C++ for programming competitions

a.k.a. C+/-

Why C++

Over C: built-in powerful structures and algorithms

Over 'x': speed (and availability in competition systems)

C+/-

- *reuse C++ object-oriented structures*
- *C imperative programming style*

From C to C++

```c
#include <stdio.h>
int main() {
    int x, y;
    scanf("%d%d", &x, &y);
    printf("%d + %d = %d", x, y, x+y);
    return 0;
}
```

```cpp
#include <iostream>
int main() {
    int x, y;
    std::cin >> x >> y;
    std::cout << x << " + " << y << " = " << x+y;
    return 0;
}
```

Namespaces

C++ code is normally organized in namespaces.

Namespaces are distant cousins of Java packages.

`std` stands for C++ standard library.

Multiple headers define structures under `std`.

```
#include <iostream>
#include <string>
int main() {
    std::string s = "Hello World!";
    std::cout << s;
}
```

The scope operator (`::`)

Access (static) structures inside a scope.

E.g., a stream (cout), class (string) or function in a namespace

```
#include <iostream>
#include <string>
int main() {
    std::string s = "Hello World!";
    std::cout << s;
}
```

Import the namespace instead of typing it all the time.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s = "Hello World!";
    cout << s;
}
```

May lead to double declaration; but very unlikely for competitions.

The `#include` “cheat”

A header can `#include` other headers.

Which also become accessible in the main code. (like in Python)

Competitors commonly use:

```
#include <bits/stdc++.h>
```

- 😀 includes basically all headers you may need
- 😑 not standard (GCC-only)
- 😑 (slightly) slower compilation time

TL;DR 🙈 do not use outside competitions

The member access operator (.)

In C, you used it to access `struct` members.

Now you also use it for classes' methods and fields.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
  string s;
  cin >> s;
  printf("%s", s.c_str()); // string object to "C string" (char*)
}
```

Conditionals

```
int x;
if (x != 2 || 3) { // THIS BAD! (but compiles... silent error)
    // solve for x
}
```

```
int x;
if (x != 2 || x != 3) {
    // solve for x
}
```

The kind of silly mistake that may happen.

And cost you **precious minutes** of debugging.

TIP:

- *we are all likely to repeat mistakes*
- *take (mental) note of your recurrent ones*
- *at least you know where to start debugging*

Loops

```
do {
    int x, flag;
    cin >> x;
    // solve for x & flag
} while (flag > 0);
```

```
int x;
while (cin >> x) {
    // solve for x
}
```

```
long long x = 1;
for (int i = 0; i <= N; i++) {
    // solve for x & i
}
```

Be mindful of sizes

```
#include <iostream>
#include <climits>
using namespace std;
int main(){
  cout << "char -------------> " << CHAR_BIT << "\n";
  cout << "short ------------> " << sizeof(short)*CHAR_BIT << "\n";
  cout << "int/float --------> " << sizeof(int)*CHAR_BIT << "\n";
  cout << "long/double ------> " << sizeof(long)*CHAR_BIT << "\n";
  cout << "long long/double -> " << sizeof(long long)*CHAR_BIT << "\n";
  return 0;
}
```

`long – long int`
`long long – long long int`

Need more bits? BigInt

IO

Get your input and output right first.

They may inform how to structure your code.

Input

```
1    2 name
...
```

```
1 2
name
...
```

```
1
2
name
...
```

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
  int x, y;
  string s;
  cin >> x >> y >> s; // skip all blanks between values
  printf("%d %d %s", x, y, s.c_str());
}
```

Not specific how many items in one line?

Get the whole line and split the string.

```
1 b c
1 a d r
...
```

```
//...
int main() {
  string s;
  getline(cin, s);
  vector<string> items = split_str(s, ' '); // needs to be implemented
  for (int i = 0; i < items.size(); i++) {
    cout << items[i];
  }
  return 0;
}
```

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
using namespace std;
vector<string> split_str(string s, const char delim) {
    vector<string> out;
    istringstream ss(s);
    string e;
    while (getline(ss, e, delim)){
        out.push_back(e);
    }
    return out;
}
```

Input synchronization

Disable synchronization safety nets:

```
ios::sync_with_stdio(false);
cin.tie(NULL);
```

Give it a ride:

https://www.spoj.com/problems/INTEST/

With synchronization: 1.48

```
#include <iostream>
using namespace std;
int main() {
    int n,k,e;
    int t = 0;
    cin >> n >> k;
    for (auto i = 0; i < n; i++){
        cin >> e;
        if (!(e%k)) t++;
    }
    cout << t;
    return 0;
}
```

Without synchronization: 0.41

```cpp
#include <iostream>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n,k,e;
    int t = 0;
    cin >> n >> k;
    for (auto i = 0; i < n; i++){
        cin >> e;
        if (!(e%k)) t++;
    }
    cout << t;
    return 0;
}
```

With optimized reading: 0.16

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

void scan_int(int &number) {
    bool n = false;
    register int c;
    number = 0;
    c = getchar();
    if (c=='-') { n = true; c = getchar(); }
    for (; (c>47 && c<58); c=getchar())
        number = number *10 + c - 48;
    if (n)
        number *= -1;
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
```

A solution template

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    //solution
}
```

Unknown amount of data?

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    // process until end of input
    while (cin >> x) {
        // solution
    }

    return 0;
}
```

Got an input file?

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    freopen("input.txt", "r", stdin);
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    //solution

    return 0;
}
```

Shortening code

Speed up writing.

But be careful to not confuse yourself.

using

Create definition shortcuts.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
using ld = long double;
using vi = vector<ll>;
int main() {
    //freopen("input.txt", "r", stdin);
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    //solution

    return 0;
}
```

Similar to `typedef` but c++ friendly.

Macros

Literal code substitutions.

```
#define REP(i,a,b) for (auto i = a; i < b; i++)
```

Consider this previous solution to INTEST.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n,k,e;
    int t = 0;
    cin >> n >> k;
    for (auto i = 0; i < n; i++){
        cin >> e;
        if (!(e%k)) t++;
    }
    cout << t;
    return 0;
}
```

Now with a loop macro now.

```cpp
#include <bits/stdc++.h>
using namespace std;
#define REP(i,a,b) for (auto i = a; i < b; i++)
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n,k,e;
    int t = 0;
    cin >> n >> k;
    REP(i,0,n) {
        cin >> e;
        if (!(e%k)) t++;
    }
    cout << t;
    return 0;
}
```

Macros can be tricky

What is the output?

```
#include <bits/stdc++.h>
using namespace std;
#define POW(a) a*a
int main() {
    cout << POW(2);
    return 0;
}
```

And now?

```
#include <bits/stdc++.h>
using namespace std;
#define POW(a) a*a
int main() {
    cout << POW(2+3);
    return 0;
}
```

It is 11!

```cpp
#include <bits/stdc++.h>
using namespace std;
#define POW(a) a*a
int main() {
    cout << 2+3*2+3;
    return 0;
}
```

So be careful! This is a better macro.

```
#include <bits/stdc++.h>
using namespace std;
#define POW(a) (a)*(a)
int main() {
    cout << 2+3*2+3;
    return 0;
}
```

Data structures

It is not only about features.

Be mindful of the performance.

Pairs & Tuples

Support structures.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    pair<int,int> p = {1,2};
    cout << p.first << ' ' << p.second << '\n';

    pair<int,int> p2 = {3,4};
    p.swap(p2);
    cout << p.first << ' ' << p.second << '\n';

    tuple<int,int,int> t = {1,2,3};
    cout << get<0>(t) << get<1>(t) << get<2>(t) << '\n';
}
```

Vectors

Efficient access. Slow removal.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v;
    int value = 1;
    int index = 1;

    // size - common to all data structures
    auto size = v.size();

    // insert
    v.insert(v.begin() + index, value); // head or index | O(n)
    v.push_back(value);                 // tail          | O(1)

    // access
    auto v_head = v.front();     // head             | O(1)
    auto v_index = v.at(index); // index (or v[i]) | O(1)
    auto v_tail = v.back();      // tail             | O(1)
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    vector<int> v = {1,2,3};
    int index = 1;

    // iterate
    for(auto &i: v) {
        cout << i << ' ';
    }
    cout << '\n';

    // remove
    v.erase(v.begin() + index); // head or index | O(n)
    v.pop_back();               // tail          | O(1)

    // clear - common to all data structures
    v.clear(); // O(n)
}
```

Lists

Efficient sorting and good overall.

But bad for repetitive access to middle.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    list<int> l;
    int value = 1;
    int index = 1;

    // insert
    l.push_front(value);                // head  | O(1)
    l.insert(l.begin(), index, value); // index | O(n)
    l.push_back(value);                 // tail  | O(1)

    // access
    auto v_head = l.front();             // head  | O(1)
    auto v_value = *next(l.begin(), 1); // index | O(n)
    auto v_tail = l.back();              // tail  | O(1)
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    list<int> l = {1,2,3};
    int index = 1;

    // iterate
    for(auto &i: l) {
        cout << i << ' ';
    }
    cout << '\n';

    // remove
    auto pos = next(l.begin(),index);
    l.pop_front(); // head  | O(1)
    l.erase(pos);  // index | O(1) + O(n)
    l.pop_back();  // tail  | O(1)
}
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    list<int> l = {1,2,3};
    list<int> l2;

    l2.splice(l2.begin(), l);// transfer elements between list
    l.remove(1);             // remove an element by value
    l.unique();              // remove duplicates
    l.merge(l2);             // merge two sorted lists
}
```

Stacks & Queues

Efficient LIFO & FIFO.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    stack<int> s;
    queue<int> q;
    int value = 1;

    // insert
    s.push(value); // head | O(1)
    q.push(value); // head | O(1)

    // access
    int top = s.top();    // head | O(1)
    int head = q.front(); // head | O(1)
    int tail = q.back();  // tail | O(1)

    // remove
    s.pop(); // head | O(1)
    q.pop(); // head | O(1)
```

Maps

Performance depends on type.

operation	`map`	`unordered_map`
insert	O(log(n))	O(1)
access	O(log(n))	O(1)
remove	O(log(n))	O(1)
find/remove	O(log(n))	O(1)

```cpp
#include <bits/stdc++.h>
using namespace std;
using pss = pair<string,string>;
int main() {
    map<string,string> m;
    // unordered_map<string,string> m;

    m.insert(pss("key", "value"));             // insert
    string value = m.at("key");                // access (or m["key"])
    m.erase("key");                            // remove
    bool exists = (m.find("key") != m.end()); // find

    // iterate
    for(auto &it: m) {
        cout << it.first << ' ' << it.second;
    }
}
```

Sets

Performance depends on type.

operation	`set`	`unordered_set`
insert	O(log(n))	O(1)
remove	O(log(n))	O(1)
find/remove	O(log(n))	O(1)

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    set<int> s;
    // unordered_set<int> s;
    int value = 1;

    s.insert(value);                          // insert
    s.erase(value);                           // remove
    bool exists = (s.find(value) != s.end()); // find

    // iterate
    for(auto &i: s) {
        cout << i;
    }
    cout << '\n';
}
```

Sorting

Don’t reinvent the wheel.

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

int main() {
    vector<int> v = {4,2,5,3,5,8,3};

    // ascending order
    sort(v.begin(),v.end());
    for (auto &i: v){
        cout << i << ',';
    }
    cout << '\n';

    return 0;
}
```

Works for reverse sorting as well.

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

int main() {
    vector<int> v = {4,2,5,3,5,8,3};

    // reverse
    sort(v.rbegin(),v.rend());
    for (auto &i: v){
        cout << i << ',';
    }
    cout << '\n';

    return 0;
}
```

And for arrays!

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

int main() {
    int n = 7; // array size
    int a[] = {4,2,5,3,5,8,3};

    sort(a,a+n);
    for (auto &i: a){
        cout << i << ',';
    }
    cout << '\n';

    return 0;
}
```

Sorting comparators

`std` data structures can be sorted out-of-the-box.

But it can also work with your custom structures.

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

struct P {
    int x, y;
    bool operator< (const P &p) { // define '<' to use sort
        return (x != p.x) ? x < p.x : y < p.y;
    }
};

int main() {
    vector<P> v;
    v.push_back({7,6});
    v.push_back({1,2});
    sort(v.begin(),v.end());
    for (auto &i: v){
        cout << '(' << i.x << ',' << i.y << ')';
    }
    cout << '\n';
```

Or, you can pass the comparison function

```cpp
#include <bits/stdc++.h>
using namespace std;
using pi = pair<int,int>;

bool rcmp(pi a, pi b) {
    return a.second < b.second;
}

int main() {
    vector<pair<int,int>> v;
    v.push_back({1,7});
    v.push_back({7,2});
    sort(v.begin(),v.end(),rcmp);
    for (auto &i: v){
        cout << i.first << ',' << i.second << '\n';
    }
}
```