基于范围的 `for` 循环（C++11）

- 在一个范围上执行 for 循环。
- 用作对范围中的各个值（如容器中的所有元素）进行操作的传统 for 循环的更加可读的等价版本。

可读性

```
int n; cin >> n;
vector<int> a(n);

// before
for (int i = 0; i < n; i++) cin >> a[i];

// after
for (auto& v: a) cin >> v;
```

值访问

```
struct foo {
    int type;
    string str;
    ll val;
};

vector<foo> a(n);

// before
for (int i = 0; i < n; i++) {
    auto v = a[i];
    // do something
}

// after
for (auto v: a) {
    // do something
}
```

使用示例

```
int main() {
    vector<int> v = {0, 1, 2, 3, 4, 5};//创建一个vector

    for (const int& i : v) // 以 const 引用访问
        cout << i << ' ';
    cout << '\n';

    for (auto i : v) // 以值访问，i 的类型是 int
        cout << i << ' ';
    cout << '\n';

    for (auto&& i : v) // 以转发引用访问，i 的类型是 int&
        cout << i << ' ';
    cout << '\n';
```

```
    const auto& cv = v;

    for (auto&& i : cv) // 以转发引用访问，i 的类型是 const int&
        cout << i << ' ';
    cout << '\n';

    for (int n : {0, 1, 2, 3, 4, 5}) // 初始化器可以是花括号包围的初始化器列表
        cout << n << ' ';
    cout << '\n';

    int a[] = {0, 1, 2, 3, 4, 5};
    for (int n : a) // 初始化器可以是数组
        cout << n << ' ';
    cout << '\n';

    for ([[maybe_unused]] int n : a)
        cout << 1 << ' '; // 不必使用循环变量
    cout << '\n';
}
```

优点

- 代码简洁，可读性更强
- 能够支持**值访问**操作

结构化绑定（C++17）

- 将指定名称绑定到初始化器的子对象或元素。
- 与引用类似，结构化绑定是既存对象的别名。不同于引用的是，结构化绑定的类型不必为引用类型。

引用类型的绑定

```
struct foo {
    int type;
    string str;
    ll val;
};

int n; cin >> n;
vector<foo> a(n);

// before
for (auto& f: a) {
    cin >> f.type >> f.str >> f.val;
}

// after
for (auto& [t, s, v]: a) cin >> t >> s >> v;
```

值类型的绑定

```
struct foo {
    int type;
    string str;
    ll val;
};

vector<foo> a(n);

// before
for (auto f: a) {
    ss += f.str;
    ans += dp[f.type] * f.val;
}

// after
for (auto& [t, s, v]: a) {
    ss += s;
    ans += dp[t] * v;
}
```

优点

- 代码简洁，可读性更强
- 可与其它新特性搭配使用

std::tuple（C++11）

- 类模板 `std::tuple` 是固定大小的异质值的汇集。它是 `std::pair` 的泛化。
- 一般搭配**结构化绑定使用**

使用示例

```
std::tuple<double, char, std::string> get_student(int id){
    switch (id) {
        case 0: return {3.8, 'A', "Lisa Simpson"};
        case 1: return {2.9, 'C', "Milhouse Van Houten"};
        case 2: return {1.7, 'D', "Ralph Wiggum"};
        case 3: return {0.6, 'F', "Bart Simpson"};
    }

    throw std::invalid_argument("id");
}

int main() {
    const auto student0 = get_student(0);
    std::cout << "ID: 0, "
              << "GPA: " << std::get<0>(student0) << ", "
              << "等级: " << std::get<1>(student0) << ", "
              << "姓名: " << std::get<2>(student0) << '\n';

    const auto student1 = get_student(1);
    std::cout << "ID: 1, "
              << "GPA: " << std::get<double>(student1) << ", "
              << "等级: " << std::get<char>(student1) << ", "
              << "姓名: " << std::get<std::string>(student1) << '\n';

    double gpa2;
    char grade2;
    std::string name2;
    std::tie(gpa2, grade2, name2) = get_student(2);
    std::cout << "ID: 2, "
              << "GPA: " << gpa2 << ", "
              << "等级: " << grade2 << ", "
              << "姓名: " << name2 << '\n';

    // 结构化绑定:
    const auto [gpa3, grade3, name3] = get_student(3);
    std::cout << "ID: 3, "
              << "GPA: " << gpa3 << ", "
              << "等级: " << grade3 << ", "
              << "姓名: " << name3 << '\n';
}
```

对结构体类型的简化

```
#define all(x) x.begin(), x.end()
struct foo {
    int type;
    string str;
    ll val;
};

// before
vector<foo> a(5);
for (auto& [t, s, v]: a) cin >> t >> s >> v;
sort(all(a), [](const auto& a, const auto& b){
   if (a.type == b.type && a.str == b.str) return a.val < b.val;
   else if (a.type == b.type) return a.str < b.str;
   return a.type < b.type;
});

// after
vector<tuple<int ,string, ll>> b(5);
for (auto& [t, s, v]: b) cin >> t >> s >> v;
sort(all(b));
```

优点

- 代码简洁，可读性更强
- 对 `pair` 进行扩展，意味着无需像结构体那样**重写排序函数**或**重载运算符**
- 同时也拥有默认构造函数

类模板实参推导（C++17）

- 为了实例化一个类模板，需要知晓每个模板实参，但并非每个模板实参都必须指定。在算法竞赛的大部分语境中，编译器会从初始化式的类型推导缺失的模板实参。

使用示例

```
// before
vector<vector<int>> e(n + 1, vector<int>(m + 1, 10));

// after
vector e(n + 1, vector(m + 1, 10));

// before
vector<tuple<int, int, ll, string>> b(n, {1, 2, 3, "bar"});

// after
vector b(n, tuple(1, 2, 3ll, string("bar")));
```

优点

- 代码简洁，可读性更强

std::format（C++ 20）

- 文本格式化库提供 `printf` 函数族的安全且可扩展的替用品。其意图是补充既存的 C++ I/O 流库。
- 注意 `format` 返回值是 `string` 类型，意味着它不止可以用于输出

使用示例

```
// before
cout << "Testcase #" << ++t << " :" << ans << endl;

int a; ll b; float c; db d; string e;
printf("%d, %lld, %f, %lf, %s", a, b, c, d, e.c_str());

// after
cout << format("Testcase #{} :{}", ++t, ans);

cout << format("{}, {}, {}, {}, {}", a, b, c, d, e);
```

优点

- 避免了 `cout` 在输出时在**常量**和**变量**之间的多次切换
- 避免了 `printf` 所需要的**格式说明符**，降低程序出错率
- 可以直接使用 `std::string`