CS100
Introduction to Programming

Tutorial 8: containers & threading

Part 1
std::unordered_map

Associative container

- Stores elements as as key-value pairs
- Similar to std::map, every element needs to have unique key

Search, insertion, and removal all have approximately **constant-time complexity**!
(remember that map std::has complexity of O(log(n))

```
name                    employee
string                  Employee

map<string, Employee *> employees;
```

Associative container

- Stores elements as as key-value pairs
- Similar to std::map, every element needs to have unique key

Search, insertion, and removal all have approximately **constant-time complexity**!
(remember that std::map has complexity of O(n)

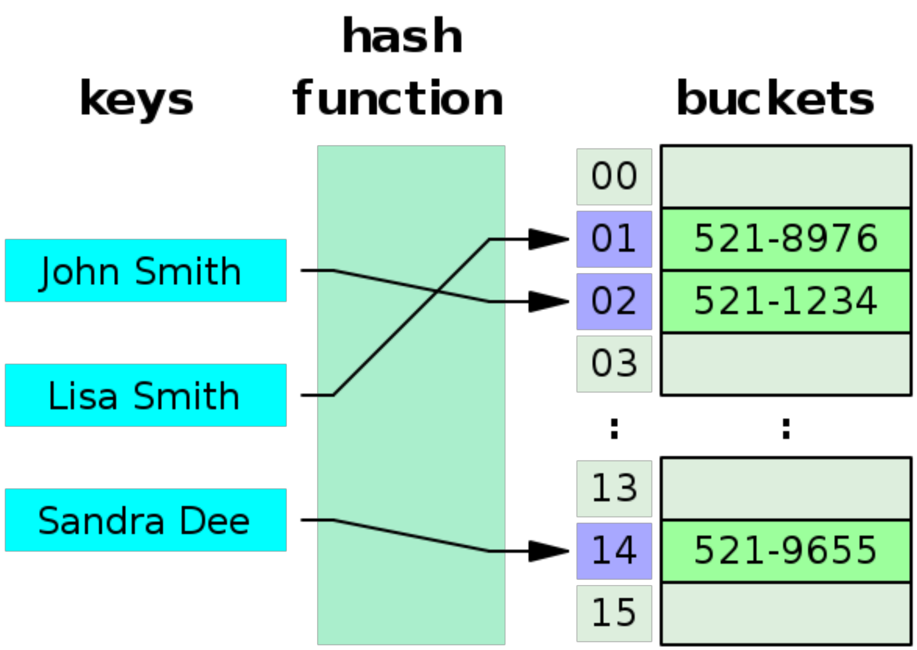
```
name            employee
string          Employee

unordered_map<string, Employee *> employees;
```

How does it work?

- Internally, the elements are organized into buckets
- Which bucket an element is placed into depends on its key
  - From an arbitrary key-type, we derive a bucket-index
  - The bucket-index is called a hash
  - A hash-function is a function that maps an arbitrary input type to a defined type with defined range
- This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into

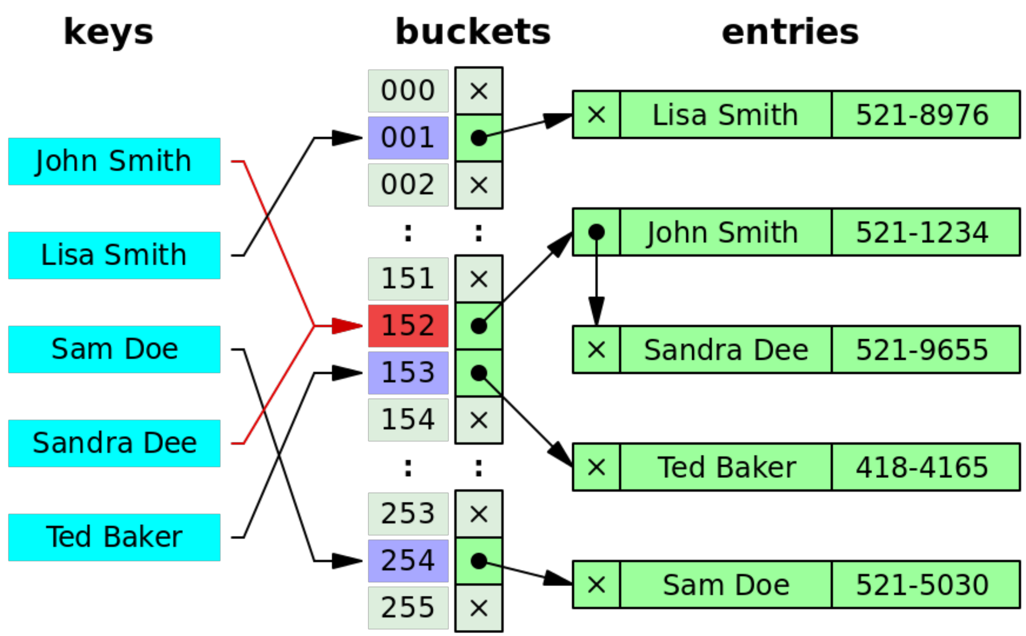
How does it work?

- Hash table
- https://en.wikipedia.org/wiki/Hash_table
- Ideally: Every key leads to an individual bucket



How does it work?

- Hash table
- https://en.wikipedia.org/wiki/Hash_table
- In practice: Limited number of bucket & extra collision resolving (e.g. chaining)



How does it work?

- Properties:

Algorithm	Average	Worst case
Space	$O(n)$[1]	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

- How does the space property come across?
  - Only a table with (initially NULL) bucket pointers is installed
  - This table has a constant size equal to the range of the hash
  - The actual buckets are growing linearly with the actual number of elements in the list

Exercise

- Create a main function in which you
  - Fill a std::map with key value pairs of the form

    `std::map<int,std::string>`

  - Notes:
    - The int can simply increase linearly for every element
    - The string can be same, dummy string everytime

- **Task 1:**
  - Add 10000 elements, then measure the time of adding 100 more elements

Exercise

- **Task 2:**
  - Do the same for an unordered map
  - What do you observe?

- **Task 3:**
  - Now increase the initial size of the container by a factor of 10.
  - What do you observe?

Part 2
MyList

Implement your own template list container

- Requirements:
  - Define a suitable type for the (doubly!) linked elements in the list
  - Implement a bidirectional STL-style iterator that iterates through the linked elements
  - Implement your own version of a template list
  - Required interface functions:

```
iterator begin();
iterator end();
bool empty();
int size();
void erase(iterator it);
void insert(iterator it, const T& x);
void push_back(const T& x);
void clear();
```

Test MyList with the below code

- Instantiate new list and fill with elements

```
#include "MyList.hpp"
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  //instantiate one of my new list object and push_back two elements
  MyList<double> myList;
  myList.push_back(0.0);
  myList.push_back(1.0);

  …
}
```

Test MyList with the below code

- Iterate through the elements and print them out

```
#include "MyList.hpp"
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  …

  //use the iterator to loop through the elements
  //and print them in the console
  MyList<double>::iterator it = myList.begin();
  while( it != myList.end() ) {
    std::cout << *it << "\n";
    it++;
  }

  …
}
```

Test MyList with the below code

- Copy the content over to an std::vector using std::copy!

```
#include "MyList.hpp”
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  …

  //create a standard vector and copy all the elements over
  //(exploit the fact that our iterators are also STL iterators!)
  std::vector<double> vec;
  std::copy( myList.begin(), myList.end(), std::back_inserter(vec) );

  …
}
```

Test MyList with the below code

- Verify that the content has been correctly copied!

```
#include "MyList.hpp"
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  …

  //verify that the content has been correctly copied
  //over to the vector
  for( int i = 0; i < vec.size(); i++ )
    std::cout << vec[i] << "\n";
  return 0;
}
```

Part 3

Implement two threads that communicate with each other

Synchronization between threads

- Apart from just protecting data, sometimes we may wish for one thread to wait until another thread has something done
- In C++:
  - Conditional variables
  - Futures

std::condition_variable

- A synchronization primitive that can be used to block a thread or multiple threads at the same time, until
  - A notification is received from another thread
  - A time-out expires

std::condition_variable

- A thread that intends to wait on std::condition_variable has to acquire a std::unique_lock first
- The wait operations atomically release the mutex and suspend the execution of the thread
- When the condition variable is notified, the thread is awakened, and the mutex is reacquired

Example

Mutex to protect resource

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Queue used to pass data

Example

When data is ready, thread locks mutex, pushes data, and calls notify_one()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

notify_one() notifies
The waiting thread

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Receiver thread puts itself into waiting mode through this call (if queue is empty). It will also release the lock

Example

It also passes a wake condition that will be checked upon notify_all()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```cpp
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

The mutex will be automatically locked once the wait terminates

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Lock only for as long as necessary

Playing with threads

- Implement two threads that are playing Ping-Pong
  - The first thread is configured to be the right-side player, and he says Ping if the ball is on the right side
  - The second thread is configured to be the left-side player, and he says Pong if the ball is on the left side
  - Realize the problem with a single function definition
  - Realize your implementation with a condition variable that sets the threads to sleep while they are waiting for the ball