

# Using C++ for competitive programming

---

Louis Jachiet

# A quick introduction to C++

---

# A brief history of C++

## Before C++ there was C

- C first released in 1972
- Multiple releases (1978, 89, 99, 11, 17, 27!)
- Bare metal
- Procedural language

## Introduction of C++

- work on C++ started in 1979
- mostly a superset of C
- C with classes
- ... but contains more than just classes !
- numerous releases (1985, 89, 98, 03, 11, 14, 17, 20, 23)

# Why C++ now?

## C++ is multi-paradigm:

- procedural and fast (just like C)
- object oriented (classes and hierarchy)
- generic (template system)
- functional
- exceptions

**All these features make C++ complicated, we will stick to all small superset of C**

- a part of the Standard Template Library
- references, foreach, auto, const, etc.

# The Standard Template Library (STL)

---

**The STL contains a lot of useful material that is:**

- sometimes hard to re-implement
- saves time
- prevents bugs

The STL contains a lot of useful material that is:

- sometimes hard to re-implement
  - saves time
  - prevents bugs
- 

Some simple examples:

- `max(a,b)`, `min(a,b)` // *returns max/min*
- `swap(a,b)` // *exchange a and b, uses references*
- `clamp(v,a,b)` // *closest value to v in [a;b]*

## Another example, storing a pair of objects

In C to store a pair we need to create a struct

```
typedef struct pair {  
    typeA first ;  
    typeB second ;  
} pair ;
```

Of course we don't necessarily need a struct...



In C++, the STL directly offers a pair data structure

- `pair<A,B>` to create a type pair using type A and B
- `make_pair(a,b)` to create a pair using a and b
- `p.first` and `p.second` to access the components
- all comparisons are implemented, e.g. `p1 == p2` is equivalent to `p1.first == p2.first && p1.second == p2.second`

In C++, the STL directly offers a pair data structure

- `pair<A,B>` to create a type pair using type A and B
  - `make_pair(a,b)` to create a pair using a and b
  - `p.first` and `p.second` to access the components
  - all comparisons are implemented, e.g. `p1 == p2` is equivalent to `p1.first == p2.first && p1.second == p2.second`
- 

Pairs are really useful to simplify your code!

In C++, the STL directly offers a “complex” template to represent complex numbers

- `complex<T>` to create a complex number type using type T
- `complex<T>(42,33)` to create the complex  $42 + 33i$
- `c.real()` and `c.imag()` to access the components
- `norm(c)`, `abs(c)`, `conj(c)`, `arg(c)`, `proj(c)` to the (squared) magnitude, conjugate, phase, normalization

In C++, the STL directly offers a “complex” template to represent complex numbers

- `complex<T>` to create a complex number type using type T
- `complex<T>(42,33)` to create the complex  $42 + 33i$
- `c.real()` and `c.imag()` to access the components
- `norm(c)`, `abs(c)`, `conj(c)`, `arg(c)`, `proj(c)` to the (squared) magnitude, conjugate, phase, normalization

---

Complex numbers are incredibly useful for geometry!

# The numeric library

**Lots of useful numeric algorithms:**

```
gcd(42,12), lcm(42,12); // pgcd / ppcm in French
```

```
abs(-2) ; // 2
```

```
fmod(5.1,3); // 2.1
```

```
exp(2), exp2(4), log(42), log2(42), log10(42) ;
```

```
sqrt(2) ;
```

```
sin(2), cos(2), tan(2) ; // works with radians
```

```
ceil(2.3), floor(2.4) ;
```

# The Standard Template Library (STL)

---

Containers and iterators

Many structures in the STL are **containers**. A container is a structure that stores a collection of objects and can be manipulated through **iterators**.

Many structures in the STL are **containers**. A container is a structure that stores a collection of objects and can be manipulated through **iterators**.

---

**Iterators are a kind of generalized pointers:**

- get the value pointed by the iterator with `*it`
- modify the value with `*it = otherVal`
- move to the next element with `++it`
- move the previous element with `--it`
- compare them (`it==it2`, `it!=it2`)



## Containers often have the following functions:

- `myCon.begin()` first element
- `myCon.end()` iterator **after** the last element
- `myCon.rbegin()`, `myCon.rend()` iterators in the reverse direction
- `myCon.size()`, `myCon.empty()` get size/emptiness
- `for(auto & x : myCon)` to loop over elements

## Containers also allow for generic algorithms, e.g.:

- `min/max_element(it_from, it_to)` to get min/max
- `accumulate(it_from, it_to, 0)` to get a sum
- `sort(it_from, it_to)` to sort
- `fill(it_from, it_to, value)` to set to a value
- `iota(from, to, start)` fills an array with consecutive values starting from the value start

## Containers also allow for generic algorithms, e.g.:

- `min/max_element(it_from, it_to)` to get min/max
- `accumulate(it_from, it_to, 0)` to get a sum
- `sort(it_from, it_to)` to sort
- `fill(it_from, it_to, value)` to set to a value
- `iota(from, to, start)` fills an array with consecutive values starting from the value start

## The above functions also work with pointers

```
int t[100];  
fill(t,t+100,42); // first included, last not included  
sort(t,t+100);
```

## Dynamic arrays: `vector<T>`

In C++, a “vector” is a dynamic array:

- `vector<T>` for the type
- `vector<T>()` to create an empty array
- `vector<T>(k)` to create an array of size `k`
- `myVector.push_back(a)` to append `a` at the end
- `myVector.pop_back()` to remove the last element
- `myVec1+myVec2` to concatenate
- `myVec1.size()` to get the current size
- `myVec[a]` to get the element `a` (0-indexed just like arrays)

## Dynamic arrays: `vector<T>`

In C++, a “vector” is a dynamic array:

- `vector<T>` for the type
- `vector<T>()` to create an empty array
- `vector<T>(k)` to create an array of size `k`
- `myVector.push_back(a)` to append `a` at the end
- `myVector.pop_back()` to remove the last element
- `myVec1+myVec2` to concatenate
- `myVec1.size()` to get the current size
- `myVec[a]` to get the element `a` (0-indexed just like arrays)

---

Vectors are typically useful when you don't know the size of the data.

## Dynamic arrays: `vector<T>`

In C++, a “vector” is a dynamic array:

- `vector<T>` for the type
- `vector<T>()` to create an empty array
- `vector<T>(k)` to create an array of size `k`
- `myVector.push_back(a)` to append `a` at the end
- `myVector.pop_back()` to remove the last element
- `myVec1+myVec2` to concatenate
- `myVec1.size()` to get the current size
- `myVec[a]` to get the element `a` (0-indexed just like arrays)

---

Vectors are typically useful when you don't know the size of the data.

Vectors also support **iterators!**

To implement a **stack**, a vector has everything you need:

```
vector<int> v ;  
v.push_back(42); // add 42  
v.push_back(23); // add 23  
v.back(); // 23  
v.pop_back();  
v.back(); // 42
```

Vectors also have `push_front`, `pop_front` and `front` but those are inefficient, **don't use them!** If you need them (e.g. for a queue), use **deque**.

**A deque (double-ended queue) can be used just like a vector:**

```
deque<int> v ;  
v.push_back(42); // add 42  
v.push_front(17); // add 17  
v.push_front(23); // add 23 at the front  
v.back(); // 42  
v.front(); // 23  
v[0]; // 0 is front so 23  
v[1]; // 1 is middle so 17  
v.pop_back();  
v.back(); // 17
```

The price of a deque is that it is less efficient (higher constant)...



## Set (Ordered / Unordered)

To represent a set efficiently you either need a **hash table** or a **binary search tree**

## Set (Ordered / Unordered)

To represent a set efficiently you either need a **hash table** or a **binary search tree**

```
// unordered_set<T> is implemented as a hash set  
unordered_set<int> mySet; // int is hashable
```

```
mySet.insert(42);  
mySet.insert(17);  
mySet.insert(42);  
mySet.size(); // 2  
for(int i : mySet) {  
    printf("%d\n",i); // order is not guaranteed  
}  
mySet.erase(42);
```

## Set (Ordered / Unordered)

To represent a set efficiently you either need a **hash table** or a **binary search tree**

```
// set<T> is implemented as a binary search tree
set<pair<int,int>> mySet; // int type is comparable,
    //             hence pair<int,int> is
mySet.insert(make_pair(42,1));
mySet.insert(make_pair(17,0));
mySet.insert(make_pair(42,1));
mySet.size(); // 2
for(auto i : mySet) {
    printf("%d %d\n",i.first,i.second); // 17 then 42
}
mySet.erase(make_pair(42,1));
```

## Set (Ordered / Unordered)

To represent a set efficiently you either need a **hash table** or a **binary search tree**

```
struct Hasher {  
    size_t operator()(const pair<int,int> & p) const {  
        return p.first ^ (7*p.second) ;  
    }  
};  
unordered_set<pair<int,int>,Hasher> mySet;  
// pair<int,int> is not natively hashable so we need to  
// create a hash ``functor'' i.e. a type as above...  
// QUITE COMPLICATED OVERALL!
```

Hash set have better asymptotic complexities but the advantage is not that big in practice... Use **set<T>** on complex types.

**C++ sets are more powerful than just maintaining sets:**

```
set<myType> s;  
*s.begin(); // the minimal value  
*s.rbegin(); // the maximal value  
s.find(v) ; // return an iterator pointing to v or s.end  
s.lower_bound(v); // return an iterator pointing to  
// first element that is greater or equal to v  
s.upper_bound(v); // return an iterator pointing to  
// first element that is strictly greater than v
```

## C++ sets are more powerful than just maintaining sets:

```
set<myType> s;  
*s.begin(); // the minimal value  
*s.rbegin(); // the maximal value  
s.find(v) ; // return an iterator pointing to v or s.end  
s.lower_bound(v); // return an iterator pointing to  
// first element that is greater or equal to v  
s.upper_bound(v); // return an iterator pointing to  
// first element that is strictly greater than v
```

## Priority queues

STL has a dedicated `priority_queue` that implements a max-heap but you can use the (slightly) less efficient sets.

**C++ sets can be used to maintain associative arrays:**

```
map<keyType, valueType> myMap;  
myMap[key1] = value1;  
myMap[key2] = value2;  
myMap[key1] = value3;  
myMap[key1] ; // returns value3;  
myMap[key3] ; // creates an entry with default value
```

**C++ sets can be used to maintain associative arrays:**

```
map<keyType, valueType> myMap;  
myMap[key1] = value1;  
myMap[key2] = value2;  
myMap[key1] = value3;  
myMap[key1] ; // returns value3;  
for(auto & t : myMap) {  
    t.first ;// cannot modify t.first  
    t.second = something ;  
}
```

Shares similarities with a `set<pair<keyType, valueType>>` thus `begin`, `find(key)`, `lower_bound(key)`, etc. return iterators towards pairs.



# Fast manipulation of bits

## The `vector<bool>`

In C++, a `bool` is usually stored as 8 bits. `vector<bool>` is optimized to store each bit as a bit.

## The `bitset<N>`

STL has a `bitset` of a fixed sized `N` (known at compile time). This create a structure that is efficiently stored and supports fast logical operations: and for intersection, or for union, etc.

## string

```
string a = "Hello!"; // convert from const char *
```

```
a < b ; // applies lexicographical comparison
```

```
"a" == "b" ; // comparison of pointers => probably an error!
```

```
string("a") == string("b") ; // compares strings
```

```
string("a") + a ; // works as expected
```

```
a.substr(pos, len) ; // extract sub string
```

```
a[0] ; // access individual chars
```

```
a.c_str() ; // access internal char *
```

```
a.find("pattern") ; // look for first occurrence
```

```
a.find("pat", 42) ; // first occurrence after position=42
```

```
unordered_set<string> mySet ; // strings are hashable too!
```

## Using the STL in practice

---

`https://en.cppreference.com/w/`

## The headers you need to include:

- `#include <algorithm>` for most algorithms, pairs, etc.
- `#include <numeric>` for everything numerical
- each container has its own include, e.g. `#include <vector>`
- all the STL lies in the namespace `std`, so put `using namespace std;`

## Storing graphs

```
const int NB_NODES_MAX = 1000*1000;
vector<pair<int,long long> > nxt[NB_NODES_MAX];
// pair of (next node, weight)
int nbEdges,nbNodes ;

void readGraph() {
    scanf("%d %d",&nbNodes,&nbEdges);
    for(int i = 0 ; i < nbEdges ; i++) {
        int a,b,p ;
        scanf("%d %d %d",&a,&b,&p);
        nxt[a].push_back({b,p});
        nxt[b].push_back({a,p}); // if undirected
    }
}
```

# Dijkstra

```
long long dist[NB_NODES_MAX];  
//...  
fill(dist,dist+NB_NODES_MAX,INF);  
set<pair<long long,int>> p_queue; // (weight, node)  
p_queue.insert(make_pair(0,start_node));  
dist[start_node] = 0;  
while(!p_queue.empty()) {  
    auto [node_dist, node] = *p_queue.begin() ; // c++17  
    p_queue.erase(p_queue.begin());  
    for(auto v : nxt[node])  
        if(node_dist + v.second < dist[v.first]) {  
            p_queue.erase(make_pair(dist[v.first],v.first));  
            dist[v.first] = node_dist + v.second;  
            p_queue.insert(make_pair(dist[v.first],v.first));  
        }  
}
```

# Dijkstra

```
long long dist[NB_NODES_MAX];
typedef pair<long long,int> pill;
fill(dist,dist+NB_NODES_MAX,INF);
// priority queue in reverse order
priority_queue<pill,vector<pill>,greater<pill>> p_queue;
p_queue.push(make_pair(0,start_node));
dist[start_node] = 0;
while(!p_queue.empty()) {
    auto [node_dist, node] = p_queue.top() ;
    p_queue.pop();
    if(dist[node] == node_dist) // lazy priority queue
        for(auto v : nxt[node])
            if(node_dist + v.second < dist[v.first]) {
                dist[v.first] = node_dist + v.second;
                p_queue.push(make_pair(dist[v.first],v.first));
            }
}
```