directory

- 命名空间
- STL
  - pair
  - tuple
  - vector
  - list
  - set
  - map
- 类模板

命名空间

- `namespace`
- `std`
- 使用整个命名空间 `using namespace A`
- 使用命名空间中部分成员 `using A::func1`

```
#include<iostream>
namespace A{
    int x;
    static int y;
}
int main(){
    using A::x;
    x = 1;
    std::cout << x << std::endl;
    A::y = 2;
    std::cout << A::y << std::endl;
}
```

STL

标准模板库
算法、容器、函数、迭代器

- 容器：简单容器、序列容器、关系容器
  `pair`

```
template<class T1, class T2>struct pair {
    T1 first;
    T2 second;
};
```

```
#include <string>
int main(){
  std::pair<std::string, double> p1("Alice", 90.5);
  std::pair<std::string, double> p2;

  p2.first = "Bob";
  p2.second = 85.0;

  auto p3 = std::make_pair("David", "95.0");
  return 0;
}
```

手动实现

```
#include<iostream>
template<class T1, class T2>
class pair {
private:
    T1 a;
    T2 b;
public:
    pair() : a(), b(){};
    pair(const T1& f, const T2& s) : a(f), b(s) {}
    static pair make_pair(const T1& f, const T2& s) {
        return pair<T1, T2>(f, s);
    }
    T1& first() {return a;}
    T2& second() {return b;}
    T1 first() const {return a;}
    T2 second() const {return b;}
};
int main() {
    pair<int, std::string>arr = pair<int, std::string>::make_pair(1, "123");
    std::cout << arr.first() << " " << arr.second() << std::endl;
    arr.first() = 2;
    arr.second() = "abc";
    std::cout << arr.first() << " " << arr.second() << std::endl;
}
```

`tuple`

- `auto t1 = std::make_tuple("abc", 1, 2.3, 'a');` 创建
- `std::get` 函数获取数据
  - 其下标需要在编译时确定：不能设定运行时可变的长度，不能当做数组
- `std::tie` 函数返回左值引用的元组

```
 v0 = std::get<0>(tuple1);
 v1 = std::get<1>(tuple2);
```

```
int i = 0;
v = std::get<i>(tuple); //编译错误
```

```
std::string x; double y; int z;
std::tie(x, y, z) = std::make_tuple("abc", 7.8, 123);
//等价于 x = "abc"; y = 7.8; z = 123
```

用于函数多返回值的传递：

```
#include <tuple>
#include <iostream>
std::tuple<int, double> f(int x){
    return std::make_tuple(x, double(x)/2);
}
int main() {
    int xval;
    double half_x;
    std::tie(xval, half_x) = f(7);
    std::cout << xval << " " << half_x << std::endl;
}
```

`vector`

- 创建
  - `std::vector<int>x;`
  - `vector<vector<int>> m(M,vector<int>(N));` // 行M,列N,值为0
  - `vector<vector<int>> m(M);` // 行M,列不固定
- 数组长度 `x.size();`
- 清空 `x.clear();`
- 末尾添加/删除 `x.push_back(1)`，`x.pop_back();`
- 中间添加/删除，使用迭代器
  - `x.insert(x.begin()+1, 5);`
  - `x.erase(x.begin()+1);`

迭代器

- `vector<int>::iterator iter;`
  - 定义了一个名为iter的变量，它的数据类型是由`vector<int>`定义的iterator类型。
- `begin` 和 `end` 函数构成所有元素的左闭右开区间
- begin函数：`x.begin()`，返回vector中第一个元素的迭代器
- end函数：`x.end()`，返回vector中最后一个元素之后的位置的迭代器
- 下一个元素：`++iter`
- 上一个元素：`--iter`
- 下n个元素：`iter += n`
- 上n个元素：`iter -= n`
- 访问元素值——解引用运算符 `*`：`*iter = 5;`
  - 解引用运算符返回的是左值引用
- 迭代器移动：与整数作加法: `iter += 5;`
- 元素位置差：迭代器相减: `int dist = iter1 - iter2;`
- 本质都是重定义运算符

```cpp
#include <iostream>
#include <vector>
int main() {
    using std::cout; using std::endl;
    std::vector<int> vec = {1,2,3,4,5};
    cout << vec.end() - vec.begin() << endl;
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it *= 2; cout << *it << " ";
    }
    return 0;
}
/*
5
2 4 6 8 10
*/
```

- C++11中按范围遍历vector, 直接利用元素: `for(auto x : vec)`
- 与使用 *it(it是指向元素的指针)等价: `for(vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)`

迭代器失效

- insert/erase后，所修改位置**之后**的所有迭代器失效（原先的内存空间存储的元素被改变）
- vector是会自动扩展容量的数组
- size达到capacity，会另外申请一片capacity*2的空间，并整体迁移vector中的内容
- 整体迁移使所有迭代器失效

```cpp
vector<int> vec = {1,2,3,4,5};
auto first = vec.begin();
auto second = vec.begin() + 1;
auto third = vec.begin() + 2;
auto ret = vec.erase(second);
cout << *ret << endl;
//first指向1，second和third失效
//ret指向3
```

`list` 链表容器

- 底层实现是双向链表
- `template<class T, class Allocator = std::allocator<T>>class list;`
  - Allocator 是用于内存分配的一个泛型类模板
  - `Allocator = std::allocator<T>` 表明分配器类型默认为 `std::allocator<T>`
- 不支持下标等随机访问
- 支持高速的在任意位置插入/删除数据
- 访问主要依赖迭代器
- 插入和删除操作不会导致迭代器失效（除指向被删除的元素的迭代器外）
- 插入前端：`l.push_front(1);`
- 插入末端：`l.push_back(2);`
- 查询：`std::find(l.begin(), l.end(), 2);` //返回迭代器
- 插入指定位置：`l.insert(it, 4);` //it为迭代器

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

int main() {
    using std::cout;
    std::list<int>l;
    l.push_front(1);
    l.push_front(2);
    l.push_back(3);
    for(auto it = l.begin(); it != l.end(); it ++)
        cout << *it << " ";
}
// 2 1 3
```

`set` 无序集合

- 不重复元素构成的无序集合

```
`template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<T>
>class list;
```

- 内部按大小顺序排列，比较器由函数对象Compare完成。
- 注意：无序是指不保持插入顺序，容器内部排列顺序是根据元素大小排列。
- 定义：`std::set<int> s;`
- 插入（不允许出现重复元素）：`s.insert(val);`
- 查询值为val的元素：`s.find(val);` //返回迭代器
- 删除：`s.erase(s.find(val));` //导致迭代器失效
- 统计：`s.count(val);` //val的个数，总是0或1

`map` 关联数组

- 每个元素由两个数据项组成，map将一个数据项映射到另一个数据项中
- 键值对
    - key
    - value

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator =
        std::allocator<std::pair<const Key, T> >
> class map;
```

- 值类型为`pair<Key, T>`
- map中的元素key必须互不相同
- 可以通过下标访问（即使key不是整数）。下标访问时如果元素不存在，则创建对应元素
- `insert`函数进行插入

```cpp
#include <string>
#include <map>
int main() {
    std::map<std::string, int> s;
    s["Monday"] = 1;
    s.insert(std::make_pair(std::string("Tuesday"), 2));
    return 0;
}
```

- 查询键为key的元素：`s.find(key);` // 返回迭代器
- 统计键为key的元素个数：`s.count(key);` // 返回0或1
- 删除：`s.erase(s.find(key));` // 导致被删元素的迭代器失效
- map常用作稀疏数组或以字符串为下标的数组

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

int main() {
    std::map<std::string, std::string> M;
    M["fp"] = "c";
    M["oop"] = M["fp"] + "++"; // M["oop"] = "c++"
    for(auto it = M.begin(); it != M.end(); it ++) {
        std::cout << it->first << ":" << it->second << std::endl;
    }
    for(auto const& [key, value] : M) {
        std::cout << key << ":" << value << std::endl;
    }
}
//都输出
//fp:c
//oop:c++
```

Set和Map所用到的数据结构都是红黑树（一种二叉平衡树）
几乎所有操作复杂度均为O(logn)

- 序列容器：vector、list
- 关联容器：set、map
- 序列容器与关联容器的区别：
    - 序列容器中的元素有顺序，可以按顺序访问
    - 关联容器中的元素无顺序，可以按数值（大小）访问
    - vector中插入删除操作会使操作位置之后全部的迭代器失效
    - 其他容器中只有被删除元素的迭代器失效

类模板

提高代码复用性

```cpp
#include <iostream>
using namespace std;

template <typename T> class A {
    int data;
public:
```

```
    T sum(T a, T b) { return a + b; }
};
int main() {
    A<int> a;
    cout << a.sum(1, 2) << endl;
    A<string> s;
    cout << s.sum("12", "ab") << endl;
}
// 3
// 12ab
```

函数模板特化

```
#include <iostream>
using namespace std;

template<class T>
T div2(const T& val) {
    cout << "using template" << endl;
    return val / 2;
}

template<>//函数模板特化
int div2(const int& val) {
    cout << "better solution!" << endl;
    return val >> 1; //右移取代除以2
}

int main() {
    cout<<div2(1.5) << endl;
    cout<<div2(2) << endl;
    return 0;
}
/*
using template
0.75
better solution!
1
*/
```

```
#include <iostream>
using namespace std;

template<class T, class A>
T sum(const A& val1, const A& val2) {
    cout << "using template" << endl;
    return T(val1 + val2);
}

template<class A>
int sum(const A& val1, const A& val2) {
   //不是部分特化，而是重载函数
    cout << "overload" << endl;
    return int(val1 + val2);
}
```

```
int main() {
    float y = sum<float, float>(1.4, 2.4);
    cout << y << endl;
    int x = sum(1, 2);
    cout << x << endl;
    return 0;
}
```

函数模板重载解析顺序：

- 类型匹配的普通函数--->基础函数模板--->全特化函数模板
    - 如果有普通函数且类型匹配，则直接选中，重载解析结束
    - 如果没有类型匹配的普通函数，则选择最合适的基础模板
    - 如果选中的基础模板有全特化版本且类型匹配，则选择全特化版本，否则使用基础模板

```
#include <iostream>
using namespace std;

template<class T> void f(T) {
  //func1为基础模板
  cout<< "full template" <<endl;
}

template<class T> void f(T*) {
  //func2为func1的重载，仍是基础模板
  cout<< "full template -> overload template" <<endl;
}

template<> void f(char*) {
  //func3为func2的特化版本(T特化为char)
  cout<< "overload template -> specialized" <<endl;
}

int main() {
    char *p;
    f(p);
    return 0;
}
// overload template -> specialized
```

类模板特化

- 全部特化
- 部分特化

```
#include <iostream>
using namespace std;
template<typename T1, typename T2> class Sum { // 类模板
public:
    Sum(T1 a, T2 b) {
        cout << "Sum general: " << a + b << endl;
    }
};

template<> class Sum<int, int> { // 类模板全部特化
public:
```

```
    Sum(int a, int b) {
        cout << "Sum specific: " << a + b << endl;
    }
};

template<typename T1> class Sum<T1, int> { // 类模板部分特化
public:
    Sum(T1 a, int b) {
        cout << "part specific: " << a + b << endl;
    }
};

int main(){
    Sum<int, int> s1(1, 2);
    Sum<int, double> s2(1, 2.5);
    Sum<double, int>s3(1.1, 1);
    return 0;
}

/*
Sum specific: 3
Sum general: 3.5
part specific: 2.1
*/
```

总结

- 类模板可以部分特化或者全部特化，编译器会根据调用时的类型参数自动选择合适的模板类
- **函数模板**只能**全部特化**，但可以通过重载代替部分特化的实现。编译器在编译阶段决定使用特化函数或者标准模板函数
- 函数模板的**全特化**版本的**匹配优先级**可能低于**重载的非特化基础函数模板**，因此最好不要使用全特化函数模板而**直接使用重载函数**