うさぎ小屋

競技プログラミングにおける個人的 C++ コーディングスタイル (2020)

- date: Oct 25, 2020
- tags: competitive

注意

この記事は競技プログラミングをする人に向けて、私個人の好みのコーディングスタイルを紹介する記事です[1]。コーディングスタイルなどは好みの問題が大きいので、まったくすべて同意できる人は少ないだろうことに注意してください[2]。想定読者は AtCoder 水色以上ぐらいです。

なお、業務プログラミングしかしない人[3]や実際のプログラミングよりも言語仕様に興味がある人[4]は、この記事を読む必要はありません。

TL;DR

おおよそ以下のようなテンプレを使っています[5]。

```
#include <bits/stdc++.h>
#define REP(i, n) for (int i = 0; (i) < (int)(n); ++ (i))
#define REP3(i, m, n) for (int i = (m); (i) < (int)(n); ++ (i))
#define REP_R(i, n) for (int i = (int)(n) - 1; (i) >= 0; -- (i))
#define REP3R(i, m, n) for (int i = (int)(n) - 1; (i) >= (int)(m); -- (i))
#define ALL(x) std::begin(x), std::end(x)
using namespace std;

int64_t solve(int n, const vector<int64_t>& a) {
    // ...
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    constexpr char endl = '¥n';

    // input
    int n; cin >> n;
    vector<int64_t> a(n);
    REP (i, n) {
        cin >> a[i];
    }

    // solve
    auto ans = solve(n, a);

    // output
    cout << ans << '¥n';
}
```

競プロで特有な書き方が多く含まれていますが、どれも有用なものです。それぞれは以下のような欠点と利点を持ちます。

- `#include <bits/stdc++.h>` は、「可搬性の低下 (このヘッダは libstdc++ にしかない)」という欠点と「コンパイルエラーの減少 (include 忘れが減る)」「快適さの上昇」という利点を持ちます。
- REP マクロは、「(他人にとっての) 可読性の低下」という欠点と「(自分にとっての) 可読性の増加」「バグの減少 (変数名の修正漏れが減る)」「快適さの上昇」と利点を持ちます。
- `using namespace std;` (ただしソースファイル (`.cpp`) 内のもの) は、「コンパイルエラーの増加 (名前衝突)」「(他人にとっての) 可読性の低下」という欠点と「快適さの上昇」という利点を持ちます。「バグの増加 (意図しない形に名前解決される)」の可能性が指摘されていますが、実際に発生することはおそらくなく、あくまで可能性があるだけです。

これらの持つ欠点はどれを見ても競プロの文脈においては無視できるものです。比較をすれば利点の方がはるかに大きいため、利用を回避する必要はないでしょう[6]。

目次

- 注意
- TL;DR
- 目次
- `REP` マクロを使う
  - 利点など
  - 批判について
  - 名前の選択について
- `ALL` マクロを使う
- `using namespace std;` は使ってもよい
  - 批判について
  - 批判そのものについて
- 他の言語との比較
- `#include <bits/stdc++.h>` は使ってもよい
  - 批判について
- `#define int long long` はあまり使わない方がよい
- グローバル変数を使わない
  - 理由
- 簡単なコメントを書く
- 計算だけをする `solve` 関数を作る
- `assert` を書く
- `std::vector` を使い、領域はぴったり確保する
- おまけ: その他の細かい事項
- おまけ: あまり重要視しないこと
- 注釈

`REP` マクロを使う

バグを防げてかつ可読性[7]が上がります。ぜひ使いましょう。 具体的には以下を使っています。

```
#define REP(i,n) for (int i = 0; (i) < (int)(n); ++ (i))
#define REP3(i,m,n) for (int i = (m); (i) < (int)(n); ++ (i))
#define REP_R(i,n) for (int i = (int)(n) - 1; (i) >= 0; -- (i))
#define REP3R(i,m,n) for (int i = (int)(n) - 1; (i) >= (int)(m); -- (i))
```

利点など

利点はふたつです。

1. 変数名を修正したときに起こりがちなバグを防げる。
2. 通常のループに `for` 文が使われなくなるので可読性[7]が上がる。

(1.) について。`for (int i = 0; i < n; ++i) ...` と毎回書いているとバグが発生しがちです。 たとえば変数名を `i` から `j` に修正するときに、うっかり `for (int j = 0; j < n; ++i) ...` のように修正漏れをしてしまうことはよくあります[8][9]。 `REP` マクロであれば 1 箇所のみ修正すればよいのでこの問題を防げます。 修正の際のミスが問題であるので、スニペットを用いては解決できないことに注意してください[10]。

(2.) について。通常のループに `for` 文の直書きでなくマクロを使うようにすると、`for` 文が使われているループは何か特殊なことをしていることが分かります。 特殊な処理には特殊な記法が使われ目立つようになるため、これは可読性[7]に寄与します。

欠点や注意点がないわけではありませんが、利点と比べて無視できる範囲内でしょう。以下のふたつです。

1. マクロであるため、副作用のある式を与えると挙動が壊れる。
2. カウンタ変数の型が `int` だとオーバーフローの可能性がある。

(1.) は、たとえば `REP(i, ++n)` などです。しかし、このことを知ってさえいれば、これで実際に困ることはないでしょう。 (2.) は、たとえば `const long long K = 1e18; REP3(i, K, K + 3) ...` のようなものです[8]。一方で、常に `long long` を使うとその他のループがすこし遅くなります。これには注意する必要があります。

批判について

「`REP` マクロは汚ない」として批判されることがたまにあります[11]。 しかし「マクロを利用して構文を拡張する」というのは、C言語の時代からの伝統であり、この批判はまったく不適切です。

C言語において、データ構造を提供するようなライブラリは抽象化のために `XXX_FOREACH` のようなマクロを同時に提供することが多いです。たとえば Linux のソースコードを検索 しても `*_foreach` という構文拡張マクロはたくさん見つかります。また C++ においても (range-based for の提供が遅かったこともあって) この手のマクロは多く存在しています。たとえば Boost Foreach Library はまさにその例でしょう。 この手のマクロは頻繁に使われるものであ

るので、標準的なフォーマッタである `clang-format` においても ForEachMacros として特別なサポートがされています。

代替案として Boost Range Library の `boost::irange` を使うというものもありますが、最適化のかかり方や融通の効かなさ (`irange(first, last)` と書くときに `first > last` だとエラーになる) などの問題があり、あまりおすすめはできません。

名前の選択について

自分が間違えない限りはマクロ名にどんなものを使ってもよいでしょう。 しかし、あまりにも分かりにくいものを使うと、他人にコードを見てもらうとき (バグの原因が分からず Twitter などで助けを求めるときや ICPC などでのペアプロ) に困りがちです。 たとえば `rer(i,l,u)` と `reu(i,l,u)` の差を定義を見ずに推測できる人はほぼいないと思います[12]。

`ALL` マクロを使う

`REP` マクロと同様です。これも使いましょう。

```
#define ALL(x) std::begin(x), std::end(x)
```

利点だけでなく注意点も `REP` マクロと同様であることに注意しましょう。

`using namespace std;` は使ってもよい

`std::` を省略できて便利です。便利なので使っています。これは好みの問題であり、好きな人は使えばよいですし、嫌いな人は使わなくてもよいです。

`using namespace std;` すると `max` や `min` のような変数名を使いにくくなるという問題はあります。しかし、そのときだけ `int max = std::max(a, b);` のように書けば済みます。また、そもそも `using namespace std;` をしていなかったとしても `std` 内の識別子と変数名を被せていくのはあまりおすすめできません。

批判について

「名前衝突が起こって危ない」などと主張して避けようとする人が[13]多いように思われますが[14]、まともなコードを書いていれば、競プロをしていて名前衝突が発生することはありません。 「通常のやり方で競プロをしていて `using namespace std;` のせいで WA が発生した」

ことの例が複数発見されれば「`using namespace std;` は危険である」を主張できますが、そのような例はおそらくまったく知られていません[15]。

危険性として挙げられる点は主に以下のふたつに分類できるでしょう。

1. 「名前の衝突により、コンパイルに失敗するかもしれない」
2. 「もしコンパイルに成功したとしても、偶然名前解決に成功しただけで全く違う処理が呼ばれるかもしれない」

しかし、(1.) はまったく危険ではありません。コンパイルができなければ、修正すればよいためです。 もしこれが危険なのだとすれば、Haskell や Rust などの言語は危険すぎて使いものにならない言語ということになってしまいます。 しかしこれらの言語は通常は安全な言語であると認識されているはずです。 また、名前の衝突は「グローバル変数の利用」という `using namespace std;` と比べてより危険な行為に起因するものがたいていであり、グローバル変数の利用の側を禁止した方が利益が大きいでしょう。これについては以降の節で説明します。

(2.) は C++ 特有の overloading に由来するものであり、危険性としては正当なものですが、競技プログラミングにおいては問題にならないでしょう。 また、`using namespace xxx; ...` と書くのも `namespace xxx { ... }` と書くのも名前解決や overloading のされ方についてはさほど差はなく、後者はむしろ推奨される書き方である以上、`std` 名前空間特有の「具体的な危険性」を提出する必要があるはずです。

そして、競技プログラミングにおいては、`std` 名前空間内にある関数と同名の関数を定義してかつそれが運悪く overloading に由来するバグを引き起すことは無視できるほど稀でしょう。少なくともそのような具体例を (私は) まだ知りません。 `std::count` `std::find` `std::size` `std::distance` あたりの名前の衝突はありそうですが、`std::iterator_traits` などが効くために実際に問題になることは少ないはずです。 `std::gcd` や `std::lcm` については、これらの存在を知らなければ意図しない overloading が起こる可能性があるでしょうが、`::std::gcd` と独自定義の `::gcd` で結果が異なるような場合があれば、それは `::gcd` の側のバグです。これを気にする必要はないでしょう。 C 言語に由来する `::pow` と C++ の `::std::pow` の衝突や、同様に C 言語に由来する `::abs` と C++ の `::std::abs` の衝突なども発生しますが、これらがバグに繋がることはおそらくないでしょう。

批判そのものについて

そもそも、競プロ界隈の外側からであったとしても、`using namespace std;` に対する批判があること自体が奇妙に思えます。これについての検討をしておきましょう。 結論は、「ヘッダファイル (`.hpp`) 内で `using namespace` するのはやめろ」が伝言ゲームされ、また「競

プロerは `using namespace std;` しがち」「競プロerのコードは汚なくて危険」という印象があわさって、「(ソースファイル (`.cpp`) であれなんであれ) `using namespace std;` は危険」という風潮が発生した可能性がある、ということです。ただし、この主張は状況証拠のみを元にした飛躍の大きい推量であり、あくまで可能性にすぎないことも注意しておきます。

根拠は以下の 3 点です。

1. 英語圏のコミュニティと日本語圏のコミュニティでは `using namespace std;` に対する態度が異なるように見えること
2. 日本語圏の `using namespace std;` の議論では、「ヘッダファイル内で `using namespace std;` するな」という最も重要であるはずの点についてほとんど触れられていないように見えること

`using namespace` の危険性についての意見は英語圏と日本語圏で差があります。 英語圏では「`using namespace` 文をヘッダファイル (`.hpp`) の中で使うのはやめろ。それ以外ならたぶん大丈夫」などの意見が多そうに見えますが、これと比較して、日本語圏では「`using namespace std;` はとにかくやめろ。ソースファイル (`.cpp`) の中でもまずい」のような意見が多そうに見えます[16]。

英語圏のコミュニティに目を向けると、`using namespace std;` はそれほど強い批判はされていないように見えます[17]。 特に、ソースコード (`.cpp`) 内での利用が批判されることは少なめです。 たとえば大手の質問サイトからいくつかの質問 c++ - Why is "using namespace std;" considered bad practice? - Stack Overflow や Why is it not a good practice to write using namespace std? - Quora や [C++] Why do people say not to use "using namespace std"? : learnprogramming あるいは Why "using namespace std" is considered bad practice - GeeksforGeeks などを見ても、「`std` に限らず `using namespace` に一般の問題である」「ヘッダファイル (`.hpp`) の中で使うのは絶対にやめろ」「ソースファイル (`.cpp`) の中で使うのは (あまり推奨されないとはいえ) だめではない。もちろん overloading に注意する必要はある」程度の穏かな意見が主流に見えます。

具体的には c++ - Why is "using namespace std;" considered bad practice? - Stack Overflow の解答 (https://stackoverflow.com/a/1452738/8090005) は

> *But consider this: you are using two libraries called Foo and Bar: ...*

として `using namespace` 一般の話をしていますし、別の解答 (https://stackoverflow.com/a/1452759/8090005) では

> *The problem with putting `using namespace` in the header files of your classes is that ...*

かつ

> *However, you may feel free to put a using statement in your (private)* `*.cpp` *files.*

のように、ヘッダファイルでの利用のみを問題視しています。

一方で日本語の記事では、英語圏とは議論の内容が異なります[17]。 たとえば日本語のページを `using namespace std` で検索して見つかる[18]記事としては、競プロ外の文脈で書かれた using namespace std; の危険性と注意点・代替案【なぜ使わないほうがいいのか】 | MaryCore や using namespace stdをmainの外に書くな などと、競プロを踏まえた文脈で書かれた なぜusing namespace std;を避けるべきか - yumetodoの旅とプログラミングとかの記録 や namespaceの賢い使い方 - Qiita があります。 これらの記事からは、ソースファイル (`.cpp`) 内のグローバルでの `using namespace std;` の是非が (そのページの著者の立場はどうであれ) 頻繁に議論されていることが分かります。

一方で、ヘッダファイル (`.hpp`) 内での `using namespace std;` についてはまったく触れられていません。前の段落で挙げて 4 つの記事のうち、最も適切に書かれた記事であるように見える なぜusing namespace std;を避けるべきか - yumetodoの旅とプログラミングとかの記録 を除いて、その他の 3 つの記事ではこのことについての言及がありません。 これは実際のソフトウェア開発が複数ファイルを用いるのが通常であることを踏まえると、`using namespace std;` を批判する文脈でヘッダファイルに言及しないのはたいへんに奇妙なことです。すると、これらの記事が「あまり深く理解せずに、印象や雰囲気だけで主張をしている」「現実のソフトウェア開発のことを忘れて、仕様についての議論をしている」「現実のソフトウェア開発のことを忘れて、競プロについての議論をしている」のいずれかであることが疑われます。 そして実際、これら 3 種のうち「あまり深く理解せずに、印象や雰囲気だけで主張をしている」に当てはまるであろう議論がいくつか見られるように思われます。このことから「ヘッダファイル (`.hpp`) 内で `using namespace` するのはやめろ」が伝言ゲームされて「`using namespace std;` はよくない」という風潮になったという可能性が思い浮かびます。

他の言語との比較

名前空間の一括での展開の言語機能は、他の多くの言語も備えています。これらとの比較をしてみましょう。たとえば、Python の `from ... import *` 文や Haskell の (`qualified` なしの) `import` 文、OCaml の `open` 文などです。もちろん TypeScript のようにこの機能を持たない言語もあります。

Haskell や OCaml のように静的で overloading を許さない言語においては、名前が衝突してもただコンパイルが通らないだけです。これは比較的安全であるためか、そのような言語では、名前空間の展開が非推奨だと議論されることはあまり見かけません。 Python のような動的な言語においては、変数の上書きがあるために同様に C++ と同様に危険です。しかし PEP 8 では可読性を下げることを理由に非推奨という弱い形に留まっています。 PEP 8 での主張が弱いことについては、静的型付けと動的型付けの違いと考えれば不思議ではありません。つまり、C++ では `using namespace` によって元々あった「コンパイルが通ればある程度は安全」という性質が消えてしまいますが、Python ではそのような性質が元からなく変化しないためです。

このような比較を踏まえると「C++ の `using namespace` 文は他の言語と比較して危険である」だけならば言えるかもしれません。 もちろんこれを主張する場面においては「他より危険であるのはどのような点においてなのか」に注意する必要があります。 「より危険な状態に近づく」とはいえ「最初からそのような危険な状態のはずの Python などの言語では、その状態の危険性をあまり重要視していない」という背景があるためです。

`#include <bits/stdc++.h>` は使ってもよい

テンプレ部分をすっきりさせたいので `#include <bits/stdc++.h>` を使っています。 使いもしないファイルについて `#include` 文を書き並べるよりも `#include <bits/stdc++.h>` をひとつだけ書く方が好きです。これは好みの問題であり、好きな人は使えばよいですし、嫌いな人は使わなくてよいです。

ただし、コンテスト中の害は少ないとはいえ、競プロ界隈の外の人にコードを見せるとき (Stack Overflow など) では避けた方がよいでしょう。環境によってはコンパイルができないので相手をかなり困らせることになります。

批判について

`#include <bits/stdc++.h>` も[13]批判されることが多いように思います[19]。 これらの批判は主に以下のふたつに分類できるでしょう。

1. 「他の環境に持っていったとき、コンパイルに失敗するかもしれない」
2. 「もしコンパイルに成功したとしても、仕様にない機能を使うのはきれいでない」

しかしどちらも、競技プログラミングにおいては、妥当な批判ではありません。 まず (1.) ですが、コンテスト中に開発環境を大きく入れ替えるなどということはありません。 コンパイラの気持ちになって定数倍最適化をして嘘解法をむりやり AC にしたりしたい人は libstdc++

と libc++ の差の影響を受けることがありますが、そうでない普通の競技プログラミングをする限りは `#include <bits/stdc++.h>` をして困ることはありません。

(2.) について、競技プログラミングはコードのきれいさを競う競技ではないため、無効な批判です。 コードがきれいでも WA が AC になることはありません。 また、「どうせテンプレなので、すべての標準ライブラリのヘッダについて `#include` 文を書いておけばよい」という代案が提示されることがあります[20]。この代案は妥当なものですが、「実際には不要なヘッダを大量に `#include` する」ことは同様に「きれいではない」とされる書き方でもあります。

`#define int long long` はあまり使わない方がよい

これも好みの問題ですが、できるだけ避けた方がよいでしょう。特に利点がありません。これをするぐらいなら `using ll = long long;` して `ll` を使った方がよいように思います[21]。ただし、原因不明の WA が出たときに `#define int long long` を追加して提出し直すことはあるかもしれません。 その場合、 `printf` `scanf` を使っている人は `%d` `%lld` の差には注意しましょう[8]。

コンテスト中の害は少ないですが、コンテスト外の Twitter などで「WA の原因が分かりません。たすけて！」をするときにこれを仕込んでいると嫌がられます。親切心からコードを呼んで「オーバーフローしてそう」と伝えて「`#define int long long` してるので大丈夫です」と返ってくると悲しい気持ちになりがちです[8]。また、ICPC などでペアプロをするときにも混乱を招きがちです。

グローバル変数を使わない

いくつかの欠点があり、利点はほぼありません。できる限り避けましょう。

理由

競技プログラミングにおいて、グローバル変数の問題点は主に以下のふたつです。

1. 複数テストケースの問題などで、再初期化を忘れやすい。
2. 再利用がしにくく、ライブラリ化を妨げる。

(1.) が重要です。 複数テストケースの問題を解くとき、グローバル変数の適切な再初期化をしていなければ「最初の 1 ケース目だけは正しく動作するが 2 ケース目以降では動かない」という形のバグになります。 サンプル入出力には t = 1 のケースしかない場合なども多く、発見が困難なものになりがちです[8]。 すべて関数内のローカル変数として持てば関数を呼び出すた

びに常に初期化してくれるので (あるいは常に初期化を忘れて 1 ケース目でも壊れるので)、面倒なバグの発生を防げます。

(2.) は本番の解法コードよりもライブラリとして保存しておくコードについてのものです[22]。ライブラリに「どう入力を与えてどう出力を受けとればよいか」は、もしそれがひとつの関数の形で閉じていれば単に呼び出すだけなので明らかですが、グローバル変数を多用するものであると明らかではありません。かなり丁寧なドキュメントを書くことが要求されてしまうでしょう。 また、複数のライブラリを貼ってグローバル変数の名前が衝突したりすると面倒な事態になります。 単にグローバル変数を除去するだけで、このような困難は解決できます。

簡単なコメントを書く

可読性[23]が上がります。 コードの塊の間に空行を入れたくなったら雑に 1 行だけコメントする (`// input` とか `// dijkstra`) 程度のものです。 手軽なわりにかなり効果があります。おすすめです。

たとえば以下のようにします。

```
int main() {
    // input
    int n; cin >> n;
    vector<int64_t> a(n);
    REP (i, n) {
        cin >> a[i];
    }

    // solve
    auto ans = solve(n, a);

    // output
    cout << ans << '¥n';
}
```

計算だけをする `solve` 関数を作る

`main` 関数の中では入出力だけをし、答えの計算は `solve` のような名前の関数に閉じ込めます。 可読性[23]が上がり、入出力部分と計算部分を分離できるのでコードが書きやすくなります。 おすすめです。

たとえば以下のようにします。

```
int64_t solve(int n, const vector<int64_t> &a) {
    ...
}

int main() {
    int n; cin >> n;
    vector<int64_t> a(n);
    REP (i, n) {
        cin >> a[i];
    }
    auto ans = solve(n, a);
    cout << ans << '¥n';
}
```

`assert` を書く

デバッグの効率が上がります。書いておきましょう。 たとえば区間 $[l, r)$ を受けとる関数の冒頭で `assert (l <= r);` と書くなどです。

ごくまれに `assert` を書きすぎたせいで AC だったものを RE にしてしまうことがありますが[8]、そのリスクよりもデバッグ速度に貢献するメリットの方がはるかに大きいです。

`std::vector` を使い、領域はぴったり確保する

コンパイルオプションに `-fsanitize=undefined` および `-D_GLIBCXX_DEBUG` を付けることと合わせれば、バグの早期発見が可能になります。つまり、バグを埋め込んだときに、提出して WA になってから気付くのではなく、手元でサンプルを試す段階で気付けるようになります。ペナルティが減ります。

生配列だとだめなので避けましょう。定数倍高速化をしたいときは `std::array` を使いましょう。

おまけ: その他の細かい事項

- `int` と `long long` を使い分ける

    - 四則演算を伴なった値としての整数の型として `long long` や `int64_t` を使い、基本的に加減算のみしか考えない添字としての整数の型として `int` を使うようにしています[24]。 「`int` の乗算をしているとなにか怪しい」「添字の中に `long long` がでてくるとなにか怪しい」などといった判別ができるようになるという形で、可読性[7]に寄与します。 これはかなり好みが分かれると思います。
- `std::cin` `std::cout` を使う場合は速度に注意する
    - これは重要です。なぜなら、TLE するからです[825]。以下を書き、また `std::endl` を使わないようにしましょう。 あるいは、 `main` 関数の中などで `constexpr char endl = '¥n';` と書いて、 `endl` と書けば単なる `char` 型のローカル変数が使われるように shadowing しておいてもよいでしょう。

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

      この 2 行は C++ 側の `std::cin` `std::cout` と C 側の `scanf` `printf` の連携を解除し入出力を高速化します。 副作用として、これを書いた上で `std::cin` と `scanf` を混ぜたりすると壊れます。 また、 `std::endl` を避けるのは不要な flush を避けるためです。

- `printf` `scanf` を使う場合はフォーマット指定子や引数に注意する
    - 間違えていてもコンパイルが通ってしまうので注意しましょう[825]。コンパイルオプションに `-Wall` を付けておくと警告がでます。 `int n; scanf("%d", n);` や `long long x; scanf("%d", &x);` などです。

      ただし、 `int64_t` と `long long` が同じであることを仮定するのは問題ないと思っています。

- 標準エラー出力を使う
    - `fprintf(stderr, ...)` や `std::cerr` を使いましょう。 デバッグ用の出力を残したまま消しても (たいていのオンラインジャッジでは) WA にならなくなります。 もちろん、出力があまり多すぎると出力処理に時間がかかって TLE します。
- 初期化は必ず宣言と同時か直後に行う
    - 未初期化の変数を使用してしまうことによるバグが防げます。関数の冒頭で `int i, j, k;` などとするのはやめ、使う直前に `int i = 0;` のように初期化と同時に宣言しましょう。また、これは好みによるでしょうが、入力についても可能ならば `int n;` と `cin >> n;` をまとめて 1 行に書くようにしています。
- 変数のスコープはできるだけ狭める

    - `{` と `}` でブロックを作ることで、変数のスコープを制限できます。 変数名の衝突を回避したり同じ変数を使い回したりしなくてよくなったり、使い終わって不要になったはずの変数を参照しようとするとコンパイルエラーになるなどするため、バグが減ります。メモリ使用量もいくらか減ります。
- ラムダ式による再帰を使っていく
    - 再帰をするときにも無闇に関数をグローバルに定義せず、ラムダ式を使ってローカル変数として定義します。 グローバル変数の利用や、引数をたくさん持ち回ることを避けるためです。 下手に書くとスタックのオーバーフローが不安になります[8]が `auto go = [&](auto && go, ...) { ... };` と定義して `go(go, ...);` と呼び出せばましになります。 これは比較的好みの問題でしょう。

おまけ: あまり重要視しないこと

`clang-format` などのフォーマッタをかけたときに失われるような差や単なる識別子の選択の差については、それほど重要でないと考えています。 それが現実的な意味でバグや実行速度低下に繋がる可能性がなければ、C++ として多少不適切な書き方であっても気にしません。 そのような差として、たとえば以下があります。

- インデントの幅
- 括弧の付け方
- 空白の入れ方
- 変数名の選び方
- `long long` なのか `int64_t` なのか
- `using ll = long long;` と書くか `#define ll long long` と書くかそれともなにも書かないか
- `template <class T> using V = vector<T>;` などをするかしないか
- `constexpr int MOD = 100000007;` なのか `#define MOD 1000000007` なのか
- `goto` を使うか使わないか

注釈

1. 以前 (5 年前) にも同様の記事を書きましたが、さすがに古くなってきたので更新です。過去記事は見ずに新規に書き直しました。こういう主観的な主張や興味は時間が経つと自分の中でも変化しがちなので、比較してみると楽しいです。ところで、あのころは C++ に限らず全体的に「美しい理想のものを求めていく」系の熱意 (たとえば QWERTY でなく Dovrak を使うような) がありましたが、月日の経過とともにどんどん「無難で標準的なものの方が楽だよ」という方向に寄っていっているんですよね。老化こわいですね ↩

2. わりと強い主張が散りばめられてるのでこわいんですよね ↩

3. 自動採点機能付き数学パズルコンテストは業務プログラミングとは異なります。業プロerにとって、この記事に読む価値はないでしょう。 ↩

4. たとえば Python と言ったときに基本的に CPython 以外の処理系で動くかどうかはあまり気にしないのと同様に、競技プログラミングの文脈で C++ と言ったときは基本的に GCC や Clang のみを考えるのが妥当でしょう。 ↩

5. ただし、自作のツール oj-template を使って、入出力パートを含めて main 関数ごと毎回自動で生成しています。古の `oj` コマンドや atcoder-tools にもある機能であり、main 関数を手動で書く時代はもう終わったと思っています。 ↩

6. もし「コンパイルエラーの増加」「可読性の低下」「バグの増加」などの様々な作用を「危険性」のような単一の語でまとめてしまったり、もし「ソフトウェア開発において」「競技プログラミングにおいて」「コードゴルフにおいて」などの文脈を無視してしまったりすれば、これらの書き方は「危険な書き方」になります。しかしそのような議論はあまりにも雑であり、無視してしまってよいでしょう。そんなことを言いだせば、そもそも C++ を使うこと自体が「危険」です ↩

7. ただし、そのスタイルを採用している当人にとっての可読性であり、無関係な他人にとっての可読性ではありません。チーム戦でない競プロのコンテストに複数人で参加することはルール違反であることに注意してください。 ↩ ↩2 ↩3 ↩4

8. なんどかやらかしました ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8 ↩9

9. `i` `j` `k` でなく `x` `y` `z` を使うという追加の対策もおすすめです。 ↩

10. かしこい IDE の rename 機能を必ず使うようにするという解決策もあるにはあります ↩

11. 私が競プロを初めたころ (2014 年) と比べるとかなり減った気がします。最近はあまり聞きません。 ↩

12. 後輩が一時期これらを利用しており、ICPC のためのチーム練習のたびに私が混乱していたという記憶があります。 ↩

13. 特に、あまり競プロをしていない層に ↩ ↩2

14. この批判はまだよく見かける気がします。 ↩

15. もしあれば教えてください ↩

16. おそらくは「ヘッダファイル (`.hpp`) 内で `using namespace` するのはやめろ」が伝言ゲームされて「`using namespace std;` するのはよくない」という雰囲気に変化し、これと月刊競技プログラミングは役に立たないによる「競プロerは `using namespace std;` しがち」「競プロerのコードは汚なくて危険」という印象が併さり、「(ソースファイル (`.cpp`) であれなんであれ) `using namespace std;` は危険」という風潮が発生したのではないか、と考えています。ただし、この主張は何の証拠もないまったくの当て推量であり、真偽の保証はできません。 ↩

17. これは主観によるものです。特に統計などは取っていません。やる気のある人がいればたのむ ↩ ↩2

18. Google 検索の結果は魚拓などを取れないぽくて困った ↩

19. これについての批判は AtCoder Programming Guide for beginners (APG4b) が始まってから増えたのではないかと疑っています。私が競プロを初めたころ (2014 年) に `#include <bits/stdc++.h>` が頻繁に話題になっていたという記憶はありません。 ↩

20. 初心者に「おまじない」として教えるのなら、環境依存の少ないこちらの方が無難であるとは思います。オンラインコンパイラなどの利用を前提にしていたとしても、おまじないの行数を削ることにどれほどの利点があるかはよく分かりません。 ↩

21. `#define int long long` と `using ll = long long;` は同じ文字数であり、それ以降において `int` が `ll` になる分だけ `using ll = long long;` の方が (コードゴルフ的な意味で) 有利です。 ↩

22. ライブラリはヘッダファイル (`.hpp`) の形で書いておいて、利用の際は `#include` を経由して呼び出し、自作のツール oj-bundle で `#include` された内容を自動で埋め込んで提出しています。最近は AtCoder Library が `expander.py` を提供したりもしていますし、「単一ファイル + スニペット」という形のライブラリ管理の時代は終わったと思っています。 ↩

23. そのコードを書いた当人以外の他人を含めての意味の可読性です。 ↩ ↩2

24. 実数 $\mathbb{R}$ や複素数 $\mathbb{C}$ の部分集合としての自然数 $\mathbb{N}$ と、順序数 ON の部分集合としての自然数 ω との違い、のようなイメージです。 ↩

25. これはコーディングスタイルではありません。書かれるコードに選択の余地はなく、単に「バグに気を付けましょう」でしかないためです ↩ ↩2

View history / Report issues / Edit this page