

# Phương pháp Lập trình hướng đối tượng

---

Tuần 10  
Template – Exception - STL

# Template – Đặt vấn đề

Các hàm khác nhau cho các kiểu dữ liệu khác nhau

```
int max(int a, int b)
{
    return (a>b) ? a : b;
}
```

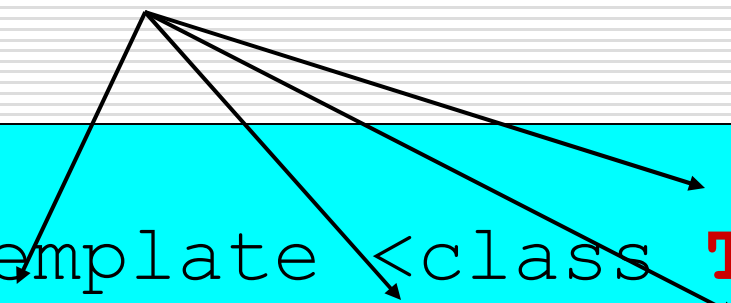
```
float max(float a, float b)
{
    return (a>b) ? a : b;
}
```

Mong muốn viết 1 lần cho bất kỳ kiểu dữ liệu nào:

```
TYPE max(TYPE a, TYPE b)
{
    return (a>b) ? a : b;
}
```

# Template

Kiểu dữ liệu được  
xem là tham số



```
template <class TYPE>
TYPE max(TYPE a, TYPE b)
{
    return (a>b) ? a : b;
}
```

Phép so sánh < phải  
được định nghĩa cho  
kiểu dữ liệu **TYPE**

# Template - Ví dụ

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

template <class T>
T GetMax (T a, T b) {
    T result;
    result = (a>b)? a : b;
    return (result);
}

int main ()
{
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

# Template - Bài tập về nhà: `sort()`

- ❑ Sử dụng việc tham số hóa kiểu dữ liệu cho hàm, viết lại hàm `sort()` cho 1 mảng `a` có `n` phần tử với kiểu dữ liệu bất kỳ cho trước.
- ❑ Cho phép: cài đặt bằng bất kỳ thuật toán sắp xếp nào mà bạn quen thuộc.

# Template - Xử lý bên trong thế nào?

Xét lại hàm `TYPE max (TYPE a, TYPE b)`

□ Nếu chúng ta gọi `max` cho 2 biến kiểu `int`.

Khi đó:

- Compiler phát sinh ra 1 hàm, ví dụ `maxint` và thay thế tất cả các từ `TYPE` trong hàm thành `int`
- Khi chương trình chạy thật sự, hàm `maxint` kia sẽ được chạy thay vì hàm `max` tổng quát

# Template - Ví dụ 2

```
template <class T, class U>
T GetMin (T a, U b) {
    return (a<b?a:b);
}
```

# Class template

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

template <class T>
class mypair
{
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax () ;
};
```

```
template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () '
{
    mypair <int>
        a(100, 75);
    cout << a.getmax();
    return 0;
}
```



# Class template

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

template <class T>
class mypair
{
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax () ;
};
```

```
template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () '
{
    mypair <int>
        a(100, 75);
    cout << a.getmax();
    return 0;
}
```

# Class template

- ❑ Tương tự như function template, lớp **MyPair** trong ví dụ trên là class template.
- ❑ Để có được tên lớp cụ thể, khi khởi tạo lớp là **MyPair<int>** hay **MyPair<PhanSo>**, compiler sẽ phát sinh ra lớp cụ thể tương ứng và thay thế tất cả các tham số kiểu dữ liệu bằng kiểu dữ liệu khởi tạo đó

# Ví dụ

```
template <class TYPE>
class MyArr
{
    public:
        MyArr() ;
        MyArr(int) ;
        ~MyArr() ;
        TYPE& operator[] (unsigned) ;
        const TYPE& operator[] (unsigned) const;
        MyArr<TYPE>& operator=(const MyArr<TYPE>&) ;
    private:
        TYPE* pArr;
};

int main()
{
    MyArr<int> a(100) ;
    MyArr<PhanSo> arrPhanSo(50) ;
}
```

# Class template – Ví dụ 2

```
template <class TYPE, int size>
class List
{
    public:
    ...
    private:
        TYPE arr[size];
};

int main()
{
    List<int, 100> a;
    return 0;
}
```

# Metaprogramming

- ❑ Template trong C++ cho phép trình biên dịch làm việc như là 1 trình thông dịch (interpreter). Các chương trình có thể được thông dịch ngay tại thời điểm biên dịch chương trình.
- ❑ Chẳng hạn, các vòng lặp hay kiểm tra điều kiện có thể được thay bằng cách sử dụng tham số hóa và đệ qui.

# Metaprogramming - ví dụ:

```
template<int N>
class GiaiThua
{
    public:
        enum { value = N * GiaiThua<N-1>::value };
};
class GiaiThua<1>
{
    public:
        enum { value = 1 };
};
void main()
{
    int i=GiaiThua<5>::value;
}
```

Các chương trình viết theo kiểu này không cần phải chạy (execute) mà kết quả sẽ được phát sinh ra ngay thời điểm biên dịch

Kỹ thuật này trở nên vô cùng tiện lợi và mạnh mẽ khi được kết hợp với kiểu lập trình C++ thông dụng.

# Class Template - BubbleSort

```
void bubbleSort(int* data, int N)
{
    for (int i = N - 1; i > 0; --i)
    {
        for (int j = 0; j < i; ++j)
        {
            if (data[j] > data[j+1])
                swap(data[j], data[j+1]);
        }
    }
}
```

```
void bubbleSort(int* data, int N)
{
    for (int j = 0; j < N - 1; ++j)
    {
        if (data[j] > data[j+1])
            swap(data[j], data[j+1]);
    }
    if (N > 2) bubbleSort(data, N-1);
}
```

```
template<int N>
class IntBubbleSort
{public:
    static inline void sort(int* data)
    {    IntBubbleSortLoop<N-1,0>::loop(data);
        IntBubbleSort<N-1>::sort(data);
    }
};
class IntBubbleSort<1>
{public:    static inline void sort(int* data)    { };
```

```
template<int I, int J>
class IntBubbleSortLoop
{private:    enum { go = (J <= I-2) };
public:
    static inline void loop(int* data)
    {    IntSwap<J,J+1>::compareAndSwap(data);
        IntBubbleSortLoop<go?I:0,go?(J+1):0>::loop(data);
    }
};
class IntBubbleSortLoop<0,0>
{public:
    static inline void loop(int*)    { }
};
```



# Swap

```
template<int I, int J>
class IntSwap
{
public:
    static inline void compareAndSwap(int* data)
    {
        if (data[I] > data[J])
            swap(data[I], data[J]);
    }
};
```

# Try-throw-catch: Một số cách xử lý khi phát hiện lỗi thường biết trước đây

- Kết thúc chương trình ngay lập tức
- Trả về giá trị **đặc biệt** thể hiện chương trình đang bị lỗi.
- Trả về 1 giá trị hợp lệ nhưng chuyển chương trình vào trạng thái “**bị lỗi**”
- Gọi 1 hàm đã được cung cấp sẵn trước khi có trường hợp lỗi xảy ra.

# **[1]. Kết thúc chương trình ngay lập tức**

- Đối với đa số các lỗi, chúng ta có thể có rất nhiều cách tốt hơn để xử lý các lỗi này thay vì ngừng ngay chương trình đang chạy.
- Chẳng hạn, 1 chương trình phức tạp không chỉ đơn thuần sẽ kết thúc tất cả khi bị phát sinh 1 lỗi đơn giản mà chúng ta có thể xử lý hay điều chỉnh được.

## [2]. Trả về giá trị lỗi

- Trả về giá trị lỗi không phải lúc nào cũng hợp lý vì có những trường hợp chúng ta không thể trả về giá trị **đặc biệt** nào đó để thể hiện lỗi được.
  - Ví dụ hàm `int tinh()` có thể trả về bất kỳ giá trị `int` nào trong miền giá trị của `int` → không còn giá trị *đặc biệt* để dành cho báo lỗi.
  - Các constructor không có giá trị trả về
  - Phải kiểm tra lỗi mỗi lần gọi hàm này → dễ gây nhầm chán, dễ quên và tăng kích thước chương trình lên đáng kể

```
int main()
{
    //...
    fd=open("file",O_RDWR) ;
    if(fd==-1)
        ...
}
```

# Ví dụ báo lỗi thông qua giá trị trả về

- Phải kiểm tra lỗi mỗi lần gọi hàm này → dễ gây nhầm chán, dễ quên và tăng kích thước chương trình lên đáng kể

```
int main()  
{  
    //...  
    fd=open("file",O_RDWR) ;  
    if (fd==-1)  
        ...  
}
```

### [3]. Trả về kết quả hợp lệ và chuyển trạng thái chương trình sang “bị lỗi”

- Hàm gọi đến hàm bị lỗi có thể không biết hay không để ý rằng chương trình đang được chuyển sang trạng thái “bị lỗi”
- Chẳng hạn, trong ngôn ngữ C, nhiều thư viện chuyển biến toàn cục `errno` sang 1 giá trị nào đó để báo hiệu chương trình đang bị lỗi. Tuy nhiên nhiều chương trình không kiểm tra giá trị này đủ thường xuyên để phát hiện và xử lý.
- Ngoài ra, dùng biến toàn cục cho báo lỗi không phù hợp với các chương trình xử lý song song.

# Exception handling

- Cơ chế thông báo và xử lý lỗi/ngoại lệ cho phép tách biệt thành phần xử lý lỗi với các đoạn mã chương trình bình thường.
- Trong nhiều trường hợp, trong khi chạy, các hàm khi phát hiện ra lỗi nhưng không biết xử lý lỗi các lỗi đó như thế nào. Trái lại các hàm gọi hàm bị lỗi thì biết cách xử lý nhưng lại không thể phát hiện lỗi. Cơ chế exception handling sẽ giúp chương trình xử lý các tình huống này.

# Exception handling

- C++ cung cấp 1 cơ chế để thông báo khi gặp phải lỗi/ngoại lệ: **throw-catch**

```
void f1()
{
    if(...)
        throw "something wrong";
};

int main()
{
    try
    {
        f1();
    }
    catch(char* s)
    {
        cout << "Error: " << s << endl;
    }
    return 0;
};
```



Ví dụ: tính  $x*y/(x-y)$

```
double tinh(double x, double y)
{
    if(x == y)
        throw "divide by zero";
    return x*y/(x-y)
};

int main()
{
    double a, b;
    ...
    try
    {
        a = tinh(a, b);
    }
    catch(char* s)
    {
        cout << "Error: " << s << endl;
    }
    return 0;
};
```

```
class bad_index{};
```

```
class no_memory{};
```

```
void test()
```

```
{
```

```
    if(...)
```

```
    throw bad_index();
```

```
    if(...)
```

```
    throw no_memory();
```

```
}
```

```
int main()
```

```
{
```

```
    ...
```

```
    try
```

```
    {
```

```
        test();
```

```
    }
```

```
    catch(bad_index& bi)
```

```
    { ... }
```

```
    catch (no_memory& nm)
```

```
    { ... }
```

```
}
```

Các lớp exception khác nhau  
dùng để phân biệt các lỗi  
khác nhau

Báo lỗi

Bắt & xử lý lỗi

# Bắt và xử lý lỗi: **catch**

- **Catch** có thể truy xuất đến giá trị của biến lỗi được thông báo nhưng mọi thay đổi (ngay cả khi biến đó được truyền là tham chiếu) đều chỉ xảy ra trên biến tạm cục bộ
- Nếu lệnh **throw** không có giá trị trả về được gọi trong block **try{ }** thì block **catch** sẽ không được xử lý. Thay vào đó, chương trình sẽ bị terminate

# Bắt và xử lý lỗi: **catch**

- Phải có ít nhất 1 **catch** block theo liền sau **try{ }**
- Bắt đầu bằng từ khóa **catch** và theo sau là các kiểu dữ liệu & biến nhận giá trị trả về của **throw** trong **try{ }**.
- **Catch** chỉ được xem xét thực hiện nếu trong trong **try{ }** có xảy ra lệnh **throw**.

# Các trường hợp `catch` được kích hoạt

```
void test()
{
    try
    {
        throw E();
    }
    catch (H)
    {
        //trong trường hợp nào chương trình chạy vào đây?
    }
}
```

1. H có cùng kiểu với E
2. H là lớp cơ sở của E
3. H & E là con trỏ và (1) hoặc (2) thỏa
4. H là tham chiếu và (1) hoặc (2) thỏa

# Bắt và xử lý lỗi: **catch**

- Nếu chúng ta muốn bắt tất cả các exception bất kể kiểu dữ liệu của chúng là gì, ta dùng: **catch (...)**. Lệnh **catch** này thường được dùng ở cuối các khối lệnh **catch** có chỉ định kiểu dữ liệu khác để bắt các lỗi còn sót lại.

# Bắt và xử lý lỗi: **catch**

- Trong khối lệnh của **catch**, chúng ta có thể **throw** 1 exception ra cấp cao hơn. Có 2 loại **throw** từ bên trong **catch**:
  - **Throw** kèm với 1 operand với 1 kiểu dữ liệu nào đó
  - **Throw** không có operand (đồng nghĩa **throw** lại lỗi cần xử lý vừa rồi cho cấp cao hơn)

# Các trường hợp đặc biệt sau khi **throw**

- Nếu không tìm thấy **catch** block phù hợp kiểu với operand của **throw** thì quá trình **unwinding stack** sẽ được thực hiện cho đến khi tìm được catch block phù hợp
- Nếu vẫn không có catch nào hết thì chương trình sẽ bị terminate



# Khai báo các kiểu **throw** cho 1 hàm

- Default: 1 hàm có thể **throw** bất kỳ lỗi gì
- Nếu muốn chỉ định các kiểu exception sẽ được **throw** từ 1 hàm cho trước, chúng ta viết chúng ở cuối của khai báo hàm

Ví dụ:

```
int tinh(int x) throw(char, int);
```

- Nếu viết: `int tinh(int x) throw();` hàm này sẽ không throw exception nào.

# Một số vấn đề của exception handling

- Có thể gây ra rò rỉ bộ nhớ nếu không xử lý các vùng nhớ hợp lý
- Exception handling không hoạt động tốt với template vì các hàm **template** có thể **throw** các lỗi khác nhau dựa vào từng loại tham số hóa cụ thể của hàm đó.

# Ví dụ: vấn đề rò rỉ bộ nhớ

```
int doSomething(int size)
{
    int* arrTest;
    arrTest = new int[size];
    ...
    if (condition)
        throw bad_exception();
    ...
    delete [] arrTest;
    return 0;
}
```

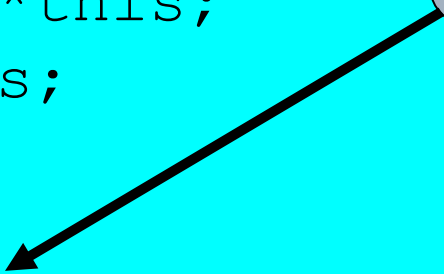
# Một cách giải quyết

```
int doSomething(int size)
{
    arrTest = new int[size];
    ...
    try
    {
        if (condition)
            throw bad_exception();
    }
    catch (bad_exception& e)
    {
        delete [] arrTest;
        throw;
    }
}
```

# Một ví dụ khác

```
MyStr& MyStr::operator=const MyStr& src)
{
    if (this == &src)
        return *this;
    delete [] s;
    if (src.s)
    {
        s = new char [strlen(src.s) + 1];
        strcpy (s, src.s);
    }
    else s = NULL;
    return *this;
}
```

**throw an error**  
(hết bộ nhớ)



# Ví dụ: sau khi đã sửa

```
MyStr& MyStr::operator=const MyStr& src)
{
    if (this == &src)
        return *this;
    char* tmpS;
    if (src.s)
    {
        tmpS = new char [strlen(src.s) + 1];
        strcpy (tmpS, src.s);
    }
    else tmpS = NULL;
    delete [] s;
    s = tmpS;
    return *this;
}
```

# Một số câu hỏi

- Thông báo lỗi từ **constructor**?
- Thông báo lỗi từ initialization list?
- Tìm hiểu các khối lệnh **try{ }** lồng nhau
- Tính kế thừa và đa xạ của các lớp **exception()**
- Tại sao hàm **pop()** trong lớp Stack nên viết với kiểu trả về là void: **void pop()**

# Thư viện chuẩn của C++

<algorithm>	<b>&lt;ios&gt;</b>	<map>	<stack>
<bitset>	<iosfwd>	<memory>	<stdexcept>
<complex>	<b>&lt;iostream&gt;</b>	<new>	<streambuf>
<deque>	<b>&lt;istream&gt;</b>	<numeric>	<b>&lt;string&gt;</b>
<b>&lt;exception&gt;</b>	<iterator>	<b>&lt;ostream&gt;</b>	<typeinfo>
<b>&lt;fstream&gt;</b>	<limits>	<queue>	<utility>
<functional>	<list>	<set>	<valarray>
<b>&lt;iomanip&gt;</b>	<locale>	<sstream>	<b>&lt;vector&gt;</b>



# Thư viện STL

- Thư viện khuôn mẫu chuẩn (Standard Template Library – STL) cung cấp các containers (cấu trúc dữ liệu), giải thuật và iterators đã được định nghĩa sẵn để có thể phát triển các ứng dụng khác nhau trên C++
- STL được đề xuất bởi Alexander Stepanov nhằm xây dựng tư tưởng lập trình tổng quát.
- Các khái niệm của STL được phát triển độc lập với C++

# Thư viện STL (tt)

- Các thành phần của STL không là OOP mà là lập trình generic
- Hầu hết các containers đều được thiết kế và cài đặt dựa trên template để có thể chứa bất kỳ kiểu dữ liệu gì.
- Dễ sử dụng, mạnh mẽ và hiệu quả
- 2 containers phổ biến nhất của STL là **vector** và **string**

# Các thành phần chính của STL

STL bao gồm 3 thành phần chính:

- Containers: các cấu trúc dữ liệu đã được định nghĩa sẵn dựa trên template.
- Iterator: tương tự như 1 con trỏ. Dùng để truy xuất các thành phần của container
- Algorithm: gồm các thuật toán phổ biến như sắp xếp, tìm kiếm và các thuật toán thao tác trên dữ liệu...

# STL containers

Các container có thể được gom nhóm như sau

- Sequence containers (container chuỗi)
- Associative containers (container liên kết)
- Ordered sets (các tập hợp có thứ tự)
- Container adapters
- Các containers chuyên biệt khác

# Sequence containers

- Các containers này lưu trữ các thành phần dữ liệu theo 1 chuỗi tuyến tính (sequence)
- Sequence containers gồm:
  - vector
  - deque
  - list

# Sequence containers: **vector**

**vector** có các tính chất sau

- Sử dụng cấu trúc mảng động (dynamic array), cho phép truy xuất đến các phần tử bất kỳ 1 cách nhanh chóng.
- Thêm và xóa phần tử cuối nhanh.
- Có kiểm tra việc truy cập ngoài mảng.

# Sequence containers: **deque**

**deque** có các tính chất sau

- Tương tự **vector**: sử dụng mảng động để thể hiện
- Thêm và xóa phần tử đầu hay cuối nhanh (hơi chậm hơn **vector** 1 chút do cho phép xử lý từ 2 đầu)

# Sequence containers: **list**

**List** có các tính chất sau:

- Sử dụng danh sách liên kết đôi để thể hiện
- Không cho phép truy xuất 1 phần tử bất kỳ trong **list**
- Thêm/xóa tại bất kỳ phần tử nào: nhanh



# Associative containers

- Các associative containers chứa các cặp khóa/giá trị:
  - Truy xuất đến từng giá trị thông qua khóa.
  - Các phần tử được sắp xếp theo khóa.
  - Thường được cài đặt bằng cây nhị phân cân bằng.
- Có 2 container thuộc loại này
  - **Map**
  - **Multimap**

# Associative containers

- **map** cung cấp khả năng truy xuất đến các phần tử thông qua khóa với bất kỳ kiểu dữ liệu gì. Map tổng quát hóa từ ý tưởng truy xuất đến phần tử chỉ thông qua chỉ số **int** trong kiểu dữ liệu **vector**.
- **multimap** tương tự như **map** nhưng nó cho phép 1 khóa có thể liên kết với nhiều hơn 1 phần tử.

# Ordered sets (tập hợp có thứ tự)

- Đôi khi nhóm các containers này được xếp vào loại associative containers. Có các tính chất sau:
  - Lưu trữ các phần tử theo thứ tự
  - Thường cài đặt sử dụng cây nhị phân cân bằng.
  - Tuy nhiên, các tập hợp có thứ tự này không có các phép toán trên tập hợp như (phép hội, phép giao...)

# Ordered sets (tập hợp có thứ tự)

- **set**

- sắp xếp các phần tử theo thứ tự khi chúng được thêm vào tập hợp
- Mỗi tập hợp chỉ chứa các phần tử không trùng nhau.

- **multiset** tương tự như **set** nhưng cho phép nhiều entry hơn cho mỗi giá trị.

# Container adapters

- Các container này được xây dựng dựa trên các container có sẵn khác và chỉ áp dụng thêm các luật khác nhau trong việc truy xuất đến các phần tử của chúng.
- Bởi vì áp dụng các luật khác nhau trên việc truy xuất đến các phần tử nên các container này không có **iterator**.

# Container adapters

- **stack** chỉ cho phép truy xuất theo kiểu LIFO (Last In, First Out).
- **queue** chỉ cho phép truy xuất theo kiểu FIFO (First In, First Out).
- **priority\_queue** luôn luôn trả về phần tử có độ ưu tiên (priority) cao nhất.

# Các containers chuyên biệt khác

- Các container khác được thiết kế chuyên biệt thông qua: kiểu dữ liệu đặc biệt hay các phương thức hỗ trợ đặc biệt v.v...
- **string**: thể hiện chuỗi. Tương tự như **vector<char>** nhưng có các hàm chức năng hữu ích khác.

# Các containers chuyên biệt khác (tt)

- **bitset**
  - Cấu trúc dữ liệu lưu trữ các bit một cách hiệu quả.
  - Có các phương thức thực thi chuyên biệt trên bit.
- **valarray** là một cài đặt đặc biệt hiệu quả cho mảng. Tuy nhiên nó không có tất cả các phương thức chuẩn như các container khác.



# Các hàm thành viên của STL

- Mọi container đều có
  - `default copy constructor, destructor`
  - `empty`
  - `max_size, size`
  - Các toán tử: `= < <= > >= == !=`
  - `swap`
- Chỉ có trong sequence, associative containers và ordered sets
  - `begin, end`
  - `rbegin, rend`
  - `erase, clear`

# Giới thiệu: **iterator**

- **iterator** tương tự như con trỏ
  - Trỏ tới các phần tử trong một container
- Các toán tử của **iterator**
  - **\*** truy nhập phần tử được trỏ tới
  - **++** trỏ tới phần tử tiếp theo
  - **begin()** trả về iterator trỏ tới phần tử đầu tiên
  - **end()** trả về iterator trỏ tới phần tử cuối container

# Các loại **iterator**

- **Input:** đọc các phần tử từ một container, hỗ trợ `++`, `+=` (chỉ tăng dần).  
Chẳng hạn: **`istream_iterator`**
- **Output:** ghi các phần tử vào container, hỗ trợ `++`, `+=` (chỉ tăng dần).  
Chẳng hạn: **`ostream_iterator`**
- **Forward:** chẳng hạn **`hash_set<T>`** iterator
  - Kết hợp input iterator và output iterator
  - Multi-pass: duyệt nhiều lần

# Các loại `iterator` (tt)

- **Bi-directional**: tương tự forward nhưng còn có thể giảm (`--`, `--=`)  
Chẳng hạn: `list<T>` iterator
- **Random access**: tương tự bi-directional nhưng còn có khả năng truy xuất đến 1 phần tử bất kỳ  
Chẳng hạn: `vector<T>` iterator

# Các phép toán trên `iterator`

- Input iterator: `++`, `=*p`, `->`, `==`, `!=`
- Output iterator: `++`, `*p=`, `p=p1`
- Forward iterator: của input và output iterator
- Bidirectional iterator: các toán tử của forward và `--`
- Random access: các toán tử cho bidirectional và `+`, `+=`, `-`, `-=`, `>`, `>=`, `<`, `<=`, `[]`

# Container hỗ trợ các loại `iterator`

- Sequence containers
  - `vector`: random access
  - `deque`: random access
  - `list`: bidirectional
- Associative containers: bidirectional
- Ordered sets: bidirectional
- Container adapters: không hỗ trợ `iterator`
- `Bitset` và `valarray`: không hỗ trợ `iterator`