10. Streams and Container Classes

10.1. Streams

10.2. Container Classes

10.1. Streams

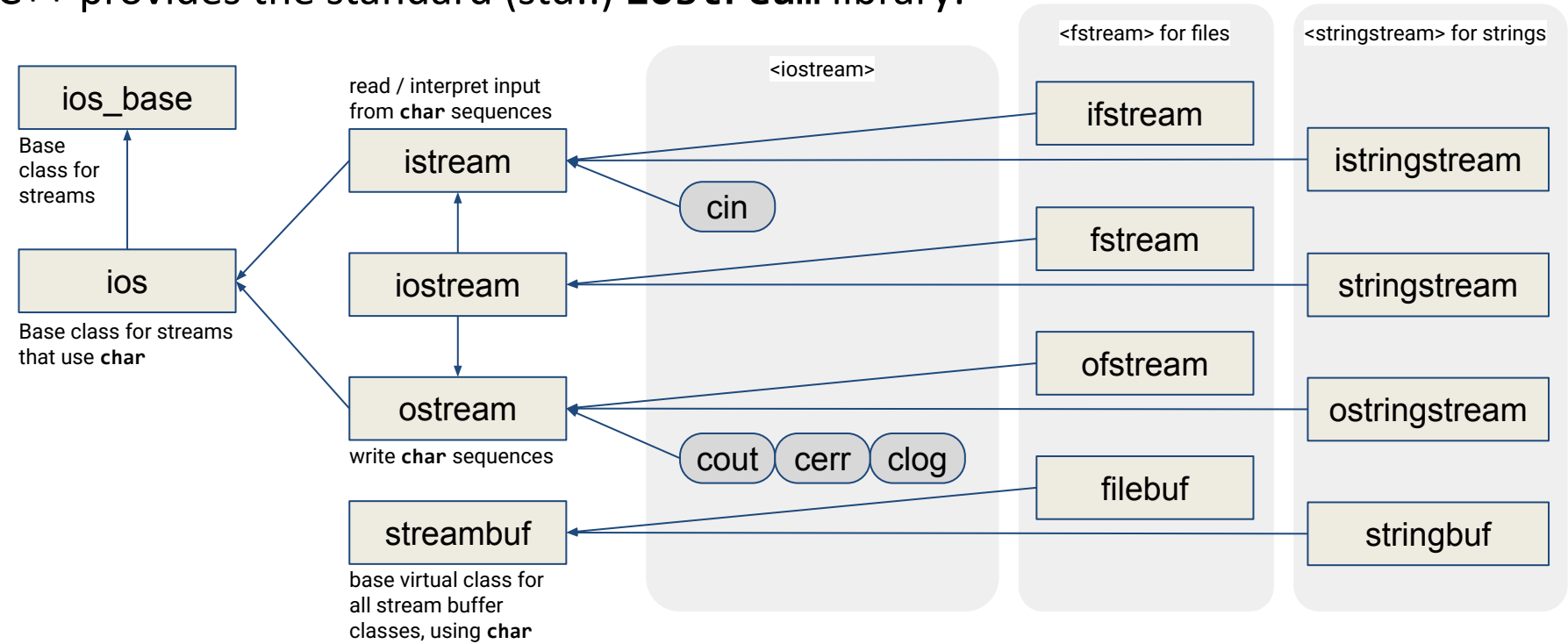
A stream class is a class which provides input and output functionality, as a standard abstraction across devices where input and output operations are performed. A stream can be represented as a source or destination, of characters of indefinite length.

For examples sending characters to and receiving characters from …

disk files, the keyboard and the console, a network connection

10.1. Streams

C++ provides the standard (std::) **`iostream`** library:



10.1. Streams

`std::streambuf` buffers manage the actual buffer on a lower level, with methods such as **`overflow()`** to handle output (writing), **`underflow()`** to handle input (reading), or **`sync()`** to flush data.

`std::iostream` Streams wrap a **`std::streambuf`** buffer, providing several common methods and operators, including:

- **`get`** / **`put`** a single character from/to a stream before returning
- **`read`** / **`write`** a certain amount of data from/to a stream before returning
- **`getline`** reads characters from an input stream until a delimiter character (usually **`'\n'`** is found) and places them into a string
- stream insertion operator **`<<`** for output to the stream
- stream extraction operator **`>>`** for input to the stream

10.1. Streams

Example 00 (difficulty level: 🌶🌶🌶): Install boost and compile:

```
/* change the following code to output the server reply straight to a local html file
   and add exception handling in case of file or connection problems */
#include <fstream>
#include <boost/asio.hpp>

int main() {
  char reply[4096];
  boost::asio::ip::tcp::iostream socket("www.example.com", "http");  // socket stream
  std::ofstream outputFile("myTest.txt");  // stream to output file
  socket << "GET / HTTP/1.1\r\nHost: www.example.com\r\nConnection: Close\r\n\r\n";
  socket << std::flush;
  socket.read(reply, 4096);
  outputFile << "Reply of server:\n" << reply;  // output reply of server to text file
}
```

10.1. Streams

Example 01 (difficulty level: 🌶🌶🌶):

```
/* An example of dynamically transforming output (similar to filters in Boost) */
#include <iostream>
class Uppercase : public std::streambuf {  // Class for a custom stream buffer
  std::streambuf* target;
 public:
  Uppercase(std::streambuf* buf) : target(buf) {}
 protected:
  // overflow is virtual std::streambuf function, called by stream writing a char:
  int overflow(int ch) override {
    if (ch != EOF) ch = std::toupper(ch); return target->sputc( ch );
  }
};
int main() {
  Uppercase u( std::cout.rdbuf() );  // buffer links to cout
  std::ostream upperOutput( &u );  // output stream using u
  upperOutput << "this should be uppercase\n";
}
```

10.1. Streams

Example 02 (difficulty level: 🌶🌶🌶):

```
/* Implement the two override methods of RLEstreambuf to implement a basic run
   length encoding compression of a string being sent to an output stream. The
   output of the program below should give "6A5B2CD2A6C3A".
   Note that EOF is a special terminator character that should not be handled.
   Use sputc() to write characters, and '0'+count to write the ascii digits. */
#include <iostream>
class RLEstreambuf : public std::streambuf {  // Class for a custom stream buffer
 public:
  RLEstreambuf(std::streambuf* dest) : dest(dest), last_char(EOF), count(0) {}
 protected:
  int overflow(int ch) override;  // override overflow to handle the compression
  int sync() override;            // flush last data (last character) when done
 private:
  std::streambuf* target;
  int last_char;          // previous character
  int count;              // count of consecutive same characters
};
```

10.1. Streams

Example 02 (difficulty level: 🌶️🌶️🌶️):

```
int main() {
  RLEstreambuf rleBuf(std::cout.rdbuf());
  std::ostream rleOut(&rleBuf);
  std::string input = "AAAAAABBBBBCCDAACCCCCCA";  // Input string to compress
  rleOut << input;  // the output is automatically RLE compressed
  rleOut.flush();   // flush leads to sync being called to output last character
  std::cout << '\n';
}
```

10.2. Container Classes

A container class is a class which implements a data structure containing objects of other classes, with well-defined access patterns (e.g., inserting, finding, removing, or sorting objects), independent of the type of objects stored inside.

Examples: Array, Stack, Queue, List, Tree

10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```
class Queue {  // Class for a queue of integers
 public:
  Queue(int size = 100) : maxSize(size), tail(0), head(0), filled(0) { items = new int[size]; }
  ~Queue() { delete[] items; items = nullptr; };
  void put(int data);
  int get();
  bool isFull() const { return filled == maxSize; }
  bool isEmpty() const { return filled == 0; }
  void clear() { filled = 0; head = 0; tail = 0; }  // clear whole queue
 private:
  int * items;   // array of integers
  int maxSize;   // size of items
  int tail;      // position in array to put
  int head;      // position in array to get from
  int filled;    // number of elements in queue
};
```

10.2. Container Classes

Illustration of a container: A queue of predefined size for integers

```
// put element at the tail of the queue, for example put(17) updates:
// items:  [  ][  ][  ][  ][  ][17][  ] … [  ][  ]
//              head              tail→
void Queue::put(int data) {   // put element at tail
  if (!isFull()) {
    items[tail] = data;
    tail = (tail+1) % maxSize;
    filled++;
  } else {
    throw std::runtime_error("queue: full on put");
  }
}
```

10.2. Container Classes

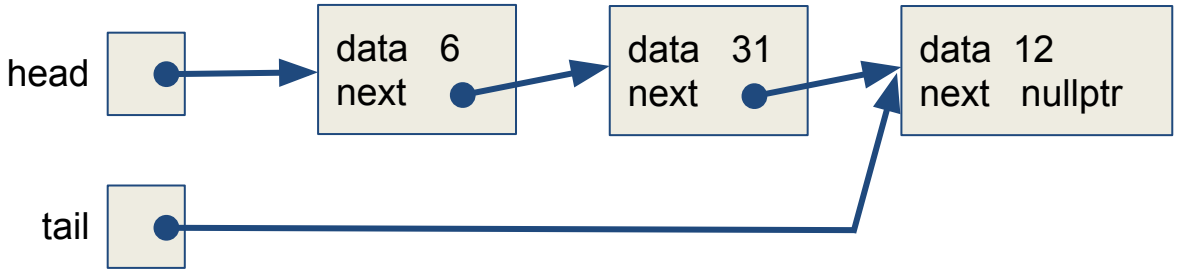
Illustration of a container: A queue of predefined size for integers

```
// gets element at the head of the queue, for example get() updates:
// items:  [  ][  ][  ][  ][  ][  ][  ] … [  ][  ]
//              head→           tail
int Queue::get() {  // get and remove element from head
  int retval;
  if (!isEmpty()) {
    retval = items[head];
    head = (head+1) % maxSize;
    filled--;
  } else {
    throw std::runtime_error("queue: empty on get");
  }
  return retval;
}
```

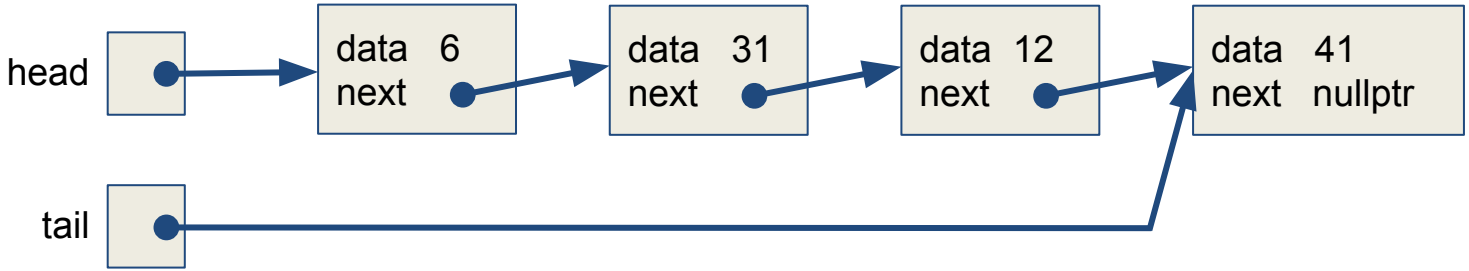
10.2. Container Classes

Illustration of a container: A queue of unlimited size for **`QueueElement`**s

```
UQueue q;
q.put(6);
q.put(31);
q.put(12);
```



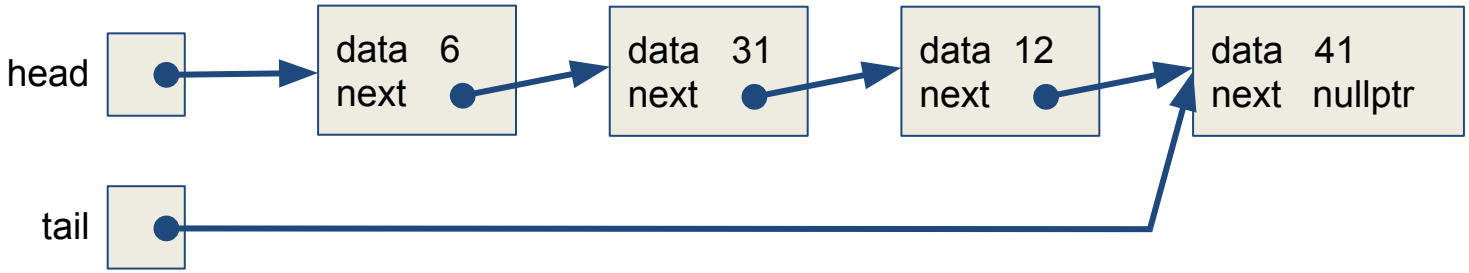

```
q.put(41);
```



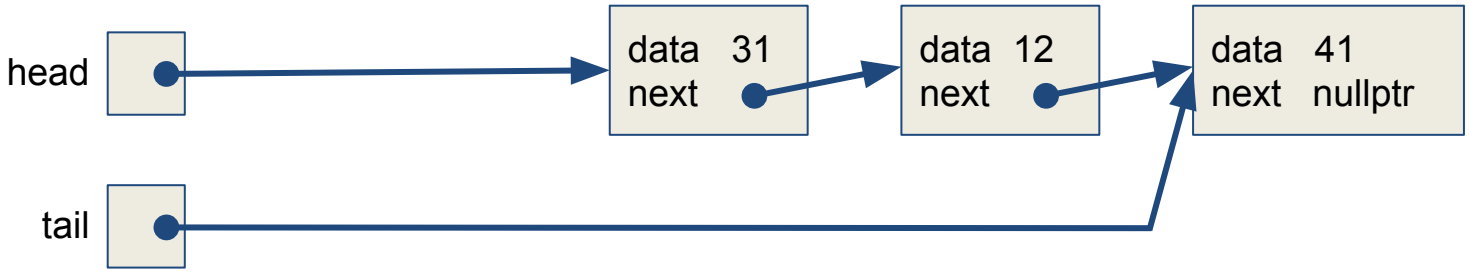
10.2. Container Classes

Illustration of a container: A queue of unlimited size for **`QueueElement`**s

```
q.put(41);
```




```
int v;
v = q.get();
```



10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
class QueueElement;  // declaration of the element class

class UQueue {  // Class for an unlimited queue of QueueElements
 public:
  UQueue() { head = tail = nullptr; }
  ~UQueue() { clear(); };
  void put(int data);
  int get();
  bool isEmpty() const;
  void clear();
 private:
  QueueElement * head;  // pointer to element to put
  QueueElement * tail;  // pointer to element to get from
};
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
class QueueElement {  // element class, hidden from users
 public:
  QueueElement(int data) : data(data) , next(nullptr) {}
  int data;
  QueueElement * next;
};

void UQueue::clear() {  // iteratively clear the queue of all elements
  QueueElement * elem, * elem_next;
  for (elem = head; elem != nullptr; elem = elem_next) {
    elem_next = elem->next;
    delete elem;
  }
  head = tail = nullptr;
}
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
bool UQueue::isEmpty() const {  // check whether the queue is empty
  return head == nullptr;
}

void UQueue::put(int data) {  // put in new data element at tail
  QueueElement * node = new QueueElement(data);
  if ( isEmpty() ) {
    head = tail = node;
  } else {
    tail = tail->next = node;  // tail is guaranteed to be valid
  }
}
```

10.2. Container Classes

Illustration of a container: A queue of unlimited size for **QueueElement**s

```
int UQueue::get() {  // get and remove element from head
  int retVal;
  if ( !isEmpty() ) {
    retVal = head->data;
    QueueElement * second = head->next;
    delete head;
    head = second;
  } else {
    throw std::runtime_error("uqueue: empty on get");
  }
  return retVal;
}
```

10.2. Container Classes: Sequence Containers

Sequence Containers maintain the order of elements:

- **std::vector** for dynamic arrays. Use this when you need fast access of any element, and efficient append operations at end of the vector.
- **std::deque** for double-ended queues. Use this when you need fast inserts and removals at both ends of the queue.
- **std::list** for doubly-linked lists. Use this for fast insertions and deletions anywhere in the list.
- **std::forward_list** for singly-linked list. Use this for light-weight lists that only need forward iterations.
- **std::array** Fixed-size array on the stack (no heap). Use this when the size is known at compile time.

10.2. Container Classes: Associative Containers

Associative Containers maintain sorted key-based element collections:

- **std::set** for sorted, unique elements. Use this when elements are unique and must remain ordered.
- **std::multiset** for any sorted elements. Use this when elements can be duplicates and need to be ordered.
- **std::map** for key-value pairs that are sorted by unique keys. Use this when it makes sense to have each element linked to its key.
- **std::multimap** for key-value pairs that are sorted by keys. Use this when elements can be grouped under the same keys.

10.2. Container Classes: Unordered Associative Containers

Unordered Associative Containers maintain *hash-based* collections:

- **std::unordered_set** for unique elements. Use this when elements are unique and order is not needed.
- **std::unordered_multiset** for any elements. Use this when elements can be duplicates and need to be ordered.
- **std::unordered_map** for key-value pairs, hash-based, with unique keys. Use this when it makes sense to have each element linked to its key and they do not need to be sorted.
- **std::unordered_multimap** for key-value pairs, hash-based. Use this when elements do not need to be ordered and can be grouped under the same keys.

10.2. Container Classes: Container Adapters

Container Adapters are wrappers for other containers to limit their interfaces.

- **std::stack** for last-in, first-out (LIFO) access of typically **deques**
- **std::queue** for first-in, first-out (FIFO) access of typically **deques**
- **std::priority_queue** for elements that are kept in sorted priority order (typically using **`vector`** with heap)

10.2. Container Classes

Example of a container: **`std::multiset`**

```
#include <iostream>
#include <set>
int main() {
  std::multiset<double> grades;  // the multiset to record grades
  // inserting student grades in any order, with duplicates:
  grades.insert(1.7); grades.insert(2.3);  grades.insert(1.0);
  grades.insert(2.3); grades.insert(5.0);  grades.insert(2.3);
  grades.insert(4.0); grades.insert(1.0);  grades.insert(5.0);
  std::cout << "all grades sorted:\n";
  for (auto grade : grades) std::cout << grade << ' ';
  // count how often certain grades occured:
  std::cout << "\ngrade 5.0 was given " << grades.count(5.0) << " times.";
  std::cout << "\ngrade 2.3 was given " << grades.count(2.3) << " times.";
  // erase all 5.0 grades:
  grades.erase(5.0);
  std::cout << "\nall passing grades:\n";
  for (auto grade : grades) std::cout << grade << ' ';
  std::cout << '\n';
}
```

10.2. Container Classes

Example of a container: **`std::priority_queue`**

```
#include <iostream>
int main() {
  // a package has a weight and destination, deliveries are a priority queue
  using Package = std::tuple<double, std::string>;
  // Comparator for max-heap (heavier packages have higher priority):
  auto cmp = [](const Package & a, const Package & b) {
      return std::get<0>(a) < std::get<0>(b);  // index-based access
    };
  std::priority_queue<Package, std::vector<Package>, decltype(cmp)> deliveries(cmp);
  // add deliveries, duplicates possible:
  deliveries.push({2.5, "Berlin"});  deliveries.push({1.2, "Siegen"});
  deliveries.push({3.1, "Berlin"});  deliveries.push({2.5, "Hamburg"});
  deliveries.push({4.1, "Cologne"}); deliveries.push({2.5, "Berlin"});
  // emptying the priority queue:
  std::cout << "handling all deliveries (packages sorted by weight):\n";
  while (!deliveries.empty()) {
    auto [weight, city] = deliveries.top();  // structured binding
    std::cout << weight << " kg - " << city << '\n';
    deliveries.pop();
  }
}
```

10.2. Container Classes

comparison.cpp

```
// create three containers, unordered_set, vector, and forward_list:
std::unordered_set<int> u_set(data.begin(), data.end());
std::vector<int> vec(data.begin(), data.end());
std::forward_list<int> f_list(data.begin(), data.end());

std::cout << "Timing element lookup:\n";
benchmark("unordered_set", u_set, queries, [](const auto & c, int x) {
        return c.find(x) != c.end();
      });
benchmark("vector", vec, queries, [](const auto & c, int x) {
        return std::find(c.begin(), c.end(), x) != c.end();
      });
benchmark("forward_list", f_list, queries, [](const auto & c, int x) {
        return std::find(c.begin(), c.end(), x) != c.end();
      });
```