C++ 面向对象

极夜酱

目录

1 STL 标准模板库 **1**

1.1 模板 1

1.2 容器 7

1.3 STL 数组 10

1.4 STL 字符串 15

1.5 STL 链表 18

1.6 容器适配器 21

Chapter 1 STL 标准模板库

1.1 模板

1.1.1 泛型编程（Generic Programming）

面向对象编程（OOP）和泛型编程（GP）都能处理在编写程序时类型未知的情况，OOP 能处理运行时获取类型的情况，GP 能处理编译期可获取类型的情况。

模板是泛型编程的基础，泛型编程就是以一种独立于任何特定类型的方式编写代码。C++ 标准库的容器、迭代器、算法都是泛型编程的例子。

1.1.2 函数模板

通过定义一个通用的函数模板可以处理参数为多种类型的情形，而不是为每个类型都定义一个重载。模板定义使用 template 关键字，后跟模板参数列表。模板参数表示函数或类定义中用到的类型，使用模板时需要隐式或显式提供模板实参，将其绑定到模板参数。

函数模板

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <typename T>
7  inline T getMax(const T& val1, const T& val2) {
8      return val1 > val2 ? val1 : val2;
9  }
10
11 int main() {
12     int iVal1 = 28;
```

```
13      int iVal2 = 92;
14      cout << getMax(iVal1, iVal2) << endl;
15
16      double dVal1 = 3.14;
17      double dVal2 = 3.71;
18      cout << getMax(dVal1, dVal2) << endl;
19
20      string sVal1 = "hello";
21      string sVal2 = "world";
22      cout << getMax(sVal1, sVal2) << endl;
23      return 0;
24  }
```

运行结果

```
92
3.71
world
```

函数模板仅仅是函数的规范，本身并不会占用内存。当编译器遇到对模板函数的调用时，才会在内存中创建该函数的实例。

1.1.3 类模板

类模板用来生成类的蓝图，与函数模板不同的是，类模板在实例化时编译器无法为类模板推导模板参数类型，而是必须在模板名后用【<>】提供实参。根据显式提供的模板实参列表，编译器使用这些模板参数来实例化特定的类。

编译器从类模板实例化一个类时，会重写模板，将模板参数的每个实例替换为给定的模板实参。因此类模板的每个实例都是独立的类，使用不同模板实参实例化出的类之间没有关联，也没有特殊的访问权限。

类模板

```
1   #include <iostream>
```

```
2  #include <sstream>
3  #include <algorithm>
4
5  using namespace std;
6
7  template<class T>
8  class SortedArray {
9  public:
10     SortedArray(int capacity = 1);
11     SortedArray(T *arr, int capacity);
12     ~SortedArray();
13
14     string data();
15     void add(T val);
16     void remove(T val);
17
18 private:
19     T *arr;
20     int len;
21     int capacity;
22     void resize(int size);
23 };
24
25 template<class T>
26 SortedArray<T>::SortedArray(int capacity) {
27     this->len = 0;
28     this->capacity = capacity;
29     this->arr = new T[capacity];
30 }
31
32 template<class T>
33 SortedArray<T>::SortedArray(T *arr, int len) {
34     this->len = len;
35     this->capacity = len;
36     this->arr = new T[len];
37     for(int i = 0; i < len; i++) {
38         this->arr[i] = arr[i];
```

```
39      }
40  }
41
42  template<class T>
43  SortedArray<T>::~SortedArray() {
44      delete arr;
45  }
46
47  template<class T>
48  string SortedArray<T>::data() {
49      if(len == 0) {
50          return "[]";
51      }
52
53      sort(this->arr, this->arr + len);
54      ostringstream out;
55      out << "[";
56      for(int i = 0; i < len; i++) {
57          out << arr[i] << ", ";
58      }
59      out << "\b\b]";
60      return out.str();
61  }
62
63  template<class T>
64  void SortedArray<T>::resize(int size) {
65      T *temp = new T[size];
66      for(int i = 0; i < len; i++) {
67          temp[i] = arr[i];
68      }
69      delete arr;
70      arr = temp;
71  }
72
73  template<class T>
74  void SortedArray<T>::add(T val) {
75      if(len == capacity) {
```

```
76          capacity *= 2;
77          resize(capacity);
78      }
79      arr[len++] = val;
80  }
81
82  template<class T>
83  void SortedArray<T>::remove(T val) {
84      for(int i = 0; i < len; i++) {
85          if(arr[i] == val) {
86              arr[i] = arr[len-1];
87              len--;
88              if(len <= capacity / 2) {
89                  capacity /= 2;
90                  resize(capacity);
91              }
92              break;
93          }
94      }
95  }
96
97  int main() {
98      int arr[] = {7, 7, 3, 9, 7, 1, 3};
99      int n = sizeof(arr) / sizeof(arr[0]);
100
101     SortedArray<int> sortedArray(arr, n);
102     cout << sortedArray.data() << endl;
103
104     sortedArray.add(28);
105     sortedArray.add(12);
106     cout << sortedArray.data() << endl;
107
108     sortedArray.remove(7);
109     sortedArray.remove(9);
110     cout << sortedArray.data() << endl;
111
112     return 0;
```

```
113 }
```

运行结果

```
[1, 3, 3, 7, 7, 7, 9]
[1, 3, 3, 7, 7, 7, 9, 12, 28]
[1, 3, 3, 7, 7, 12, 28]
```

1.2 容器

1.2.1 容器（Container）

容器是特定类型对象的集合，容器分为顺序容器和关联容器：

- 顺序容器：元素的顺序与其加入容器的位置对应。
- 关联容器：元素的顺序由其关联的关键字决定，关联容器分为有序关联容器和无序关联容器。

所有容器类共享公有接口，不同容器按不同方式扩展。

C++ 新标准容器的性能比旧版本快很多，其性能与最精心优化过的同类数据结构一样好。现代 C++ 程序应该使用标准库容器，而不是更原始的数据结构。

1.2.2 顺序容器

每个容器都定义于一个头文件中，文件名与容器名相同。容器都定义为模板类，顺序容器几乎可以保存任意类型的元素，还可以在容器中保存容器。

顺序容器包括 array、vector、string、deque、list 和 forward_list。

容器	描述
array	固定大小数组，支持快速随机访问，不能增删元素
vector	可变大小数组，支持快速随机访问，非尾部位置增删较慢
string	专门用于保存字符，随机访问快，在尾部增删速度快
deque	双端队列，支持快速随机访问，在头尾位置增删速度很快
list	双向链表，支持双向顺序访问，在任何位置增删都很快
forward_list	单向链表，只支持单向顺序访问，在任何位置增删都很快

表 1.1: 顺序容器

array 和内置数组一样大小固定，但操作更安全。除固定大小的 array 外，其它容器都提供高效灵活的内存管理，可以添加、删除、扩展和收缩容器的大小。

vector 和 string 将元素存储在连续空间中，故通过下标的随机访问很快。在尾部添加元素很快，但中间和头部插入或删除很慢。添加元素可能造成空间的重新分配和元素拷贝。

deque 支持快速随机访问，在两端插入或删除很快，但在中间插入或删除元素很慢。

list 和 forward_list 的设计目的是让任何位置的插入或删除都快速高效且不需重新分配内存，但是不支持随机访问，为访问一个元素需要遍历整个链表。

1.2.3 迭代器（Iterator）

迭代器比下标访问更通用，所有标准库容器都支持迭代器，但只有几种支持下标。迭代器提供了对容器对象的间接访问，类似于指针。begin() 返回指向首元素的迭代器，end() 返回指向尾元素下一位置（尾后）的迭代器。如果容器为空，则 begin() 和 end() 返回的都是尾后迭代器。

任何可能改变容器容量的操作都会使容器的迭代器失效。

容器	**描述**
iterator	容器的迭代器
begin()	返回指向首元素的迭代器
end()	返回尾后迭代器
const_iterator	只读迭代器
cbegin()	返回指向首元素的只读迭代器
cend()	返回尾后只读迭代器
reverse_iterator	按逆序寻址元素的迭代器
rbegin()	返回指向尾元素的逆序迭代器
rend()	返回首前逆序迭代器
const_reverse_iterator	只读逆序迭代器
crbegin()	返回指向尾元素的只读逆序迭代器
crend()	返回首前只读逆序迭代器

表 1.2: 迭代器

迭代器可以进行算术运算，将迭代器与整数相加减可以得到向前或向后若干位置的迭代器。使用关系运算符【<】、【<=】、【>】、【>=】和【==】可以对迭代器所指位置比较大小。将两个迭代器相减，结果是两个迭代器的距离。

迭代器

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s = "hello world";

    string::iterator iter = s.begin();
    cout << "[";
    while(iter != s.end()) {
        cout << *iter << ", ";
        iter++;
    }
    cout << "\b\b]" << endl;

    return 0;
}
```

运行结果

```
[h, e, l, l, o,  , w, o, r, l, d]
```

1.3 STL 数组

1.3.1 array

array 容器是 C++11 标准中新增的序列容器，它在普通数组的基础上，添加了一些成员函数和全局函数。在使用上，它比普通数组更安全，且效率并没有因此变差。和其它容器不同，array 的大小是固定的，无法动态的扩展或收缩。与内置数组不同的是，array 允许做整个容器的拷贝和赋值，要求两个 array 大小和元素类型都一样。

array 以类模板的形式定义在 <array> 头文件，array 具有固定大小，其大小也是类型的一部分，定义时模板参数包含元素类型和大小。

成员函数	功能
size()	返回容器中当前元素的数量
max_size()	返回容器可容纳元素的最大数量
empty()	判断容器是否为空
at(n)	返回容器中第 n 个元素的引用
front()	返回容器中第一个元素的直接引用
back()	返回容器中最后一个元素的直接应用
data()	返回一个指向容器首个元素的指针
fill(val)	将 val 赋值给容器中的每个元素
arr1.swap(arr2)	交换相同长度和类型的 arr1 和 arr2 中所有元素

表 1.3: array 成员函数

array

```
1 #include <iostream>
2 #include <array>
3
4 using namespace std;
5
6 int main() {
7     array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
8     cout << "size = " << arr.size() << endl;
9     array<int, 10>::iterator begin = arr.begin();
10    array<int, 10>::iterator end = arr.end();
11    while(begin != end) {
12        cout << *begin << " ";
13        begin++;
14    }
15    cout << endl;
16    return 0;
17 }
```

运行结果

```
size = 10
0 1 2 3 4 5 6 7 8 9
```

1.3.2 vector

vector 表示对象的集合，由于 vector 容纳其它的对象，所以是一种容器。使用 vector 需要包含头文件 <vector>。vector 是一个类模板，模板可以看作编译器生成类或函数的一份说明。

vector 的初始化包括直接初始化、拷贝初始化和列表初始化。

初始化	**功能**
vector<T> v	创建一个空的 vector
vector<T> v2(v1)	用 v1 中所有元素的副本创建 v2
vector<T> v2 = v1	等价于 v2(v1)
vector<T> v(n, val)	v 中包含了 n 个值为 val 的元素
vector<T> v(n)	v 中包含了 n 个值为默认初始化的元素
vector<T> va, b, c, ...	用列表元素初始化 v
vector<T> v = a, b, c, ...	等价于 va, b, c, ...
vector<T> v (begin, end)	根据迭代器范围 [begin, end) 复制到 vector 中

表 1.4: vecor 初始化

vector 初始化

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

template <typename T>
ostream& operator<<(ostream& out, const vector<T>& v) {
    if(!v.empty()) {
        out << "[";
        copy(v.begin(), v.end(), ostream_iterator<T>(out, ", "));
        out << "\b\b]";
    }
    return out;
}

int main() {
    vector<int> v1(10);          // 有10个元素，都是0
    vector<int> v2{10};          // 有1个元素，值是10
    vector<int> v3(10, 1);       // 有10个元素，都是1
    vector<int> v4{10, 1};       // 有2个元素，10和1
    vector<string> v5{"hello"}; // 有1个元素，是字符串"hello"

    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;
    cout << "v3 = " << v3 << endl;
    cout << "v4 = " << v4 << endl;
    cout << "v5 = " << v5 << endl;
    return 0;
}
```

运行结果

```
v1 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
v2 = [10]
v3 = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
v4 = [10, 1]
v5 = [hello]
```

1.3.3 vector 操作

操作	功能
v.empty()	判断 vector 是否为空
v.size()	返回 vector 元素个数
v[n]	返回 vector 中第 n 个元素的引用
v1 = v2	用 v2 中的元素拷贝替换 v1 中的元素
v1 == v2、v1 != v2	v1 和 v2 相等当且仅当元素个数和对应元素都相同
v.push_back(val)	向 vector 尾部添加一个元素
v.insert(iter, val)	向迭代器指向元素前添加一个元素
v.pop_back()	删除 vector 最后一个元素
v.erase(iter)	删除迭代器指向元素
v.erase(begin, end)	删除迭代器返回 [begin, end) 范围元素
v.clear()	清空 vector
v.swap(vector)	交换两个同类型 vector 数据
v.assign(n, val)	设置 vector 中前 n 个元素值为 val

表 1.5: vector 操作

vector 不能使用下标添加元素，否则会造成缓冲区溢出，确保下标合法的一种有效手段就是尽可能使用 for-each 循环。如果循环体内部包含向 vector 添加元素的语句，则不能使用 for-each 循环。

vector

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v;
    for(int i = 0; i < 10; i++) {
        v.push_back(i * i);
    }

    for(int& item : v) {
        cout << item << " ";
    }
    cout << endl;
    return 0;
}
```

运行结果

```
0 1 4 9 16 25 36 49 64 81
```

1.4 STL 字符串

1.4.1 string

string 是标准库中的类型，表示可变长字符序列，使用需要包含头文件 <string>。

string 的初始化分为：

1. 直接初始化：使用括号初始化，调用构造函数。
2. 拷贝初始化：使用赋值初始化，调用重载的赋值运算符。

string 初始化

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s1;              // 默认初始化，为空字符串
    string s2(s1);          // 直接初始化，s2是s1的副本
    string s3 = s1;         // 拷贝初始化，s3是s1的副本，等价s3(s1)
    string s4("hello");     // 直接初始化，初始化为字面值常量
    string s5 = "hello";    // 拷贝初始化，初始化为字面值常量
    string s6(10, 'x');     // 直接初始化，初始化为10个字符'x'

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    cout << "s3 = " << s3 << endl;
    cout << "s4 = " << s4 << endl;
    cout << "s5 = " << s4 << endl;
    cout << "s6 = " << s4 << endl;

    return 0;
}
```

运行结果

```
s1 =
s2 =
s3 =
s4 = hello
s5 = hello
s6 = hello
```

1.4.2 string 操作

操作	功能
out « s	将 s 写到输出流 out 中
in » s	从输入流中读取字符串赋给 s，字符串以空白符分割
getline(in, s)	从输入流中读取一行赋给 s
s.empty()	判断 s 是否为空
s.size()	返回 s 中字符个数
s[n]	返回 s 中第 n 个字符的引用
s1 + s2	返回 s1 和 s2 连接后的结果
s1 = s2	用 s2 的副本替换 s1
s1 == s2、s1 != s2	判断 s1 和 s2 是否相等
<、<=、>、>=	字典序比较，对大小写敏感
s1.append(s2)	尾部插入
s1.insert(pos, s2)	在第 pos 个位置插入 s2
s.erase(pos, n)	从第 pos 个位置删除 n 个字符
s1.replace(pos, n, s2)	从第 pos 个位置开始替换 n 个字符为 s2
s.substr(pos, n)	返回一个从 pos 开始的 n 个字符的拷贝
s1.find(s2)	查找 s1 中 s2 第一次出现的位置
s1.rfind(s2)	查找 s1 中 s2 最后一次出现的位置

表 1.6: string 操作

string

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    string s("Hello");

    s.append("world");          // Helloworld
    s.insert(s.size(), "!");    // Helloworld!

    s.replace(1, 4, "i");       // Hiworld!
    s.erase(2, 5);              // Hi!
    s.insert(2, " C++");        // Hi C++!

    cout << s << endl;

    cout << s.substr(3, 3) << endl;
    cout << s.substr(3) << endl;
    cout << s.find("C++") << endl;

    return 0;
}
```

运行结果

```
Hi C++!
C++
C++!
3
```

1.5 STL 链表

1.5.1 list

list 双向链表通过指针连成逻辑意义上的线性表，由于 list 中结点并不要求在一段连续内存中，因此 list 不支持快速随机存取，迭代器只能通过【++】或【–】移动。

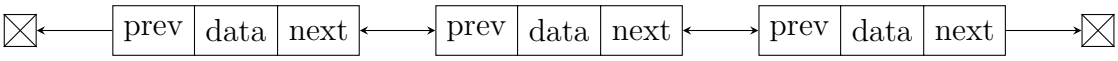

图 1.1: 双向链表

操作	功能
list<T> lst	创建空的 list
list<T> lst(n)	创建含有 n 个元素的 list
list<T> lst1(lst2)	使用 lst2 初始化 lst1
lst.size()	返回 list 元素个数
lst.clear()	删除所有元素
lst.empty()	判断 list 是否为空
lst.front()	返回第一个元素
lst.back()	返回最后一个元素
lst.insert()	插入一个元素
lst.erase()	删除一个元素
lst.push_front()	在头部添加一个元素
lst.push_back()	在尾部添加一个元素
lst.pop_front()	删除第一个元素
lst.pop_back()	删除最后一个元素
lst.remove()	删除元素
lst.reverse()	反转 list
lst.sort()	排序
lst.unique()	去除相邻的重复元素
lst.merge()	合并两个有序 list

表 1.7: list 操作

其中，lst.unique() 并不是把重复的元素删除，而是全部放到数组尾部，返回去重后的尾地址。unique() 中不自带 sort()，因此需要先使用 sort() 进行排序。

list

```cpp
#include <iostream>
#include <list>

using namespace std;

void printList(list<int> lst) {
    for(list<int>::iterator iter = lst.begin();
                            iter != lst.end();
                            iter++) {
        cout << *iter << " ";
    }
    cout << endl;
}

int main() {
    list<int> lst;

    lst.push_back(11);       // [11]
    lst.push_front(22);      // [22, 11]
    cout << lst.front() << endl;    // 22
    cout << lst.back() << endl;     // 11

    lst.insert(++lst.begin(), 3);   // [22, 3, 11]
    lst.insert(--lst.end(), 2);     // [22, 3, 2, 11]
    lst.push_back(2);               // [22, 3, 2, 11, 2]
    printList(lst);

    lst.pop_front();                // [3, 2, 11, 2]
    lst.sort();                     // [2, 2, 3, 11]
    lst.unique();                   // [2, 3, 11]
    printList(lst);
```

```
32
33      lst.sort();                         // [2, 3, 11]
34      printList(lst);
35
36      list<int> lst2{1, 2, 8};
37      lst.merge(lst2);                    // [1, 2, 2, 3, 8, 11]
38      printList(lst);
39
40      return 0;
41  }
```

运行结果

```
22
11
22 3 2 11 2
2 3 11
2 3 11
1 2 2 3 8 11
```

1.5.2 forward_list

forward_list 和 list 的区别在于前者是单向链表，每个元素内部只有一个链接指向下一个元素，因此在存储方面 list 会消耗更多的空间。

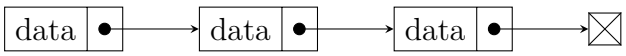

图 1.2: 单向链表

forward_list 不支持反向迭代器，并且没有指向尾元素的迭代器，因此不提供 back()、push_back()、pop_back() 等操作。

1.6 容器适配器

1.6.1 stack

栈，又名堆栈，是一种运算受限的线性数据结构，栈只能在表尾进行插入和删除操作。

栈中的元素只能先进后出（FILO, First In Last Out）。最早进入栈的元素所存放的位置叫作栈底（bottom），最后进入栈的元素存放的位置叫作栈顶（top）。