GAME2001 Data Structures and Algorithms

Fall 2020

GEORGE BROWN COLLEGE

Week 13

Heaps

Abstract data types (ADTs)

- data structures that can be implemented using a variety of underlying data structures

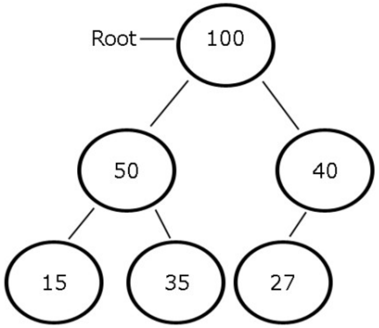
- Priority queue is an ADT
  - can be implemented using an array, link list, or another data structure such as a tree

- So are heaps

Heaps

- binary tree data structure that is not used for searching
- the node keys are always larger than their children nodes
- their children nodes are not in any particular order

An example of a heap as a tree…



Heaps

- Major characteristics:
  - A heap is a binary tree
  - A heap is a complete data structure
  - Every node in a heap is larger than or equal to its child nodes

Heaps

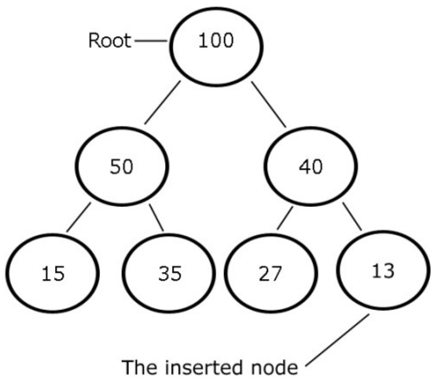
- A weakly ordered binary tree

- Search:
  - Every node would potentially be visited
  - Early exit from paths is possible
  - O(N)

Heaps

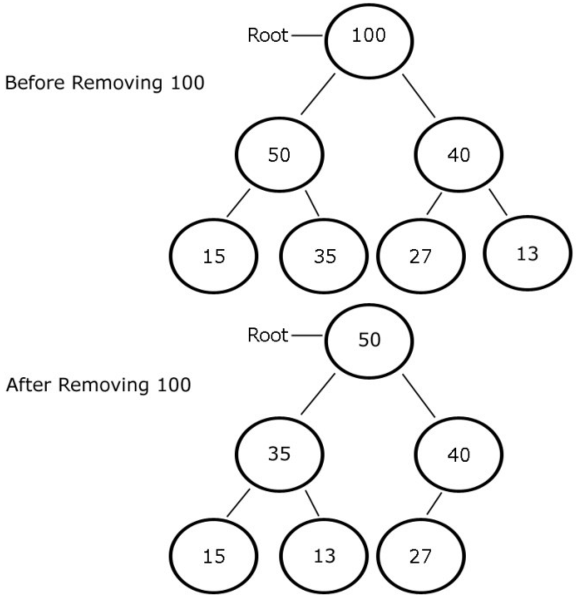
- Insertion:
  - Item is initially placed on the bottom of the list
  - moved up through the list until it finds an index where the element is smaller than its parent but larger than its children
  - O(logN)

An example of a heap as a tree...



Heaps

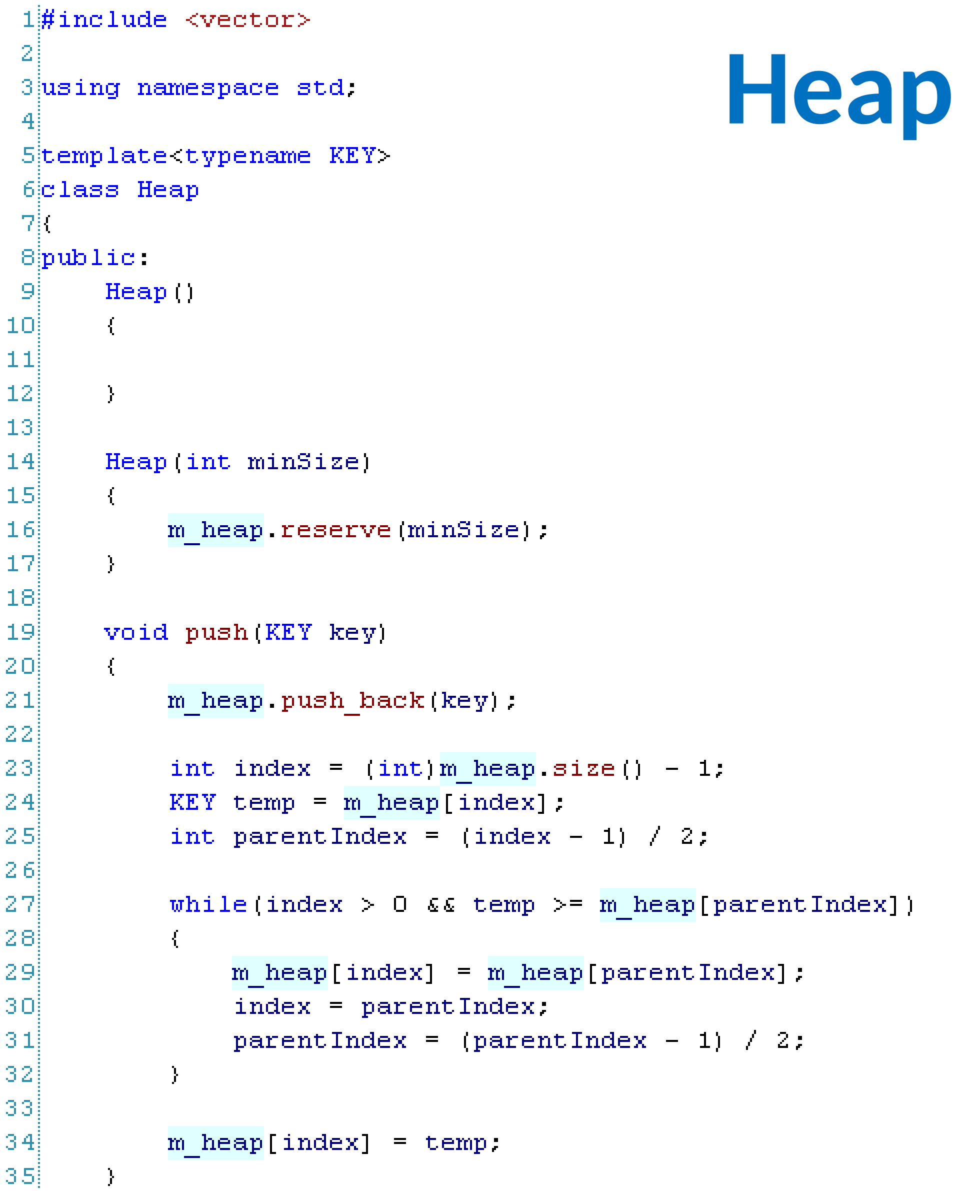
- Removal:
  - done from the top of the heap
  - take the last element in the array, place it at index 0, and move that element down to its correct position
  - O(logN)



Heaps

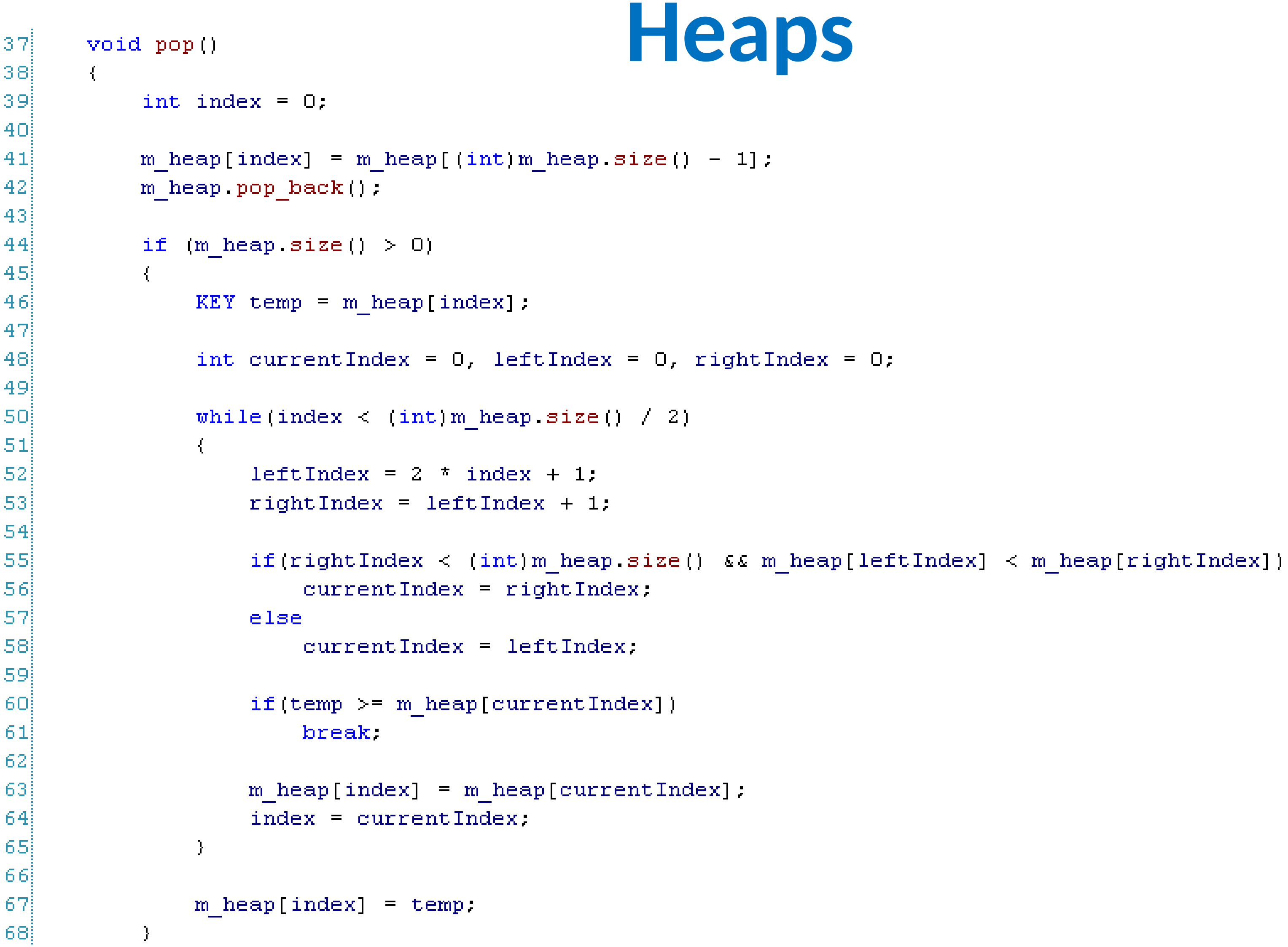
- Resizing:
  - Depends on the base implementation

Heaps



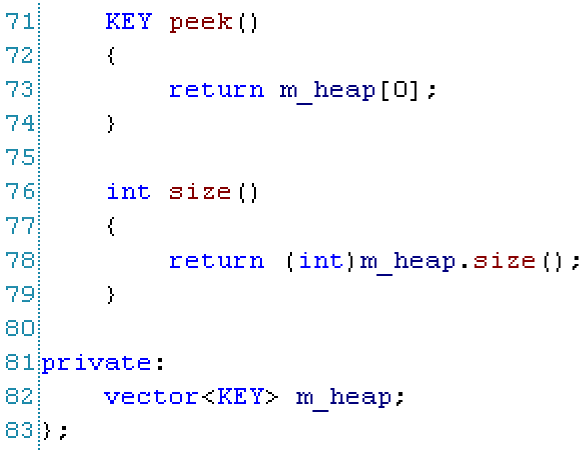
```cpp
#include <vector>

using namespace std;

template<typename KEY>
class Heap
{
public:
    Heap()
    {

    }

    Heap(int minSize)
    {
        m_heap.reserve(minSize);
    }

    void push(KEY key)
    {
        m_heap.push_back(key);

        int index = (int)m_heap.size() - 1;
        KEY temp = m_heap[index];
        int parentIndex = (index - 1) / 2;

        while(index > 0 && temp >= m_heap[parentIndex])
        {
            m_heap[index] = m_heap[parentIndex];
            index = parentIndex;
            parentIndex = (parentIndex - 1) / 2;
        }

        m_heap[index] = temp;
    }
```

Heaps



```
void pop()
{
    int index = 0;

    m_heap[index] = m_heap[(int)m_heap.size() - 1];
    m_heap.pop_back();

    if (m_heap.size() > 0)
    {
        KEY temp = m_heap[index];

        int currentIndex = 0, leftIndex = 0, rightIndex = 0;

        while(index < (int)m_heap.size() / 2)
        {
            leftIndex = 2 * index + 1;
            rightIndex = leftIndex + 1;

            if(rightIndex < (int)m_heap.size() && m_heap[leftIndex] < m_heap[rightIndex])
                currentIndex = rightIndex;
            else
                currentIndex = leftIndex;

            if(temp >= m_heap[currentIndex])
                break;

            m_heap[index] = m_heap[currentIndex];
            index = currentIndex;
        }

        m_heap[index] = temp;
    }
```

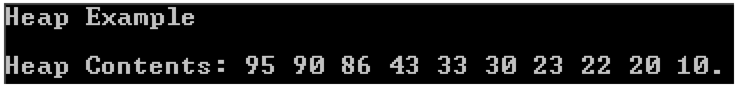
Heaps



```
    KEY peek()
    {
        return m_heap[0];
    }

    int size()
    {
        return (int)m_heap.size();
    }

private:
    vector<KEY> m_heap;
};
```

Heaps

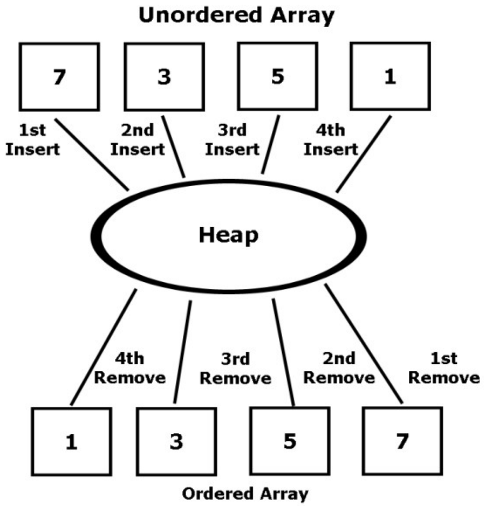
```cpp
#include <iostream>
#include "Heap.h"

using namespace std;

int main(int args, char **argc)
{
    cout << "Heap Example" << endl << endl;

    Heap<int> heap(10);

    heap.push(30);
    heap.push(33);
    heap.push(43);
    heap.push(23);
    heap.push(20);
    heap.push(10);
    heap.push(22);
    heap.push(90);
    heap.push(95);
    heap.push(86);

    cout << "Heap Contents:";

    while(heap.size() != 0)
    {
        cout << " " << heap.peek();
        heap.pop();
    }

    cout << "." << endl << endl;

    return 1;
}
```



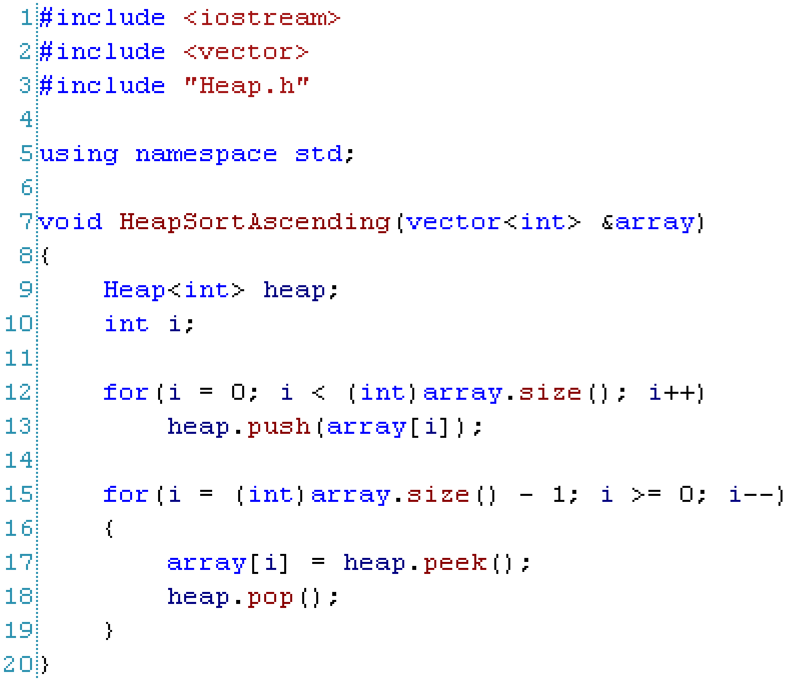
```
Heap Example

Heap Contents: 95 90 86 43 33 30 23 22 20 10.
```

Heap Sort

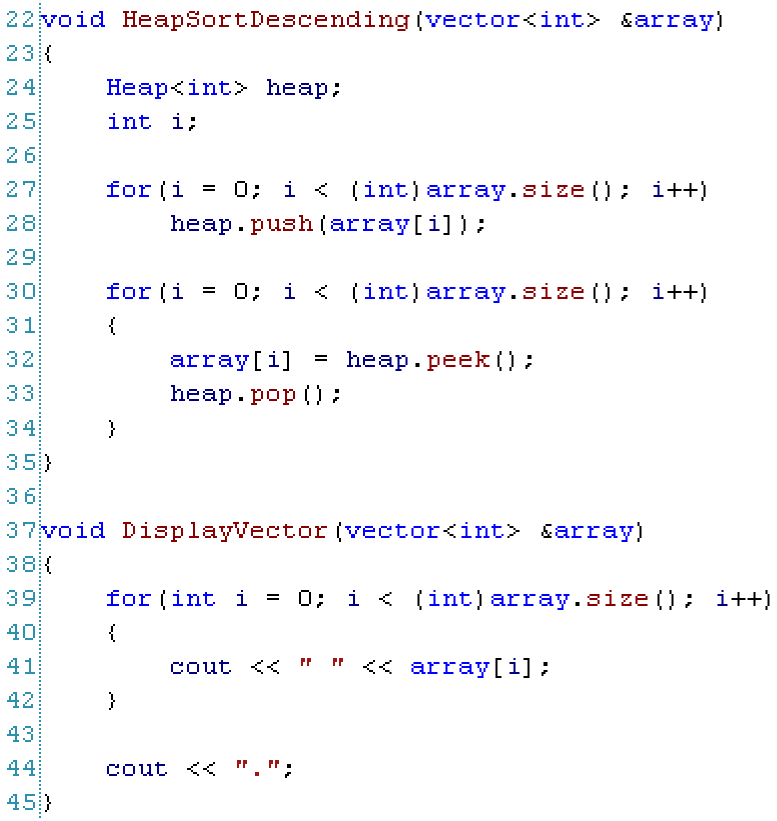
- an algorithm that uses a heap to sort the elements of another data structure
- done by inserting all the elements of an unordered data structure into a heap
- then moving the elements from the heap one by one back into the original data structure
- O(N*logN)



Heap Sort Example

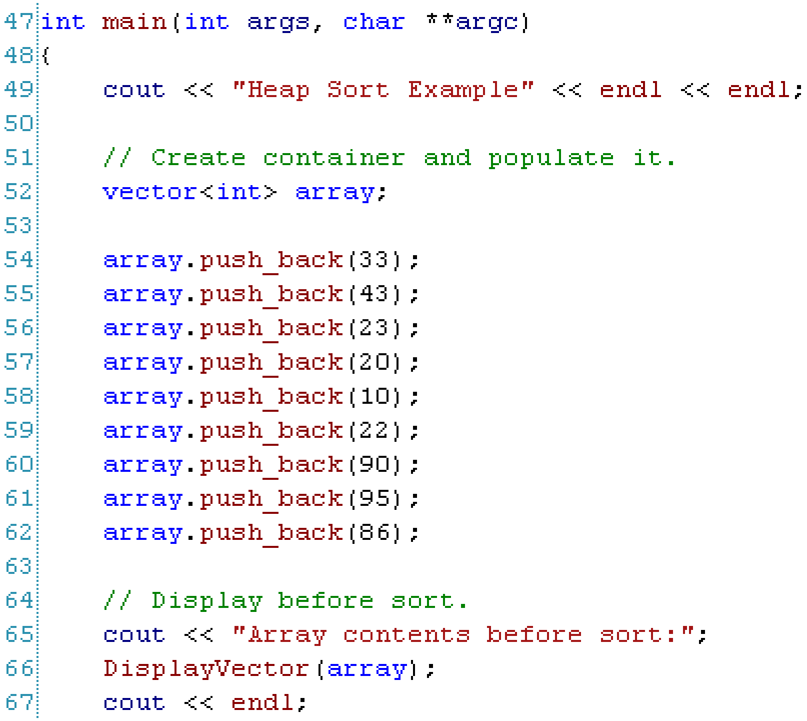


```
#include <iostream>
#include <vector>
#include "Heap.h"

using namespace std;

void HeapSortAscending(vector<int> &array)
{
    Heap<int> heap;
    int i;

    for(i = 0; i < (int)array.size(); i++)
        heap.push(array[i]);

    for(i = (int)array.size() - 1; i >= 0; i--)
    {
        array[i] = heap.peek();
        heap.pop();
    }
}
```

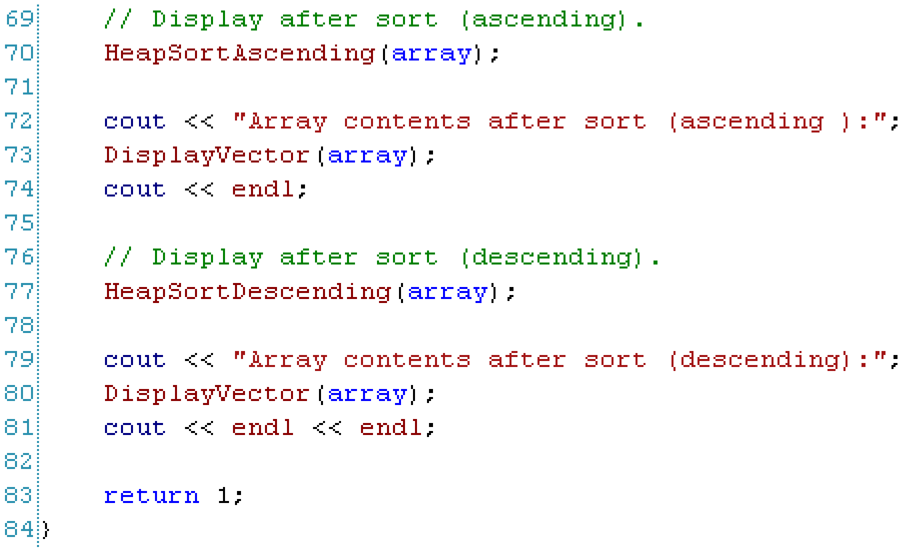


```
void HeapSortDescending(vector<int> &array)
{
    Heap<int> heap;
    int i;

    for(i = 0; i < (int)array.size(); i++)
        heap.push(array[i]);

    for(i = 0; i < (int)array.size(); i++)
    {
        array[i] = heap.peek();
        heap.pop();
    }
}

void DisplayVector(vector<int> &array)
{
    for(int i = 0; i < (int)array.size(); i++)
    {
        cout << " " << array[i];
    }

    cout << ".";
}
```

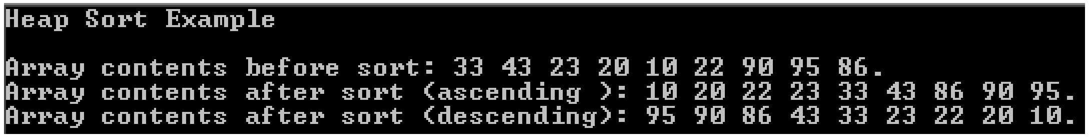
Heap Sort Example



```
int main(int args, char **argc)
{
    cout << "Heap Sort Example" << endl << endl;

    // Create container and populate it.
    vector<int> array;

    array.push_back(33);
    array.push_back(43);
    array.push_back(23);
    array.push_back(20);
    array.push_back(10);
    array.push_back(22);
    array.push_back(90);
    array.push_back(95);
    array.push_back(86);

    // Display before sort.
    cout << "Array contents before sort:";
    DisplayVector(array);
    cout << endl;
```



```
    // Display after sort (ascending).
    HeapSortAscending(array);

    cout << "Array contents after sort (ascending ):";
    DisplayVector(array);
    cout << endl;

    // Display after sort (descending).
    HeapSortDescending(array);

    cout << "Array contents after sort (descending):";
    DisplayVector(array);
    cout << endl << endl;

    return 1;
}
```



```
Heap Sort Example

Array contents before sort: 33 43 23 20 10 22 90 95 86.
Array contents after sort (ascending ): 10 20 22 23 33 43 86 90 95.
Array contents after sort (descending): 95 90 86 43 33 23 22 20 10.
```

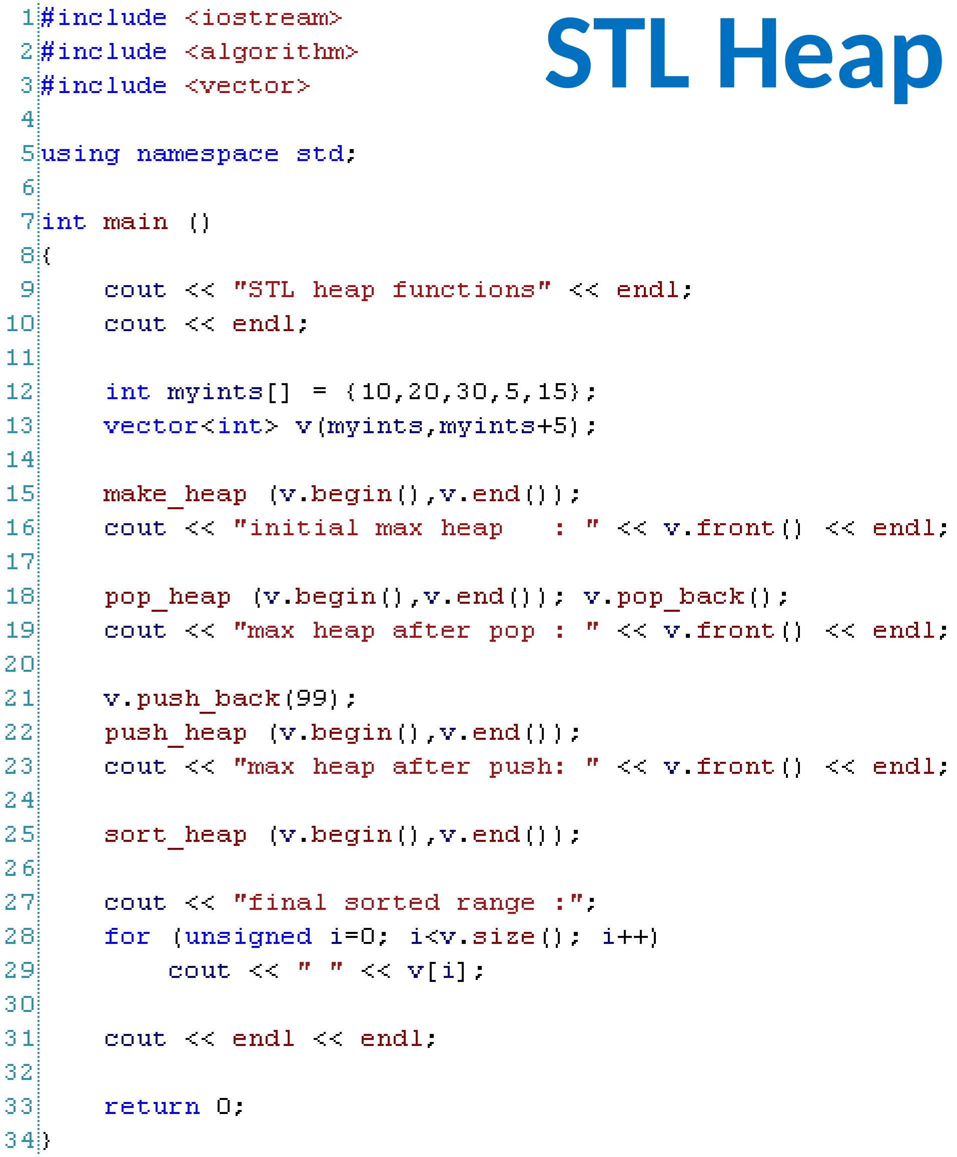
STL Heap Functions

- no heap data structure in the STL
- a few heap-related functions are part of the STL
- Using the STL heap functions to push, pop, and create the elements can create a heap without a heap container

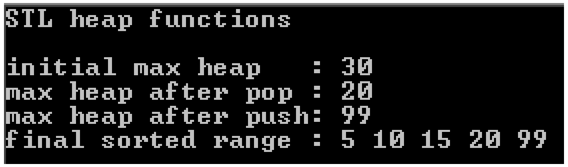
STL Heap Functions

- make_heap()
  - take a range of elements and create a heap out of them
- push_heap()
  - adds a range of elements to a heap
- pop_heap()
  - removes the largest element from the container
- sort_heap()
  - sorts the elements in the range

STL Heap Functions Example



```cpp
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main ()
{
    cout << "STL heap functions" << endl;
    cout << endl;

    int myints[] = {10,20,30,5,15};
    vector<int> v(myints,myints+5);

    make_heap (v.begin(),v.end());
    cout << "initial max heap   : " << v.front() << endl;

    pop_heap (v.begin(),v.end()); v.pop_back();
    cout << "max heap after pop : " << v.front() << endl;

    v.push_back(99);
    push_heap (v.begin(),v.end());
    cout << "max heap after push: " << v.front() << endl;

    sort_heap (v.begin(),v.end());

    cout << "final sorted range :";
    for (unsigned i=0; i<v.size(); i++)
        cout << " " << v[i];

    cout << endl << endl;

    return 0;
}
```



```
STL heap functions

initial max heap    : 30
max heap after pop  : 20
max heap after push: 99
final sorted range  : 5 10 15 20 99
```