
第9课 模板、STL

内容概述

1. 模板：引入
2. 函数模板
3. 类模板
4. 控制实例化
5. 特例化
6. 静多态和动多态
7. Traits
8. 引用折叠
9. 完美转发
10. 模板元编程

1. STL概述
2. STL组件
3. STL容器
4. STL迭代器
5. 顺序容器:vector
6. 顺序容器:array
7. 顺序容器:deque
8. 顺序容器:list
9. 顺序容器:forward_list
10. 容器适配器

模板：引入

模板是C++泛型编程的基础。一个模板就是一个创建类或者函数的蓝图。

模板就是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现了真正的代码可重用性。模版分为两类，一个是函数模版，另外一个为是类模版。

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

//通过函数重载方式的swap:
//虽然函数实现方式雷同,
//但是:有一种类型就要写一个函数

void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}

void swap(double& a, double& b) {
    double tmp = a;
    a = b;
    b = tmp;
}
```

抽象：将类型变为可传入的参数

//用函数模板，只写一套代码解决问题

```
template<typename T>
void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a = 10, b = 20;
    swap(a, b);
    double a1 = 1.0, b1 = 2.0;
    swap(a1, b1);
    const char * c1 = "abc", *c2 = "123";
    ::swap(c1, c2);
    ::swap<const char*>(c1, c2);
    char cc1 = '1', cc2 = '2';
    ::swap(cc1, cc2);
    ::swap<char>(cc1, cc2);
    return 0;
}
```

函数模板：模板类型参数

函数模板的格式：

`template <typename 形参名1, typename 形参名2,> //typename可以写为class`

`返回类型 函数名(参数列表) {函数体}`

<>括号中的参数叫**模板形参**（类似函数形参），编译器根据传入的参数(或实参推导)确定实际的模板实参类型，并**实例化**。

参数类型：**模板类型参数**，**非类型模板参数**。

```
class A {};  
class B {  
public:  
    bool operator<(const B& rhs)const {  
        //...  
        return true;  
    }  
};
```

```
//参数类型,可以作为返回类型使用  
//可以值返回(包括指针返回),引用返回  
template<typename T1,typename T2>  
T1 *f1(T1& t1, T2& t2) {  
    //...  
    return &t1;  
};
```

```
//调用myswap的类型T必须支持拷贝构造和赋值  
template<typename T>  
void myswap(T& a, T& b) {  
    T tmp = a;  
    a = b;  
    b = tmp;  
}  
//调用compare的类型T必须支持 < 操作  
template<typename T>  
int compare(const T& a, const T& b) {  
    if (a < b) return -1;  
    if (b < a) return 1;  
    return 0;  
}
```

```
int main() {  
    A a1, a2; //类类型  
    myswap(a1, a2);  
    myswap<A>(a1, a2);  
  
    //compare(a1, a2); 错(A没有<操作)  
    B b1, b2;  
    compare(b1, b2);  
  
    int i1 = 1, i2 = 2;  
    f1(i1, i2); //T1 T2 都是 int类型  
    double d1 = 1.0;  
    f1(i1, d1); //T1 int,T2 double  
    return 0;  
}
```

函数模板：非类型模板参数

参数类型：**非类型模板参数**。

如：`template<typename T, int N> void fun(T t){}`; 其中**int N**就是**非类型的模板形参**。

在模板的内部是**常量**。只能是**整型**、对象的**指针或引用**，必须能在**编译时确定**。(局部变量、局部对象的地址和引用都不行，`const`类型的整形变量可以，全局对象或**static**对象的地址和引用可以)

//创建静态数组,类型为T,长度为N, 返回其首地址

```
template<typename T,int N>
T* f1(const T& t) {
    static T tmp[N];
    for (auto &tt : tmp) tt = t;
    return tmp;
}
```

//比较2个字符串常量 注意：内联声明inline的位置

```
template<unsigned M,unsigned N>
inline int compare(const char(&p1)[M], const char(&p2)[N]) {
    cout << "M=" << M << ",N=" << N << endl;
    return strcmp(p1, p2);
}
```

```
int main() {
    int *p = f1<int,10>(33);
    for (int i = 0; i < 10; i++)
        cout << *p++ << " ";
    cout << endl;
    int flag = compare("abc","abcd");
    return 0;
}
```

```
33 33 33 33 33 33 33 33 33 33
M=4,N=5
请按任意键继续. . .
```

编译时：编译器通过函数实参来**推断**模板实参。无法推断,则必须显式给出。

函数模板：实例化

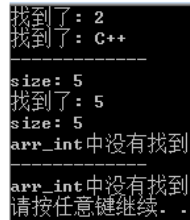
当调用一个函数模板时，编译器通常会利用给定的函数实参来推断模板参数，用此实际实参来代替模板参数来创建出模板的一个新的“实例”，也就是一个真正可以调用的函数，这个过程称为实例化。

```
vector<int>::iterator my_find(const vector<int>::iterator& b, const vector<int>::iterator& e, const int& value);  
list<string>::iterator my_find(const list<string>::iterator& b, const list<string>::iterator& e, const string& value);  
int* my_begin(int(&arr)[5]);  
int* my_end(int(&arr)[5]);
```

实例化了4个函数。 编译时实例化，模板要放在头文件中。

```
#include <iostream>  
#include <vector>  
#include <list>  
#include <string>  
using namespace std;  
//模仿std::find,在两个迭代器之间找到匹配的元素,返回迭代器  
//T为迭代器类型,V为元素类型  
template<typename T,typename V>  
T my_find(const T& b,const T& e,const V& value) {  
    T it = b;  
    while (it != e && *it != value)  
        ++it;  
    return it;  
}  
//编写my_begin,my_end,返回数组的头尾指针  
template<typename T,unsigned N>  
T* my_begin(T(&arr)[N]) {  
    cout << "size: " << N << endl;  
} return &arr[0];  
template<typename T,unsigned N>  
T* my_end(T(&arr)[N]) {  
    return &arr[N - 1];  
}
```

```
int main() {  
    vector<int> vec = { 1,4,7,2,5,8 };  
    vector<int>::iterator it1 = my_find(vec.begin(),vec.end(),2);  
    if (it1 != vec.end()) cout << "找到了: " << *it1 << endl;  
    else cout << "vec中没有找到\n";  
    list<string> ls = { "abc","C++","C#" };  
    list<string>::iterator it2 = my_find(ls.begin(), ls.end(), "C++");  
    if (it2 != ls.end()) cout << "找到了: " << *it2 << endl;  
    else cout << "list中没有找到\n";  
    cout << "-----\n";  
    int arr_int[] = { 1,3,5,7,9 };  
    int * p_int = my_find(my_begin(arr_int), my_end(arr_int),5);  
    if (p_int != my_end(arr_int)) cout << "找到了: " << *p_int << endl;  
    else cout << "arr_int中没有找到\n";  
    p_int = my_find(my_begin(arr_int), my_end(arr_int), 6);  
    if (p_int != my_end(arr_int)) cout << "找到了: " << *p_int << endl;  
    else cout << "arr_int中没有找到\n";  
    cout << "-----\n";  
    p_int = std::find(std::begin(arr_int), std::end(arr_int), 6);  
    if (p_int != std::end(arr_int)) cout << "找到了: " << *p_int << endl;  
    else cout << "arr_int中没有找到\n";  
    return 0;  
}
```



类模板

类模板：用来生成类的蓝图。编译器**不能**为类模板**推断**模板参数类型，必须提供实参列表。

类模板的格式：**template <typename 形参名1, typename 形参名2,> class 类名 { ... }**

```
template <typename T,int MAXSIZE>
class MyStack{
public:
    MyStack():size(0){}
    MyStack(const MyStack& other):size(other.size){ //类内实现
        for (int i = 0; i < size; i++)
            data[i] = other.data[i];
    }
    MyStack& operator=(const MyStack& other);
    bool empty()const { return size == 0; }//类内实现,隐式内联
    bool full()const { return size == MAXSIZE; }
    void push(const T& elem) {
        assert(!full()); data[size++] = elem;
    }
    void pop() { assert(!empty()); size--; }
    inline T& top();
private:
    T data[MAXSIZE]; //栈数组
    int size; //元素个数
};
```

1. 一个类模板的**每个实例**都会形成一个**独立的类**，类和类之间没有任何关联。
2. 可以在类内部和外部定义类模板的成员函数，模板内的**隐式声明为inline**。
3. 默认情况下，对于一个实例化的类模板，其**成员只有在使用时才被实例化**。
4. 在类模板自己的作用域中，可以**直接使用模板名**而不提供实参。

```
template<typename T,int MAXSIZE>
MyStack<T, MAXSIZE>& MyStack<T, MAXSIZE>::operator=
    (const MyStack/*<T, MAXSIZE>*/ &other){
    if (this != &other) {
        size = other.size;
        for (int i = 0; i < size; i++)
            data[i] = other.data[i];
    }
    return *this;
}
template<typename T,int MAXSIZE>
T& MyStack<T,MAXSIZE>::top() {
    assert(!empty());
    return data[size - 1];
}
int main() {
    MyStack<int, 10> st1;
    st1.push(1);
    st1.pop();
    MyStack<int, 20> st2;
    st2.empty();
    //st1和st2是同一类型的对象吗？
    //MyStack<int, 10> 和
    //MyStack<int, 20>实例化的成员函数？
    return 0;
}
```

类模板：static成员、默认参数

类模板中的**static成员**，这些成员由**相同类型的实例共享**。

模板可以设置**默认实参**，如：`template <typename T = int> class A{}`，使用时`A<> a;`(尖括号不能省)默认参数的位置习惯上靠右。

```
template <typename T=int>
class A {
public:
    A(const T& t) :data(t) { count++; }
    ~A() { count--; }
    static int get_count() { return count; }
private:
    T data;
    static int count;
};
template<typename T>
int A<T>::count = 0;
```

```
int main() {
    A<char> a1(' '),a2('c');
    //实例化 A<char>,该类对象共享 A<char>::count
    cout << A<char>::get_count() << endl; //2

    A<> b1(0), b2(0), b3(0);
    //实例化 A<int>,默认值是int,可写为 A<>
    //该类对象共享A<int>::count
    cout << A<>::get_count() << endl; //3
    cout << A<int>::get_count() << endl; //3
    cout << b1.get_count() << endl; //3
    return 0;
}
```


使用类的类型成员

```
template <typename T,unsigned N=10>
class A {
public:
    //T* 起个别名, 注意类外使用A<T,N>::iterator的方式
    typedef T* iterator;
    typedef const T* const_iterator;
    A(std::initializer_list<T> il) {
        count++;
        int i = 0;
        for (auto it=il.begin();it!=il.end();++it,i++)
            data[i] = *it;
    }
    ~A() { count--; }
    static int get_count() { return count; }
    iterator begin() { return &data[0]; }
    iterator end() { return &data[N - 1]; }
public:
    static int count;
private:
    T data[N];
};
template<typename T,unsigned N>
int A<T,N>::count = 0;
```

```
template <typename T,typename V>
typename T::iterator my_find(T &c, const V& value) {
    cout << typeid(c).name() << endl;
    typename T::const_iterator * c_it; //类的类型成员
    T::count * 3; //static成员
    typename T::iterator it = c.begin();
    while (it != c.end() && *it != value)
        ++it;
    return it;
}

template<typename T>
class B {
public:
    typename T::iterator first(T& t) {
        return t.begin();
    }
};
```

使用 `T::xxx` 时, 假如xxx是一种类型, 那么前面加上 `typename` (不能使用class)

```
int main() {
    A<int> a1 = { 1,2,3,4,5 };
    auto it = my_find(a1, 3);
    if(it!=a1.end())
        cout << *it << endl; //3

    B< A<int> > b1;
    cout << *(b1.first(a1)) <<endl;//1
    return 0;
}
```

```
class A<int,10>
3
1
请按任意键继续.
```

不加 `typename` 编译器默认认为是类的非类型成员。

成员函数模板

```
class Base{};
class D1:public Base{};
template<typename T>
class SmartPtr {
public:
    explicit SmartPtr(T* p=nullptr)
        :ptr(p), count(new int(p ? 1 : 0)) { }
    ~SmartPtr() {
    } if(--(*count)<=0){delete ptr;delete count;}
    void swap(SmartPtr& rhs) {
        std::swap(ptr, rhs.ptr);
    } std::swap(count, rhs.count);

    SmartPtr(const SmartPtr& rhs) //拷贝构造
        :ptr(rhs.ptr), count(&(++*rhs.count)) {}
    template<typename U>
    SmartPtr(const SmartPtr<U>& rhs)
        :ptr(rhs.ptr), count(&(++*rhs.count)) {}

    SmartPtr& operator=(const SmartPtr& other);
    template<typename U> //赋值
    SmartPtr& operator=(const SmartPtr<U>& other);

private:
    T* ptr; int* count;
    //所有的SmartPtr<...>都是SmartPtr<...>的友元
    template<typename X> friend class SmartPtr;
    //friend class Test<T>;
}; //Test<int> 是 Smart<int>的友元,匹配参数
```

类模板的**成员函数模板**：
类里面的函数也是一个函数模板，成员函数模板**不能是虚函数**。

```
template<typename T>
template<typename U> //次序不能错
SmartPtr<T>& SmartPtr<T>::operator=
    (const SmartPtr<U>& other){
    SmartPtr(other).swap(*this);
    return *this;
}
```

```
template<typename T>
SmartPtr<T>& SmartPtr<T>::operator=
    (const SmartPtr& other) {
    SmartPtr(other).swap(*this);
    return *this;
}
```

```
int main() {
    //基类指针指向派生类对象
    Base *pb = new D1;
    delete pb;
    //要实现类似上面的效果,但是类模板并没有继承关系
    //只能通过构造、赋值函数来实现
    SmartPtr<Base> s_pb1 = SmartPtr<D1>(new D1);
    SmartPtr<Base> s_pb2;
    s_pb2 = s_pb1;
    s_pb2 = SmartPtr<D1>(new D1);
    return 0;
}
```

控制实例化

模板使用时才会实例化意味着**相同的实例**可能出现在多个对象文件中。可通过**显式实例化**来避免。显式实例化会**实例化所有成员**（正常的实例化只会实例化用到的成员）。

```
#ifndef TEMPLATE_H
#define TEMPLATE_H
#include <vector>
template <typename T>
int compare(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
template <typename T>
class my_vector {
public:
    my_vector() { data.resize(10); }
    my_vector(const T& t, int n=10) {
        for (int i = 0; i < n; i++)
            data.push_back(t);
    }
private:
    std::vector<T> data;
};
#endif // !TEMPLATE_H
```

文件名: 11_template.h

```
#include "11_template.h"
template int compare(const int&, const int&); //实例化定义
template class my_vector<int>; //实例化定义
//其他代码
```

文件名: 11_abc.cpp

```
#include <iostream>
#include "11_template.h"
using namespace std;
//实例化声明
extern template int compare(const int&, const int&);
extern template class my_vector<int>;
struct A{ A(int i){} };
//template class my_vector<A>;
//错误,A没有默认构造,无法实例化my_vector<A>的默认构造
int main() {
    int a = 10, b = 20;
    cout << compare(a, b) << endl; //实例化在其他地方
    my_vector<int> v1; //实例化在其他地方
    //my_vector<double> 和 带参构造函数 在本文件实例化
    my_vector<double> v2(1);
    //my_vector<A> 和 带参构造函数 在本文件实例化
    my_vector<A> v3(A(1),3);
    return 0;
}
```

文件名: 11_main.cpp

特例化

```
template<typename T>
void f1(const T&) { cout << "const T&\n"; }
//f1函数模板的一个特例化, T 为 int,实现可以不同
template<> void f1(const int&) {
    cout << "const int &\n";
}
//特例化的本质是实例化一个模板,而不是重载。
```

```
//类模板A
template<typename T1,typename T2>
class A {
public:
    A(const T1& _t1,const T2& _t2)
        :t1(_t1),t2(_t2){}
    inline void show()const;
private:
    T1 t1;
    T2 t2;
};
template<typename T1, typename T2>
inline void A<T1,T2>::show()const {
    cout << t1 << " " << t2 << endl;
}
```

```
//1.类模板A 的一个特例化,T1=int,T2=double
template<>
class A<int, double> {
public:
    A(const int& _t1,const double& _t2)
        :t1(_t1),t2(_t2){}
    inline void show()const;
private:
    int t1;
    double t2;
};
//注意类外成员函数的写法(template)
inline void A<int, double>::show()const{
    cout << "特:"<<t1<<" "<< t2 << endl;
}
//2.类模板A 只特例化一个成员函数
template<>
inline void A<char, char>::show()const {
    cout <<"特show:"<<t1<<" "<<t2<<endl;
}
```

```
const T&
const int &
2.2 c
特:1 1.2
特show:a b
请按任意键继续.
```

```
//3.部分特例化 类模板B
template <typename T1,typename T2> class B {};
//类模板B的三个特例化
template<typename T> class B<T, T> {};
template<typename T> class B<T, int> {};
template<typename T1,typename T2> class B<T1*, T2*>{};
int main() {
    f1(1.2); //根据函数模板实例化 f1(const double&)
    f1(1); //使用特例化的 f1(const int&)
    A<double, char> a1(2.2, 'c'); //实例化
    a1.show(); // 2.2 c
    A<int, double> a2(1,1.2); //使用特例
    a2.show(); //特: 1 1.2
    A<char, char> a3('a', 'b');//实例化,show函数用特例
    a3.show(); //特show: a b
    B<double, char> b1; //使用 B<T1,T2>
    B<double, double> b2; //使用 B<T,T>
    B<double, int> b3; //使用 B<T,int>
    B<int*, double*> b4; //使用 B<T1*,T2*>
    //B<int, int> b5; 错 B<T,T>和B<T,int>同等程度匹配
    //B<int*, int*> b6;错 B<T,T>和B<T1*,T2*>同等程度匹配
    //int*,int* 可提供另外一个特例化
    //template<typename T> class B<T*,T*>{};
    return 0;
}
```

函数模板的重载

函数模板的重载：函数模版可以被另一个模版或者普通非模版函数重载。

匹配规则：非模板函数优先于模板函数实例，如果没有非模板函数，则选择一个更加特例化的实例。

```
template<typename T>
int compare(const T& a, const T& b) {
    cout << "const T&\n";
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
template<typename T>
int compare(T* a, T* b) { //值传递
    cout << "T* \n";
    if (*a < *b) return -1;
    if (*b < *a) return 1;
    return 0;
}
int compare(const char* const &a, const char* const &b) {
    cout << "const char* const & \n";
    return strcmp(a, b);
}
```

```
int compare(const char* &a, const char* &b) {
    cout << "const char* & \n";
    return strcmp(a, b);
}
int main() {
    int a = 20, b = 10;
    cout << compare(a, b) << endl; //const T&
    cout << compare(&a, &b) << endl; // T* 若没有T*则调用第1个
    //如果没有2个非模板函数,则下面调用 T*
    cout << compare("abc", "abcd") << endl; //const char* const &
    const char *p1 = "abc", *p2 = "abcd";
    cout << compare(p1, p2) << endl; //const char* &
    return 0;
}
```

```
const T&
1
T*
1
const char* const &
-1
const char* &
-1
请按任意键继续. . .
```

继承关系的用法

```
template<typename T>
class Base {
public:
    Base(const T& d):data(d){}
    void set_data(const T& d) { data = d; }
    virtual void show()const { cout << "Base show: data=" << data << " "; }
    virtual ~Base(){}
private:
    T data;
};
```

```
template<> class Base<double> { //double特例, 取消了 set_data函数
public:
    Base(const double& d) :data(d) {}
    //void set_data(const double& d) { data = d; }
    virtual void show()const { cout << "Base show: data=" << data << " ";}
    virtual ~Base() {}
private:
    double data;
};
```

类模板中，派生类调用基类成员，要加上this->

```
template<typename T>
class D1 :public Base<T> {
public:
    D1(const T& d1,const T& d2):Base<T>(d1),data2(d2){}
    void set_d1_data(const T& d1, const T& d2) {
        this->set_data(d1); //加上this
    }
    data2 = d2;
    virtual void show()const override{
        Base<T>::show(); //显式调用基类成员函数
    }
    cout << "D1 show: data2=" << data2 << endl;
private:
    T data2;
};

int main() {
    D1<int> d1(1,2); //d2.set_d1_data(11, 22);
    d1.show(); //不行,double特例没有实现set_data()
    d1.set_d1_data(10, 20); //只要不调用 set_d1_data,实例化正常
    d1.show();
    Base<int> *pb = &d1;
    pb->show(); //虚函数调用正常
    D1<double> d2(111, 222);
    d2.show();
    return 0;
}
```

```
Base show: data=1 D1 show: data2=2
Base show: data=10 D1 show: data2=20
Base show: data=111 D1 show: data2=222
请按任意键继续...
```

静多态和动多态

```
class Shape {  
public:  
    virtual void show()const = 0;  
    virtual ~Shape() = default;  
};  
  
//继承自抽象基类  
class Rectangle :public Shape {  
public:  
    virtual void show()const override {  
        cout << "Rect show\n";  
    }  
};  
  
class Circle :public Shape {  
public:  
    virtual void show()const override {  
        cout << "Circle show\n";  
    }  
};  
  
//普通函数  
void my_show(const Shape& obj) {  
    //基类引用,虚函数调用实现多态  
    obj.show();  
}
```

动多态

静多态

```
class Rectangle_1 { //没有继承关系  
public:  
    void show()const {  
        cout << "Rect_1 show\n";  
    }  
};  
  
class Circle_1 {  
public:  
    void show()const {  
        cout << "Circle_1 show\n";  
    }  
};  
  
template<typename T> //函数模板  
void my_show_1(const T& obj) {  
    obj.show();  
}
```

```
int main() {  
    //动多态:  
    Rectangle r1;  
    Circle c1;  
    my_show(r1);  
    my_show(c1);  
    //静多态  
    Rectangle_1 r2;  
    Circle_1 c2;  
    //my_show_1<Rectangle_1>(const Rectangle_1 &  
    my_show_1(r2);  
    //my_show_1<Circle_1>(const Circle_1 &  
    my_show_1(c2);  
    return 0;  
}
```

```
Rect show  
Circle show  
Rect_1 show  
Circle_1 show  
请按任意键继续
```

动多态：运行期绑定

静多态：编译器绑定

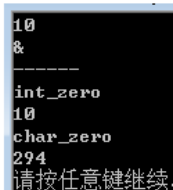
Traits

Traits (特性萃取)：需要根据**数据类型不同**进行不同的处理，但不会改变封装，可以使用Traits。

```
//计算总和：尝试(有什么问题?)
template<typename T>
T sum_test(T* beg, T* end) {
    T total = T(); // = 0 ?
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```

```
template<typename T> struct base_traits;
template<> struct base_traits<int> {
    typedef long long sum_type;
    static sum_type zero() { cout << "int_zero\n"; return 0; }
};
template<> struct base_traits<char> {
    typedef int sum_type;
    static sum_type zero() { cout << "char_zero\n"; return 0; }
};
```

```
template<typename T>
typename base_traits<T>::sum_type sum_test_new(T* beg, T* end) {
    typename base_traits<T>::sum_type total = base_traits<T>::zero();
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}
```



```
10
8
-----
int_zero
10
char_zero
294
请按任意键继续
```

```
int main() {
    int num[] = { 1,2,3,4 };
    cout << sum_test(&num[0], &num[sizeof(num)/sizeof(int)]) << endl;
    char name[] = "abc"; //97 98 99 = 294 - 256 = 38(&的ascii)
    cout << sum_test(&name[0], &name[sizeof(name)-1]) << endl; //8
    cout << "-----\n";
    cout << sum_test_new(&num[0], &num[sizeof(num) / sizeof(int)]) << endl;
    cout << sum_test_new(&name[0], &name[sizeof(name) - 1]) << endl; //294
    return 0;
}
```

计算总和的函数模板：

需要根据**传入的参数类型**来设置**总和的类型**，
如：传入类型是int, 则总和类型为long long
传入类型是char, 则总和类型为int

可以通过增加模板参数来解决(例如增加一个参
template <typename T, typename SUM_T>), 但
用traits处理感觉更好。

增加一个**基础模板**，然后**特例化**这个基础模
板，然后把这个基础模板应用到我们的函数模板
中。

这样做的好处就是函数模板还是一个，只是
增加一些基础模板的特例化实例。

参数推断、引用折叠

//模板：允许将一个 右值引用 绑定到 一个左值

//一： 将一个左值传递给函数的右值引用参数时,并且此右值引用指向模板类型参数 (如T&&) 时, 推断模板类型参数是实参的左值引用类型。

//二： 引用折叠,类型T& &、 T& &&、 T&& & 折叠成类型 T&, 类型 T&& && 折叠成类型 T&&

//实参必须是左值

```
template <typename T> void f1(T&) {}
```

//实参可以是右值

```
template <typename T> void f2(const T&) {}
```

//模板：实参可以是左值

```
template <typename T> void f3(T&&) {}
```

```
template <typename T> void f4(T&& val) {
```

```
    T t = val;
```

```
    //1.假如传入一个int左值: T 就是 int&, 那么上面相当于 int& t = val;
```

```
    //2.假如传入一个int右值: T 就是 int, 那么上面相当于 int t = val;
```

```
    t = 100; //情况1: val将会置为100, 情况2: val不会改变
```

```
    cout << "f4(): val = " << val << endl;
```

```
}
```

```
int main() {
    int i = 10;
    const int ci = 20;
    f1(i); // T: int
    f1(ci); // T: const int
    //f1(2); // 错,不能是右值
    f2(i); // T: int
    f2(ci); // T: int
    f2(2); // T: int
    f3(i); // T: int &
    f3(ci); // T: const int &
    f3(2); // T: int
    f4(i);
    cout << "左值i= " << i << endl;
    i = 10;
    f4(std::move(i));
    cout << "右值i=" << i << endl;
    return 0;
}
```

```
f4(): val = 100
左值i= 100
f4(): val = 10
右值i=10
请按任意键继续.
```

remove_reference、std::move

remove_reference实现:

```
template<typename T>
struct remove_reference {
    typedef T type;
};

template<typename T>
struct remove_reference<T&> {
    typedef T type;
};

template<class T>
struct remove_reference<T&&> {
    typedef T type;
};

remove_reference<int>::type i1;
remove_reference<int&>::type i2;
remove_reference<int&&>::type i3;
//上面三种类型都是 int (相当于去掉了引用)
```

move实现:

```
template<typename T>
inline typename remove_reference<T>::type&& move(T&& t) {
    return (static_cast<typename remove_reference<_Ty>::type&&>(t));
}

// move(string("abc")); //右值
//1.推断出 T 的类型是 string
//2.remove_reference<string> 实例化(第1个)
//3.remove_reference<string>的type成员是 string
//4.move的返回类型是 string&&
//5.move函数的参数 t 的类型是 string&&
//此调用实例化 move<string>: 就是 string&& move(string&& t);

// string s1="abc"; move(s1); //左值
//1.推断出 T 的类型是 string&
//2.remove_reference<string&> 实例化(第2个)
//3.remove_reference<string&>的type成员是 string
//4.move的返回类型是 string&&
//5.move函数的参数 t 的类型是 string& &&,折叠为 string&
//此调用实例化 move<string&>: 就是 string&& move(string& t);
```

转发: std::forward

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

template<class TT>
TT&& my_forward(typename remove_reference<TT>::type& tt) {
    cout << "my_forward: 1\n";
}
return (static_cast<TT&&>(tt));

template<class TT>
TT&& my_forward(typename remove_reference<TT>::type&& tt){
    cout << "my_forward: 2\n";
}
return (static_cast<TT&&>(tt));

int main() {
    int i = 10;
    outer_1(i); //T: int&, i的类型int&
    outer_1(42);//T: int, 虽然传递右值,但是用右值引用接收以后转为左值
    cout << "-----\n";
    outer_2(i); //T: int&, TT: int&, typename remove_reference<TT>::type --> int, tt: int&, TT&&: int& &&-->int&(左值)
    outer_2(42);//T: int, TT: int, typename remove_reference<TT>::type --> int, tt: int&, TT&&: int&&-->int&&(右值)
    cout << "-----\n";
    outer_3(i);
    outer_3(42);
    return 0;
}
```

```
void inner(const int& i){ cout << "left: " << i << std::endl;}
void inner(int&& i){ cout << "right: " << i << std::endl; }

template <typename T> void outer_1(T&& i) { inner(i); }
template <typename T> void outer_2(T&& i) { inner(my_forward<T>(i)); }
template <typename T> void outer_3(T&& i) { inner(std::forward<T>(i));}
```

std::forward<T>(u) 有两个参数: T 与 u。当T为左值引用类型时, u将被转换为T类型的左值, 否则u将被转换为T类型右值。如此定义std::forward是为了在使用**右值引用参数的函数模板中解决参数的完美转发问题**。

```
left: 10
left: 42
-----
my_forward: 1
left: 10
my_forward: 1
right: 42
-----
left: 10
right: 42
请按任意键继续
```

模板元编程 (TMP)

模板元编程 (Template metaprogramming) (**TMP**): 是编写template-based C++程序并**执行于编译期**的过程。一种元编程技术, 编译器使用模板产生暂时性的源码, 然后再和剩下的源码混合并编译。这些模板的输出包括编译时期常量、数据结构以及完整的函数。如此利用模板可以被想成编译期的运行。

TMP的特点: 由于TMP执行于C++编译期, 因此可将工作从运行期转移到编译期, 因此: 某些错误原本通常在运行期才能侦测到, 现在可在编译期找到。另外使用TMP的C++程序可能在每一方面都更高效: 较小的可执行文件、较短的运行期、较小的内存需求。

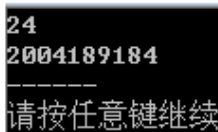
将工作从运行期转移到编译期的另一个结果是: **编译时间变长了**。

对于C++来说, 模板元编程的语法及语言特性比起传统的C++编程, 较难以令人理解。因此对于那些在模板元编程经验不丰富的程序员来说, 程序可能会变的难以维护 (**可读性变差**)。

```
template<unsigned N>
struct Factorial { //阶乘
    enum { value = N * Factorial<N - 1>::value };
};

template<>
struct Factorial<0> {
    enum { value = 1 };
};
```

```
int main() {
    cout << Factorial<4>::value << endl;
    cout << Factorial<16>::value << endl;
    cout << "-----\n";
    return 0;
}
```



```
24
2004189184
-----
请按任意键继续
```

STL概述

标准模板库 **STL** (Standard Template Library)：一个具有工业强度的，高效的C++程序库，它被容纳于C++标准程序库 (C++ Standard Library) 中。

STL：包含了常用的基本数据结构和基本算法，并提供了可扩展性、可复用性。

STL：数据结构和算法的分离。

STL：为了具有足够通用性，主要是基于模板（泛型程序设计思想）而不是面向对象。

C++程序员的神兵利器，通过学习，掌握特性，根据实际的需求选择最合适的。要避免自己再去实现。

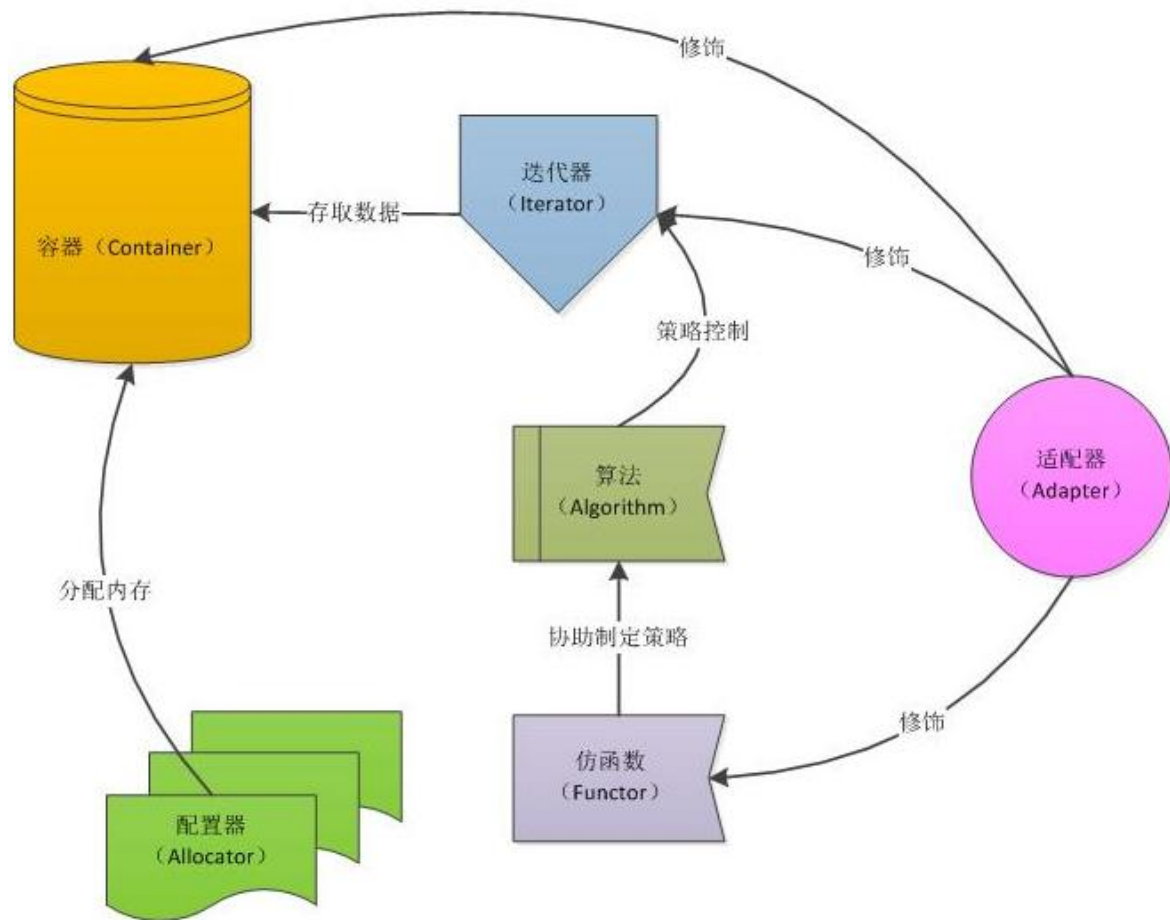
主要由六大组件构成：

容器 (container)、算法 (algorithm)、迭代器 (iterator)、仿函数 (functor)、适配器 (adapter)、配置器 (allocator)。

STL组件

STL主要由六大组件构成：**容器（container）**、**迭代器（iterator）**、**算法（algorithm）**、**仿函数（functor）**、**适配器（adapter）**、**配置器（allocator）**。

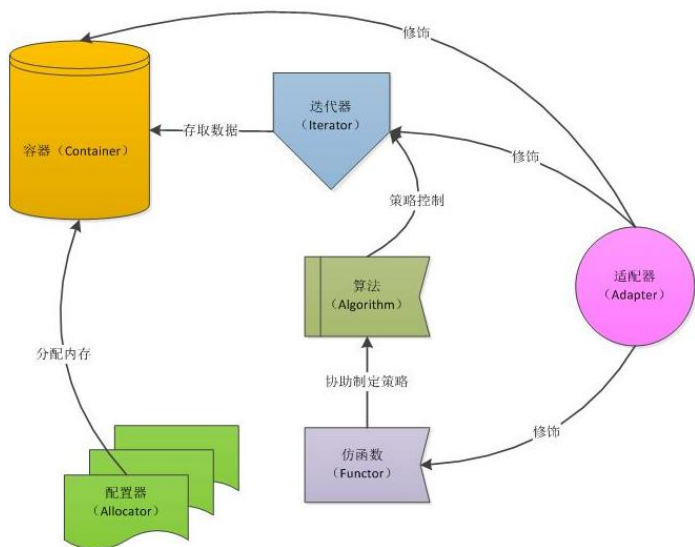
1. **容器**：各种数据结构，如vector,list,map等，用来存放数据，主要以类模板实现。
2. **迭代器**：“**泛型指针**”，提供访问容器中对象的方法，是容器与算法之间的胶合剂。
3. **算法**：是用来操作容器中的数据的功能模板。函数本身与他们操作的**数据的结构和类型无关**，因此他们可以在从简单数组到复杂容器的任何数据结构上使用。
4. **仿函数**：行为类似函数（函数对象），可作为算法的某种策略。
5. **适配器**：一种用来修饰容器、迭代器、仿函数接口的东西。如stack是一种容器适配器。
6. **配置器**：负责空间配置和管理。是一个实现了动态空间配置、管理、释放的类模板。



STL组件关系

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

class GreaterN {
public:
    GreaterN(int i=0):val(i){}
    bool operator()(const int& v)const {
        if (v > val) return true;
        return false;
    }
private:
    int val;
};
```



```
int main() {
    //vector的定义类似如下：_Alloc是空间配置器,默认值是allocator<_Ty>
    //template<class _Ty,class _Alloc = allocator<_Ty> >
    //class vector...
    std::vector<int> v1;
    std::vector<int, std::allocator<int>> v2;

    //观察：容器、迭代器、算法、仿函数之间的关系
    //数组也可看作是一种容器
    int arr[5] = { 1,2,3,4,5 };
    int* it1_beg = std::begin(arr); //相当于 &arr[0]
    int* it1_end = std::end(arr);   //相当于 &arr[5]
    //int* 指针,也可看作是一种迭代器
    int total = count_if(it1_beg, it1_end, GreaterN(3));
    //GreaterN是函数对象(仿函数),协助算法制定策略
    //count_if是算法,统计在两个迭代器之间所有元素满足条件的个数
    cout << total << endl;

    //vector是一种容器
    std::vector<int> vec = { 1,2,3,4,5 };
    //std::vector<int>::iterator 是一种迭代器
    std::vector<int>::iterator it2_beg = vec.begin();
    std::vector<int>::iterator it2_end = vec.end();
    int total2 = count_if(it2_beg, it2_end, GreaterN(2));
    //上面三句可以简写为：
    total2 = count_if(vec.begin(), vec.end(), GreaterN(2));
    cout << total2 << endl;
    return 0;
}
```

STL: 容器

一个容器就是一些特定类型对象的集合。

顺序容器: 提供控制元素存储和访问顺序的能力。这种顺序不依赖于元素的值, 而是与元素加入容器时的位置相对应。

1. **vector**: 可变大小数组, 随机存取元素, 尾部添加或删除元素快, 在中部或头部插入元素慢;
2. **deque**: double-ended queue, 随机存取元素, 头部和尾部添加删除元素快, 在中部或头部插入元素慢;
3. **array**: 大小固定的数组, 随机存取元素, 不能添加和删除元素。(类似内置数组)
4. **list**: 双向链表, 不提供随机存取(按顺序走到需存取的元素, $O(n)$), 在任何位置上插入删除快;
5. **forward_list**: 单向链表, 类似list, 但只能单向遍历;
6. **string**: 类似vector, 专门用于保存字符。随机访问快, 尾部插入和删除快, 头部和中间插入删除慢。

关联容器: 支持高效的關鍵字查找和访问。

set, map, multiset, multimap

unordered_set, unordered_map, unordered_multiset, unordered_multimap(无序容器)

前4种存入的元素必须有比较运算(如 $<$, 要保证**序关系**), 后4种根据**hash函数**和关键字类型的 $==$ 运算符。

顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。

关联容器中的元素是按**关键字**来保存和访问的。

STL: 迭代器

每种容器类型都定义了自己的迭代器类型。迭代器可通过解引用运算符访问容器中的元素，使用++、--等操作。
迭代器：类似智能指针。

```
vector<int> v1 = { 1,2,3 };  
//迭代器  
vector<int>::iterator it1 = v1.begin(); //end();  
//const迭代器  
vector<int>::const_iterator it1_c = v1.cbegin(); //cend();  
//反向迭代器  
vector<int>::reverse_iterator it1_r = v1.rbegin(); //rend();  
//反向const迭代器  
vector<int>::const_reverse_iterator it1_cr = v1.crbegin(); //crend();  
//成员函数begin/rbegin/end/rend是重载过的  
const vector<int> v2 = { 1,2,3 };  
auto it11 = v1.begin(); //it11 是 vector<int>::iterator  
auto it22 = v2.begin(); //it22 是 vector<int>::const_iterator  
*it11 = 11; //解引用访问  
//*it22 = 10; //错,const迭代器 只读  
cout << *it22 << endl; //能读不能写  
//begin 和 end  
//容器有成员函数begin/end等...  
//也可以用std::begin(xx),对内置数组也可使用 end对应  
it1 = std::begin(v1);  
it1_c = std::cbegin(v1);  
it1_r = std::rbegin(v1);  
it1_cr = std::crbegin(v1);
```

```
//迭代器范围 [begin, end) 左闭右开  
while (it1_c != cend(v1)) {  
    cout << *it1_c << " ";  
    ++it1_c;  
} // 11 2 3  
cout << endl;  
//反向迭代器,++是从后面向前  
while (it1_cr != crend(v1)) {  
    cout << *it1_cr << " ";  
    ++it1_cr;  
} // 3 2 11  
cout << endl;
```



STL: vector1

```
#include <iostream>
#include <utility>
#include <string>
#include <vector>
using namespace std;
struct A {
    A(const string& ss1 = "", int i1 = 0) :s1(ss1), d1(i1) {}
    string s1;
    int d1;
};
int main() {
    //vector 动态数组 #include <vector>
    //尾部删除、添加,速度快。遍历非常快。内存紧凑,使用的空间较少。
    //template<class _Ty, class _Alloc = allocator<_Ty>> class vector...
    //1、初始化:
    vector<int> v1; //默认构造
    vector<double> v2{ 1.0,2.0,3.0 }; //初始化列表
    vector<string> v3 = { "abc","C++","C" }; //初始化列表
    vector<double> v4(v2);
    vector<string> v5 = v3; //拷贝构造
    vector<string> v6 = std::move(v3); //移动构造
    vector<double> v7(v2.cbegin(), v2.cend()); //迭代器范围构造
    vector<int> v8(10); //10个元素,每个元素为0(值初始化)
    vector<int> v9(10, 1); //10个元素,每个元素为1
    //2.赋值(会冲掉原先的内容)
    v8 = v1;
    v8 = { 1,2,3 }; //用初始化列表赋值
    v8.assign(v9.cbegin(), v9.cend()); //迭代器范围赋值
    v8.assign({ 1,2,3 }); //初始值列表赋值
    initializer_list<int> il = { 1,2,3 };
    v8.assign(il);
    v8.assign(10, 2); //用10个2赋值
```

```
//3.swap 交换,2个vector交换 O(1)时间复杂度
vector<int> v10(20, 1); //20个元素都是1
vector<int> v11(10, 2); //10个元素都是2
v10.swap(v11);
std::swap(v10, v11); //会执行上面的函数
```

```
//4.容量相关
v10.size(); //元素个数
v10.empty(); //元素个数==0则返回true
v10.max_size(); //能存储的最大元素个数
v10.capacity(); //当前能存储的最大元素个数
v10.reserve(100); //只能扩容,不能缩减,不改变元素个数
cout << v10.size() << endl;
v10.resize(25); //改变元素个数,删掉多余元素,或以值初始化添加元素
v10.resize(30, 2); //改变元素个数,删掉多余元素,或以2添加元素
v10.shrink_to_fit(); //降低容量,但是不是强制性的,只是通知stl可以降低容量
```

```
//5.元素访问
v11[1] = 2; //下标访问,要保证下标不越界,否则未定义结果
v11.at(1) = 20; //下标越界抛out_of_range异常
if (!v11.empty()) v11.front() = 1; //第1个元素的引用
if (!v11.empty()) v11.back() = 2; //最后一个元素的引用
```

```
//6.迭代器相关
vector<int>::iterator it1 = v11.begin();
vector<int>::const_iterator it2 = v11.cbegin();
vector<int>::reverse_iterator it3 = v11.rbegin();
vector<int>::const_reverse_iterator it4 = v11.crbegin();
//end()也是一样
```

STL: vector2

```
//7.插入元素
vector<A> v12;
v12.push_back(A("123", 1)); //尾部添加元素
A a1("abc", 2);
v12.push_back(a1); //元素存入容器是拷贝 const A&, A&
v12.push_back(std::move(a1)); //push_back进右值,移动
v12.emplace_back("abcd", 3); //尾部添加元素,直接调用A的构造
//在迭代器位置前插入以参数"aac",9构造的 A对象,返回指向新插入元素的迭代器
v12.emplace(v12.begin(), "aac", 9);
A a2("acc", 3);
vector<A>::iterator it_insert = v12.insert(v12.begin(), a2);
//在迭代器位置前插入a2,返回指向新插入元素的迭代器
cout << (*it_insert).s1 << "--" << (*it_insert).d1 << endl;
//在迭代器位置前插入3个a2,返回指向第1个新元素的迭代器
v12.insert(v12.begin(), 3, a2);
vector<A> v13;
//在迭代器位置前插入 v12.begin()到v12.end()之间的元素,返回指向第1个新元素的迭代器
//若 v12.begin() == v12.end() 表示不插入一个元素,返回还是原迭代器位置
v13.insert(v13.begin(), v12.begin(), v12.end());
v13.insert(v13.begin(), { { "aa",1 }, { "bb",2 } }); //用初始值列表插入

//8.删除元素
v13.pop_back(); //删除最后1个元素
//删除迭代器位置的元素,返回指向被删元素后面元素的迭代器,若迭代器是end(),则行为未定义
v13.erase(v13.begin());
//删除迭代器范围,返回最后1个删除元素之后元素的迭代器
v13.erase(v13.begin() + 1, v13.end());
v13.clear(); //清空所有元素,但是容量不会变(capacity()不变)
```

```
//9.关系运算
//所有容器都支持 == != 运算符,除无序容器外都支持 < > <= >=
vector<int> v14 = { 1,2,3 };
vector<int> v15 = { 1,2,4 };
v14 < v15; //true ,每个元素的比较
v14 == v15; //false

//10.与C的接口
vector<char> v16(10, '\0');
strcpy(&v16[0], "abcd");
printf("%s\n", &v16[0]); //abcd
strcat(v16.data(), "1234");
printf("%s\n", v16.data()); //abcd1234
//printf("%s\n", v16.begin()); //错
printf("%s\n", &(*v16.begin())); //OK
//行为像指针,但是所指对象的地址是 &*迭代器
cout << "----\n";
return 0;
```

STL: array

```
//array(没有空间配置器) #include <array> (原生数组的一个封装)
//明确知道最多要多少个元素,可以考虑array
//直接 array<int,10> a; 元素是在栈上的。注意大小(栈空间有限)
//template<class _Ty, size_t _Size> class array{ _Ty _Elems[_Size]; ...}
//1. 初始化
array<int, 10> a1; //默认构造,值没有初始化
array<int, 3> a2{}; //值初始化,都是0
array<string, 4> a3 = { "abc" }; //第1个按列表初始化,后面的默认初始化""
array<int, 3> a4(a2);
array<int, 3> a5 = a2; //拷贝(区别于内置数组,内置数组不能这样操作)
array<string, 4> a6 = std::move(a3); //移动,每个元素都执行移动(或拷贝)
//array<int, 3> a7(a2.begin(), a2.end()); 不支持迭代器范围初始化
//array<int, 6> a8(6);
//array<int, 6> a8(6,0); 指定几个元素的初始化不行

//2. 赋值
a5 = a4; //内置数组不支持该操作
a5 = { 2,3 }; //用初始化列表赋值
//不支持 assign操作
a5.fill(9); //所有元素都赋值为9

//3. swap交换,2个array交换,0(n)时间复杂度(每个元素交换)
array<int, 10> a10 = { 1,2,3 };
array<int, 10> a11 = { 5,6,7,8 };
a10.swap(a11);
std::swap(a10, a11); //同上

//4. 容量相关
a10.size(); //元素个数,同非类型模板参数
a10.empty(); //除非 array<int,0> a10;否则不可能为true
a10.max_size(); //同size()
//下面的都不支持
//a10.capacity(); //a10.reserve(100); //a10.resize(25);
```

```
//5. 元素访问,与vector一样
a10[2] = 10; //下标访问,要保证下标不越界,否则未定义结果
a10.at(1) = 20; //下标越界抛out_of_range异常
a10.front(); //第1个元素的引用
a10.back(); //最后一个元素的引用

//6. 迭代器相关,尺寸要匹配,与vector一样
array<int, 10>::iterator it1 = a10.begin();
array<int, 10>::const_iterator it2 = a10.cbegin();
array<int, 10>::reverse_iterator it3 = a10.rbegin();
array<int, 10>::const_reverse_iterator it4 = a10.crbegin();

//7. 插入元素,不存在
//8. 删除元素,不存在 clear()也没有

//9. == != < > <= >= 都有,但是类型要一样
array<int, 2> a12 = { 1,2 };
array<int, 3> a13 = { 1,2,3 };
//cout << (a12 == a13) << endl; 不行,类型不匹配

//10. C接口 同vector
array<char, 10> a16 = {'a','b','c','\0'};
printf("%s\n", a16.data()); //abc
//printf("%s\n", a16.begin()); //错
printf("%s\n", &(*a16.begin())); //OK
return 0;
```

STL: deque

```
struct A {
    A(const string& ss1 = "", int i1 = 0) :s1(ss1), d1(i1) {}
    string s1;
    int d1;
};
int main() {
    //deque 双端队列 #include <deque>
    //随机访问元素,头部和尾部添加删除元素效率很高,中间删除和添加效率低。
    //元素的访问和迭代比vector要慢,迭代器不能是普通的指针。
    //template<class _Ty, class _Alloc = allocator<_Ty>> class deque ...

    //1.初始化,与vector一样
    deque<int> d1; //默认构造
    deque<int> d2{ 1,2,3 };
    deque<double> d3 = { 1.0,2.0,3.0 }; //初始化列表
    deque<int> d4(d2);
    deque<int> d5 = d2; //拷贝构造
    deque<double> d6(d2.cbegin(), d2.cend()); //迭代器范围,元素可转换即可
    deque<string> d7(10); //10个元素,每个都是空串
    deque<string> d8(8, "abc"); //8个元素,每个都是"abc"

    //2.赋值,与vector一样
    d8 = d7;
    d8 = { "aaa","bbb","ccc" }; //用初始化列表赋值
    d8.assign(d7.cbegin(), d7.cend()); //迭代器范围赋值
    d8.assign({ "aa","bb","cc" }); //初始值列表赋值
    initializer_list<string> il = { "a","b","c" };
    d8.assign(il);
    d8.assign(10, "abc"); //用10个"abc"赋值
```

```
//3.swap 与vector一样,0(1)
deque<int> d10, d11{ 1,2,3 };
d10.swap(d11); std::swap(d10, d11);

//4.容量相关 没有capacity和reserve,其他与vector一样
d10.size(); //元素个数
d10.empty(); //元素个数==0则返回true
d10.max_size(); //能存储的最大元素个数
//d10.capacity(); deque没有
//d10.reserve(100); deque没有
cout << d10.size() << endl;
d10.resize(25); //改变元素个数,删掉多余元素,或以值初始化添加元素
d10.resize(30, 2); //改变元素个数,删掉多余元素,或以2添加元素
d10.shrink_to_fit(); //降低容量,但是不是强制性的,只是通知stl可以降低容量

//5.元素访问 与vector一样 [] at front back
//6.迭代器相关 与vector array 一样

//7.插入元素 多了push_front,emplace_front 其他与vector一样
//push_back,emplace_back,emplace,insert
deque<A> d12;
d12.push_front(A("123", 1)); //头部添加元素
if (!d12.empty()) d12.pop_front(); //头部删除元素
d12.emplace_front("abc", 2);

//8.删除元素 多了pop_front 其他和vector一样
d12.pop_front(); //删除第1个元素
d12.clear(); //清空所有元素,有可能释放空间(一般释放空间)
//9.关系运算 和vector一样
return 0;
```

STL: list

```
//list 双向链表 #include <list>
//不支持随机访问元素,访问头部和尾部元素很快。任何位置插入删除元素都很快, O(1)
//插入和删除不会造成迭代器失效。空间成本较高(有2个指针)
//template<class _Ty, class _Alloc = allocator<_Ty>> class list ...
//1.初始化 与vector一样
list<int> l1; //默认构造
list<int> l2{ 1,2,3 };
list<double> l3 = { 1.0,2.0,3.0 }; //初始化列表
list<int> l4(l2);
list<int> l5 = l2; //拷贝构造
list<double> l6(l3.cbegin(), l3.cend()); //迭代器范围
list<string> l7(10);
list<string> l8(10, "abc"); //10个元素,每个都是"abc"

//2.赋值 与vector一样
l1 = l2; //类型要一致
l2 = { 2,3,4 }; //初始值列表赋值
l2.assign({ 1,2,3 });
l2.assign(10, 0);
l2.assign(l1.cbegin(), l1.cend());

//3.swap 与vector一样
//4.容量相关 不需要预先准备空间
l2.size(); //元素个数
l2.empty(); //元素个数==0则返回true
l2.max_size(); //能存储的最大元素个数
//l2.capacity(); list没有capacity(不存在扩容这样的操作)
//l2.reserve(100); 没有
l2.resize(25); //改变元素个数,删掉多余元素,或以值初始化添加元素
l2.resize(30, 2); //改变元素个数,删掉多余元素,或以2添加元素
//l2.shrink_to_fit(); 不需要预先准备空间,有1个元素开辟1个空间
```

```
//5.元素访问 不能随机访问,头尾快,中间慢
//l2[1] = 2; list 不行
//l2.at(1) = 20; list 不行
if (!l2.empty()) l2.front() = 1; //第1个元素的引用
if (!l2.empty()) l2.back() = 2; //最后一个元素的引用
//需要访问第5个元素:(list访问中间的元素要遍历)
list<int> l10 = { 1,2,3,4,5,6,7 };
auto it = l10.begin();
for (int i = 0; i < 4; ++i) ++it;
cout << *it << endl;
it = l10.begin();
std::advance(it, 4); //it迭代器像前移动4步
cout << *it << endl;
it = l10.begin();
auto it0=std::next(it,4);//从it向前移4步,但it不变,返回值是移后的位置
cout << *it0 << " " << *it << endl;

//6.迭代器相关 与vector一样,因为是双向链表,所有反向迭代器ok
//7.插入元素 支持 push_front(类似deque) 其他与vector一样
//push_back,emplace_back,emplace,insert
l10.push_front(100);

//8.删除元素 支持pop_back pop_front
if (!l10.empty()) l10.pop_back(); //删除最后1个元素
if (!l10.empty()) l10.pop_front(); //删除第1个元素
//删除迭代器位置的元素,返回指向被删元素后面元素的迭代器,若迭代器是end(),则行为未定义
l10.erase(l10.begin());
//删除迭代器范围,返回最后1个删除元素之后元素的迭代器
l10.erase(l10.begin() /*+1不支持*/, l10.end());
l10.clear(); //清空所有元素

//9.关系运算,类似vector == != < > <= >=
```

STL: forward_list

```
//forward_list 单向链表 #include <forward_list>
//不支持随机访问元素,访问头部元素很快。任何位置插入删除元素都很快, O(1)
//插入和删除不会造成迭代器失效。空间成本较高(有1个指针),比list节省
//与手写的C风格的单链表相比,没有任何时间和空间上的额外开销(比如没有size)
//template<class _Ty, class _Alloc = allocator<_Ty>> class forward_list
//1.初始化,和vector list一样
forward_list<int> l1; //默认构造
forward_list<int> l2{ 1,2,3 };
forward_list<double> l3 = { 1.0,2.0,3.0 }; //列表初始化
forward_list<int> l4(l2);
forward_list<int> l5 = l2; //拷贝构造
//迭代器范围初始化,可以用其他容器,只要元素类型能转换(其他容器一样)
forward_list<double> l6(l2.cbegin(), l2.cend());
forward_list<string> l7(9);
forward_list<string> l8(9, "aaa"); //9个元素,"aaa"
//2.赋值 与vector list一样
l1 = l2; //类型要一致
l2 = { 2,3,4 }; //初始值列表赋值
l2.assign({ 1,2,3 });
l2.assign(10, 0);
l2.assign(l1.cbegin(), l1.cend());
//3.swap 与vector一样
//4.容量相关 不需要预先准备空间
//l2.size(); forward_list 没有元素个数size()函数
l2.empty(); //元素个数==0则返回true
l2.max_size(); //能存储的最大元素个数
//l2.capacity(); 同list 没有
//l2.reserve(100); 同list 没有
l2.resize(25); //改变元素个数,删掉多余元素,或以值初始化添加元素
l2.resize(30, 2); //改变元素个数,删掉多余元素,或以2添加元素
//l2.shrink_to_fit(); 不需要预先准备空间,有1个元素开辟1个空间
```

```
//5.元素访问 不能随机访问,头快,尾部和中间慢
//l2[1] = 2; list 不行 //l2.at(1) = 20; list 不行
if (!l2.empty()) l2.front() = 10; //第1个元素的引用
//l2.back(); forward_list没有
//6.迭代器相关 与vector一样,因为是双向链表,所有反向迭代器ok
forward_list<int>::iterator it1 = l2.begin();
forward_list<int>::const_iterator it2 = l2.cbegin();
//forward_list<int>::reverse_iterator 没有 rbegin() 没有
//forward_list<int>::const_reverse_iterator没有 crbegin()没有
- forward_list<int>::iterator it3 = l2.before_begin(); //多的
- forward_list<int>::const_iterator it4 = l2.cbefore_begin(); //多的
//7.插入元素 与list有区别
//l2.push_back(100); 不支持
l2.push_front(100); //头部插入,ok
l2.emplace_front(101); //头部插入
//emplace没有了,改用emplace_after
- l2.emplace_after(l2.before_begin(), 102); //在before_begin()后插入102
//insert没有了,改用insert_after,在迭代器位置后面插入
//返回指向最后1个插入元素的迭代器(其他迭代器:返回第1个新元素的迭代器)
- l2.insert_after(std::next(l2.before_begin(), 2), 103);
l2.insert_after(l2.before_begin(), l1.cbegin(), l1.cend());
//8.删除元素 与list有区别
if (!l2.empty()) l2.pop_front(); //删除第1个元素
//l2.pop_back(); 没有
//删除迭代器位置后面的元素,返回void,(其他容器:指向被删元素后面元素的迭代器)
- l2.erase_after(l2.before_begin()); //删第1个元素
//删除迭代器范围,返回void,(其他容器:返回最后1个删除元素之后元素的迭代器)
for (const auto& item : l2) cout << item << " "; cout << endl;
l2.erase_after(l2.cbegin(), l2.end()); //第1个元素没有删掉
for (const auto& item : l2) cout << item << " "; cout << endl;
l2.erase_after(l2.cbefore_begin(), l2.end()); //全部删掉了
for (const auto& item : l2) cout << item << " "; cout << endl;
l2.clear(); //清空所有元素
```

STL: 容器适配器 stack

```
#include <iostream>
#include <stack>
#include <vector>
#include <string>
using namespace std;
struct A {
    A(const string& s, int d) :s1(s), d1(d) {}
    string s1; int d1;
};
int main() {
    //stack 栈(先进后出), 默认是 deque实现 【容器适配器】 #include <stack>
    //template<class _Ty,class _Container=deque<_Ty>>class stack;
    stack<int> s1; //默认构造(用deque实现)
    stack<int, vector<int>> s2; //默认构造(用vector实现)
    //stack<int> s1 = { 1,2,3 }; 不行
    stack<int> s3(s1);
    stack<int> s4 = s1; //拷贝构造
    s4 = s1; //赋值,没有assign
    s4.swap(s1);
    std::swap(s4, s1); //可以高效swap
    cout << s4.size() << endl; //当前元素个数
    cout << s4.empty() << endl; //是否为空,空返回true
```

//不支持迭代器,也没有 begin,end 等操作

//插入元素:

```
stack<A> s5;
s5.push(A("abc", 1)); //同样有const T& 和T&&
A a1("aaa", 2);
s5.push(a1);
s5.emplace("ccc", 4); //等同push,不过参数是A的构造参数
```

//访问元素: 只能访问栈顶元素

```
s5.top() = { "bbb",3 }; //top()返回的是左值引用
cout << s5.top().s1 << s5.top().d1 << endl;
```

//删除元素: 只能删除栈顶元素

```
if (!s5.empty()) s5.pop(); //pop()返回void
```

```
return 0;
```


STL: 容器适配器 queue

```
#include <iostream>
#include <queue>
#include <vector>
#include <string>
using namespace std;
struct A {
    A(const string& s,int d):s1(s),d1(d){}
    string s1; int d1;
};
int main() {
    //queue 队列(先进先出), 默认是 deque实现 【容器适配器】#include <queue>
    //template<class _Ty,class _Container=deque<_Ty>>class stack;
    queue<int> q1; //默认构造(用deque实现)
    //queue<int, vector<int>> q2; 不能vector,没有pop_front()
    //queue<int> q = { 1,2,3 }; 不行
    queue<int> q3(q1);
    queue<int> q4 = q1; //拷贝构造

    q4 = q1; //赋值,没有assign
    q4.swap(q1);
    std::swap(q4, q1); //可以高效swap
    cout << q4.size() << endl; //当前元素个数
    cout << q4.empty() << endl; //是否为空,空返回true;
```

//不支持迭代器,也没有 begin,end 等操作

//插入元素:

```
queue<A> q5;
q5.push(A("abc", 1)); //同样有const T& 和T&&
A a1("aaa", 2);
q5.push(a1);
q5.emplace("ccc", 4); //等同push,不过参数是A的构造参数
```

//访问元素: 只能访问队列头部和尾部元素

```
q5.back() = { "bbb",3 }; //back()返回的是左值引用
q5.front(); //队列头部元素
```

//删除元素: 只能删除栈顶元素

```
if (!q5.empty()) q5.pop(); //pop()返回void
```

```
return 0;
```

STL: 容器适配器 priority_queue

```
#include <iostream>
#include <queue>
#include <vector>
#include <string>
using namespace std;

struct A {
    A(const string& s, int d) :s1(s), d1(d) {}
    string s1; int d1;
};

struct compare_A {
    bool operator()(const A& lhs, const A& rhs)const {
        //先比较 s1,再比较 d1
        if (lhs.s1 < rhs.s1) return true;
        if (!(rhs.s1 < lhs.s1) && lhs.d1 < rhs.d1) return true;
        return false;
    }
};

int main() {
    //priority_queue 优先级队列(符合优先条件的先出),
    //默认是vector实现 【容器适配器】#include <queue>
    //template<class _Ty, class _Container = vector<_Ty>,
    // class _Pr = less<typename _Container::value_type> >
    // class priority_queue;
```

```
//默认构造(用vector实现,优先级的比较是<,大顶堆,大数优先出队)
//使用默认的 less<T> 就是元素要实现 < 运算符
priority_queue<int> q1;
priority_queue<int> q2 = q1; //拷贝构造
q1.swap(q2);
std::swap(q1, q2); //高效交换
cout << q1.size() << endl;
cout << q1.empty() << endl;

//不支持迭代器,也没有 begin,end 等操作

//使用自定义的比较函数,要把三个模板参数都写出来
priority_queue<A, vector<A>,compare_A> q3;

//插入
q3.push(A("123",4));
q3.push(A("123",2));
q3.push(A("abc",13));
q3.push(A("abc", 5)); //传元素类型对象
q3.emplace("abb", 2); //传构造参数

//访问元素,只有top(),没有front, back
cout << q3.top().s1 << "--" << q3.top().d1 << endl;

//删除元素:
if(!q3.empty()) q3.pop();
```