

# CPP 在竞程实作中的运用

郎易恒

# 前言

本节课旨在分享我在竞程实作中对 cpp 的一点粗浅认识，引入性质，以例子为主。

大概有两部分内容，一部分谈代码习惯，一部分谈实用特性。

核心想法是尝试将团队协作中的代码可读性置于首位，其次兼顾鲁棒性，最后考量代码量。

由于我无论在竞程还是 cpp 上的积累都相当浅薄，囿于能力所限，本次分享必然存在许多不足之处。恳切盼望同学们在听课时保持独立思考，务必批判性地辨别取舍，并不吝赐教。

若能博君一笑，甚至能带来一二启发，已是我的莫大荣幸。

# 代码习惯

习惯是很主观的，无法断言好坏，接下来的内容更多的是我的一些习惯分享，再次请大家批判性地辨别取舍。

不要重新造轮子

如果你足够了解一个标准库的函数，并且认为他完全符合你的需求，那就用它而不是自己写。

平时遇到一些常见的需求，也可以询问 lilm 或者查询 cppreference，进行积累。

## 一个例子

统计数组 a 内偶数的个数：

```
int cnt{}; {
    for (auto&& ai : a) {cnt += bool(ai % 2 == 0);}
}
```

可以这样实现：

```
auto cnt = std::ranges::count_if(a, [](const auto& v) { return v % 2 == 0; });
```

# #define int long long?

当然是不好的，但如果非要用的话，提供一个我认为更好些的实作 by [hos.lyric](#)：

```
using Int = long long;
```

然后代码写完文本批量替换 `int -> Int`

- `using Int = long long;` 明确地声明这是一个别名，语义清晰。
- 避免一些副作用：
  - 不能 `int main()`
  - 不能 `printf("%d\n", x); -> printf("%lld\n", x);`

# 变量与类型规范

## 避免延迟声明

降低可读性，阅读者不知道这个变量在哪里用到了。

```
// ✗ 提前: int temp;
{
    int temp;

    for (int i = 0; i < n; i++) {
        temp = i * i + 1;
    }
}

// 用到时再声明
{
    for (int i = 0; i < n; i++) {
        auto temp = i * i + 1;
    }
}
```

## = 拷贝初始化

- `auto` 初始化
  - 进行变量拷贝操作时
  - 进行引用时
  - 接受函数返回值时
  - 接受算式计算结果时
- 类型转化初始化
  - 比如为了转换 `size(a)` 为 `signed`
    - `int n = size(a)`

## { } 统一初始化

- 除了上述情况之外的情况。
- 禁止窄化、更安全

# assert

```
// 1. 输入合法性  
assert(1 <= n && n <= 2e5);  
  
// 2. 容器越界  
assert(i < (int)a.size());  
  
// 3. 不变式  
assert(ans >= 0);  
  
// 4. 算法边界  
assert(i != -1);
```

# 尽量避免双下划线函数

用户代码调用双下划线函数本身就是一种 UB，标准库没有义务保证这些函数的使用体验。

无论是可读性还是鲁棒性，都是标准库的函数完胜，使用双下划线只应在标准库没有实现该功能的场景。

`std::gcd` 甚至显著快于 `__gcd`：

因为 `__gcd` 使用的是普通的辗转相除法，而 `std::gcd` 用 Stein 算法显著提高了效率

- `__builtin_clz(x) ; __builtin_clzl(x) ; __builtin_clzll(x)`
  - `std::countl_zero(x)`
- `__builtin_popcount(x) ; __builtin_popcountl(x) ; __builtin_popcountll(x)`
  - `std::popcount(x)`
- `__gcd(a, b)`
  - `std::gcd(a, b)`

可以发现一些双下划线函数甚至没有做基本的函数重载，而利用标准库替代之后就舒服多了。

## 避免 `std::vector<bool>`

标准库特化了这个实例，而这个特化至少在竞程中非常反人类。

比如：`operator[]` 不返回 `bool&`，而返回一个代理对象。

一个比较好的替代思路是用 `std::vector<uint8_t>` 平替。

# NO VLA!

VLA (Variable-Length Array, 可变长度数组) 是指数组长度在运行时确定而不是在编译时固定的数组形式。

这是 cpp 标准明文禁止的用法，cpp 的新特性没有义务保证 VLA 的正常运行

一些萌新可能会有这样一份实作

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

int main() {
    int n; std::cin >> n;
    std::vector<int> adj[n];
    for (int i = 0; i < n - 1; i++) {
        int u, v; std::cin >> u >> v; --u; --v;
        adj[u].emplace_back(v); adj[v].emplace_back(u);
    }
    auto dfs = [&](auto self, int u, int father) -> void {
        for (auto&& v : adj[u]) if (v != father) {
            self(self, v, u);
        }
        return;
    };
    dfs(dfs, 1, 0);
    return 0;
}
```

那么在这样的环境下：

```
Using built-in specs.
COLLECT_GCC=C:\Program Files (x86)\w64devkit-1.20.0\w64devkit\bin\g++.exe
Target: x86_64-w64-mingw32
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 13.2.0 (GCC)
```

就会报错！

```
Exit code: 1 Errors while compiling:  
d:\test.cpp: In function 'int main()':  
d:\test.cpp:12:16: error: use of deleted function 'main()::<lambda(auto:54, int, int)>::~<lambda>()'  
12 |     auto dfs = [&](auto self, int u, int father) -> void {  
|             ^~~~~~  
13 |         for (auto&& v : adj[u]) if (v != father) {  
|             ~~~~~  
14 |             self(self, v, u);  
|             ~~~~~  
15 |         }  
|         ~  
16 |         return;  
|         ~~~~  
17 |     };  
|     ~  
d:\test.cpp:12:18: note: 'main()::<lambda(auto:54, int, int)>::~<lambda>()' is implicitly deleted  
because the default definition would be ill-formed:  
12 |     auto dfs = [&](auto self, int u, int father) -> void {  
|             ^
```

这个 lambda 的析构函数被删除了，何意味？

枚举所有情况！以下两种实作通过编译：

```
std::function<void(int, int)> dfs = [&](int u, int father) -> void {
    for (auto&& v : adj[u]) if (v != father) {
        dfs(v, u);
    }
    return ;
};
```

```
auto dfs = [&adj](auto self, int u, int father) -> void {
    for (auto&& v : adj[u]) if (v != father) {
        self(self, v, u);
    }
    return;
};
```

第二种耐人寻味，这样的写法能绕过 lambda 默认捕获，只捕获 `adj`。

那么问题就显露出来了：

在 C++ 中，隐式生成的析构函数会在以下两种情况下被定义为 `= delete`（也就是“删除”状态）：

1. 类中有一个非静态成员或基类，其析构函数本身就是 deleted 或者在当前上下文不可访问；
2. （同样适用于多维数组成员）如果某个成员是一个数组，而数组元素类型的析构函数 deleted 或不可访问，也会导致包含它的类析构函数被删除。

- `std::vector<int> adj[n]` 是 VLA。
- 当用 `[&]` (默认按引用) 捕获时, 编译器会把这个 VLA 当作一个非静态成员 (引用) 添加到闭包类型里。
- 在生成这个闭包类型的析构函数时, 编译器需要为那个“引用 VLA”成员调用析构, 但由于 VLA 类型本身不是一个在编译期可完整知悉的标准类型, 隐式析构定义就变得 **ill-formed**。
- 根据上面的规则, “**如果隐式析构的定义会不合法 (ill-formed) 就把它定义为 deleted**”, 于是闭包的析构函数就被标记为 deleted, 编译器报错“use of deleted function ‘...::<lambda>::~<lambda>()’”。

当改成：

```
auto dfs = [&adj](auto self, int u, int father) -> void { ... };
```

就避开了捕获 VLA 的场景，闭包类型里不再含有那个“会导致析构不合法”的成员，析构函数恢复正常定义，代码因而可以编译通过。

不过最好的实作当然是直接改掉 VLA

```
std::vector adj(n, std::vector<int>())
```

这正是 GCC 的一个已知 [Bug \(102272\)](#)：VLA 在泛型 lambda 中捕获时被错误拒绝。

## 实用特性

这里对于新同学来说可能部分知识会不知所云/不清楚有什么用处，说明还没有遇到过运用场景，此时不用深入了解，记得有这么个东西即可。

# 作用域

## 单开作用域

| 有很多场景会用到，在代码复杂时有好的简化效果。

```
std::set<int> S; int x;
// 假设我们惯用 it 来作为迭代器的命名，而又只对于这个作用域有效，那么单开作用域就避免了要写多个命名的问题
// 放在一个作用域
auto it_l = S.lower_bound(x);
auto it_u = S.upper_bound(x);
// 单开作用域
{
    auto it = S.lower_bound(x);
}
{
    auto it = S.upper_bound(x);
}
```

{}

- 中断强度：不支持中断
- 适用场景：最简单的局部隔离，最好写。
- 代码范例：

```
std::vector<int> pre(n + 1); {
    for (int i = 0; i < n; i++) {
        pre[i + 1] = pre[i] + a[i];
    }
}
```

## do {} while (false);

- 中断强度：支持 break
- 适用场景：需要简单的中断。
- 代码范例：

```
std::vector<int> a(n);
auto it = std::ranges::find(a, 1);
int ans{};
do {
    if (it == std::end(a)) {break ;}
    ans += *it;
} while (false);
```

## IILE (Immediately Invoked Lambda Expression)

- 中断强度：支持 `return`
- 适用场景：需要 强中断 或 返回值

- 代码范例：

```
// 如果没有访问过 res 相连的节点，才能继续，需要 return (因为一次 break 只能退 for)
int res{};
[&] {
    for (auto&& v : adj[res]) {
        if (vis[v]) {
            return ;
        }
    }
    // do something
}();

// 比如为一个 flag 初始化，需要返回值
bool ok = [&] {
    //do someting
    if (/*something*/) {
        return true;
    }
    return false;
}();
```

## `if` 初始化

很好解决了一些变量只在 `if` 期间用到，却不得不污染外部作用域的问题。

比如一个 *DP* 场景

```
if (j + k - 1 < n) {  
    update(dp[j + k - 1], dp[j] + w);  
}
```

可以优化成这样：

```
if (auto nxt_j = j + k - 1; j < n) { // nxt_j 在整个 if 作用域内，包括 else, else if  
    update(dp[nxt_j], dp[j] + w);  
}
```

# std::optional

如果一个函数返回值存在无解情况，建议使用 `std::optional` 的 `std::nullopt` 来表达这一状况。

```
auto create(bool b) {
    return b ? std::optional<std::string>{"Godzilla"} : std::nullopt;
}

int main() {
    auto res = create(true);
    if (res) {
        std::cout << res.value();
        std::cout << *res << "\n";
    }
    std::cout << res.value_or("empty") << "\n";
}
```

## std::optional + if 初始化的线段树上二分：

```
// 返回区间 [x, y) 内第一个满足 pred 的位置，未找到则返回 std::nullopt
template<class F>
std::optional<int> findFirst(int p, int l, int r, int x, int y, F&& pred) {
    // 当前节点区间与查询区间无交集
    if (l >= y or r <= x) { return {}; }
    // 当前节点完全在查询区间内，且整段都不满足 pred
    if (l >= x and r <= y and not pred(info[p])) { return {}; }
    // 到达叶子节点，返回该位置
    if (r - l == 1) { return l; }
    int m = (l + r) / 2;
    push(p); // 下推懒标记

    // 先在左子树查找
    if (auto res = findFirst(2 * p, l, m, x, y, pred); res.has_value()) {
        return res; // 找到则直接返回
    }
    // 左子树没找到，继续查询右子树
    return findFirst(2 * p + 1, m, r, x, y, pred);
}
```

# 引用的别名作用

- 统一用万能引用 `auto&& ref = ...;` 可同时绑定 lvalue / rvalue，且无需纠结 `const` 与否。

在用来作别名的引用上一律万能引用，个人没有想到不妥之处。

- 例子：

```
std::vector val2pos(n, std::vector<int>()); {
    for (int i = 0; i < n; i++) {
        val2pos[a[i]].push_back(i);
    }
}

for (int val = 0; val < n; ++val) {
    auto&& pos = val2pos[val]; // pos 是别名，省去多层下标
    for (auto& idx : pos) {
        // do something with idx
    }
}

auto&& [w, p] = item; // 结构化绑定
```

## **std::format**

<https://en.cppreference.com/w/cpp/utility/format/format.html>

至少可以当成一个类型安全的 `printf` ?

什么场景下用?

Takahashi: id = 71806291, score = 60.12

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

int main() {
    std::string name{"Takahashi"};
    int id{71806291};
    double score{60.1234};

    std::printf("%s: id = %d, score = %.2f\n", name.c_str(), id, score);
    std::cout << name << ": id = " << id << ", score = "
        << std::fixed << std::setprecision(2) << score << '\n';
    std::cout << std::format("{}: id = {}, score = {:.2f}\n", name, id, score);

    return 0;
}
```

## Ranges library (since C++20)

<https://en.cppreference.com/w/cpp/ranges.html>

## **std::ranges**

把很多常见的 STL 接口升级为可以直接作用于整个“范围”， 算法竞赛中可以减少码量？

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

int main() {
    std::vector a = {1, 2, 4, 3};
    int x{4};

    {
        // 排序
        std::sort(std::begin(a), std::end(a));
        // 去重
        a.erase(std::unique(std::begin(a), std::end(a)), std::end(a));
        // 查 x 在压缩后数组的位置
        int pos = std::distance(std::begin(a), std::lower_bound(std::begin(a), std::end(a), x));
    }

    {//std::ranges
        // 排序
        std::ranges::sort(a);
        // 去重
        a.erase(std::begin(std::ranges::unique(a)), std::end(a));
        // 查 x 在压缩后数组的位置
        int pos = std::distance(std::begin(a), std::ranges::lower_bound(a, x));
    }

    return 0;
}
```

## std::views

- **懒执行**: `views` 对原容器或生成器不作拷贝，只有在真正遍历（`for`、`copy`、`accumulate` 等）时才计算元素。
- **零开销抽象**: 编译期展开后，几乎等同于手写循环，没有额外的运行时代价。
- **可组合**: 可以用管道符 `|` 串联多种变换，比如 `filter` → `transform` → `take`，代码像 Unix 管道一样直观。

## 获得一个存储树中所有叶子节点编号的数组

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

int main() {
    int n = 10;

    std::vector deg = {0, 1, 0, 2, 0, 3, 4, 0, 1, 0};
    auto leaves = std::views::iota(0, n) | std::views::filter([&](int u) { return deg[u] == 0; });

    return 0;
}
```

## 遍历时需要一系列的限制条件，可以利用 views::filter，增加可读性

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

int main() {
    int n = 12;

    std::vector<char> vis(n);
    std::vector<int> a(n);

    auto checkNotVisited = [&](int i) -> bool {
        return not vis[i];
    };

    auto checkValid = [&](int i) -> bool {
        return a[i] > 0;
    };

    for (auto i : std::views::iota(0, n) |
        std::views::filter(checkNotVisited) |
        std::views::filter(checkValid)) {
        // do something
    }

    return 0;
}
```

## 二分找第一个大于 5000 的数

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

int main() {
    const int K = 5000;

    auto rng = std::views::iota(0, 1'000'000'000); // 懒执行，不会真开 1E9 的元素。
    auto check = [&](auto x) -> bool { return not (x > K); };

    auto it = std::ranges::partition_point(rng, check);

    if (it != std::end(rng)) {
        std::cout << *it << "\n";
    }

    return 0;
}
```