Data Structures and Abstract Data Types

- **Data structure** is representation of the logical relationship existing between individual elements of data.
- **Data structure** is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- Data structure affects the design of both structural & functional aspects of a program.
- An algorithm is a step by step procedure to solve a particular function.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures form a program.

Program = algorithm + Data Structure

Organizing Data and Efficiency

- Any organization for a collection of records can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.
- A solution is said to be efficient if it solves the problem within its resource constraints.
  - Space
  - Time
- The cost of a solution is the amount of resources that the solution consumes.

- Select a data structure as follows:
  1. Analyze the problem to determine the resource constraints a solution must meet.
  2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
  3. Select the data structure that best meets these requirements.

- Questions to ask:
  1. Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
  2. Can data be deleted?
  3. Are all data processed in some well-defined order, or is random access allowed?

Data Structures Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.
- After careful analysis of the problem characteristics it is possible to know the best data structure for the task.
- Bank example:
  - Start account: a few minutes
  - Transactions: a few seconds
  - Close account: overnight

STL in C++

- The Standard Template Library (STL): an extensive library of generic templates for classes and functions.
- Categories of Templates:
  - **Containers**: Class templates for objects that store and organize data (Data Structures)
  - **Iterators**: Class templates for objects that behave like pointers, and are used to access the individual data elements in a container
  - **Algorithms**: Function templates that perform various operations on elements of containers

Containers

- **Sequence Containers**
  - Stores data sequentially in memory, in a fashion similar to an array

- **Associative Containers**
  - Stores data in a non-sequential way that makes it faster to locate elements

Table 17-1 Sequence Containers

Container Class	Description
`array`	A fixed-size container that is similar to an array
`deque`	A double-ended queue. Like a `vector`, but designed so that values can be quickly added to or removed from the front and back. (This container will be discussed in Chapter 19.)
`forward_list`	A singly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
`list`	A doubly linked list of data elements. Values may be inserted to or removed from any position. (This container will be discussed in Chapter 18.)
`vector`	A container that works like an expandable array. Values may be added to or removed from a `vector`. The `vector` automatically adjusts its size to accommodate the number of elements it contains.

Table 17-2 Associative Containers

Container Class	Description
`set`	Stores a set of unique values that are sorted. No duplicates are allowed.
`multiset`	Stores a set of unique values that are sorted. Duplicates are allowed.
`map`	Maps a set of keys to data elements. Only one key per data element is allowed. Duplicates are not allowed. The elements are sorted in order of their keys.
`multimap`	Maps a set of keys to data elements. Many keys per data element are allowed. Duplicates are allowed. The elements are sorted in order of their keys.
`unordered_set`	Like a `set`, except that the elements are not sorted
`unordered_multiset`	Like a `multiset`, except that the elements are not sorted
`unordered_map`	Like a `map`, except that the elements are not sorted
`unordered_multimap`	Like a `multimap`, except that the elements are not sorted

Table 17-3 Container Adapter Classes

Container Adapter Class	Description
`stack`	An adapter class that stores elements in a `deque` (by default). A `stack` is a last-in, first-out (LIFO) container. When you retrieve an element from a `stack`, the `stack` always gives you the last element that was inserted. (This class will be discussed in Chapter 19.)
`queue`	An adapter class that stores elements in a `deque` (by default). A `queue` is a first-in, first-out (FIFO) container. When you retrieve an element from a `stack`, the `stack` always gives you the first, or earliest, element that was inserted. (This class will be discussed in Chapter 19.)
`priority_queue`	An adapter class that stores elements in a `vector` (by default). A data structure in which the element that you retrieve is always the element with the greatest value. (This class will be discussed in Chapter 19.)

STL Header Files

Table 17-4 Header Files

Header File	Classes
<array>	array
<deque>	deque
<forward_list>	forward_list
<list>	list
<map>	map, multimap
<queue>	queue, priority_queue
<set>	set, multiset
<stack>	stack
<unordered_map>	unordered_map, unordered_multimap
<unordered_set>	unordered_set, unordered_multiset
<vector>	vector

Template Class : array

- An array object works very much like a regular array
- A fixed-size container that holds elements of the same data type.
- array objects have a size() member function that returns the number of elements contained in the object.
- The array class is declared in the <array> header file.
- When defining an array object, you specify the data type of its elements, and the number of elements.

- Examples:

```
array<int, 5> numbers;
array<string, 4> names;
array<int, 5> numbers1 = {1, 2, 3, 4, 5};
array<string, 4> names2 = {"Jamie", "Ashley", "Doug", "Claire"};
```

Template Class : array

- The array class overloads the [ ] operator.
- You can use the [ ] operator to access elements using a subscript, just as you would with a regular array.
- The [ ] operator does not perform bounds checking. Be careful not to use a subscript that is out of bounds.

```
#include <iostream>
#include <array>
#include <algorithm>
#include <iterator>
using namespace std;

int main() {
    const int SIZE = 5;
    array<int, SIZE> a1 {1,2,3,4,5};

    for (int i=0;i<a1.size();i++)
        cout << a1[i] << ":";
    cout << endl;

    array<string,3> a2 = {"A","B","C"};
    cout << a2[0] << endl;

    a2[1]="D";

    for (auto x:a2)
        cout << x << ":";
    cout << endl;

    return 0;
}
```

Iterators

- Objects that work like pointers
- Used to access data in STL containers
- Five categories of iterators:

Table 17-6 Categories of Iterators

Iterator Category	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and –– operators).
Random access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with an input stream to read data from an input device or a file.
Output	Can be used with an output stream to write data to an output device or a file.

Iterators and Pointers

	Pointers	Iterators
Use the * and -> operators to dereference	Yes	Yes
Use the = operator to assign to an element	Yes	Yes
Use the == and != operators to compare	Yes	Yes
Use the ++ operator to increment	Yes	Yes
Use the -- operator to decrement	Yes	Yes (bidirectional and random-access iterators)
Use the + operator to move forward a specific number of elements	Yes	Yes
Use the - operator to move backward a specific number of elements	Yes	Yes Yes (bidirectional and random-access iterators)

- To define an iterator, you must know what type of container you will be using it with.
- The general format of an iterator definition:

***containerType*::iterator *iteratorName*;**

Where *containerType* is the STL container type, and *iteratorName* is the name of the iterator variable that you are defining.

Iterators

- For example, suppose we have defined an array object, as follows:

  array<string, 3> names = {"Sarah", "William", "Alfredo"};

- We can define an iterator that is compatible with the array object as follows:

  array<string, 3>::iterator it;

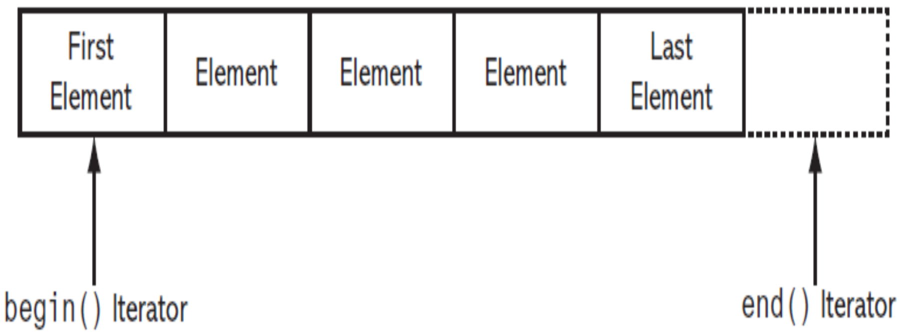
- This defines an iterator named it.
- The iterator can be used with an array<string, 3> object.
- All of the STL containers have a begin() member function that returns an iterator pointing to the container's first element.

```
int main()
{
    const int SIZE = 3;

    array<string, SIZE> names
        {"Sarah", "William", "Alfredo"};

    array<string,SIZE>::iterator it;

    it = names.begin();

    cout << *it << endl;

    it++;

    cout << *it << endl;

    return 0;
}
```

```
Sarah
William
```

Iterators

- All of the STL containers have a **end()** member function that returns an iterator pointing to the **position after** the container's **last element**.



- You typically use the end() member function to know when you have reached the end of a container.
- You can use the auto keyword to simplify the definition of an iterator.

```
int main()
{
    const int SIZE = 3;
    array<string, SIZE> names =
        {"Sarah", "William", "Alfredo"};

    array<string,SIZE>::iterator it;
    it = names.begin();
    while (it != names.end()) {
        cout << *it << ":";
        it++;
    }
    cout << endl;
    for (auto i=names.begin();
          I  != names.end();
          i++)
        {
                cout << *i << ":";
        }
    cout << endl;

    return 0;
}
```

Mutable Iterators

- An iterator of the iterator type gives you read/write access to the element to which the iterator points.
- This is commonly known as a mutable iterator.

```
int main()
{
    const int SIZE = 3;
    array<string, SIZE> names = {"Sarah", "William",
                                 "Alfredo"};
    array<string,SIZE>::iterator it;
    it = names.begin();

    *it = "Sharaf";
    it++;
    it++;
    *it = "ChienLee";

    for (auto x:names)
        cout << x << ":";

    cout << endl;

    return 0;
}
```

```
Sharaf:William:ChienLee:
```

Constant Iterators

- An iterator of the **const_iterator** type provides read-only access to the element to which the iterator points.
- The STL containers provide a **cbegin()** member function and a **cend()** member function.
- The **cbegin()** member function returns a **const_iterator** pointing to the first element in a container.
- The **cend()** member function returns a const_iterator pointing to the end of the container.
- When working with **const_iterators**, simply use the container class's **cbegin()** and **cend()** member functions instead of the begin() and end() member functions.

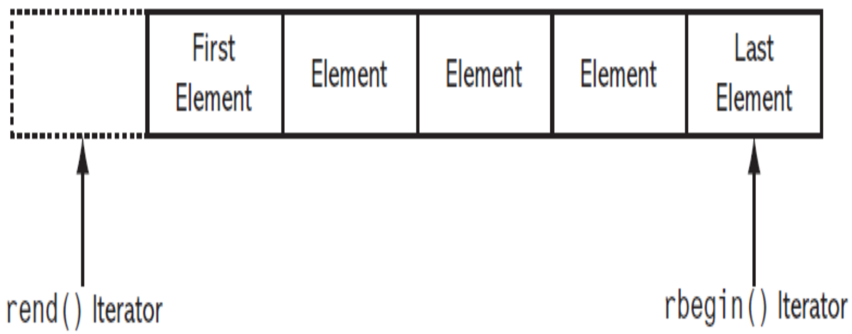
```
int main()
{
    const int SIZE = 3;
    array<string, SIZE> names
            = {"Sarah", "William", "Alfredo"};

    array<string,SIZE>::const_iterator it;
    it = names.cbegin();

 // The next line generates compile error
    *it = "Sharaf";

    cout << *it << endl;
    it++;
    cout << *it << endl;

    return 0;
}
```

Reverse Iterators

- A reverse iterator works in reverse, allowing you to iterate backward over the elements in a container.
- With a reverse iterator, the last element in a container is considered the first element, and the first element is considered the last element.
- The ++ operator moves a reverse iterator backward, and the −− operator moves a reverse iterator forward.
- The following STL containers support reverse iterators:
  - array
  - deque
  - list
  - map
  - multimap
  - multiset
  - set
  - vector
- All of these classes provide an rbegin() member function and an rend() member function.

Reverse Iterators

- The rbegin() member function returns a reverse iterator pointing to the last element in a container.
- The rend() member function returns an iterator pointing to the position before the first element.



- To create a reverse iterator, define it as reverse_iterator

```
int main()
{
    const int SIZE = 3;
    array<string, SIZE> names = {"Sarah",
                                 "William", "Alfredo"};

    array<string,SIZE>::reverse_iterator it;
    it = names.rbegin();

    *it = "Sharaf";
    it++;
    it++;
    *it = "ChienLee";

    for (auto x:names)
        cout << x << ":";
    cout << endl;

    return 0;
}
```

```
ChienLee:William:Sharaf:
```

The `vector` Class

- A vector is a sequence container that works like an array, but is dynamic in size.
- Overloaded **[]** operator provides access to existing elements
- The vector class is declared in the **<vector>** header file.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v1;
    vector<int> v2 (5,100);
    vector<int> v3 (v2.begin(),v2.end());
    vector<int> v4 (v3);
    vector<int> v5 {1,3,4,6,7,8};
    vector<int> v6 = {1,3,4,6,7,8};

    int arr[] = {1,3,5,6};
    vector<int> v7 (arr, arr + sizeof(arr) / sizeof(int) );

    cout << "vector v7:";
    for (vector<int>::iterator it = v7.begin();
         it != v7.end();
         ++it)
         cout << ' ' << *it;

    cout << '\n';

    return 0;
}
```

Accessing the elements of a vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v {1,3,4,6,7,8};

    for (int i=0;i<v.size();i++)
        cout << ' ' << v[i];

   cout << endl;

    return 0;
}
```

```
for (auto x:v)  cout << ' ' << x;
```

```
for (int i=0; i<v.size(); i++)
        cout << ' ' << v.at(i);
```

```
for (auto it = v.begin(); it != v.end(); ++it)
        cout << " " << *it;
```

Adding elements to the vector

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v{1,3,4};

    v.push_back(5);


    auto it = v.begin() + 1;
    v.insert(it, 7);

    it = v.begin() + 2;
    v.insert(it,3,0);

    for (auto x:v)
      cout << " " << x;
    cout << '\n';

    return 0;
}
```

The push_back member function adds a new element to the end of a vector

Add an element in a specific position

Add a number of elements initialized to a specific value

```
1 7 0 0 0 3 4 5
```

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v{2,4,6};
    vector<int> v2{1,2,3,4,5,6,7,8,9};

    auto vits=v.begin();
    auto vite=v.end();
    auto v2its = v2.begin()+1;

    v2.insert(v2its,vits,vite);

    for (auto x:v2)
      cout << " " << x;
    cout << '\n';

    return 0;
}
```

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

Storing Objects in a vector

```
#include <iostream>
#include <vector>
using namespace std;

class Shape {
    int x;
    int y;
public:
    Shape(int x=0,int y=0)
    :x(x),y(y)
    { }

    void print() {
        cout << "("
              << x
              <<","
              <<y
              <<")";
    }
};
```

```
int main()
{
    vector<Shape> v {Shape(1,1), Shape(2,2), Shape(3,3)};

    Shape s(4,4);
    v.push_back(s);

    v.insert(v.begin()+1,Shape(5,5));

    for (auto x:v) x.print();
    cout << '\n';

    return 0;
}
```

```
(1,1)(5,5)(2,2)(3,3)(4,4)
```

Storing pointers to objects in a vector

```
#include <iostream>
#include <vector>
using namespace std;

class Shape {
    int x;
    int y;
public:
    Shape(int x=0,int y=0)
     :x(x),y(y)
    {}

    void print() {
        cout << "("
               << x
               <<","
               <<y
               <<")";
    }
};
```

```
int main()
{
    vector<Shape*> v { new Shape(1,1),
                       new Shape(2,2),
                       new Shape(3,3)};

    Shape s(4,4);
    v.push_back(&s);

    v.insert(v.begin()+1,new Shape(5,5));

    for (auto x:v) x->print();
    cout << '\n';

    return 0;
}
```

```
(1,1)(5,5)(2,2)(3,3)(4,4)
```

Inserting Container Elements with Emplacement

- Member functions such as insert() and push_back() can cause temporary objects to be created in memory while the insertion is taking place.
- This is not a problem in programs that make only a few insertions.
- However, these functions can be inefficient for making a lot of insertions.

```
class Shape {
    int x;
    int y;
public:
    Shape(int x=0,int y=0)
    :x(x),y(y)
    {cout << "def constr...\n";}

    Shape(const Shape& s){
        x = s.x;
        y = s.y;
        cout << "Copy constr...\n";
    }
    ~Shape()
    {cout << "destructor...\n";}

    void print(){
        cout << "(" << x << ","
                << y << ")";
    }
};
```

```
int main()
{
    vector<Shape> v;
    cout << "....1......\n";
    v.push_back(Shape(1,1));
    cout << "....2......\n";
    v.push_back(Shape(2,1));
    cout << "....3......\n";
    v.push_back(Shape(3,1));
    cout << "....4......\n";

    return 0;
}
```

```
....1......
def constr...
Copy constr...
destructor...
....2......
def constr...
Copy constr...
Copy constr...
destructor...
destructor...
....3......
def constr...
Copy constr...
Copy constr...
Copy constr...
destructor...
destructor...
destructor...
....4......
destructor...
destructor...
destructor...
```

Inserting Container Elements with Emplacement

```
class Shape {
    int x;
    int y;
public:
    Shape(int x=0,int y=0)
    :x(x),y(y)
    {cout << "def constr...\n";}

    Shape(const Shape& s) {
        x = s.x;
        y = s.y;
        cout << "Copy constr...\n";
    }
    ~Shape()
    {cout << "destructor...\n";}

    void print(){
        cout << "(" << x << ","
                << y << ")";
    }
};
```

```
int main()
{
    vector<Shape> v;
    cout << "....1......\n";
    v.emplace_back(1,1);
    cout << "....2......\n";
    v.emplace_back(2,1);
    cout << "....3......\n";
    v.emplace_back(3,1);
    cout << "....4......\n";

    return 0;
}
```

```
....1......
def constr...
....2......
def constr...
Copy constr...
destructor...
....3......
def constr...
Copy constr...
Copy constr...
destructor...
destructor...
....4......
destructor...
destructor...
destructor...
```

Emplacement avoids the creation of temporary objects in memory while a new object is being inserted into a container.

Inserting Container Elements with Emplacement: noexcept

```
class Shape {
    int x;      int y;
public:
    Shape(int x=0,int y=0)
    :x(x),y(y) {
        cout << "def constr...\n";
    }
    Shape(const Shape& s) noexcept {
        x = s.x;
        y = s.y;
        cout << "Copy constr...\n";
    }
    Shape(Shape&& s) noexcept {
        x = s.x;
        y = s.y;
        cout << "move constr...\n";
    }
    ~Shape() {
        cout << "destructor...\n";
    }
    void print() {
        cout <<"(" << x <<”,"<< y << ")";
        }
};
```

```
int main()
    {
        vector<Shape> v;
        cout << "....1......\n";
        v.emplace_back(1,1);
        cout << "....2......\n";
        v.emplace_back(2,1);
        cout << "....3......\n";
        v.emplace_back(3,1);
        cout << "....4......\n";

        return 0;
    }
```

```
....1......
def constr...
....2......
def constr...
move constr...
destructor...
....3......
def constr...
move constr...
move constr...
destructor...
destructor...
....4......
destructor...
destructor...
destructor...
```

Algorithms

STL Algorithms

- The STL provides a number of algorithms, implemented as function templates, in the **<algorithm>** header file.

- These functions perform various operations on ranges of elements.

- A range of elements is a sequence of elements denoted by two iterators:
  - The first iterator points to the first element in the range
  - The second iterator points to the end of the range (the element to which the second iterator points is not included in the range).

Categories of Algorithms in the STL

- Min/max algorithms
- Sorting algorithms
- Search algorithms
- Read-only sequence algorithms
- Copying and moving algorithms
- Swapping algorithms
- Replacement algorithms
- Removal algorithms
- Reversal algorithms
- Fill algorithms
- Rotation algorithms
- Shuffling algorithms
- Set algorithms
- Transformation algorithm
- Partition algorithms
- Merge algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

Sorting

- The sort function:

  sort(*iterator1*, *iterator2*);

  iterator1 and *iterator2* mark the beginning and end of a range of elements. The function sorts the range of elements in ascending order.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v = {4,2,5,6,3,1,9,0};
    sort(v.begin(), v.end());
    for (auto x:v)
        cout << x << ", ";
    cout << endl;
   return 0;
}
```

```
0, 1, 2, 3, 4, 5, 6, 9,
```

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


bool order (int x, int y) {
    return x>y;
}

int main()
{
    vector<int> v = {4,2,5,6,3,1,9,0};
    sort(v.begin(), v.end(),order);
    for (auto x:v)
        cout << x << ", ";
    cout << endl;
   return 0;
}
```

```
9, 6, 5, 4, 3, 2, 1, 0,
```

Searching

- The binary_search function:

  binary_search(*iterator1, iterator2, value*);

  iterator1 and *iterator2* mark the beginning and end of a range of elements that are sorted in ascending order. value is the value to search for. The function returns true if value is found in the range, or false otherwise.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v = {4,2,5,6,3,1,9,0};
    int value = 9;

    //must sort the data before searching
    sort(v.begin(),v.end());

    if (binary_search(v.begin(), v.end(),value))
        cout << value << " is found\n";
    else
        cout << value << " is not Found\n";

    for (auto x:v)
        cout << x << ", ";
    cout << endl;
   return 0;
}
```

for_each

- Many of the function templates in the STL are designed to accept function pointers as arguments.
- This allows you to “plug” one of your own functions into the algorithm.
- For example:

  for_each(*iterator1*, *iterator2*, *function*)

  - *iterator1* and *iterator2* mark the beginning and end of a range of elements.
  - *function* is the name of a function that accepts an element as its argument.
  - The for_each() function iterates over the range of elements, passing each element as an argument to *function*.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void print(int x) { cout << x << ":"; }

void Double (int &x) { x*=2; }

int main()
{
    vector<int> v = {4,2,5,6,3,1,9,0};

    for_each(v.begin(),v.end(),print);
    cout << endl;

    for_each(v.begin(),v.end(),Double);

    for_each(v.begin(),v.end(),print);
    cout << endl;
   return 0;
}
```

4:2:5:6:3:1:9:0:
8:4:10:12:6:2:18:0:

count_if

count_if(*iterator1*, *iterator2*, *function*)

- *iterator1* and *iterator2* mark the beginning and end of a range of elements.
- *function* is the name of a function that accepts an element as its argument, and returns either true or false.
- The count_if() function iterates over the range of elements, passing each element as an argument to *function*.
- The count_if function returns the number of elements for which function returns true.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


bool odd(int x){
   return x%2==1;
}


int main()
{
   vector<int> v = {1,2,3,4,5,6,7};

   auto odds = count_if(v.begin(),v.end(),odd);

   cout << odds << endl;

  return 0;
}
```

Pair and Tuple
Dara structures

pair

- std::pair is a struct template that provides a way to store two heterogeneous objects as a single unit.

```
using namespace std;
int main() {
    pair<int,int> p1 = {1,1};
    pair<int,int> p2 {2,2};
    pair<int,int> p3 = make_pair(3,3);

    cout << p1.first << ":" << p1.second << endl;

    p2.first = 1;
    p2.second = 1;
    cout << p2.first << ":" << p2.second << endl;
    cout << get<0>(p3) << ":" << get<1>(p3) << endl;

    pair<int,int> p4 = p2;

    pair<int,int> *p = &p4;
    cout << p->first << ":" << p->second << endl;
    p4.swap(p1);
    cout << p4.first << ":" << p4.second << endl;

    if (p1 == p2 ) cout << "p1 and p2 are equal\n";
    else cout << "p1 and p2 are not the equal\n";
}
```

pair

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<pair<int,int>> v ={{2,2},{1,3},{2,1},{4,5},{6,6}};
    v.push_back(make_pair(0,0));
    sort(v.begin(),v.end());

    pair<int,int> pv = make_pair(4,4);
    if (binary_search(v.begin(), v.end(),pv) )
        cout << "found\n";
    else
        cout << "not found\n";

    for (auto x:v)
    {
        cout << x.first << ":" << x.second << endl;
    }

    return 0;
}
```

```
not found
0:0
1:3
2:1
2:2
4:5
6:6
```

tuple

- std::tuple is a fixed-size collection of heterogeneous values.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    tuple<int,string,double> t1 = {1,"A",3.1};
    tuple<int,string,double> t2 = {2,"B",2.8};
    tuple<int,string,double> t3 = make_tuple(3,"C",3.7);

    cout << get<0>(t2)<<get<1>(t2)<<get<2>(t2) <<endl;

    auto [id,name,gpa] = t3;
    cout << id << name << gpa << endl;

    tuple<int,string,double> t4 = t2;
    tie(id,name,gpa) = t4;
    cout << id << name << gpa << endl;

    tuple<int,string,double> t5 = tie(id,name,gpa);

    t5.swap(t1);
    cout << get<0>(t5)<<get<1>(t5)<<get<2>(t5) <<endl;

    return 0;
}
```

tuple

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<tuple<int,string,double>> v = { {1,"A",3.1}, {2,"B",2.8} };
    v.push_back(make_tuple(3,"C",3.7));

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

    tuple<int,string,double> pv = make_tuple(2,"B",2.8);
    if (binary_search(v.begin(), v.end(),pv) )
        cout << "found\n";
    else
        cout << "not found\n";

    for (auto x:v) {
        cout << get<0>(x) << ":"
              << get<1>(x) << ":"
              << get<2>(x) << endl;
    }
    return 0;
}
```

found
1:A:3.1
2:B:2.8
3:C:3.7