**Binary Search Function:**

**Header file:**

algorithm header file.

**bool binary\_search (ForwardIterator first, ForwardIterator last, const T& val);**

**template <class ForwardIterator, class T, class Compare>**

**bool binary\_search (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);**

So, binary search can accept a function pointer in which you can define Compare.

**Here is an complete Example of binary\_search consisting both versions of functions:**

// binary\_search example

#include <iostream> // std::cout

#include <algorithm> // std::binary\_search, std::sort

#include <vector> // std::vector

bool myfunction (int i,int j) { return (i<j); }

int main ()

{

int myints[] = {1,2,3,4,5,4,3,2,1};

std::vector<int> v(myints,myints+9);

// 1 2 3 4 5 4 3 2 1

// using default comparison:

std::sort (v.begin(), v.end());

std::cout << "looking for a 3... ";

if (std::binary\_search (v.begin(), v.end(), 3))

std::cout << "found!\n"; else std::cout << "not found.\n";

// using myfunction as comp:

std::sort (v.begin(), v.end(), myfunction);

std::cout << "looking for a 6... ";

if (std::binary\_search (v.begin(), v.end(), 6, myfunction))

std::cout << "found!\n"; else std::cout << "not found.\n";

return 0;

}

**Lower\_bound:**

**First version:**

**template <class ForwardIterator, class T>**

**ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val);**

**Second version:**

**template <class ForwardIterator, class T, class Compare>**

**ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val, Compare comp);**

Return iterator to lower bound

Returns an iterator pointing to the first element in the range [first,last) which does not compare less than val.

The elements are compared using operator< for the first version, and comp for the second. The elements in the range shall already be sorted according to this same criterion (operator< or comp), or at least partitioned with respect to val.

**An Example:**template <class ForwardIterator, class T>

ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val)

{

ForwardIterator it;

iterator\_traits<ForwardIterator>::difference\_type count, step;

count = distance(first,last);

while (count>0)

{

it = first; step=count/2; advance (it,step);

if (\*it<val) { // or: if (comp(\*it,val)), for version (2)

first=++it;

count-=step+1;

}

else count=step;

}

return first;

**upper\_bound:**it is necessarily same as lower\_bound. Except it will only return a pointer to the value greater than the value sent as argument.