Using C++ in contest situations

Noah Weninger

Department of Computing Science
University of Alberta
Canada

February 9, 2019
You know a little bit of C or C++ but are by no means an expert. Neither am I; if you notice any errors in these slides please send me a mail: nweninge@ualberta.ca
What is optimal contest style?

1. Unlikely to crash
2. Easy to reason about
3. High performance
4. Fast to type
General things to avoid

1. Excessive abstraction
2. Long names
3. Things you don’t understand
4. Excessive pointer arithmetic
Compiling

$ g++ -std=c++11 -Wall -Wextra -O2 yourcode.cpp

If you have a modern version of clang, and are getting segfaults:

$ clang++ -fsanitize=memory yourcode.cpp
$ clang++ -fsanitize=address yourcode.cpp
This is a special header in gcc that includes the entire C++ standard library. Because of it’s size, this can increase compile times, and make error messages massive and difficult to read. But you never need to worry about not knowing what header to include for a specific data structure.
Be very familiar with them!

en.cppreference.com

on duck.com: search with !cpp

avoid cplusplus.com, it’s often outdated
```cpp
int x, y;
cin >> x >> y;
cout << x*2 << '␣' << y*2 << endl;
```

That’s a space in quotes, this syntax highlighting is weird.
#include<iostream>
#include<iomanip>
...
cout << fixed << setprecision(5) << 3.141592653 << endl; // 3.14159
When input is terminated at EOF

```cpp
int n;
while (cin >> n) {
  ...
}
```
Reading a whole line

1 string s;
2 getline(cin, s);

NOTE: the first time you call getline after having just used >> to read a line, getline will just return a newline. This is because getline will read to the end of the current line, and >> only consumes preceding whitespace. To avoid this, just call getline twice.

1 int x;
2 cin >> x;
3 string s;
4 getline(cin, s); getline(cin, s);
When there is an unspecified number of items on a line

```cpp
string s;
getline(cin, s);
stringstream ss(s);
int n;
while (ss >> n) {
    ...  
}
```
Kattis and presentation error

Kattis doesn’t really care much about whitespace. If they ask you to print a bunch of numbers on a line, this is fine:

```cpp
for (int x : numbers)
    cout << x << '␣';
```
Fast IO is usually unnecessary, but it’s interesting to note that for many problems, IO is 90% of your running time.

```cpp
int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    ...
```
scanf is *marginally* faster than cin with the previous trick.

However, it’s also error prone. Using the wrong format specifiers often results in intermittent incorrect results, e.g.

```cpp
1 long long x;
2 scanf("%ld", &x);
```

So it’s best to avoid.
Even faster IO

Write your own integer parser!
Way faster IO

Use `getchar_unlocked`, `putchar_unlocked`, `fgetc_unlocked`, ...

Look them up for details.
Macros

#define F(i,n) for(int i=0;i<n;i++)

I don’t use these often, but some people like them.
#define dbg(a...) D("line", __LINE__, ':', a); cerr<<'\n'
void D(){} template<class A, class... T>
void D(A a, T... x){ cerr<<a<<' '<<D(x...); }
int main() {
  dbg(1,2,3); // prints "line 5: 1 2 3"
}

You can submit code with these dbg prints to kattis because they ignore stderr, but your code might run so much slower that you get TLE. Disable them with:

#define dbg(...)
Goto is often the cleanest way to break out of a nested loop.

```cpp
F(i,n) {
    F(j,n) F(k,n) {
        // Some code
        if (...) goto next:
    }
    next:;
}
```
assert(foobar);

If your assert fires, you get “Run time error” on kattis.
```cpp
#include<cmath>
...
float x = INFINITY; // C++11
assert(isinf(x));
```

Also ok for double.
2000000000 lol

Just pick a number so big it can’t arise normally. If infinities can get added, make sure they are small enough that they don’t overflow.
Globals

Globals are your friend. They look like this:

```cpp
int n;
int a[100010];
```

- Globals are always initialized to zero! No need to worry about uninitialized memory.
- Make them bigger than you need (+1 is probably enough, but might as well add 10!) Under no circumstances do you ever want to have to debug an out of bounds array access.
memset operates at the byte level. Recall -1 has all bits set. So

```
int x[101];
memset(x, -1, sizeof x);
```

sets all x to -1. Works for 0 too. But

```
unsigned int x[101];
memset(x, 1, sizeof x);
```

sets all x to 0x01010101!!
Trivia:

```c
float x[101];
memset(x, 0, sizeof x);
```

What is \(x[0]\)?
Trivia:

```c
float x[101];
memset(x, 0, sizeof x);
```

What is $x[0]$? 0.0
Trivia:

```c
float x[101];
memset(x, -1, sizeof x);
```

What is \( x[0] \)?
Trivia:

```c
float x[101];
memset(x, -1, sizeof x);
```

What is $x[0]$? -NaN
assert(isnan(x[0]));

Can be used as an in band control.
When a function returns a value outside of its codomain to indicate some metadata.

E.g. -1, -2, NaN.
Never use class.

struct is exactly the same, except you never have to type public:

(private or protected clearly have no place here)
When you mutate a STL data structure, iterators are often invalidated.

* Very common cause of crashes in contest code.

Solution: read the docs, every data structure is different. Or just make no assumptions (harder than it sounds)
Casting

static_cast<int>(x + y) is great when you’re in a templated mess or 100000 LOC deep.

But just do

1. int(x + y)

(People call (int)(x + y) a “C-style cast” but it’s really Java-style)
Handy GCC extensions

```c
__gcd(a,b); // greatest common divisor
__builtin_popcount(a); // count number of bits set
__builtin_ctz(a); // count trailing zero bits
__builtin_clz(a); // count leading zero bits
__int128_t x; // when 64 bits isn’t enough
__uint128_t y;
```