Data Structures & Algorithms – Cấu Trúc Dữ Liệu & Giải Thuật

Nguyễn Quản Bá Hồng*

Ngày 3 tháng 6 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Data Structures & Algorithms – Cấu Trúc Dữ Liệu & Giải Thuật.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_structure_algorithm/lecture/NQBH_data_structure_algorithm_lecture.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/data_structure_algorithm/lecture/NQBH_data_structure_algorithm_lecture.tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

1 Array in C++ STL

2 Vector in C++ STL

Resources – Tài nguyên.

1. Geeks4Geeks/vector in C++ STL

Definition 1. C++ vector *is a dynamic array that stores collection of elements of same type in contiguous memory. It has the ability to resize itself automatically when an element is inserted or deleted.*

– C++ vector *là một mảng động lưu trữ tập hợp các phần tử cùng loại trong bộ nhớ liền kề. Nó có khả năng tự động thay đổi kích thước khi một phần tử được chèn vào hoặc xóa.*

2.1 Create vectors – Tạo vectors

Resources – Tài nguyên.

1. Geeks4Geeks/8 ways to initialize vector in C++.

Before creating a vector, we must know that a vector is defined as the `std::vector` class template in the `<vector>` header file.

```
vector<T> v;
```

where `T` is the type of elements & `v` is the name assigned to the vector.

Now we are creating an instance of `std::vector` class. This requires us to provide the type of elements as template parameter.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // Creating an empty vector
    vector<int> v1;
    return 0;
}
```

We can also provide the values to be stored in the vector inside `{}` curly braces. This process is called *initialization*.

```
#include <bits/stdc++.h>
using namespace std;
void printVector(vector<int>& v) {
    for (auto x: v) {
        cout << x << " ";
    }
    cout << endl;
}

int main() {
    // Creating a vector of 5 elements from initializer list
    vector<int> v1 = {1, 4, 2, 3, 5};

    // Creating a vector of 5 elements with default value
        vector<int> v2(5, 9);

    printVector(v1);
```

```
    printVector(v2);
        return 0;
}
```

Output:

```
1 4 2 3 5
9 9 9 9 9
```

Statement `vector<int> v1 = {1, 4, 2, 3, 5}` initializes a vector with given values. Statement `vector<int> v2(5, 9)` creates a vector of size 5 where each element initialized to 9.

Remark 1. *Statement* `vector<int> v`$= \{v_1, v_2, \ldots, v_n\}$ *initializes a vector with given values. Statement* `vector<int> v(n, a)` *creates a vector of size* $n \in \mathbb{N}^\star$ *where each element initialized to* $a \in \mathbb{Z}$.

Initializing a vector means assigning some initial values to the `std::vector` elements. Here are 8 different ways to initialize a vector in C++.

2.1.1 Initialize vector in C++ by using initializer list – Sử dụng danh sách các giá trị khởi tạo

We can initialize a vector with the list of values enclosed in curly braces `{}` known as initializer list. The value of the list will be assigned sequentially i.e. 1st value will be assigned to the 1st element of vector, 2nd value to 2nd element, ..., nth value to nth element. Syntax:

```
vector<type> v = {val1, val2, val3, ...};
```

where `val1, val2, val3`, ... are the initial values, e.g.:

```
// C++ Program to initializ std::vector with initializer list
#include <bits/stdc++.h>
using namespace std;

int main() {
    // Initializing std::vector with list of multiple values
    vector<int> v = {11, 23, 45, 89};

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 23 45 89`.

Note 1. `for (auto i : v)` *means for each element of the type that will be determined automatically in the vector* `v`.

2.1.2 Initialize vector in C++ 1 by 1 initialization

Vector can be initialized by pushing value 1 by 1. In this method, an empty vector is created, & elements are added to it 1 by 1 using the `vector::push_back()` method. This method is mostly used to initialize vector after declaration. Syntax:

```
v.push_back(val);
```

where `val` is the value which we have to insert, e.g.:

```
// C++ Program to initialize std::vector by pushing values 1 by 1
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v;

    // Pushing Value one by one
    v.push_back(11);
    v.push_back(23);
    v.push_back(45);
    v.push_back(89);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 23 45 89`.

2.2 Initialize vector in C++ by with a single value

We can initialize all the elements of the vector to a single value. We create a vector of a specified size & initialize all elements to the same value using vector constructor. Syntax:

```
vector<type> v(n, val);
```

where $n \in \mathbb{N}$ is the size & `val` is the initial value, e.g.:

```
// C++ Program to initializ the std::vector with specific value
#include <bits/stdc++.h>
using namespace std;

int main() {
    // Initializing all the elements of a vector using a single value
    vector<int> v(5, 11);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 11 11 11 11`.

2.2.1 Initialize vector in C++ from an array

We can also initialize a vector using plain old static arrays using vector constructor. This works by copying all the elements of the array to the newly created vector. Syntax:

```
vector<type> v(arr, arra + n);
```

where `arr` is the array name & $n \in \mathbb{N}$ is the size of the array, e.g.:

```
// C++ Program to initializ the std::vector from another array
#include <bits/stdc++.h>
using namespace std;

int main() {
    int arr[] = {11, 23, 45, 89};
    int n = sizeof(arr) / sizeof(arr[0]);

    // initialize the std::vector v by arr
    vector<int> v = {arr, arr + n};

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 23 45 89`.

2.2.2 Initialize vector in C++ from another vector

We can also initialize a newly created vector from an already created vector if they are of same type. Syntax:

```
vector<type> v2(v1.begin(), v1.end());
```

where `v1` is the already existing vector, e.g.:

```
// C++ Program to initializ the std::vector from another vector
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v1 = {11, 23, 45, 89};

    // initialize the vector v2 from vector v1
    vector<int> v2(v1.begin(), v1.end());

```

```
    for (auto i : v2)
        cout << i << " ";
    return 0;
}
```

Output: `11 23 45 89`.

2.2.3 Initialize vector in C++ by from any STL container

Vectors are flexible containers that can be initialized by any other already existing containers e.g. set, multiset, map, etc. if they are of same type. Syntax:

```
vector<type> v(first, last);
```

where `first, last` are the iterator to the 1st element & the element just after the last element in the range of STL container.

2.2.4 Initialize vector in C++ by using `std::fill()` function

We can also use the `std::fill` function to initialize the whole or a part of a vector to the same value. Syntax:

```
fill(first, last, val);
```

where `first, last` are the iterator to the 1st element & the element just after the last element in the range of STL container & `val` is the value to be initialized with, e.g.:

```
// C++ Program to initialize the std::vector using std::fill() method
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v(5);

    // initialize vector v with 11
    fill(v.begin(), v.end(), 11);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 11 11 11 11`.

2.2.5 Initialize vector in C++ by using `std::iota()` function

The `std::iota()` function from the `<numeric>` library allows us to initialize a vector with consecutive values starting from the given value. Syntax:

```
std::iota(first, last, val);
```

where `first, last` are the iterator to the 1st element & the element just after the last element in the range of the vector & `val` refers to the starting value, e.g.:

```
// C++ Program to initializ the std::vector using std::iota()
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v(5);

        // using std::iota() to initialize vector v with 11
    iota(v.begin(), v.end(), 11);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `11 12 13 14 15`.

2.3 Insert elements – Chèn phần tử

An element can be inserted into a vector using vector insert() method which takes linear time. But for the insertion at the end, the vector `push_back()` method can be used, which is much faster, taking only constant time.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'f', 'd'};

    // Inserting 'z' at the back
    v.push_back('z');

    // Inserting 'c' at index 1
    v.insert(v.begin() + 1, 'c');

    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    return 0;
}
```

Output: `a c f d z`.

2.4 Access or update elements – Tiếp cận hoặc cập nhật các phần tử

Just like arrays, vector elements can be accessed using their index inside the [] subscript operator. While accessing elements, we can also update the value of that index using assignment operator =. The `[] subscript operator` doesn't check whether the given index exists in the vector or not. So, there is another member method vector al() for safely accessing or update elements.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

        // accessing & printing values
    cout << v[3] << endl;
    cout << v.at(2) << endl;

    // updating values using indexes 3 & 2
    v[3] = 'D';
    v.at(2) = 'F';

    cout << v[3] << endl;
    cout << v.at(2);
    return 0;
}
```

2.5 Find vector size – Tìm cỡ/kích thước của vector

1 of the common problems with arrays was to keep a separate variable to store the size information. Vector provides the solution to this problem by providing size() method.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

    // finding size
    cout << v.size();
    return 0;
}
```

Output: `5`.

2.6 Traverse vector – Duyệt vector

Vector in C++ can be traversed using indexes in a loop. The indexes start from 0 & go up to a vector size −1. To iterate through this range, we can use a loop & determine the size of the vector using the vector size() method.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

    // traversing vector using range based for loop
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    return 0;
}
```

Output: `a c f d z`.

We can also use a range-based loop for simple traversal.

2.7 Delete elements – Xóa phần tử

An element can be deleted from a vector using vector erase() but this method needs iterator to the element to be deleted. If only the value of the element is known, then `find()` function is used to find the position of this element.

For the deletion at the end, the vector `pop_back()` method can be used, & it is much faster, taking only constant time.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<char> v = {'a', 'c', 'f', 'd', 'z'};

    // deleting last element 'z'
    v.pop_back();

    // deleting element 'f'
    v.erase(find(v.begin(), v.end(), 'f'));

    for (int i = 0; i < v.size(); i++) {
        cout << v[i] << " ";
    }
    return 0;
}
```

Output:

```
a c f d
a c d
```

2.8 Other operations – Các thao tác khác

Vector is 1 of the most frequently used containers in C++. It is used in many situations for different purposes. The following examples aim to help you master vector operations beyond the basics.

3 2D Vector in C++

Resources – Tài nguyên.

1. Geeks4Geeks/2D vector in C++.

A 2D vector is a vector of the vectors, i.e., each element is a vector in itself. It can be visualized as a matrix where each inner vector represents a row, & the number of rows represents a row, & the number of rows represents the maximum columns. A 2D vector is dynamically resizable in both dimensions. Syntax:

```
vector<vector<data_type>> v;
```

where `data_type` is the type of elements & `V` is the name assigned to the 2D vector.

3.1 Creating a 2D vector – Tạo 1 vector 2D

In C++, we can create/declare a 2D vector by using the vector container defined in the C++ Standard Template Library (STL). We can simply create a 2D vector by creating a vector with the vector data type.

Just like vectors, a 2D vector can be created & initialized in multiple ways:

1. **Default.** An empty 2D vector can be created using the declaration:

   ```
   vector<vector<data_type>> v;
   ```

   It can be filled in later on in the program.

2. **With user defined size & default value.** A vector of a specific size can also be declared & initialized to the given value as default value.

   ```
   vector<vector<T>> v(n, vector<T>(m, value));
   ```

   where $n \in \mathbb{N}^\star$ is the number of rows, $m \in \mathbb{N}^\star$ is the number of columns, `val` is the new default value for all of the elements of the vector.

3. **Using initializer list.** Vector can also be initialized using a list of values enclosed in `{}` braces separated by comma. The list must be nested according to the 2 dimensions as it helps in determining the row size & column size.

   ```
   vector<vector<T>> v = {{x1, x2, ...}, {y1, y2, ...}, ...};
   ```

   E.g.:

```cpp
#include <bits/stdc++.h>
using namespace std;

void printV(vector<vector<int>>& v) {
    for (auto i: v) {
        for (auto j: i) {
            cout << j << " ";
        }
        cout << endl;
    }
    cout << endl;
}

int main() {
    // an empty 2D vector
    vector<vector<int>> v1;

    // 2D vector with initial size and value
        vector<vector<int>> v2(2, vector<int>(3, 11));

    // a 2D vector initialized with initializer list
    vector<vector<int>> v3 = {
        {1, 2, 3},
        {4, 5, 6},
    };

    printV(v1);
    printV(v2);
    printV(v3);
    return 0;
}
```

Output:

```
11 11 11
11 11 11

1 2 3
4 5 6
```

Basic operations of 2D vector:

1. Inserting elements in a 2D vector
2. Accessing & updating elements
3. Deleting elements
4. Traversing the vector

3.2 Inserting elements in a 2D vector

In 2D vectors, there are 2 types of insertion:

1. Insert a new row.
2. Insert a value in an existing row.

These can be inserted at any given position using `vector_insert()` & at the end using `vector push_back()`. As vector can dynamically grow, each row can have different size like Java's jagged arrays, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

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

    // insert a new row at the end
    v.push_back({7, 8, 9});

    // insert value in 2nd row at 2nd position
    v[1].insert(v[1].begin() + 1, 10);

    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++) {
            cout << v[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
1 2 3
4 10 5 6
7 8 9
```

3.3 Accessing & uploading elements

As 2D vectors are organized as matrices with row & column, we need 2 indexes to access an element: 1 for the *row number i* & other for the *column number j*. We can then use any access method e.g. [] operator or vector at() method.

The value of the accessed element can be changed by assigning a new value using = operator, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

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

    // access 3rd element in 2nd row
    cout << "3rd element in 2nd row: "<< v[1][2] << endl;

    // access 2nd element in 1st row
    cout << "2nd element in 1st row: "<<v[0][1]<<endl;

    // updating the 2nd element in 1st row
    v[0][1] = 9;
```

```
    cout << "2nd element in 1st row after updating: " <<v[0][1]<<endl;

    return 0;
}
```

Output:

```
3rd element in 2nd row: 6
2nd element in 1st row: 2
2nd element in 1st row after updating: 9
```

3.4 Deleting elements from a 2D vector

Similar to insertion, there are 2 types of deletion in 2D vector:

1. Delete a row

2. Delete a value in an existing row.

Elements can be deleted using vector erase() for a specific position or range & using vector pop_back() to remove the last element, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

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

    // delete the 2nd row
    v.erase(v.begin() + 1);

    // delete 2nd element in 1st row
    v[0].erase(v[0].begin() + 1);

    for (int i = 0; i < v.size(); i++) {
        for (int j = 0; j < v[i].size(); j++)
            cout << v[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

Output: `1 3`.

3.5 Traversing 2D vectors

Traversing a 2D vector involves iterating through rows & columns using nested loops & access the elements by indexes, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

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

    // loop through rows
    for (int i = 0; i < v.size(); i++) {
        // loop through columns
        for (int j = 0; j < v[i].size(); j++)
            cout << v[i][j] << " ";
        cout << endl;
    }

    return 0;
}
```

Output:

```
1 2 3
4 5 6
```

C++ provides more methods to traverse 2D vector.

3.6 Finding size of 2D vector

Finding the size of a 2D vector involves finding its row size & column size which can be done using the `vector.size()` method. The size vector used on the outer vector gives the number of rows in the 2D vector while using them on the inner vector gives the number of columns in that row (as all rows can have different number of columns). Syntax:

```
//finding the number of rows
int rows = vec.size();

//finding the number of columns
int rows = vec[0].size();
```

where `vec` is the name of the vector for which the size is to be determined. Since each element of a 2D vector is a vector itself we can use the `size()` method on the elements `vec[0]` to find the size of each row separately, e.g.:

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

int main() {
    // creating a 2D vector
    vector<vector<int>> vec = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // finding the number of rows  (size of the outer vector)
    int rows = vec.size();
    cout << "Number of rows: " << rows << endl;

    // finding the number of columns (size of any inner vector first row)
    int cols = vec[0].size();
    cout << "Number of columns: " << cols << endl;

    return 0;
}
```

Output:

```
Number of rows: 3
Number of columns: 3
```

3.7 Common operations & applications

Apart from the basic operations, there are many operations that can be performed on 2D vectors.

4 Advantages of Vector Over Array in C++

Resources – Tài nguyên.

1. Geeks4Geeks/advantages of vector over array in C++.

In C++, both vectors & arrays are used to store collections of elements, but vector offers significant advantages over arrays in terms of flexibility, functionality, & ease of use. This section explores the benefits of using vectors in C++ programming.

4.1 Dynamic resizing

Unlike arrays, vectors can dynamically resize themselves, i.e., you don't need to know the size of the vector in advance, it can grow & shrink according to the number of elements present in it.

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<int> v;

    // initial size
    cout << v.size() << endl;

    // add elements dynamically
    for (int i = 1; i <= 5; ++i)
        v.push_back(i);

    // size after inserting elements
    cout << v.size();
    return 0;
}
```

Output:

```
0
5
```

4.2 Built-in functions & operations

Vectors come with a variety of member functions, e.g. `push_back()`, `pop_back()`, `insert()`, `erase()`, & more, which simplify many operations. Apart from that, vectors can be easily copied from one to another using assignment operator.

```cpp
#include <bits/stdc++.h>
using namespace std;

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

    // remove the last element
    v.pop_back();

    // insert a new element at the beginning
    v.insert(v.begin(), 0);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `0 1 2 3 4`.

4.3 Memory management

Vectors handle memory allocation & deallocation automatically, whereas arrays require manual allocation & deallocation.

```cpp
#include <bits/stdc++.h>
using namespace std;

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

    // no manual reallocation needed
    v.push_back(4);

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `1 2 3 4`.

4.4 Bounds checking

Vector supports bounds checking in `at()` method & throw an `out_of_range` exception if the index is out of bounds, offering a safer way to access elements.

```
#include <bits/stdc++.h>
using namespace std;

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

    try {
        // attempting to access out of range index
        cout << v.at(5) << endl;
    } catch (const out_of_range& e) {
        cout << "Exception: " << e.what() << endl;
    }

    return 0;
}
```

Output:

```
Exception: vector::_M_range_check: __n (which is 5) >= this->size() (which is 4)
```

4.5 Standard Template Library (STL) integration

Vectors are fully compatible with STL algorithms like sort, find, `remove_if`, making it easier to make use of inbuilt functionality of the language.

```
#include <bits/stdc++.h>
using namespace std;

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

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

    // reverse sorted vector
    reverse(v.begin(), v.end());

    for (auto i : v)
        cout << i << " ";
    return 0;
}
```

Output: `5 4 3 2 1`.

4.6 Seamless working with functions

When arrays are passed to a function, a separate parameter for size is also passed whereas in case of passing a vector to a function, there is no such need as vector maintains variables which keeps track of size of container at all times. Also, it can be easily passed & returned as both value & reference.

– Khi mảng được truyền cho một hàm, một tham số riêng cho kích thước cũng được truyền trong khi trong trường hợp truyền một vector cho một hàm, không cần thiết vì vector duy trì các biến theo dõi kích thước của container mọi lúc. Ngoài ra, nó có thể dễ dàng được truyền & trả về dưới dạng cả giá trị & tham chiếu.

```
#include <bits/stdc++.h>
using namespace std;

// take vector as argument as reference but return by value
vector<int> rev(vector<int>& v) {
    reverse(v.begin(), v.end());
    return v;
}

```

```
int main()  {
    vector<int> v1 = {1, 2, 3, 4, 5};
    vector<int> v2 = rev(v1);

    for (auto i: v2) cout << i << " ";
    return 0;
}
```

Output: `5 4 3 2 1`.

5 Pair in C++ STL

In C++, pair is used to combine together 2 values that may be of different data types of same data types as a single unit. The 1st element is stored as a data member with name `first` & the 2nd element as `second`, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // creating a pair of int & string
    pair<int, string> p1 = {1, "Geeks"};

    cout << p1.first << ": " << p1.second;
    return 0;
}
```

Output: `1: Geeks`. Explanation: In this program, we created a pair `p1` of type `int` & `string` with values `{1, "Geeks"}`.

Syntax. The pair container is defined in `<utility>` header file.

```
pair <T1, T2> p;
```

where `T1, T2`: data types of the 1st- & 2nd elements, respectively, `p`: name assigned to the pair.

5.1 Declaration & initialization – Khai báo & khởi tạo

In C++, pair can be declared and initialized in multiple ways as shown below:

1. **Default initialization.** We can declare an empty pair using the declaration

   ```
   pair <T1, T2> p;
   ```

2. **Declaration & initialization with values.** We can initialize a pair directly by assigning values to 1st & 2nd.

   ```
   pair<T1, T2> p = {v1, v2};
   ```

3. **Initialization with `make_pair()`.** We can use `make_pair()` method to initialize pair.

   ```
   pair<T1, T2> p = make_pair(v1, v2);
   ```

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // creating an empty pair
    pair<int, string> p1;

    // insert values using curly braces {}
    pair<int, string> p2 = {1, "Geeks"};

    // insert values using make_pair method
    pair<int, string> p3 = make_pair(2, "ForGeeks");

    cout << p2.first << " " << p2.second << endl;
    cout << p3.first << " " << p3.second;
    return 0;
}
```

Output:

```
1 Geeks
2 ForGeeks
```

Explanation: In the above program, we created 3 pairs:

1. `pair<int, string> p1` creates an empty pair of type `int, string`.
2. `pair<int, string> p2 = {1, "Geeks"}` creates a pair & initializes it with the values `{1, "Geeks"}` using curly braces.
3. `pair<int, string> p3 = make_pair(2, "ForGeeks")` creates a pair & initializes it with the value `{2, "ForGeeks"}` using the `make_pair` method.

All the values should match the type of the pair. Otherwise, a compiler error will be displayed.

Remark 2. *If a pair is not initialized, the compiler automatically assigns the 1st & 2nd members default values according to their types.*

The basic operations on pairs are as follows.

5.2 Accessing values – Tiếp cận giá trị

In pair, 1st & 2nd values are stored as data members. So, we can access them by using their name with `.` operator, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    pair<int, string> p = {1, "Geeks"};

    // accessing the elements of the pair
    cout << p.first << " " << p.second;
        return 0;
}
```

Output: `1 Geeks`. Explanation: In the above program, `p.first` accesses the `first` values of the pair, which is `1`. `p.second` accesses the `second` element of the pair, which is `"Geeks"`.

5.3 Update values – Cập nhật giá trị

We update the elements of pair like accessing elements from pair but in place of access, we just assign new data using `assignment` operator, e.g.:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    pair<int, string> p = {1, "Geeks"};

    // update first and second value of pair
    p.first = 2;
    p.second = "ForGeeks";
    cout << p.first << " " << p.second;
    return 0;
}
```

Explanation: In the above code, `p.first = 2` directly changes the 1st value of the pair to `2`, & `p.second = "ForGeeks"`; directly changes the 2nd value of the pair to `"ForGeeks"`.

5.4 Compare pairs – So sánh cặp

Just like other data types, we can use relational operators with pairs. They initially compare the 1st value. If it does not give result, then 2nd value is compared. The following table describes the behavior of these operators for pairs:

Operator	Description
`==`	Return `true` if both pairs are equal, otherwise `false`
`!=`	Return `true` if pairs are not equal, otherwise `false`
`>`	Return `true` if the LHS pair is greater than the RHS pair, otherwise `false`
`<`	Return `true` if the LHS pair is less than the RHS pair, otherwise `false`
`>=`	Return `true` if the LHS pair is greater than or equal to the RHS pair, otherwise `false`
`<=`	Return `true` if the LHS pair is less than or equal to the RHS pair, otherwise `false`

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

int main() {
    pair<int, int> p1 = {3, 5};
    pair<int, int> p2 = {3, 7};
    pair<int, int> p3 = {2, 5};

    // printing result of comparision
    cout << boolalpha;
    cout << "p1 == p2: " << (p1 == p2) << endl;
    cout << "p1 != p3: " << (p1 != p3) << endl;
    cout << "p1 > p3: " << (p1 > p3) << endl;
    cout << "p1 < p2: " << (p1 < p2) << endl;
    cout << "p1 >= p3: " << (p1 >= p3) << endl;
    cout << "p3 <= p1: " << (p3 <= p1);
    return 0;
}
```

Output:

```
p1 == p2: false
p1 != p3: true
p1 > p3: true
p1 < p2: true
p1 >= p3: true
p3 <= p1: true
```

5.5 Unpacking a pair – Tháo một cặp

We can extract & store the 2 values of the pair in 2 different variables of same type using `tie()` function, e.g.:

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    pair<int, string> p = {1, "Geeks"};

    // variables to store extracted values
    int a;
    string s;

    // extracting values using tie()
    tie(a, s) = p;

    cout << "First value: " << a << endl;
    cout << "Second value: " << s;
    return 0;
}
```

Output:

```
First value: 1
Second value: Geeks
```

Explanation: In this program, the 1st & 2nd value of the pair `p` is extracted into the variable `a, s` using the function `tie()`.

5.6 Some common applications of pairs – Vài ứng dụng phổ biến của cặp

A pair is commonly used for the following purposes:

1. Returning multiple values from functions.
2. Storing key-value pairs in other containers, especially maps.
3. Sorting containers on the basis of multiple criteria.

6 Miscellaneous