C++ Priority Queue With Comparator

`priority_queue` is categorized as a STL container adaptor. It is like a queue that keeps its element in sorted order. Instead of a strict FIFO ordering, the element at the head of the queue at any given time is the one with the highest priority.

The template class definition of `priority_queue` is as follow

template definition

```
template <
   class Type,
   class Container=vector<Type>,
   class Compare=less<typename Container::value_type> >
class priority_queue
```

A user-provided compare can be supplied to change the ordering, e.g. using `std::greater` would cause the smallest element to appear as the top(). We also can create custom comparator for our need.

Many samples available on net about `priority_queue` with default compare parameter. In this article let’s create samples by specifying the compare parameter template.

priority_queue with std::greater

```
//helper function displays sorted data
template<class T>
void printQueue(T& q)
{
    while (!q.empty())
    {
        cout << q.top() << endl;
        q.pop();
    }
}

void SamplePriorityQueue()
{
    std::priority_queue<int, std::vector<int>, std::greater<int> > q;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);

    printQueue(q);
}
```

The code above uses `std::greater` as a compare parameter template.

output

```
0
1
2
3
4
5
6
7
8
9
```

Beside the `std::less` or `std::greater`, we can create our custom comparator with lamda or custom class or struct.

lamda as compare parameter

```cpp
void SamplePriorityQueueWithLamda()
{
    // using lambda to compare elements.
    auto compare = [](int lhs, int rhs)
                {
                    return lhs < rhs;
                };

    std::priority_queue<int, std::vector<int>, decltype(compare)> q(compare);

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);


    printQueue(q);
}
```

To use the custom comparator, we just need to pass it as the third parameter of `priority_queue` template

custom comparator

```cpp
struct CustomCompare
{
    bool operator()(const int& lhs, const int& rhs)
    {
        return lhs < rhs;
    }
};
```

sample with custom comparator

```cpp
void SamplePriorityQueueWithCustomComparator()
{
    priority_queue<int,vector<int>, CustomCompare > pq;

    pq.push(3);
    pq.push(5);
    pq.push(1);
    pq.push(8);

    printQueue(pq);
}
```

The data stored in `priority_queue` is not limited to basic data type. We can store object in it. Let's create a sample of it. Let's say we have a `Person` class.

Person.hpp

```
#ifndef Person_hpp
#define Person_hpp

#include <stdio.h>
#include <string>

using namespace std;

class Person
{
public:
    Person();
    Person(string name, int age);
    virtual ~Person();

    string getName() const;
    int getAge() const;

    friend bool operator < (const Person& lhs, const Person& rhs);
    friend bool operator > (const Person& lhs, const Person& rhs);

private:
    string name;
    int age;
};

#endif /* Person_hpp */
```

Person.cpp

```
#include "Person.hpp"

bool operator < (const Person& lhs, const Person& rhs)
{
    return lhs.getAge() < rhs.getAge();
}
```

```

bool operator > (const Person& lhs, const Person& rhs)
{
    return lhs.getAge() > rhs.getAge();
}

Person::Person()
{
}

Person::Person(string name, int age):name(name), age(age)
{
}

Person::~Person()
{
}

string Person::getName() const
{
    return name;
}

int Person::getAge() const
{
    return age;
}
```

On the `Person` class, we have friend overloading methods, right angle bracket and left angle bracket. The methods act as comparation operator. The operator overloading is needed if we want to use `std::less` or `std::greater`.

sample priority_queue stores object

```
void SamplePriorityQueueStoreObject()
{
    vector<Person> personVector =
    {
        Person("Person 1", 25),
        Person("Person 2", 17),
        Person("Person 3", 35),
        Person("Person 4", 7),
        Person("Person 5", 50)
    };

    cout << "======== Less Priority Queue ======= " << endl;

    priority_queue<Person, vector<Person>, less<vector<Person>::value_type>> pqueue_

    //fill pqueue_less
    for (auto it = personVector.cbegin(); it!=personVector.cend(); it++)
    {
        pqueue_less.push(*it);
    }

    //iterate,display and pop
    while (!pqueue_less.empty())
    {
        Person value = pqueue_less.top();
        cout << value.getName() << " : " << value.getAge() << endl;

        pqueue_less.pop();
    }


    cout << endl << endl;

    cout << "======== Greater Priority Queue ======= " << endl;

    priority_queue<Person, vector<Person>, greater<vector<Person>::value_type>> pque
    //fill pqueue_greater
    for (auto it = personVector.cbegin(); it!=personVector.cend(); it++)
    {
        pqueue_greater.push(*it);
    }

    //iterate,display and pop
    while (!pqueue_greater.empty())
    {
        Person value = pqueue_greater.top();
        cout << value.getName() << " : " << value.getAge() << endl;

        pqueue_greater.pop();
    }
}
```

References

1. http://en.cppreference.com/w/cpp/container/priority_queue
2. https://support.microsoft.com/en-us/kb/837697
3. http://www.wrox.com/WileyCDA/WroxTitle/Professional-C-2nd-Edition.productCd-0470932449.html

Posted by neutro • Dec 29th, 2016 2:52 pm • cpp

Print

« Reset Git Credential on OS X Keychain C++ Set with Custom Comparator »