an `image`

- Straight forward solution:

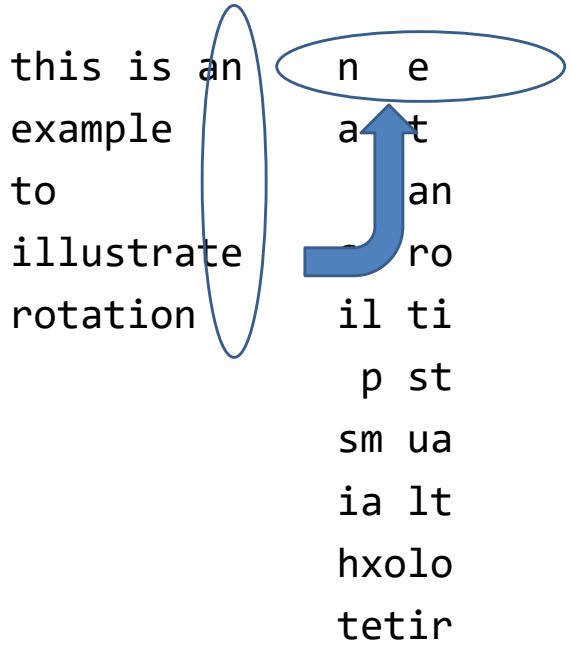
```
image vflip(image const& v)
{
    image ret;
    for (auto it = v.rbegin(); it != v.rend(); ++it)
        ret.append(*it);
    return ret;
}
```

- Using Standard algorithm:

```
image vflip(image const& v)
{
    image ret;
    std::reverse_copy(v.begin(), v.end(), std::back_inserter(ret.data));
    return ret;
}
```

Left-rotate an image

- For example:



Left-rotate an image

```
// left-rotate an image
image rotate_left(image const& img)
{
    image ret;

    // take a letter from each string, starting at the end of the line
    for (auto i = img.width(); i != 0; --i)
    {
        std::string line;                   // new line
        for (auto const& current: img) {    // for all lines in the input image
            if (current.size() < i) line += ' ';
            else                    line += current[i-1];
        }
        ret.append(line);                   // store the new line in the result picture
    }
    return ret;
}
```

Generating Output On the Fly

```
int main()
{
    std::vector<std::string> example = {
        "this is an", "example", "to illustrate", "framing"};

    image img(example);

    std::cout << frame(img, 3);        // creates vector just to discard it

    return 0;
}
```

Generating Output On the Fly

- If all we need is to generate the framed image
  - No need to materialize the `std::vector<std::string>`
  - It will be discarded right away
- Anther application of the decorator pattern
- Create special type `framed_image`

```
struct framed_image
{
    image const& base;    // underlying image to frame
    int gap;              // gap for spacing
};
```

Generating Output On the Fly

```
std::ostream& operator<<(std::ostream& os, framed_image const& img)
{
    auto maxlen = img.base.width();    // find longest string

    // create top line and append to result
    std::string border(maxlen + 2 + 2 * img.gap, '*');
    os << border << '\n';

    // append each line from v to result after adding '*'
    std::string spaces(img.gap, ' ');
    for (auto const& s : img.base) {
        os << "*" + spaces + s + std::string(maxlen - s.size(), ' ') + spaces + "*\n";
    }

    os << border << '\n';    // 'write' the bottom border
    return os;
}
```

Generating Output On the Fly

```
int main()
{
    std::vector<std::string> example = {
        "this is an", "example", "to illustrate", "framing"};

    image img(example);

    std::cout << frame(img, 3);         // creates vector just to discard it
    std::cout << framed_image(img, 3); // generates output on the fly

    return 0;
}
```

The Type `std::queue`

Introduction to Queue

- A queue is a first-in-first-out (FIFO) data structure
- Limited access vector (or list, or deque)
- Main operations:
  - Adding an item: `queue.push(e)`
    - Referred to as pushing it onto the queue (enqueue an item, adding to the end)
  - Removing an item: `queue.pop()`
    - Referred to as popping it from the queue (dequeue an item, remove from the front)
  - Access front item: `auto e = q.front()`
    - Note that pop does not return the front item, it just removes it

Introduction to Queues

- Definition:
  - An ordered collection of data items
  - Can be read at only one end (the front) and written to only at the other end (the back)
- Operations:
  - Construct a queue (usually empty)
  - Check if it is empty
  - push: add an element to the back
  - front: retrieve the front element
  - pop: remove the front element
  - size: returns number of elements in queue

Introduction to Queues

- Useful for
  - All kind of simulations (traffic, supermarket, etc.)
  - Computers use queues for scheduling
    - Handling keyboard events
    - Handling mouse events
    - Scrolling the screen
    - Printer spooler
  - C++ I/O streams are queues
    - Even if we only access one end, the other end is managed by the operating system

Example: Queue of Strings

```
void manage_queue()
{
    std::queue<std::string> waiting_strings;     // queue of 'waiting' strings
    while (true) {
        std::cout << "?> ";                      // ask for next line of text
        std::string response;
        std::getline(std::cin, response);

        if (response.empty()) break;
        if (response == "next") {                // try to dequeue
            if (waiting_strings.empty())
                std::cout << "No one waiting!" << std::endl;
            else {
                std::cout << waiting_strings.front() << std::endl;
                waiting_strings.pop();
            }
        }
        else {                                   // enqueue the line read
            waiting_strings.push(response);
        }
    }
}
```

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP