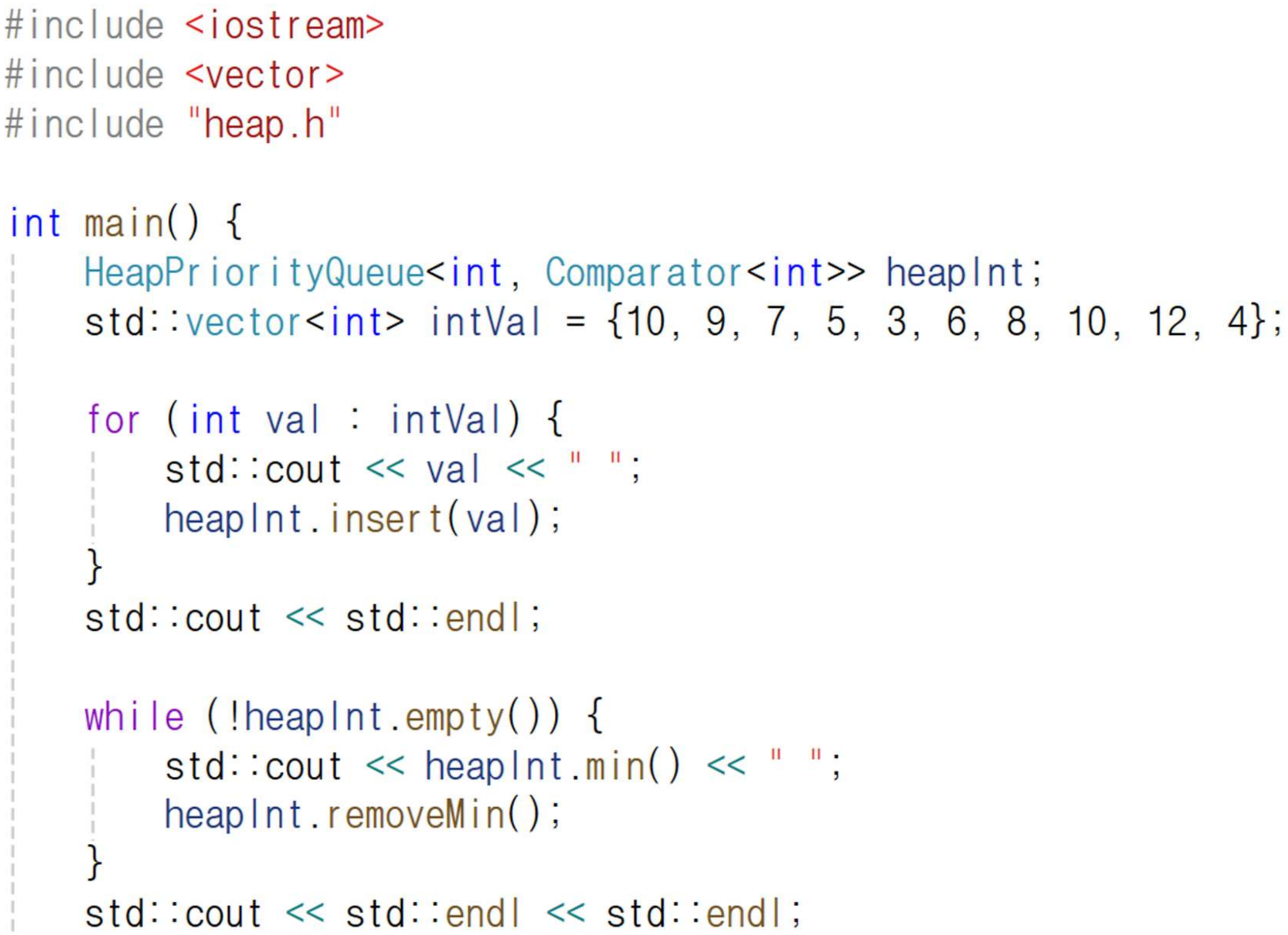
Homework #2: Heap-Based Priority Queue

- Deadline: Jun. 3, 11:59 PM
- Recommended: Use Visual Studio 2022 or Visual Studio 2019

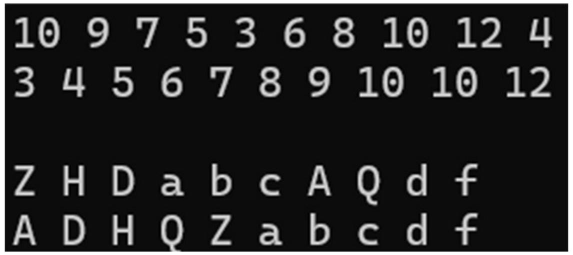
- Submission Requirements: 김철수_20251111.zip
  1. Report: PDF format (2~3 pages, plus appendices for code & comments only)
  2. Source Code: Files in .cpp and .h format
  3. Executable: Compiled .exe file

- Grading Criteria:
  - Code / Executable: 50%
  - Report: 50%

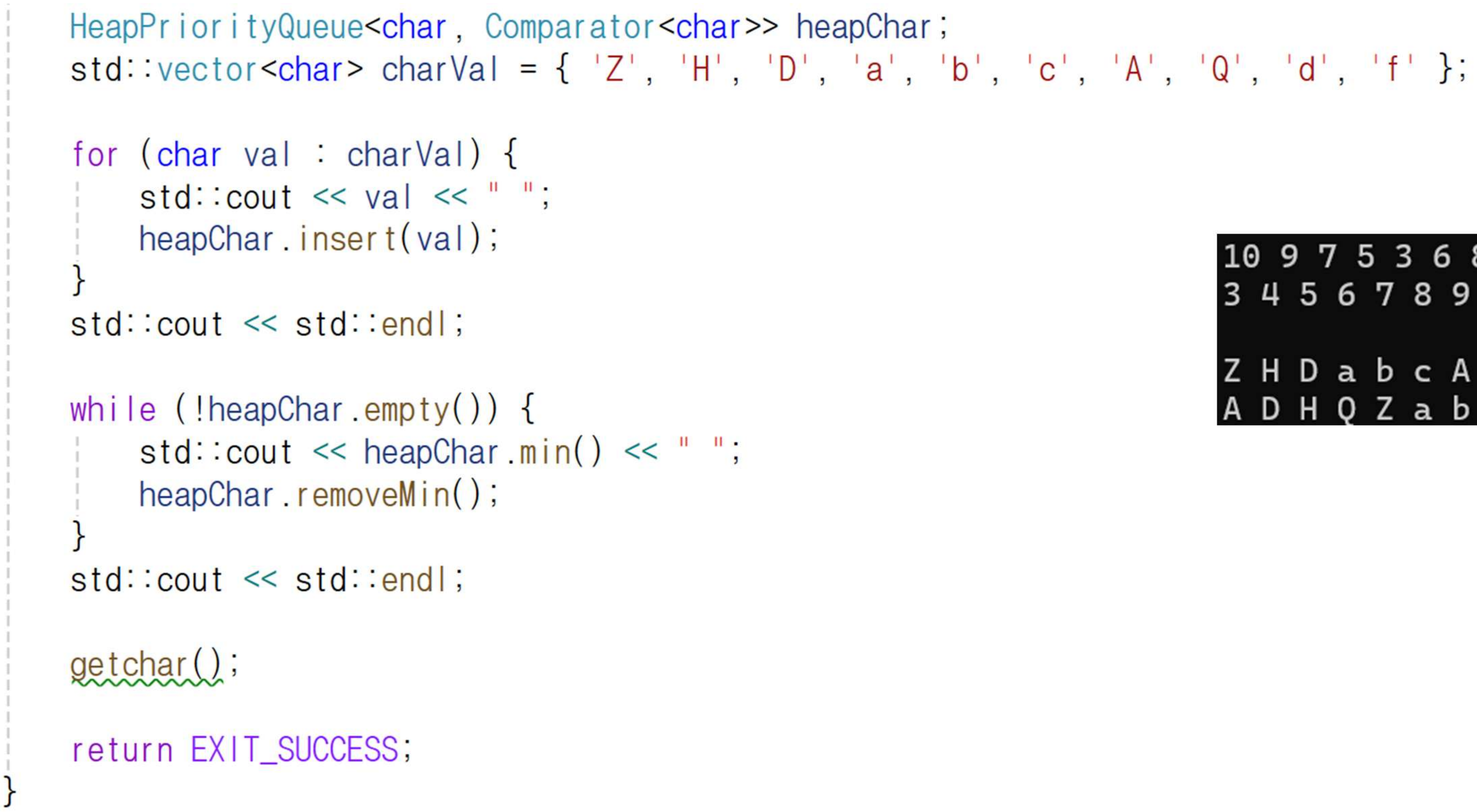
main.cpp



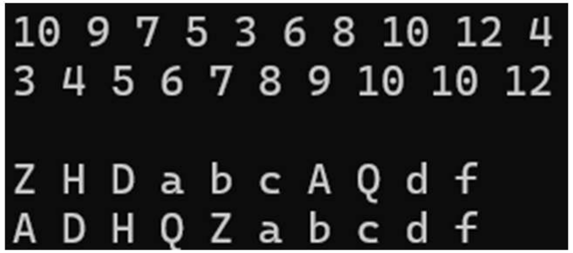
```cpp
#include <iostream>
#include <vector>
#include "heap.h"

int main() {
    HeapPriorityQueue<int, Comparator<int>> heapInt;
    std::vector<int> intVal = {10, 9, 7, 5, 3, 6, 8, 10, 12, 4};

    for (int val : intVal) {
        std::cout << val << " ";
        heapInt.insert(val);
    }
    std::cout << std::endl;

    while (!heapInt.empty()) {
        std::cout << heapInt.min() << " ";
        heapInt.removeMin();
    }
    std::cout << std::endl << std::endl;
```



```
10 9 7 5 3 6 8 10 12 4
3 4 5 6 7 8 9 10 10 12

Z H D a b c A Q d f
A D H Q Z a b c d f
```

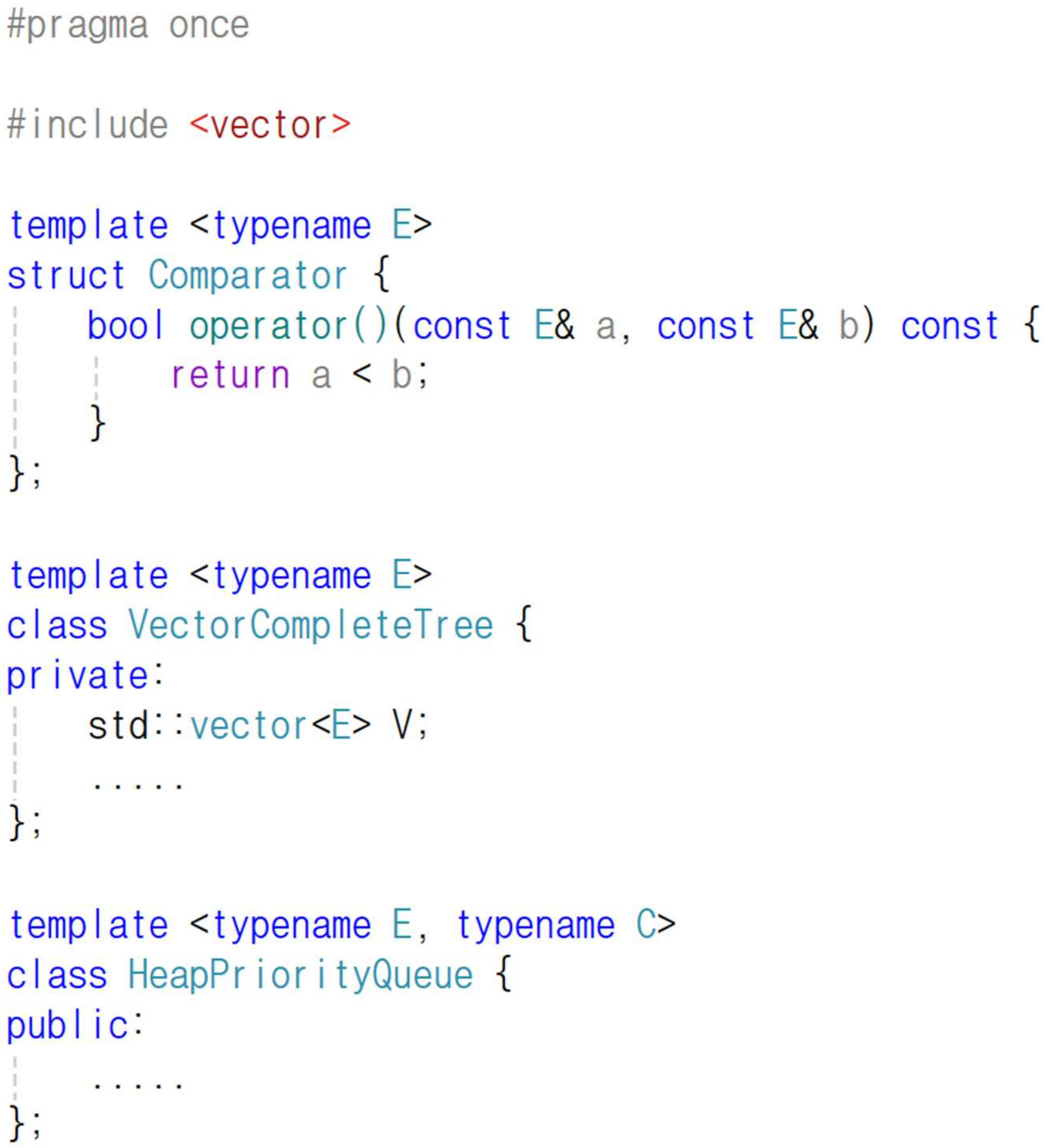


```
    HeapPriorityQueue<char, Comparator<char>> heapChar;
    std::vector<char> charVal = { 'Z', 'H', 'D', 'a', 'b', 'c', 'A', 'Q', 'd', 'f' };

    for (char val : charVal) {
        std::cout << val << " ";
        heapChar.insert(val);
    }
    std::cout << std::endl;

    while (!heapChar.empty()) {
        std::cout << heapChar.min() << " ";
        heapChar.removeMin();
    }
    std::cout << std::endl;

    getchar();

    return EXIT_SUCCESS;
}
```

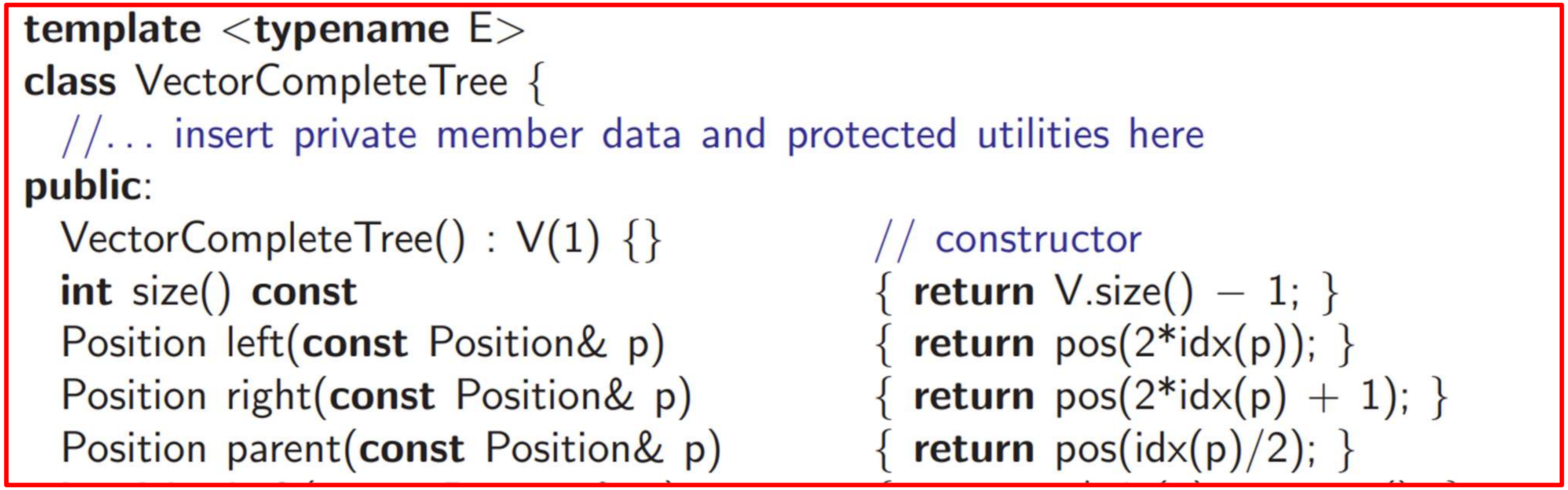


```
10 9 7 5 3 6 8 10 12 4
3 4 5 6 7 8 9 10 10 12

Z H D a b c A Q d f
A D H Q Z a b c d f
```

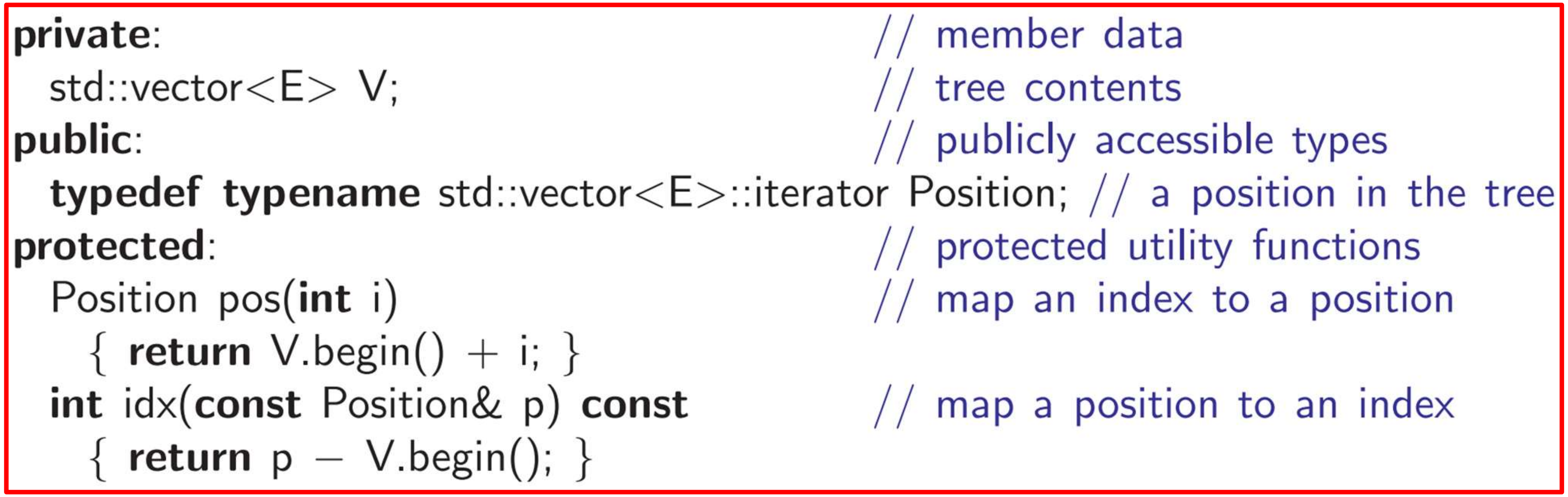
heap.h



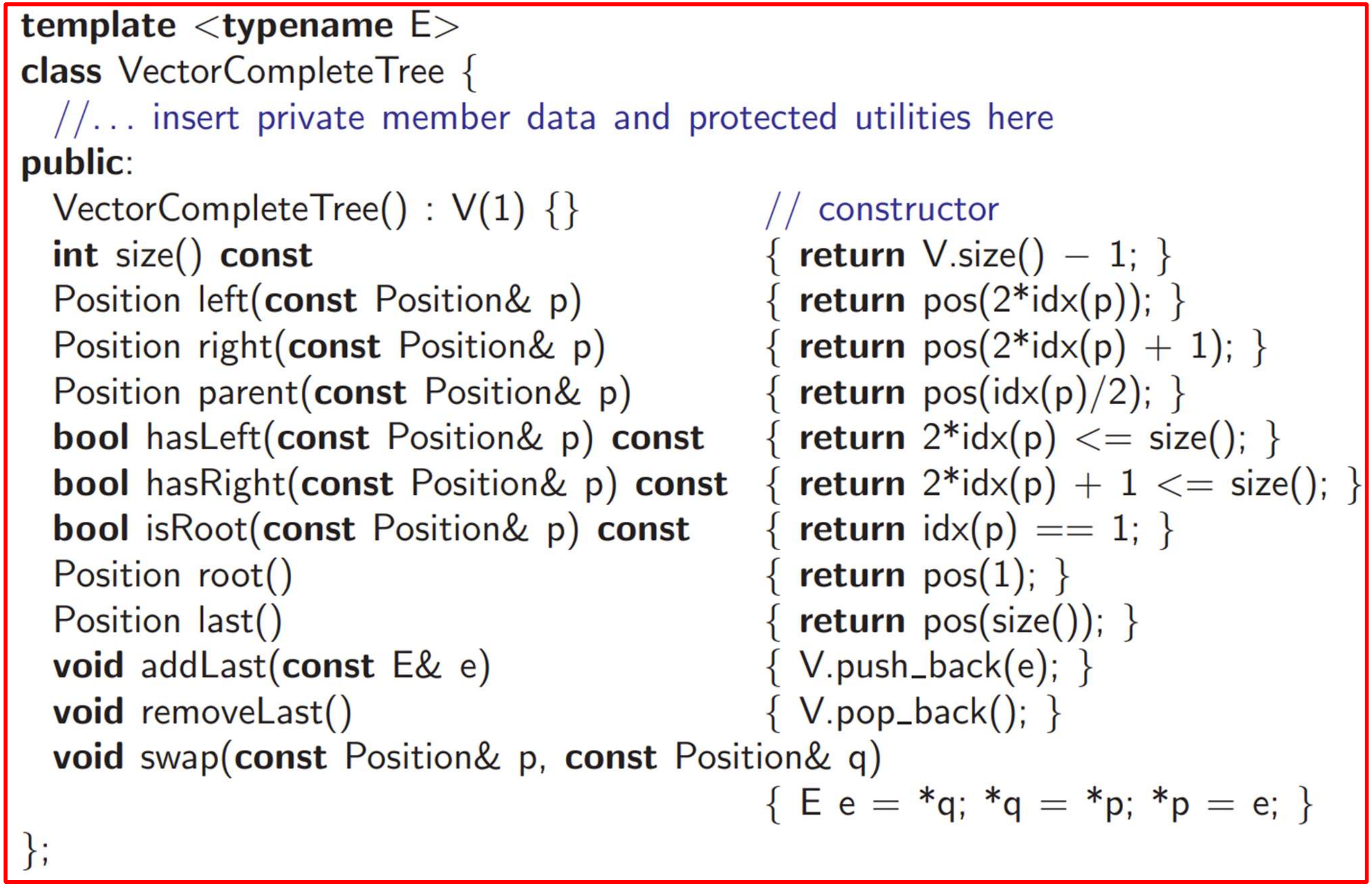
```
#pragma once

#include <vector>

template <typename E>
struct Comparator {
    bool operator()(const E& a, const E& b) const {
        return a < b;
    }
};

template <typename E>
class VectorCompleteTree {
private:
    std::vector<E> V;
    .....
};

template <typename E, typename C>
class HeapPriorityQueue {
public:
    .....
};
```



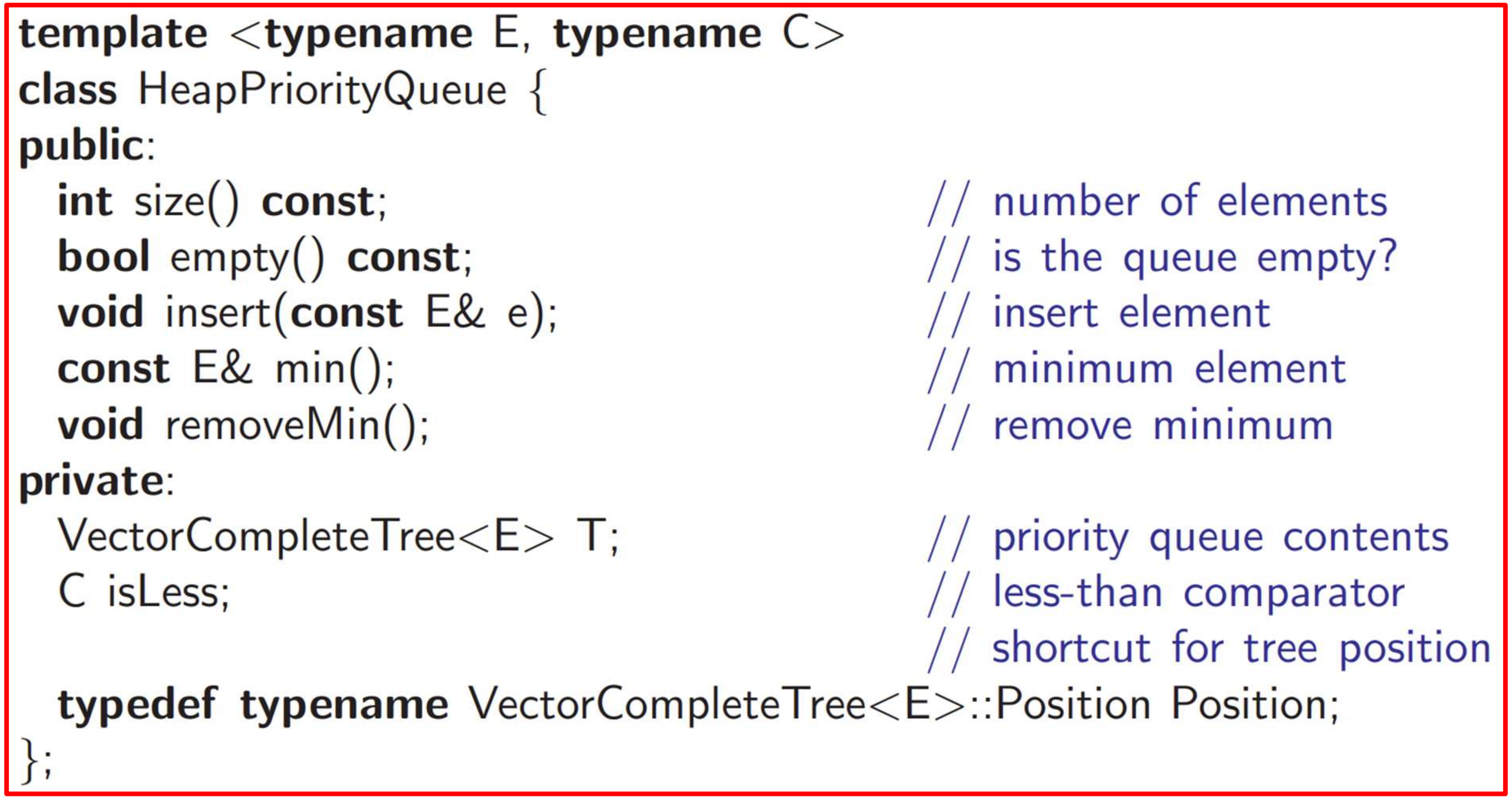
```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}                      // constructor
  int size() const                                    { return V.size() - 1; }
  Position left(const Position& p)                    { return pos(2*idx(p)); }
  Position right(const Position& p)                   { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)                  { return pos(idx(p)/2); }
```



```
private:                                              // member data
  std::vector<E> V;                                   // tree contents
public:                                               // publicly accessible types
  typedef typename std::vector<E>::iterator Position; // a position in the tree
protected:                                            // protected utility functions
  Position pos(int i)                                 // map an index to a position
    { return V.begin() + i; }
  int idx(const Position& p) const                    // map a position to an index
    { return p - V.begin(); }
```

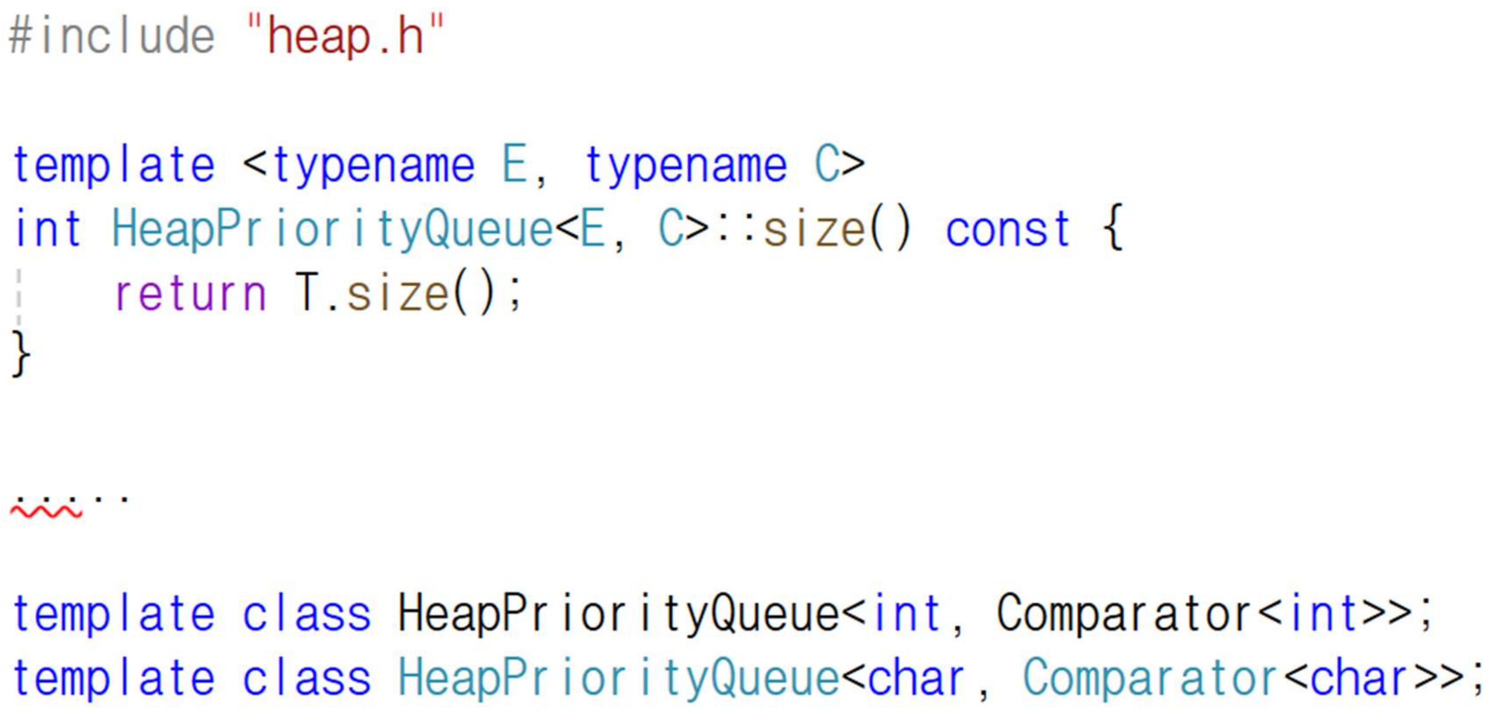


```
template <typename E>
class VectorCompleteTree {
  //... insert private member data and protected utilities here
public:
  VectorCompleteTree() : V(1) {}                     // constructor
  int size() const                                   { return V.size() − 1; }
  Position left(const Position& p)                   { return pos(2*idx(p)); }
  Position right(const Position& p)                  { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)                 { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const              { return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const             { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const               { return idx(p) == 1; }
  Position root()                                    { return pos(1); }
  Position last()                                    { return pos(size()); }
  void addLast(const E& e)                           { V.push_back(e); }
  void removeLast()                                  { V.pop_back(); }
  void swap(const Position& p, const Position& q)
                                                     { E e = *q; *q = *p; *p = e; }
};
```

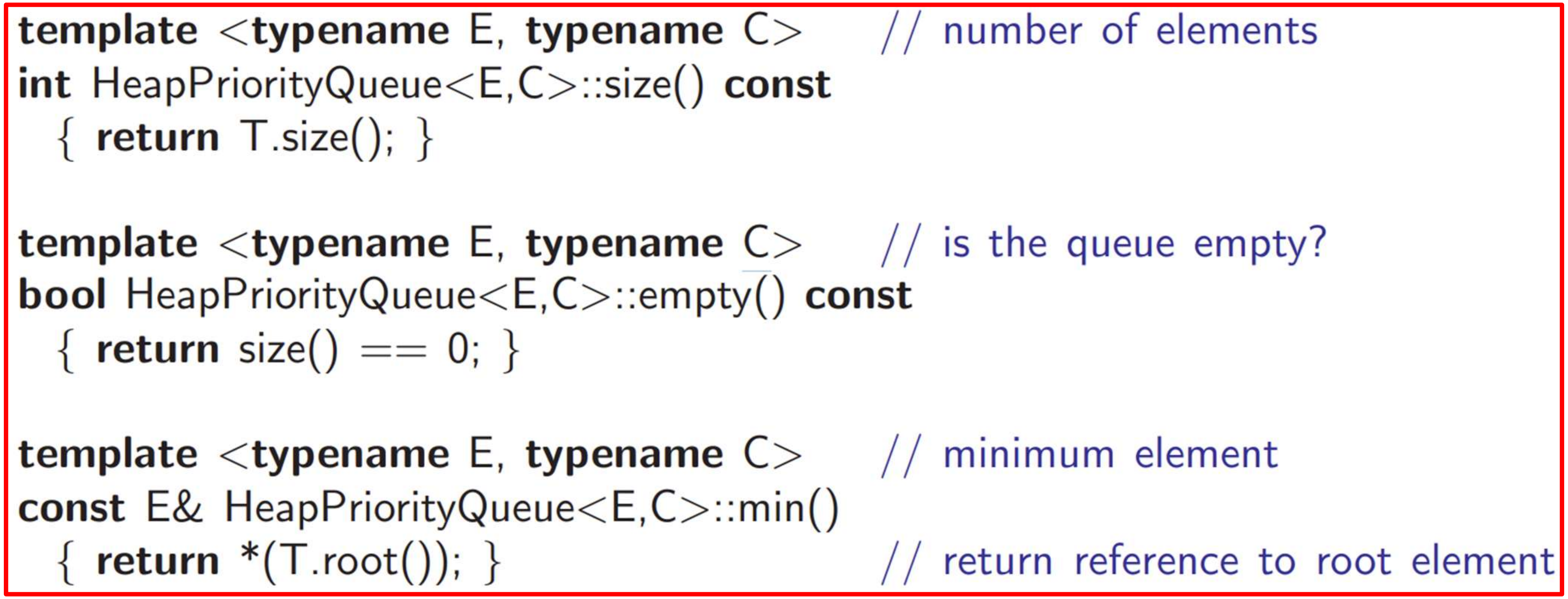


```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;                                       // number of elements
  bool empty() const;                                     // is the queue empty?
  void insert(const E& e);                                // insert element
  const E& min();                                         // minimum element
  void removeMin();                                       // remove minimum
private:
  VectorCompleteTree<E> T;                                // priority queue contents
  C isLess;                                               // less-than comparator
                                                          // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

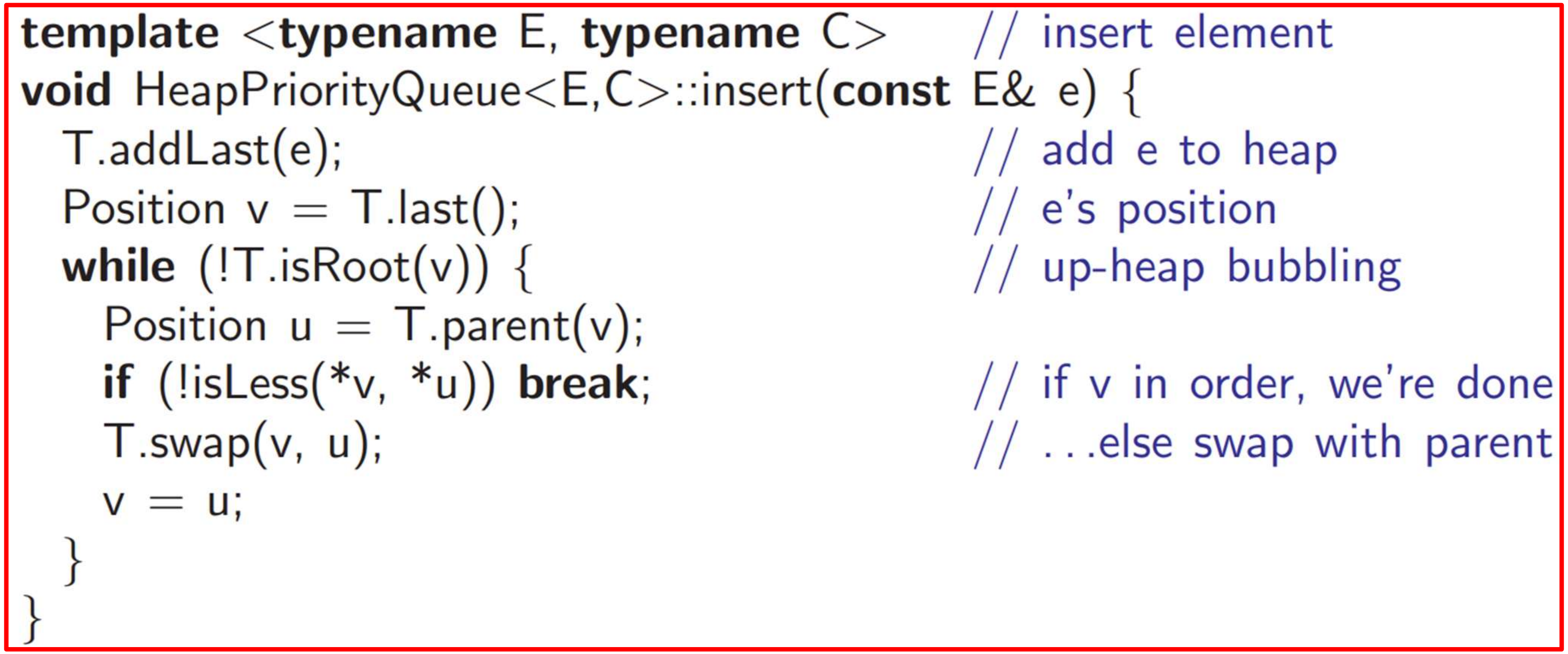
heap.cpp



```
#include "heap.h"

template <typename E, typename C>
int HeapPriorityQueue<E, C>::size() const {
    return T.size();
}


.....

template class HeapPriorityQueue<int, Comparator<int>>;
template class HeapPriorityQueue<char, Comparator<char>>;
```



```
template <typename E, typename C>          // number of elements
int HeapPriorityQueue<E,C>::size() const
  { return T.size(); }

template <typename E, typename C>          // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const
  { return size() == 0; }

template <typename E, typename C>          // minimum element
const E& HeapPriorityQueue<E,C>::min()
  { return *(T.root()); }                  // return reference to root element
```



```
template <typename E, typename C>             // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                               // add e to heap
  Position v = T.last();                      // e's position
  while (!T.isRoot(v)) {                      // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;               // if v in order, we're done
    T.swap(v, u);                             // ...else swap with parent
    v = u;
  }
}
```

```
template <typename E, typename C>                  // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)                                 // only one node?
    T.removeLast();                                // ...remove it
  else {
    Position u = T.root();                         // root position
    T.swap(u, T.last());                           // swap last with root
    T.removeLast();                                // ...and remove last
    while (T.hasLeft(u)) {                         // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
        v = T.right(u);                            // v is u's smaller child
      if (isLess(*v, *u)) {                        // is u out of order?
        T.swap(u, v);                              // ...then swap
        u = v;
      }
      else break;                                  // else we're done
    }
  }
}
```