```cpp
int main() {
    // Part 1: Iterator
    vector<int> v = {1, 2, 3, 4, 5};
    vector<int>::iterator start = v.begin();
    cout << start[0] << endl;  // []访问
    auto end = v.end();  // 使用auto
    cout << *(end - 1) << endl;  // 使用*访问
    auto t = *start;  // 只能用auto
    for (auto it = start; it != end; it++) {  // 使用for循环
        cout << *it << endl;
    }
    // 数组指针也可以当迭代器使用

    // Part 2: Container
    vector<int> a(start, end);  // 使用迭代器初始化，stack，queue等不能这样初始化，pq可以
    vector<int> b(5);  // 这里是实际尺寸，不是reserve的空间
    cout << b.size() << endl;  // 5
    vector<int> c(5, 4);  // 全部初始化为4
    a.reserve(0); a.pop_back(); a.push_back(0); a.back(); a.size();
    a.insert(a.begin() + 2, 10);  // a[2] = 10，vector没有front操作，只能这样插入
    a.erase(a.begin() + 2);  // 删除a[2]
    int* data = a.data();  // 返回指向vector的指针

    stack<int> s;
    s.push(1); s.top(); s.pop(); s.empty(); s.size(); s.empty();
    queue<int> q;
    q.push(1); q.front(); q.back(); q.pop(); q.empty(); q.size();
    // stack 和 queue 的 push 和 pop 全部不带 front 和 back
    // stack<int> s2(data, data+5); stack<int> s3(start, end); // 都不可以
    deque<int> dq;
    dq.push_back(1); dq.push_front(1); dq.pop_back(); dq.pop_front();
    dq.front(); dq.back(); dq.size(); dq.empty(); // 全部都要指定front和back

    priority_queue<int> pq;
    priority_queue<int> pq2(data, data + 5); // 可以
    priority_queue<int> pq3(start, end); // 可以
    priority_queue<int, vector<int>, greater<int>> pq4; // 指定运算符，最小堆

    // Part 3: 自增、自减
    int i = 0;
    cout << i++ << endl;  // 0
    cout << i << endl;  // 1
    cout << ++i << endl;  // 2
    cout << i << endl;  // 2

    // Part 4: 引用
    // 传参数时使用引用不会复制整个对象，可以修改原值
    // 使用const可以防止表面上的修改，但是内部还是可以修改
    // 返回const引用也是一样道理
    // 构造运算符，有输入都传const引用，不返回（新建时使用=也会调用这个）
    // 拷贝运算符，输入是const引用，输出引用（非const），返回*this，需要在内部定义一个新的变量
    // 重载[]，输入是const引用，输出是引用（非const）
    // 重载运算符，输入输出都是const引用，需要在内部定义一个新的变量，返回这个变量
    // 自身操作运算符，输入输出都是const引用，返回*this
```

Tiancheng Jiao
EECS 281
tcjiao

```cpp
    // Part 5: 自定义比较运算符
    // 在class中定义的，直接像 greater<int> less<int> 一样使用
    class IndexComparator {
      public:
        inline bool operator()(int a, int b) const {
            return zombies_ptr[a] > zombies_ptr[b];
        }
    };
    priority_queue<int, vector<int>, IndexComparator> orders; // 最小堆

    // bool operator<(const A& a, const B& b) {
    //     return true;
    // }
    cout << (a1 < b1) << endl;  // 1

    sort(v.begin(), v.end());  // 默认从小到大
    sort(v.begin(), v.end(), [](int a, int b) { return a > b; });  // 从大到小
    sort(v.begin(), v.end(), IndexComparator());  // 自定义比较函数
    // bool compareFunction(int a, int b) {
    //     return a > b; // 降序排序
    // }
    sort(v.begin(), v.end(), compareFunction);  // 从大到小

    // Part 6: string
    string s1 = "abcdef";
    s1.substr(1, 3);  // bcd，开始位置，长度
    s1.find("cd");  // 2，找到的第一个位置，找不到返回-1
    s1.length(); s1[0]; s1.empty(); s1.clear();
    s1.insert(1, "123");  // a123bcdef

    // Part 7: Pair
    pair<int, int> p1 = {1, 2};
    p1.first; p1.second;
}

// Part 8: Template
template<typename T>
T add(T a, T b) {
    return a + b;
}

template<typename T>
class Box {
  private:
    T content;
};
```

$$T(n) = kT(n/b) + O(n^d)$$

$$= \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d \end{cases}$$

Sort	Best	Average	Worst	Memory	Stable?	Adaptive?
Bubble	Ω(n)	Θ(n^2)	O(n^2)	O(1)	Yes	Yes
Selection	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)	No	No
Insertion	Ω(n)	Θ(n^2)	O(n^2)	O(1)	Yes	Yes
Heap	Ω(n log n) (distinct keys)	Θ(n log n)	O(n log n)	O(1)	No	No
Merge	Ω(n log n)	Θ(n log n)	O(n log n)	O(n)	Yes (if merge is stable)	No
Quick	Ω(n log n)	Θ(n log n)	O(n^2)	O(log n)	No	No