

第7章 模板与STL

- 本章主要教学内容

- 泛型程序设计的技术原理
- 函数模板、类模板程序设计（包括内联函数模板、常量函数模板）
- 模板参数（类型参数、非类型参数、默认参数）、模板实例化（函数模板实例化、类模板实例化、类成员函数实例化）
- 模板重载和模板特化
- 仿函数应用、可变参数函数模板和元编程基础
- **STL**程序设计：函数对象，常用容器、迭代器和算法

- 本章教学重点

- 函数模板，类模板设计
- 函数模板实例化、特化
- **STL**应用：用**list**、**vector**、**string**、**set**、**map**、**tuple**等容器、迭代器和算法处理自定义类
- 仿函数设计及应用

- 教学难点

- 模板可变参数和元编程
- 模板非类型参数
- 元组程序设计

第7章 模板与STL

- 模板（**template**）是**C++**实现代码重用机制的重要工具，是泛型技术（即与数据类型无关的通用程序设计技术）的基础。
- 模板是**C++**中相对较新的语言机制，它实现了与具体数据类型无关的通用算法程序设计，能够提高软件开发的效率，是程序代码复用的强有力工具。
- 本章主要介绍了函数模板和类模板两类，以及**STL库**中的几个常用模板数据类型及其应用。

7.1 模板的概念

1、有关模板的几个重要概念

(1) 模板

- 模板是对具有相同特性的函数或类的再抽象，模板是一种**参数多态性**的工具，可以为**逻辑功能相同而类型不同的程序**提供一种**代码共享的机制**。
- 一个模板并非一个实实在在的函数或类，仅仅是一个函数或类的描述，但它可以**接受数据类型作为其调用参数并生成可用的函数或类**，是参数化的函数和类，是创建函数或类的自动化工具。

(2) 模板的类型

- 函数模板
- 类模板

7.1 模板的概念

(3) 抽象 → 模板

— 从多个具有相同特性的同类事物推导出对该类事物共同特征的统一描述的过程。

- 变量→类型
- 对象→类
- 类→类模板
- 函数→函数模板



抽象

动物： 有
机物为食，是
能够自主运动
或能够活动的
有感觉的生物

```
int min(int a, int b){return (a<b)?a:b;}  
float min(float a, float b){return (a<b)?a:b;}  
double min(double a, double b){return (a<b)?a:b;}
```

抽象

T min(T a,T b) 函数模板
{ return a<b?a:b; }

(4) 实例化 → 模板函数

— 实例化是抽象的逆过程：由抽象类型定义具体变量的过程。模板实例化过程是：

- 用实际数据类型代入模板
- 每一种不同数据类型的实例化，都将生成一份不同的代码

实例化

T min(T a,T b)
{ return a>b?a:b; }

7.1 模板的概念

(5) 模板的抽象与定义

- 某些程序除了所处理的数据类型之外，程序代码和功能完全相同，但为了实现它们，却不得不编写多个与具体数据类型紧密结合的程序。
- 例如，为了求两个int、float、double、char类型数中的最小数，需要编写下列函数：

```
int min(int a, int b){return (a<b)?a:b;}
```

```
float min(float a, float b){return (a<b)?a:b;}
```

```
double min(double a, double b){return (a<b)?a:b;}
```

```
char min(char a, char b){return (a<b)?a:b;}
```

.....

.....

- 有什么办法只写一次代码，却可以处理不同数据类型呢？

7.1 模板的概念

① C语言通过#define定义宏

```
define min(x,y) ((x)<(y) ? (x) : (y))
```

– 特点：

- “自动具备多态特征”
- 不适合表达复杂的逻辑
- 简单的文本替代，不进行任何语法检查，不安全。

② C++方法一：函数重载

– 缺点：

- 相同代码，多次重复编写！

③ C++方法二：函数模板

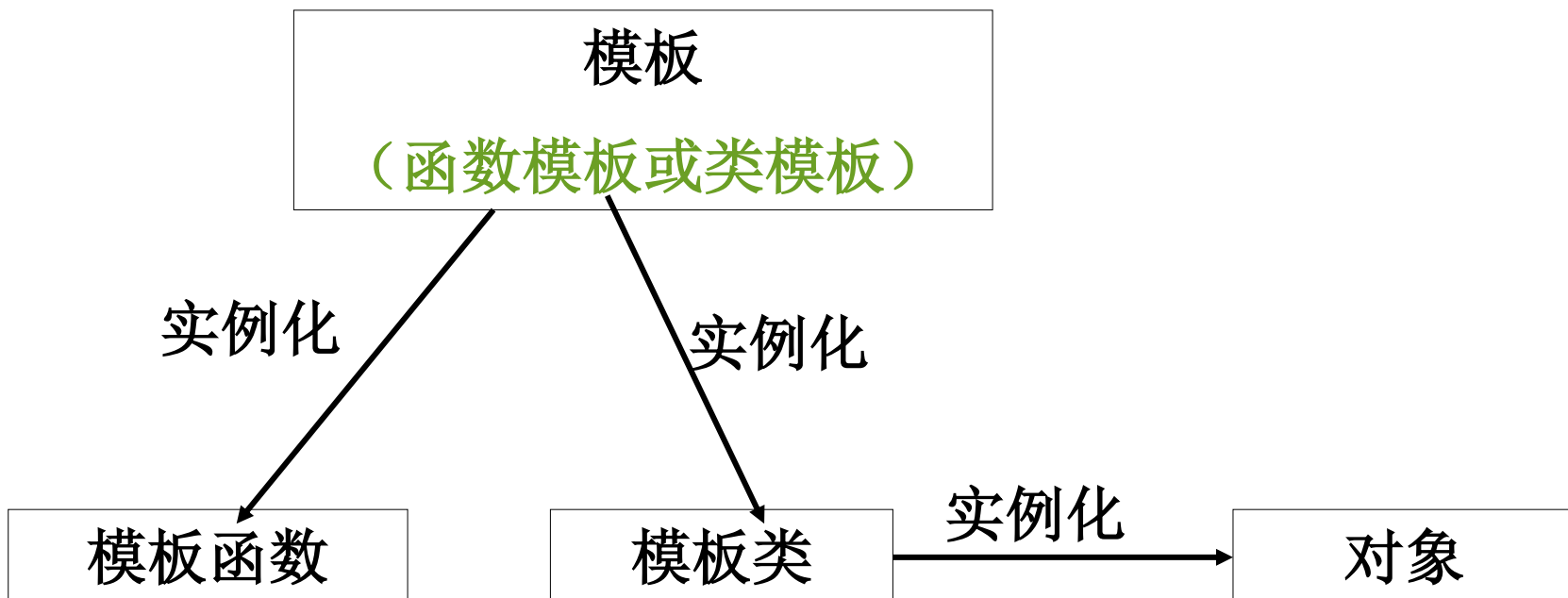
```
template <typename T>
```

```
T min(T a,T b){ return (a<b)?a:b; }
```

– 特点：代码复用的有效机制，可以根据类型生成相应程序代码。

7.1 模板的概念

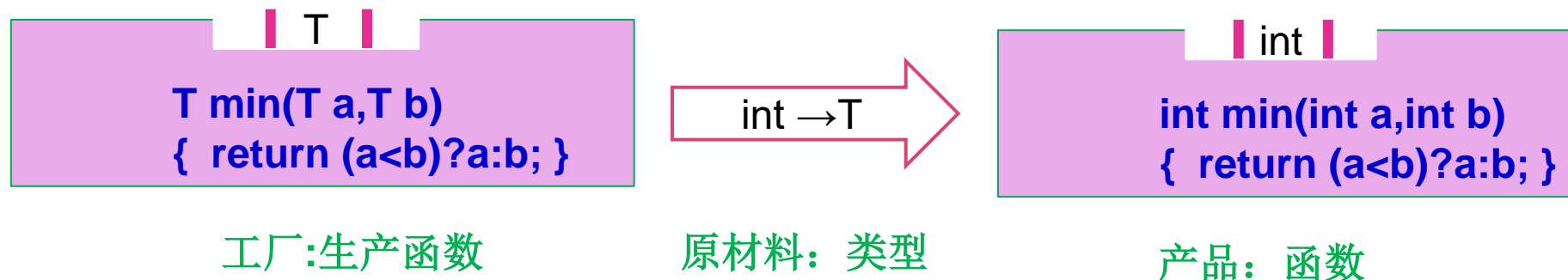
④ 模板、模板函数、模板类和对象之间的关系



7.2 函数模板与模板函数

1、函数模板的功能

- 函数模板提供了一种通用的函数行为，该函数行为可以用多种不同的数据类型进行调用，编译器会根据调用类型自动将它实例化为具体数据类型的函数代码，也就是说函数模板代表了一个函数家族。



2、函数模板与函数重载的区别

- 与普通函数相比，函数模板中某些函数元素的数据类型是未确定的，这些元素的类型将在使用时被参数化；与重载函数相比，函数模板不需要程序员重复编写函数代码，它可以自动生成许多功能相同但参数和返回值类型不同的函数。

7.2.1 函数模板的定义

1、函数模板的定义

template <typename T1, typename T2,...>

返回类型 函数名(参数表){

..... //函数模板定义体

}

- **template**是定义模板的关键字；写在一对<>中的T1，T2，...是**模板参数**，其中的**typename**表示其后的参数可以是任意类型。
- **模板参数**常称为**类型参数**或**类属参数**，在模板实例化（即调用模板函数时）时需要传递的实参是一种数据类型，如**int**或**double**之类。
- 函数模板的参数表中常常出现模板参数，如T1， T2

7.2.1 函数模板的定义

【例7-1】 求两数最小值的函数模板。

```
//Eg7-1.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
template <typename T>
```

```
T min(T a,T b) {
```

```
    return (a<b)?a:b;
```

```
}
```

```
void main(){
```

```
    double a=2,b=3.4;
```

```
    float c=2.3,d=3.2;
```

```
    cout<<"2, 3 的最小值是: "<<min(2,3)<<endl;
```

```
    cout<<"2, 3.4 的最小值是: "<<min(a,b)<<endl;
```

```
    cout<<"'a', 'b' 的最小值是: "<<min('a','b')<<endl;
```

```
    cout<<"2.3, 3.2的最小值是: "<<min(c,d)<<endl;
```

```
}
```

程序运行结果如下:

2, 3 的最小值是: 2

2, 3.4 的最小值是: 2

'a', 'b' 的最小值是: a

2.3, 3.2 的最小值是:

2.3

7.2.1 函数模板的定义

2、使用函数模板的注意事项

- ① 在定义模板时，不允许**template**语句与函数模板定义之间有任何其他语句。

```
template < typename T>
```

```
int x;           //错误，不允许在此位置有任何语句
```

```
T min(T a,T b){...}
```

- ② 函数模板可以有多个**类型参数**，但每个类型参数都必须用关键字**class**或**typename**限定。此外，模板参数中还可以出现确定类型参数，称为**非类型参数**。例：

```
template < typename T1, typename T2, typename T3,int T4>
```

```
T1 fx(T1 a, T2 b, T3 c){...}
```

在传递实参时，**非类型参数T4只能使用常量**，如整数**6**

7.2.1 函数模板的定义

③模板类型参数关键字

- `template <typename T1, class T2>。`
- 这里的**class**和**typename**含义相同，与类没有任何关系，它仅表示T是一个类型参数，可以是任何数据类型，如int、float、char等，或者用户定义的struct、enum或class等自定义数据类型。建议用**typename**，以示区别。

④模板类与普通类文件组织的区别

- 普通函数或类通常把函数或类的声明放在头文件中，把实现代码放在另一个实现文件中，以达到接口与实现分离的目的。
- 模板（包括函数模板和类模板）则不一样，由于在用模板创建（实例化）模板函数或模板类时，编译器必须掌握函数模板或类成员函数模板的确切定义。因此，**必须把模板的声明和定义保存在同一文件中，通常保存在同一头文件中。**

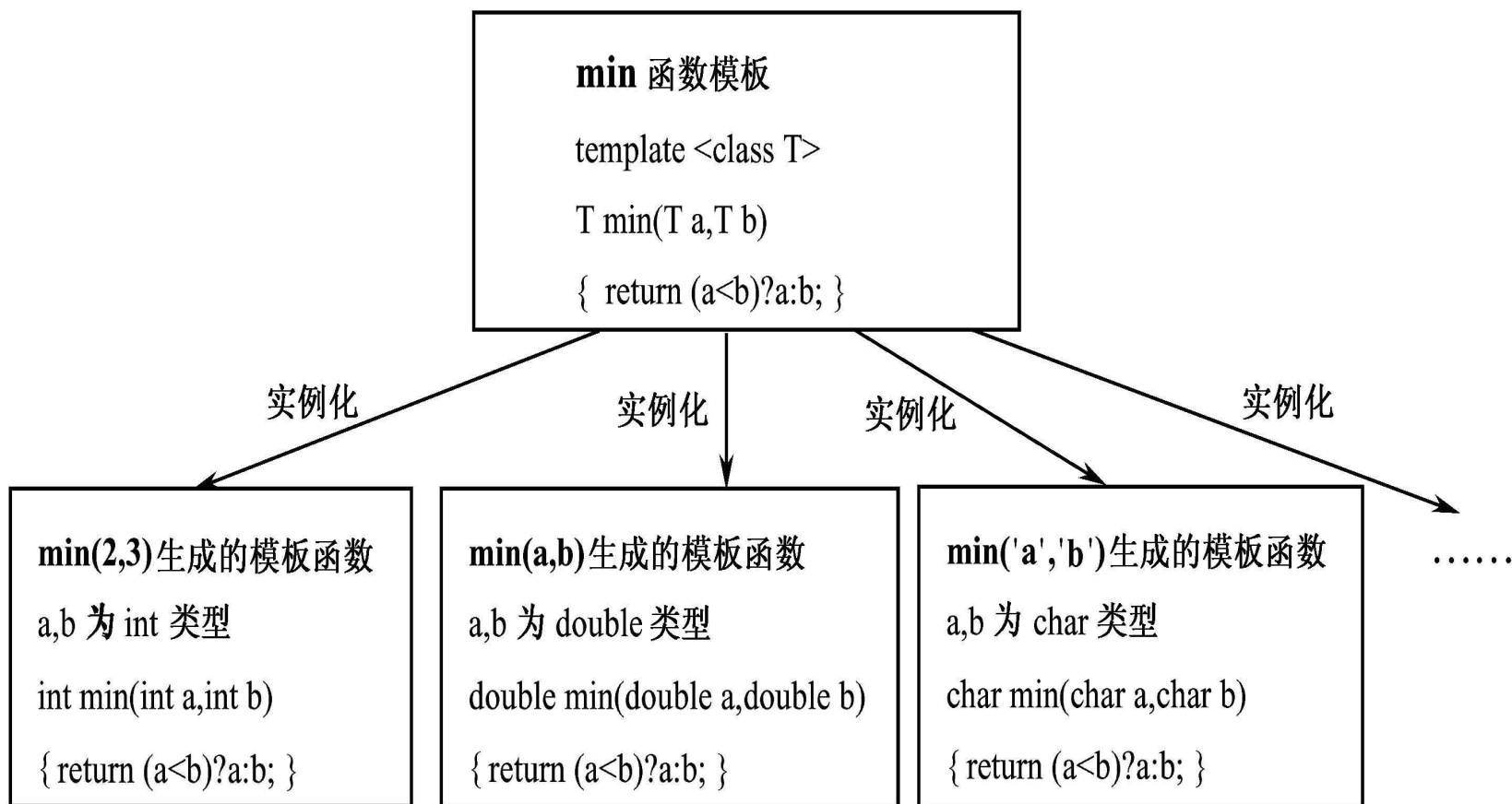
7.2.2 函数模板的实例化

1、实例化发生的时机

- 模板实例化发生在调用模板函数时。
- 当编译器遇到程序中对函数模板的调用时，它才会根据调用语句中实参的具体类型，确定模板参数的数据类型，并用此类型替换函数模板中的模板参数，生成能够处理该类型的函数代码，即模板函数。

7.2.2 函数模板的实例化

- 函数模板实例化情形



7.2.2 函数模板的实例化

2、当多次发生类型相同的参数调用时，**只在第1次进行实例化。**

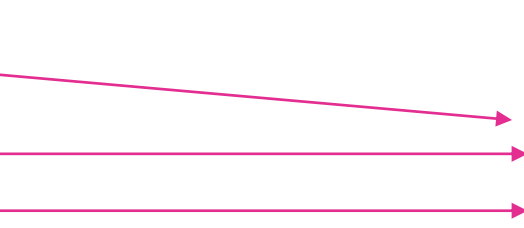
- 对min模板的函数调用：

`int x=min(2,3);`

`int y=min(3,9);`

`int z=min(8,5);`

```
template <class T>
T min(T a,T b) {
    return (a<b)?a:b;
}
```



```
int min(int a,int b) {
    return (a<b)?a:b;
}
```

编译器只在第1次调用时生成模板函数，当之后遇到相同类型的参数调用时，不再生成其它模板函数，它将调用第1次实例化生成的模板函数。

7.2.2 函数模板的实例化

3、实例化的方式

— 隐式实例化

- 编译器能够判断模板参数类型时，自动实例化函数模板为模板函数

```
template <typename T> T max (T, T);
```

```
...
```

```
int i = max (1, 2);
```

```
float f = max (1.0, 2.0);
```

```
char ch = max ('a', 'A');
```

```
...
```

- 隐式实例化，表面上是在调用模板，实际上是调用其实例。

7.2.2 函数模板的实例化

— 显式实例化

- 时机

- 编译器不能判断模板参数类型或常量值
- 需要使用特定数据类型实例化

- 语法形式

- 模板名称<数据类型,...,常量值,...> (参数)

其中数据类型提供给类型参数，常量值提供给非类型参数

- 示例1

```
template <typename T> T max (T, T);
```

```
...
```

```
int i = max (1, '2'); // error: data type can't be deduced
```

```
int i = max<int> (1, '2');
```

```
...
```

7.2.3 模板参数

1、模板参数匹配的问题

- C++在实例化函数模板的过程中，只是简单地将模板参数替换成调用实参的类型，并以此生成模板函数，不会进行参数类型的任何转换！

7.2.3 模板参数

【例7-2-01】 求最大值的函数模板。

```
//Eg.cpp
#include <iostream>
using namespace std;
template <class T>
T max(T a,T b) {
    return (a>b)?a:b;
}
void main(){
    double a=2,b=3.4;
    float c=5.1,d=3.2;
    cout<<"2, 3.2 的最大值是: "<<max(2,3.2)<<endl;
    cout<<"a c 的最大值是: "<<max(a,c)<<endl;
    cout<<"'a', 3 的最大值是: "<<max('a',3)<<endl;
}
```

7.2.3 模板参数

- 编译本程序，将会产生3个编译错误：
C2782, “T max(T,T) ”：模板参数“T”不明确
.....
- 在普通函数的调用过程中，C++会对类型不匹配的参数进行隐式的类型转换。但是，模板实例化过程中不会进行任何形式的参数类型转换，从而导致模板函数的参数类型不匹配，因此产生上述编译错误。

7.2.3 模板参数

- 模板参数不匹配的解决方法

(1) 在模板调用时进行参数类型的强制转换

```
cout<<max(double(2),3.2)<<endl;
```

(2) 显式指定函数模板实例化的类型参数

```
cout<<max<double>(2,3.2)<<endl;
```

```
cout<<max<int>('a',3)<<endl;
```

(3) 指定多个模板参数

【例7-2】 用两个模板参数实现求最大值的函数。

```
#include <iostream>
using namespace std;
template <class T1,class T2>
T1 max(T1 a,T2 b) {
    return (a>b)?a:b;
}
void main(){
    double a=2,b=3.4;
    float c=5.1,d=3.2;
    cout<<"2, 3.2  的最大值是: "
        <<max(2,3.2)<<endl;
    cout<<"a, c  的最大值是: "<<max(a,c)<<endl;
    cout<<"a', 3  的最大值是: "<<max('a',3)<<endl;
}
```

7.2.2 函数模板的实例化

2. 类型与非类型模板参数

- 函数模板参数可以是类属参数，也可以包括普通类型的参数。

【例7-3】 用函数模板实现数组的选择法排序，

//Eg7-3.cpp

```
#include <iostream>
```

```
using namespace std;
```

```
template <typename T>
```

```
void sort(T & a, int n) {
```

```
    for (int i=0;i<n;i++){
```

```
        int p=i;
```

```
    for(int j=i;j<n;j++)
```

```
        if(a[p]<a[j])
```

```
            p=j;
```

```
    int t=a[i];
```

```
    a[i]=a[p];
```

```
    a[p]=t;
```

```
}
```

```
}
```

T是类型参数，n是非类型参数

```

template <typename T>
void display(T& a,int n) {
    for(int i=0;i<n;i++)
        cout<<a[i]<<"\t";
        cout<<endl;
}

```

T是类型参数，**n**是非类型参数

```

void main(){
    int a[]={1,41,2,5,8,21,23};
    char b[]={'a','x','y','e','q','g','o','u'};
    sort(a,7);
    sort(b,8);
    display(a,7);
    display(b,8);
}

```

向模板函数传递非类型参数时只能传递常量

程序运行结果如下：

41	23	21	8	5	2	1	
y	x	u	q	o	g	e	a

7.2.3 模板参数

3. 模板参数的作用域

- 模板参数遵循普通的作用域范围规则：模板参数的可用范围在其声明之后，直到模板声明或定义结束之前；
- 同局部变量一样，模板参数会隐藏外层作用域中声明的相同名称。但与普通函数不同的是，在模板内不能重用模板参数名。

```
struct A {};
```

```
template<typename A, typename B>
```

```
void f(A a, B b) {
```

```
    A t = a;
```

```
    //t是typename A指定的类型
```

```
    int B;
```

```
    //错误，重用模板参数定义变量名称。
```

```
}
```

- 在f模板内，模板参数A隐藏了外层作用域定义的结构类型A。

7.3 类模板

7.3.1 类模板的概念

- 类模板可用来设计**结构和成员函数完全相同**，但所处理的**数据类型不同的通用类**。如栈，

双精度栈：

```
class doubleStack{  
    private:  
        double data[size];  
    public:  
        push(double);  
        double pop()  
        double top();  
};
```

字符栈：

```
class charStack{  
    private:  
        char data[size];  
        .....  
};  
    public:  
        push(char);  
        char pop();  
        char top();  
};
```

- 这些栈除了数据类型之外，操作完全相同，就可用类模板实现。

7.3.2 类模板的定义

1、类模板的声明

```
template<typename T1,typename T2,...>  
class 类名{  
    .....  
    // 类成员的声明与定义  
}
```

- 其中T1、T2是类型参数
- 类模板中可以有多多个模板参数，包括类型参数和非类型参数

7.3.2 类模板的定义

- **非类型参数**是指某种具体的数据类型，在调用模板时只能为其提供用相应类型的常数值。

`template<class T1,class T2,int T3>`

- T1、T2是类型参数，T3是非类型参数。
- 在实例化时，必须为T1、T2提供一种数据类型，为T3指定一个常整数（如10），该模板才能被正确地实例化。
- **非类型参数的限制**
 - 通常可以是整型、枚举型、对象或函数的引用，以及对对象、函数或类成员的指针
 - 但不允许用浮点型（或双精度型）、类对象或void作为非类型参数。

7.3.2 类模板的定义

2、类模板的成员函数的定义

方法1：在类模板外定义，语法

template <模板参数列表>

返回值类型 类模板名<模板参数名称列表>::成员函数名 (参数列表)

{

.....

};

其中

- <模板参数列表>引入的“类型标识符”作为数据类型使用
- <模板参数名称列表>引入的“普通数据类型常量”作为常量使用

方法2:

- 直接在模板内定义成员函数，与常规成员函数的定义方法相同。

【例7-4】 设计一个堆栈的类模板**Stack**，在模板中用类型参数**T**表示栈中存放的数据，用非类型参数**MAXSIZE**代表栈的大小。

```
//Eg7-4.cpp
```

```
//Stack.h
```

```
template<typename T, int MAXSIZE>
class Stack{
private:
    T elems[MAXSIZE];
    int top;
public:
    Stack(){top=0;};
    void push(T e);
    T pop();
    bool empty(){return top==0;}
    bool full(){return top==MAXSIZE;}
};
```

7.3.2 类模板的定义

```
template<typename T, int MAXSIZE>
void Stack<T, MAXSIZE>::push(T e) {
    if(top==MAXSIZE){
        cout<<"栈已满，不能再加入元素了！";
        return;
    }
    elems[top++]=e;
}
template<class T, int MAXSIZE>
inline T Stack<T, MAXSIZE>::pop(){
    if(top<=0){
        cout<<"栈已空，不能再弹出元素了！"<<endl;
        return 0;
    }
    top--;
    return elems[top];
}
```

<模板参数列表>

<模板参数名称列表>

7.3.3 类模板实例化

1、类模板实例化的内容

包括**模板实例化**和**成员函数实例化**

2、类模板实例化的时间

当用**类模板定义对象**时，引起类模板的实例化

3、实例化的方法：

在实例化类模板时，如果模板参数是类型参数，则必须为它指定具体的类型；如果模板参数是非类型参数，则必须为它指定一个常量值。

7.3.3 类模板实例化

4、实例化案例

如对Stack类模板，下面的定义将引起实例化

```
Stack<int,10> iStack;
```

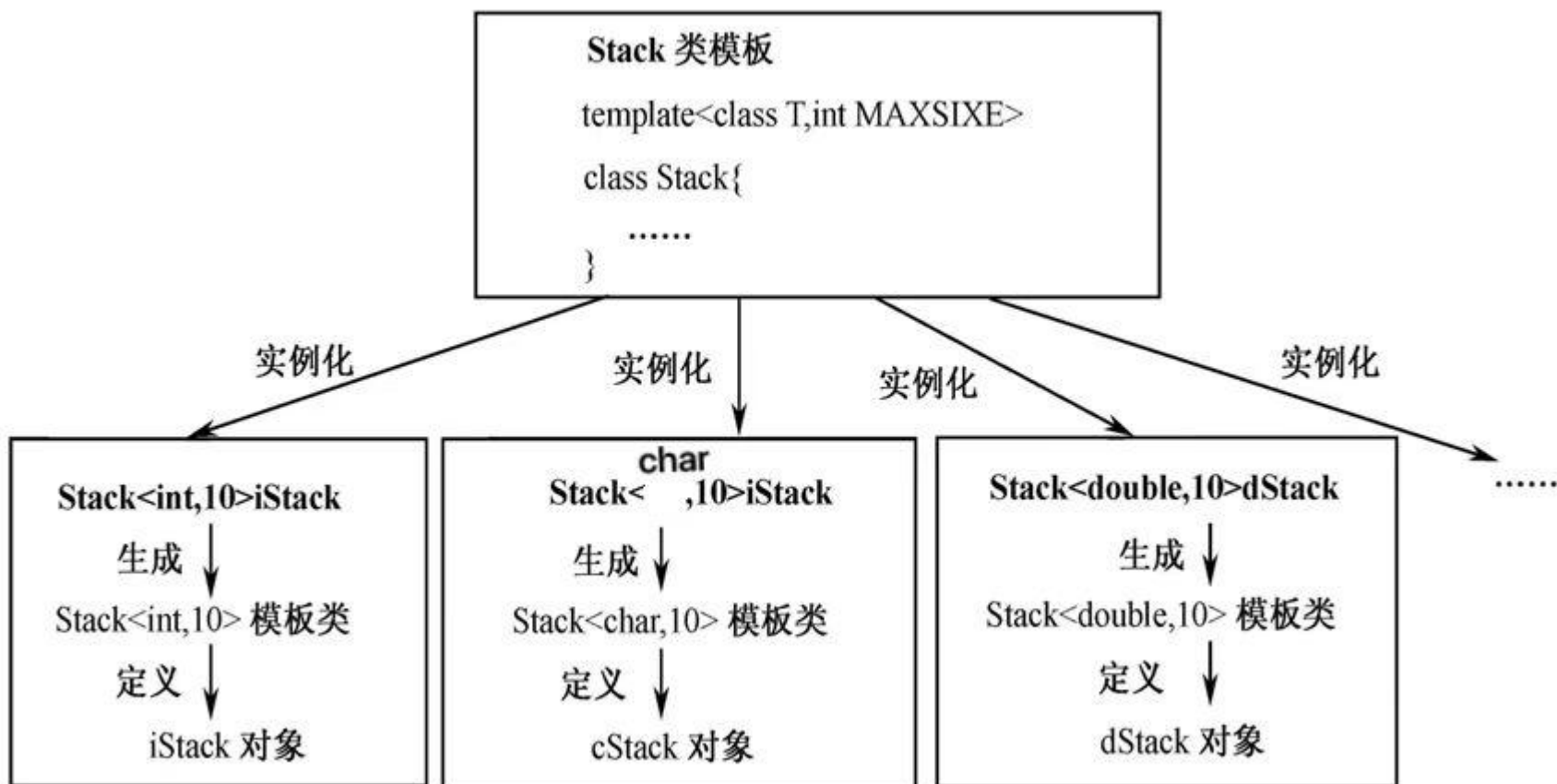
— 编译器实例化iStack的方法是：

- ① **数据成员实例化**：将Stack模板声明中的所有数据成员的类型参数T替换成int，将所有的非类型参数MAXSIZE替换成10，生成了一个int类型的模板类。
- ② **成员函数实例化**：
 - 用int替换无参构造函数中的T，实例化出被调用的构造函数。
 - **未被调用的成员函数不被实例化**（即不生成相应的成员函数）

```
class Stack{
private:
    int elems[10];
    int top;                                //栈顶指针
public:
    Stack(){top=0;};
    void push(int e);                       //入栈操作
    int pop();                              //出栈操作
    bool empty(){return top==0;}
    bool full();}
};
```

7.3.3 类模板实例化

Stack模板能够实例化出无穷多的模板类



7.3.4 类模板的使用

- 为了使用类模板对象，**显式指定模板实参**进行模板实例化。

//Eg7-4b.cpp

#include "stack.h"

//该头文件的内容见例7-4所示的程序清单

#include <iostream>

using std::cout;

//只使用std域名空间中的cout

using std::endl;

//只使用std域名空间中的endl

void main(){

Stack<int,10> iStack;

Stack<char,10> cStack;

cout<<"-----intStack----\n";

int i;

for(i=1;i<10;i++) iStack.push(i);

for(i=1;i<10;i++) cout<<iStack.pop()<<"\t";

cout<<"\n\n-----charStack----\n";

cStack.push('A'); cStack.push('B');

cStack.push('C'); cStack.push('D');

cStack.push('E');

for(i=1;i<6;i++) cout<<cStack.pop()<<"\t";

cout<<endl;

}

7.3.4 类模板的使用

【例7-5】 对例7-4建立的Stack类模板编写一个函数 **display**，该函数能够读取并显示Stack模板类建立的栈中的所有元素。

```
//Eg7-5.cpp
#include "stack.h"
#include <iostream>
using namespace std;
template<class T>
void display(Stack<T,10> &s) {
    while( !s.empty())
        cout<<s.pop()<<"\t";
    cout<<endl;
}
void main(){
    Stack<int,10> iStack;
    cout<<"-----intStack----\n";
    for(int i=1;i<10;i++)
        iStack.push(i);
        display(iStack);
}
```

7.5 STL

- STL就是标准模板库（Standard Template Library），是惠普实验室开发的一系列软件的统称。STL当前已是C++的一部分，因此不用安装额外的库文件。
- 本质上讲，STL是一些“容器”、算法和其他一些组件的集合，其目的是标准化组件，减少重复开发，可以使用现成的组件。。
- 在C++标准中，常用STL库被组织为下面的13个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。

7.5.1 函数对象

1、函数对象的概念

- STL标准库为每个算术运算和关系运算定义了一个对应的运算符模板类，能够实现对应的运算符操作，称为**函数对象**。
- 函数对象的定义在**functional**头文件中，如表所示。

算术对象	关系对象	逻辑运算对象
plus<T>	equal_to<T>	logical_and<T>
minus<T>	not_equal_to<T>	logical_or<T>
multiplies<T>	greater<T>	logical_not<T>
divides<T>	greater_equal<T>	
modulus<T>	less<T>	
negate<T>	less_equal<T>	

7.5.1 函数对象

2、函数对象的应用

- 在程序中可以用这些模板定义对象，实现相应的运算。

【例7-19】 函数对象应用示例。

```
//Eg7-19.cpp
```

```
#include<iostream>
```

```
#include<functional>
```

```
#include<algorithm>
```

```
using namespace std;
```

```
void main() {  
    int a[] = { 3,1,7,0,-3,2,8,-5 };
```

```
    plus<int> iadd;
```

```
    minus<double> dm;
```

```
    less<int> les;
```

```
    int s = iadd(5, 6);
```

```
    double d = dm(25, 5);
```

```
    cout << "s=" << s << "\td=" << d << "\t";
```

```
    if (les(5, 7)) cout << "5<7"<<endl;
```

```
    else cout << "5>7" << endl;
```

```
    sort(a, a + 8, less<int>());           // 从小到大排序a数组
```

```
    for (int i = 0; i < 8; i++) cout << a[i]<<"\t";
```

```
    cout << endl;
```

```
    sort(a, a + 8, greater<int>());       //从大到小排序a数组
```

```
    for (int i = 0; i < 8; i++) cout << a[i]<<"\t";
```

```
    cout << endl;
```

```
}
```

程序运算结果如下：

s=11 d=20 5<7

-5	-3	0	1	2	3	7	8
8	7	3	2	1	0	-3	-5

7.5.2 顺序容器

C++容器的概念及类型

- 容器（**container**）是用来存储其他对象的对象，它是用模板技术实现的。
- STL的容器包括以下三类。
 - ① 顺序容器
向量（**vector**）、链表（**list**）、双端队列（**deque**）。
 - ② 关联容器
集合（**set**）、多重集合（**multiset**）
 - ③ 容器适配器
堆栈（**stack**）、队列（**queue**）、**priority_queue**（优先队列）

表7-2 STL中的容器及头文件名

容器名	头文件名	说 明
vector	<vector>	向量，从后面快速插入和删除，直接访问任何元素
list	<list>	双向链表
deque	<deque>	双端队列
set	<set>	元素不重复的集合
multiset	<set>	元素可重复的集合
stack	<stack>	堆栈，后进先出（LIFO）
map	<map>	一个键只对于一个值的映射
multimap	<map>	一个键可对于多个值的映射
queue	<queue>	队列，先进先出（FIFO）
priority_queue	<queue>	优先级队列

表7-3 所有容器都具有的成员函数

成员函数名	说 明
默认构造函数	对容器进行默认初始化的构造函数，常有多，用于提供不同的容器初始化方法
拷贝构造函数	用于将容器初始化为同类型的现有容器的副本
析构函数	执行容器销毁时的清理工作
empty()	判断容器是否为空，若为空返回true，否则返回false
max_size()	返回容器最大容量，即容器能够保存的最多元素个数
size	返回容器中当前元素的个数
operator=	将一个容器赋给另一个同类容器
operator<	如果第1个容器小于第2个容器，则返回true，否则返回false
operator<=	如果第1个容器小于等于第2个容器，则返回true，否则返回false
operator>	如果第1个容器大于第2个容器，则返回true，否则返回false
operator>=	如果第1个容器大于等于第2个容器，则返回true，否则返回false
swap	交换两个容器中的元素

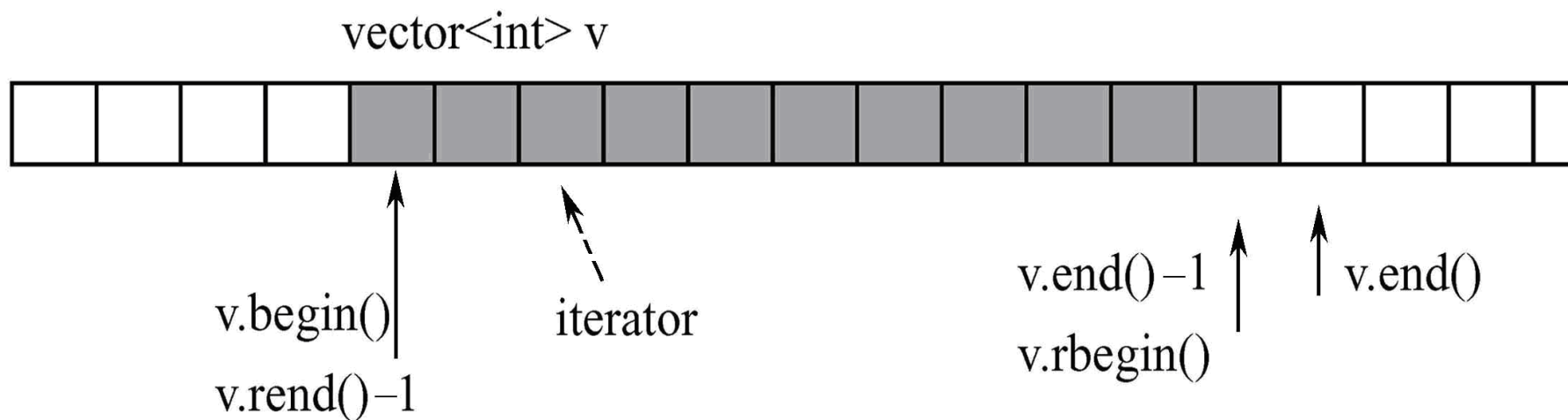
表7-4 顺序和关联容器共同支持的成员函数

成员函数名	说 明
begin()	指向第一个元素
end()	指向最后一个元素的后一个位置
rbegin()	指向按反顺序的第一个元素的前一个位置
rend()	指向按反顺序的末端位置
erase()	删除容器中的一个或多个元素
clear()	删除容器中的所有元素

7.5.2 顺序容器

1. vector

- vector**是向量容器，它具有存储管理的功能，在插入或删除数据时，**vector**能够自动扩展和压缩其大小。可以像数组一样使用**vector**，通过运算符[]访问其元素，但它比数组更灵活，当添加数据时，**vector**的大小能够自动增加以容纳新的元素。图是向量的一个示意图。



7.5.2 顺序容器

【例7-20】 **vector**向量的应用举例。

```
//Eg7-20.cpp
```

```
#include<iostream>
```

```
#include<vector>
```

//向量头文件

```
using namespace std;
```

```
void display(vector<int> &v) {
```

```
    while(!v.empty()){
```

```
        cout<<v.back()<<"\t"; //输出向量的尾部元素
```

```
        v.pop_back();           //删除向量尾部元素
```

```
    }
```

```
    cout<<endl;
```

```
}
```

7.5.2 顺序容器

```
void main(){
    int a[]={1,2,3,4,5,6};
    vector<int> v1, v2;
    vector<int> v3(a,a+6);
    组的两个迭代器
    vector<int> v4(6);
    v1.push_back(10);
    v1.push_back(11);
    v1.push_back(12);
    v1.insert(v1.begin(),30);
    v2=v1;
    v3.assign(3,10);
    cout<<"v1: "; display(v1);
    cout<<"v2: "; display(v2);
    cout<<"v3: "; display(v3);
    v4[0]=10; v4[1]=20;
    v4[2]=30; v4[3]=40;
    cout<<"v4: ";
    for(int i=0;i<6;i++)
        cout<<"v4["<i>i</i>]<<"\t";
    cout<<endl;
    v4.resize(10);
    cout<<"v4: "; display(v4);
}
```

//定义只有0个元素的向量v1、v2

//定义向量v3，并用a数组初始化该向量，参数为a数

//定义具有6个元素的向量v4

//在v1向量的尾部加入元素10

程序运行结果如下：

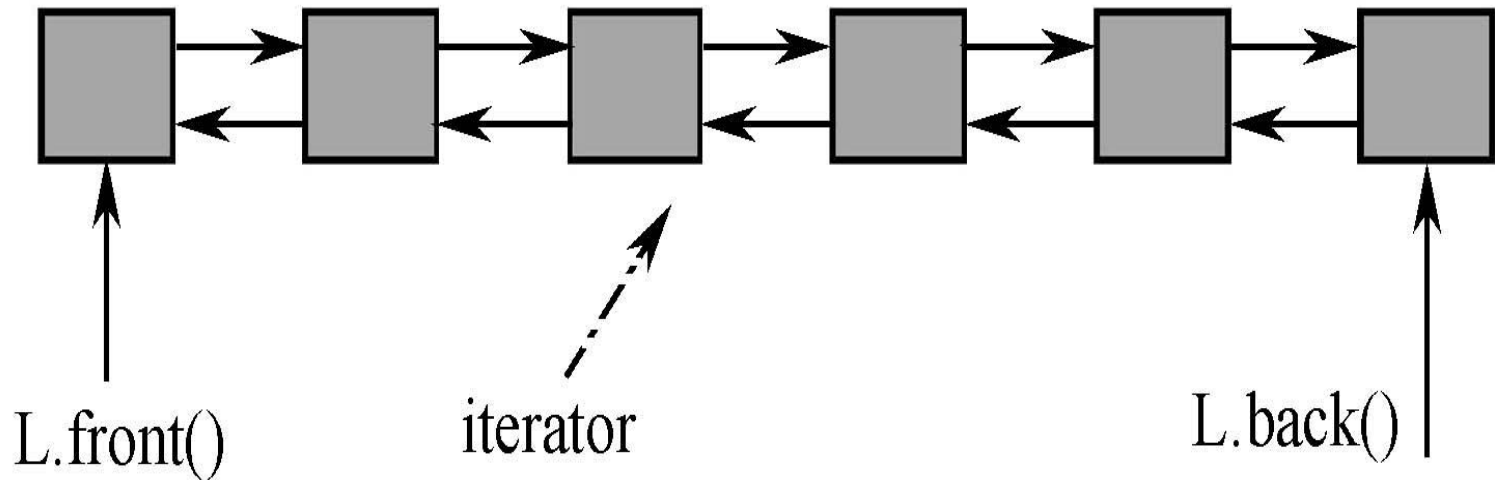
v1:	12	11	10	30		
v2:	12	11	10	30		
v3:	10	10	10	4	5	6
v4:	10	20	30	40	0	0
v4:	0	0	0	0	0	0
	0	40	30	20	10	

//重置向量v4的大小，已有元素不受影响

7.5.2 顺序容器

2. List

- STL中的list是一个双向链表，可以从头到尾或从尾到头访问链表中的节点，节点可以是任意数据类型。链表中节点的访问常常通过迭代器进行。下图是一个链表的示意图。



链表的操作

(1) 链表的构造 (模板参数T是链表的数据类型)

<code>list<T> c</code>	建立一个空链表c
<code>list<T> c1(c2)</code>	建立与c2同型的链表c1 (c2的每个元素都被复制)
<code>list<T> c(n)</code>	建立具有n个元素的链表c, 元素值由默认构造函数产生
<code>list<T> c(n,e)</code>	建立n个元素的链表c, 每个元素的值都是e
<code>list<T> c(beg, end)</code>	建立链表c, 并用[beg, end]区间内的元素作初始化
<code>c.~list<e>()</code>	销毁链表c, 释放内存

(2) 链表赋值

<code>c1=c2</code>	将c2链表的全部元素赋值给c1链表
<code>c1.assign(n,e)</code>	将元素e拷贝n次到c1链表
<code>c.assign(beg,end)</code>	将区间[beg,end]的元素赋值给c
<code>c1.swap(c2)</code>	将链表c1和c2的全部元素互换

(3) 链表存取

c.front()	返回第一个元素，不检查元素存在与否
c.back()	返回最后一个元素，不检查元素存在与否

(4) 链表插入与删除

c.insert(pos,e)	在pos位置插入元素e的副本，并返回新元素的位置
c.insert(pos,n,e)	在pos位置插入元素e的n个副本，没有返回值
c.insert(pos, beg, end)	在pos位置插入区间[bed, end]内的全部元素
c.push_back(e)	在尾部追加一个元素e的副本
c.push_front(e)	在表头插入元素e的一个副本
c.pop_back(e)	删除最后一个元素
c.pop_front()	删除第一个元素
c.remove(val)	删除值为val的元素
c.remove_if(op)	删除所有“造成op(e)结果为true”的元素
c.erase(pos)	删除pos指向的元素，返回下一元素的位置
c.erase(beg, end)	删除区间[beg,end]内的元素，返回下一元素位置
c.resize(n)	将链表c的大小重新设置为n
c.clear()	删除链表所有元素，将整个容器置空

链表的操作

(5) 链表的特殊操作

c.unique()	删除相邻重复元素，只留一个
c.unique(op)	若存在若干相邻且使op()操作为true的元素，删除重复，只留一个
c1.splice(pos, c2)	将c2内的所有元素转换到c1内，pos之前
c1.splice(pos, c2, c2pos)	将c2链表的c2pos所指元素移到c1内的pos指向的位置
c1.splice(pos, c2, c2beg, c2end)	将c2内[c2beg, c2end]区间的所有元素转换到c1内pos之前
c.sort()	以operator<为准则，对所有元素排序
c.sort(op)	以op()为准则，对所有元素排序
c1.merge(c2)	c2合并到c1，若合并前有序则合后仍有序
c.reverse()	将所有元素反序

7.5.2 顺序容器

【例7-21】 list应用的一个例子。

```
//Eg7-21.cpp
#include<iostream>
#include<list>                                //链表头文件
using namespace std;
void main(){
    int i;
    list<int> L1,L2;
    int a1[]={100,90,80,70,60};
    int a2[]={30,40,50,60,60,60,80};
```

7.5.2 顺序容器

```
for(i=0;i<5;i++)  
    L1.push_back(a1[i]);           //将a1数组加入到L1链表中  
for(i=0;i<7;i++)  
    L2.push_back(a2[i]);           //将a2数组加入到L2链表中  
L1.reverse();                     //将L1链表倒序  
L1.merge(L2);                     //将L2合并到L1链表中  
cout<<"L1的元素个数为: "<<L1.size()<<endl;  
L1.unique();                       //删除L1中相邻位置的相同元素, 只留1个  
while(!L1.empty()){  
    cout<<L1.front()<<"\t";  
    L1.pop_front();               //删除L1的链首元素  
}  
cout<<endl;  
};
```

本程序的运行结果如下:

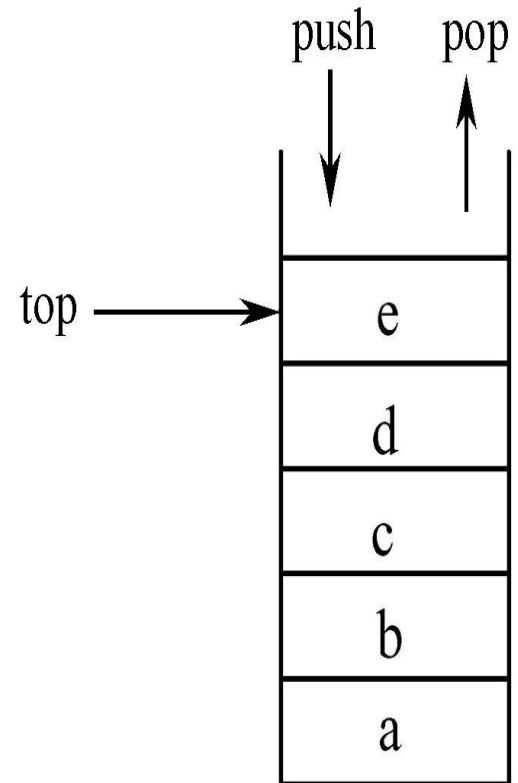
L1的元素个数为: 12

30 40 50 60 70 80 90 100

7.5.2 顺序容器-适配器

3. stack

- 堆栈（**stack**）是一种较简单的常用容器，它是一种受限制的向量，只允许在向量的一端存取元素，后进栈的元素先出栈，即**LIFO**（**last in first out**）。右图是一个字符堆栈的示意图。
- 可以用**list**、**vector**实现**stack**，因此是一种容器适配器。



7.5.2 顺序容器

- STL中的堆栈提供的主要操作如下：
 - push() 将一个元素加入stack内，加入的元素放在栈顶
 - top() 返回栈顶元素元素的值
 - pop() 删除栈顶元素
- top与pop是不同的，top只返回栈顶元素的值，不删除元素，而pop只删除栈顶元素，不返回值。

7.5.2 顺序容器

【例7-22】 STL stack应用的例子。

```
//Eg7-22.cpp
#include<iostream>
#include<stack>
using namespace std;
void main(){
    stack<int> s;
    s.push(10);      s.push(20); s.push(30);
    cout<<s.top()<<"\t";
    s.pop();         s.top()=100;
    s.push(50);      s.push(60);
    s.pop();
    while(!s.empty()){
        cout<<s.top()<<"\t";
        s.pop();
    }
    cout<<endl;
}
```

程序运行结果（分析其由来）：

30	50	100	10
----	----	-----	----

7.5.2 顺序容器

4. String

- **string**可以被看成是以字符（**characters**）为元素的一种容器，字符构成序列（字符串），有时需要在字符序列中进行遍历，标准**string**类提供了**STL**容器接口，具有成员函数**begin()**和**end()**，迭代器可以用这两个函数进行定位。
- **STL**中的**string**是一种特殊类型的容器，原因是它除了可作为字符类型的容器外，更多的是作为一种数据类型——字符串，可以像**int**、**double**之类的基本数据类型那样定义**string**类型的数据，并进行各种运算。

表7-5 string的重载运算符

运算符	举例（s1、s2是string类型）	说 明
=	s2=s1	赋值运算，将s1赋值给s2
>	s1>s2	若s1大于s2，结果为真，否则为假
==	s1==s2	若s1等于s2，结果为真，否则为假
>=	s1>=s2	若s1大于或等于s2，结果为真，否则为假
<	s1<s2	若s1小于s2，结果为真，否则为假
<=	s1<=s2	若s1小于或等于s2，结果为真，否则为假
!=	s1!=s2	若s1不等于s2，结果为真，否则为假
+=	s1+=s2	将s2连接在s1后面，并赋值给s1
[]	s[1]='a'	string可用数组方式访问元素，起始下标为0

(1) string的常用成员函数

在下面的string成员函数介绍中，假设s1、s2的定义如下：

```
string s1="ABCDEFGFH";  
string s2="0123456123";  
string s;
```

- ① **substr(n1, n)** 取子串函数，从当前字符串的n1下标开始，取出n个字符。如
“s=s1.substr(2, 3)”的结果为：s="CDE"
- ② **swap(s)** 交换字符串。如“s1.swap(s2)”的结果为：
s1="0123456123", s2="ABCDEFGFH"
- ③ **size()/length()** 计算字符串中当前存放的字符个数。如“s1.length()”的结果为：7
- ④ **capacity()** 计算字符串的容量（可容纳的字符个数）。如
“s1.capacity()”的结果为：31
- ⑤ **max_size()** 计算string类型数据的最大容量。如“s1.max_size()”的结果为：
4294967293
- ⑥ **find(s)** 在当前字符串中查找子串s，如找到就返回s在当前串中的起始位置；若没有找到，返回常数string::npos。如“s1.find("EF")”的结果为：4
- ⑦ **rfind(s)** 同find，但从后向前进行查找。如“s1.rfind("BCD")”的结果为：1

(1) string的常用成员函数

- ① **find_first_of(s)** 在当前串中查找子串s第一次出现的位置。如
“s2.find_first_of("123")”的结果为：1
- ② **find_last_of(s)** 在当前串中查找子串s最后一次出现的位置。如
“s2.find_last_of("123")”的结果为：9
- ③ **replace(n1, n, s)** 替换当前字符串中的字符，n1是替换的起始下标，n是要替换的字符个数，s是用来替换的字符串。如“s1.replace(2, 3, s2)”的结果为：s1="AB0123456123FH"
- ④ **replace(n1, n, s, n2, m)** n1是替换的起始下标，n是替换掉的字符个数，s是用来替换的字符串，n2是s中用来替换的起始下标，m是s中用于替换的字符个数。如“s1.replace(2, 3, s2, 2, 3)”的结果为：s1="AB234FH"
- ⑤ **insert(n, s)** 在当前串的下标位置n之前，插入s串。如“s1.insert(2, "88888")”的结果为：s1="AB88888CDEFH"
- ⑥ **insert(n1, s, n2, m)** 在当前串的n1下标后插入s串，n2是s串中要插入的起始下标，m是s串中要插入的字符个数。如“s1.insert(2, s2, 3, 2)”的结果为：s1="AB34CDEFH"

7.5.2 顺序容器

【例7-23】 string应用的例子。

//Eg7-23.cpp

```
#include<iostream>
```

```
#include<string>
```

```
using namespace std;
```

```
void main(){
```

```
    string s1="中华人民共和国成立了";
```

```
    string s2="中国人民从此站起来了！";
```

```
    string s3,s4,s5;
```

```
    s3=s1+"， "+s2;
```

7.5.2 顺序容器

```
int n=s1.find_first_of("人民");
    if (n!=string::npos)
        cout<<"人民在s1中的位置: "<<n<<endl;
    else
        cout<<"在s1中没有该子串! ";
s4=s1.substr(4,10);
cout<<"s1= "<<s1<<endl;
cout<<"s2= "<<s2<<endl;
cout<<"s3= "<<s3<<endl;
cout<<"s4= "<<s4<<endl;
if (s1>s2)    cout<<"s1>s2= true "<<endl;
else         cout<<"s1>s2= false"<<endl;
s3.replace(s3.find("从此"),4,"从1949年");
cout<<"s3 after replace= "<<s3<<endl;
s3.insert(s3.find("站"),"10月");
cout<<"s3 after insert= "<<s3<<endl;
```

```
}
```

7.5.2 顺序容器

- 程序运行结果如下：

人民在s1中的位置： 4

s1= 中华人民共和国成立了

s2= 中国人民从此站起来了！

s3= 中华人民共和国成立了， 中国人民从此站起来了！

s4= 人民共和国

s1>s2= true

s3 after replace= 中华人民共和国成立了， 中国人民从1949年站起来了！

s3 after insert= 中华人民共和国成立了， 中国人民从1949年10月站起来了！

7.5.3 迭代器

1、迭代器的概念

- 迭代器（**iterator**）是一个**对象**，常用它来遍历容器，即在容器中实现“取得下一个元素”的操作。
- 迭代器的操作类似于指针，但它是基于模板的“功能更强大、更智能、更安全的指针”，用于指示容器中的元素位置，通过迭代器能够遍历容器中的每个元素。
- 迭代器是**STL**的核心，定义了哪些算法在哪些容器中使用，把算法和容器连接起来，使算法、容器和迭代器能够协同工作，实现强大的程序功能。
- 若某个容器要使用迭代器，就必须定义迭代器。定义迭代器时，必须指定迭代器所使用的容器类型。

7.5.3 迭代器

2、迭代器的基本操作

- ① 在容器中的特定位置定位迭代器。
- ② 在迭代器指示位置检查是否存在对象。
- ③ 获取存储在迭代器指示位置的对象值。
- ④ 改变迭代器指示位置的对象值。
- ⑤ 在迭代器指示位置插入新对象。
- ⑥ 将迭代器移到容器中的下一个位置。

7.5.3 迭代器

3、迭代器提供的主要操作

- `operator *` 返回当前位置上的元素值
- `operator++` 将迭代器前进到下一个元素位置
- `operator--` 将迭代器后退到前一个元素位置
- `operator==` 或 `operator!=`
- `operator=` 为迭代器赋值
- `begin()` 指向容器起点（即第一个元素）位置
- `end()` 指向容器的结束点
- `rbegin()` 指向按反向顺序的第一个元素位置之前
- `rend()` 指向按反向顺序的最后一个元素后的位置

7.5.3 迭代器

【例7-24】 链表迭代器应用举例。

//Eg7-24.cpp

```
#include<iostream>
```

```
#include<list>
```

```
using namespace std;
```

```
int main(){
```

```
    int i;
```

```
    list<int> L1, L2, L3(10);
```

```
    list<int>::iterator iter;           //定义迭代器iter
```

```
    int a1[]={100,90,80,70,60};
```

```
    int a2[]={30,40,50,60,60,60,80};
```

```
    for(i=0;i<5;i++)
```

```
        L1.push_back(a1[i]);
```

```
    for(i=0;i<7;i++)
```

```
        L2.push_front(a2[i]);
```

```

for(iter=L1.begin(); iter!=L1.end(); iter++)
    cout<<*iter<<"\t"
cout<<endl;
int sum=0;
//通过迭代器反向输出L2除第一个的所有元素
for(iter=--L2.end();iter!=L2.begin();iter--){
    cout<<*iter<<"\t";
    sum+=*iter; //计算L2所有链表节点的总和
}
cout<<"\nL2: sum="<<sum<<endl;
int data=0;
//通过迭代器修改L3链表的内容
for(iter=L3.begin();iter!=L3.end();iter++)
    *iter=data+=10;
for(iter=L3.begin();iter!=L3.end();iter++)
    cout<<*iter<<"\t";
cout<<endl;
return 0;
}

```

程序运行结果如下：

100	90	80	70	60
30	40	50	60	60
	60			
L2: sum=300				
10	20	30	40	50
	60	70	80	90
	100			

7.5.4 pair和tuple容器

1、关于pair和tuple容器

- tuple与pair都是一种容器模板,pair只能有两个元素;
- tuple称为元组,可以有任意多个不同类型的元素(但一个定义好的tuple类型的成员数目是固定的)。
- tuple可以用STL中的其它容器,如list、vector、map、set等作为元组的成员,建立随意而功能强大的数据结构。

7.5.4 pair和tuple容器

2、tuple对象的构造

- 定义一个tuple时，需要指出每个成员的类型。
tuple<T1,T2.....,Tn> t; //使用默认构造函数
tuple<T1,T2,.....Tn> t(v1,v2.....vn); //使用指定值初始化
tuple<T1,T2,.....Tn> t{v1,v2.....vn}; //用值列表初始化
- 例如，
- tuple<int, string,char*> t1,t2{ 1,"数据结构","3.5学分" };
- tuple<string, vector<double>, int, list<int>> someVal("constants", { 3.12,2.34,32 }, 42, { 0,1,1,1 });
- 注意：tuple的构造函数是explicit 的，可以像t2那样用列表直接初始化，但不能用列表赋值方式初始化，如下所示：
- tuple<int,int,int > t={1,2,3}; //错误原因tuple禁用了operator=
- tuple<int,int,int > t{1,2,3}; //正确

7.5.4 pair和tuple容器

3、tuple元素访问

- tuple中的每个成员都是public属性，且都可以是对象、数组之类的复杂数据类型。C++标准模板中提供了一个get函数模板，它以类似于数组下标索引的方式访问元组中指定位置的成员，用法如下：

get<i>(t) //t是元组，i是元组中的元素位置，第1个元素位置为0

- 例如
- tuple<char *, int, vector<string>, list<int>>
 tue("tom", 101, { "语文","数学","英语" }, { 76,87,91 });
- get<0>(tue) //返回考生姓名: "tom"
- get<1>(tue) //返回考生考号: 101
- get<2>(tue) //返回考试科目: { "语文","数学","英语" }
- get<3>(tue) //返回科目成绩: { 76,87,91 }
- get<3>(tue)[0] //返回{ 76,87,91 }中的第一个元素，即76

7.5.4 pair和tuple容器

4、tuple操作

- 同类型的tuple可以进行 <、==、=运算。
- 用make_tuple函数构造tuple对象，或者调用参数的类型推断tuple类型等。
- 例如，

```
auto t=make_tuple("string",3,2.1);
```

编译器会据实参推断元组类型t为：tuple<const char*,int,double>，等价于下面的定义：

```
tuple<const char *,int,double> t("string", 3, 2.1);
```

7.5.4 pair和tuple容器

5、tuple应用

- 当希望将一些类型不同但具有联系的数据组合成单一对象，而又不想定义类或结构时，用元组把这些数据组合起来（快而随意的数据结构）就显得非常有用。
- 用元组作为函数的返回类型，可以一次返回多个数据。

【例7-26】 在一个成绩系统中，要求函数返回的成绩数据包括：学生姓名，专业，班主任，大学英语，数学，C语言等5科成绩，用元组处理数据可以简化程序设计。

- 下面的程序用来说明元组的解决本应用的方法。其中，`inputData`函数用于说明为元组中的成员输入数据，并通过函数返回元组的方法；`display`函数用于说明向函数传递元组参数的方法。

7.5.4 pair和tuple容器

//Eg7-26.cpp

```
#include <tuple>
```

```
#include<string>
```

```
#include<list>
```

```
#include <vector>
```

```
#include<iostream>
```

```
using namespace std;
```

```
struct Grade {                                //表示学生科目、成绩的数据结构
```

```
    string courseName;
```

```
    double grade;
```

```
    Grade(string s, double g) :courseName(s), grade(g) {}
```

```
};
```

```
typedef tuple<string, int, string, vector<Grade>> Student;
```

inputData函数返回的Student是一种复杂的元组数据类型，该元组保存学生的姓名，学号，专业，以及任意多门课程的成绩。

```
typedef tuple<string, int, string, vector<Grade>> Student;
```

```
Student inputData() {  
    Student stu;  
    cout << "输入学生数据: 姓名, 学号, 专业" << endl;  
    cin >> get<0>(stu) >> get<1>(stu) >> get<2>(stu);    //元组元素访问方法  
    string cName;  
    double score=0;  
    int i = 1;  
    while (cName != "exit") {  
        cout << "输入第 " << i++ << " 科目名称, 输入exit结束:\t";  
        cin >> cName;  
        if (cName == "exit") break;  
        cout << "成绩:\t";  
        cin >> score;  
        get<3>(stu).push_back(Grade(cName, score));    //向元组向量添加对象  
    }  
    return stu;  
}
```

```
//typedef tuple<string, int, string, vector<Grade>> Student;
```

```
void display(Student student) {
```

```
    cout << get<0>(student) << "\t" << get<1>(student) << "\t"
        << get<2>(student) << "\t" << endl;
```

//for循环示范了访问元组中具有不确定个数的向量元素的访问方法。

```
for (int i = 0; i < get<3>(student).size(); i++)
```

```
    cout << get<3>(student)[i].courseName << "\t"
        << get<3>(student)[i].grade << endl;
```

```
}
```

```
void main(){
```

```
    auto t = make_tuple("string", 3, 20.01);
```

```
    tuple<char *,int,double> tt("string", 3, 20.01);
```

```
    //tuple<int, int, int > t = { 1,2,3 };
```

//错误

```
    tuple<int, int, int > t5{ 1,2,3 };
```

//正确

```
    tuple<int, string,char*> t1,t2{ 1,"数据结构","3.5学分" };
```

```
    t1 = t2;
```

//同类型元组可以赋值

```
    cout << get<0>(t1) << "\t" << get<1>(t1) << "\t"
```

```
        << get<2>(t1) << endl;
```

//元组访问的常规方法

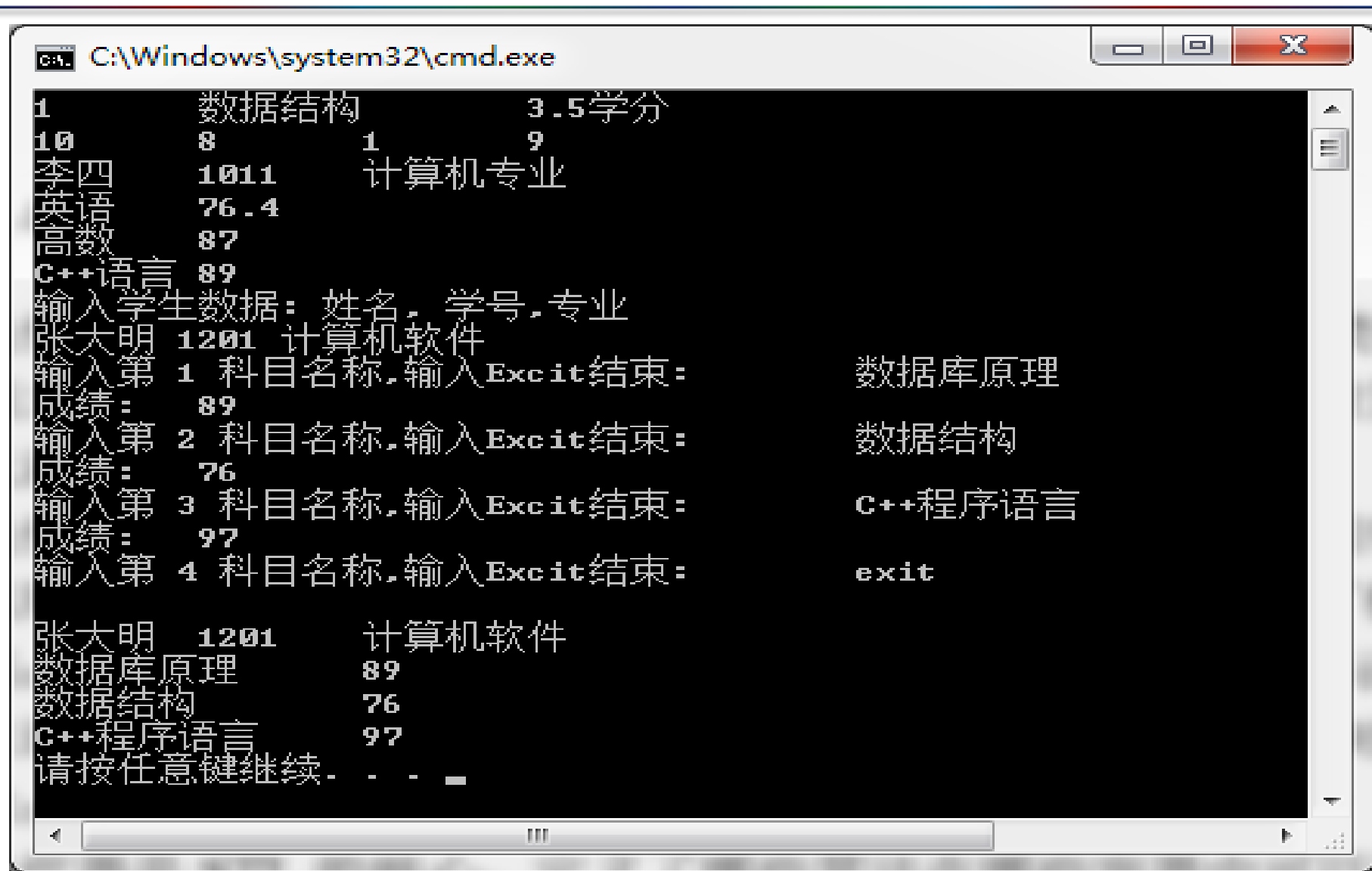
//下面的代码段示例了具有链表、向量元素的元组定义方法，以及元组中的链表访问方法

```
tuple<string,vector<double>,int,list<int>>vtable("tomoto",{3.12,2.34}, 42, {10,8,1,9});  
list<int>::iterator iter;           //访问链表的迭代器  
for (iter = get<3>(vtable).begin(); iter != get<3>(vtable).end(); iter++)  
    cout << *iter << "\t";  
    cout << endl;
```

//student对象示例了向元组中的对象数组赋值的方法。

```
Student student{"李四",1011,"计算机专业",{{"英语",76.4},{ "高数",87},{ "C++语言",89}}};  
display(student);  
student = inputData();  
cout << endl;  
display(student);  
}
```

程序运行结果



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of a program that processes student data. The output is as follows:

```
1      数据结构      3.5学分
10     8      1      9
李四   1011   计算机专业
英语   76.4
高数   87
C++语言 89
输入学生数据: 姓名, 学号, 专业
张大明 1201  计算机软件
输入第 1 科目名称, 输入Exit结束:      数据库原理
成绩:      89
输入第 2 科目名称, 输入Exit结束:      数据结构
成绩:      76
输入第 3 科目名称, 输入Exit结束:      C++程序语言
成绩:      97
输入第 4 科目名称, 输入Exit结束:      exit

张大明 1201  计算机软件
数据库原理  89
数据结构    76
C++程序语言 97
请按任意键继续. . .
```

7.5.6 算法

- 算法（**algorithm**）是用模板技术实现的适用于各种容器的通用程序。算法常常通过迭代器间接地操作容器元素，而且通常会返回迭代器作为算法运算的结果。
- **STL**大约提供了100个算法，每个算法都是一个模板函数或者一组模板函数，能够在许多不同类型的容器上进行操作，各个容器则可能包含着不同类型的数据元素。**STL**中的算法覆盖了在容器上实施的各种常见操作，如遍历、排序、检索、插入及删除元素等操作

7.5.6 算法

1. find和count算法

- find用于查找指定数据在某个区间中是否存在，该函数返回等于指定值的第一个元素位置，如果没有找到就返回最后元素位置；count用于统计某个值在指定区间出现的次数，
- 其用法如下：
 - `find(beg,end,value)`
 - `count(beg,end,value)`
 - [beg, end]是指定的区间，常用迭代器位置描述该区间，value是要查找或统计的值。

7.5.6 算法

【例7-31】 find算法举例。

//Eg7-31.cpp

```
#include<iostream>
```

```
#include<list>
```

```
#include<algorithm>
```

```
using namespace std;
```

```
void main(){
```

```
    int arr[]={100,200,300,400,500,500,600,700,800,900,1000};
```

```
    int *ptr;
```

```
    ptr=find(arr,arr+9,400);
```

//查找400在arr数组中的地址

```
    cout<<"400在数组中的下标是： "
```

```
        <<ptr<<endl;
```

//find返回地址，

```
    list<int> L1;
```

//定义链表L1

```
    int a1[]={30,40,50,60,60,60,80};
```

```
    for(int i=0;i<7;i++)
```

```
        L1.push_back(a1[i]);
```

//将a1数组加入到L1链表中

7.5.6 算法

```
list<int>::iterator pos;
pos=find(L1.begin(),L1.end(),80);           //在L1中查找80，结果放于pos中
if(pos!=L1.end())
cout<<"L1链表中存在数据元素： "<<*pos;      //输出找到的数据
cout<<"，它是链表中的第： "<<distance(L1.begin(),pos)+1
    <<"个节点！ "<<endl;                    //计算迭代器与链首元素间隔的元素个数
int n1=count(arr,arr+10,500);               //统计arr数组中500的个数
int n2=count(L1.begin(),L1.end(),60);       //统计L1链表中60的个数
cout<<"arr 数组中有： "<<n1<<"个"<<500<<endl;
cout<<"L1链表中有： "<<n2<<"个"<<60<<endl;
}
```

程序运行结果如下：

400在数组中的下标是： 3

L1链表中存在数据元素： 80，它是链表中的第： 7个节点！

arr 数组中有： 2个500

L1链表中有： 3个60

7.5.6 算法

2. merge

`merge`可对两容器进行合并，将结果存放在第3个容器中，

- 其用法如下：

`merge(beg1, end1, beg2, end2, dest)`

- `merge`将`[beg1, end1]`与`[beg2, end2]`区间合并，把结果存放在`dest`容器中。如果参与合并的两个容器中的元素是有序的，则合并的结果也是有序的。
- 说明：`list`链表也提供了一个`merge`成员函数，它能够把两个`list`类型的链表合并在一起。同样，如果合并前的链表是有序的，则合并后的链表仍然有序。

【例7-32】 merge算法与list的merge成员函数的应用。

```
#include<iostream>
#include<list>
#include<algorithm>
using namespace std;
void main(){
    int a1[]={10,20,30,40,50,60,70};
    int a2[]={40,50,60};
    int a[10];
    merge(a1,a1+7,a2,a2+3,a);           //将a1、a2合并，结果放在a数组中
    for(int i=0;i<10;i++)                cout<<a[i]<<"\t";
    cout<<endl;
    list<int> L1,L2;
    list<int>::iterator pos;              //pos迭代器用于输出链表元素
    for(int i=0;i<7;i++)
        L1.push_back(a1[i]);             //插入L1的链表元素
    for(int j=0;j<3;j++)
        L2.push_back(a2[j]);             //插入L2的链表元素
    L1.merge(L2);                         //用list的merge成员合并L1、L2
    for(pos=L1.begin();pos!=L1.end();pos++) //用迭代器输出合并后的L1
        cout<<*pos<<"\t";
    cout<<endl;
}
```

7.5.6 算法

3. search算法

- **search**算法则是从一个容器查找由另一个容器所指定的顺序值。
- **search**用法形式如下：
search(beg1,end1,beg2,end2)

search将在[beg1, end1]区间内查找有无与[beg2, end2]相同的子区间，如果找到就返回[beg1, end1]内第一个相同元素的位置，如果没找到，返回end1.

7.5.6 算法

【例】 search算法举例：查找某数据区间在另一数据区间中是否存在

```
#include<iostream>
#include<vector>
#include<list>
#include<algorithm>
using namespace std;
void main(){
    int a1[]={10,20,30,40,50,60,70,80};
    int a2[]={40,50,60};
    int *ptr;
    ptr=search(a1,a1+8,a2,a2+3);    //查找a2数组在a1中的位置
    if(ptr==a1+8)
        cout<<"no match found\n";
    else
        cout<<"a2 match a1 at:"<<(ptr-a1)<<endl; //输出第一个匹配元
        素的位置
```

7.5.6 算法

```
vector<int> v;  
list<int> L;  
for(int i=0;i<8;i++)  
    v.push_back(a1[i]);    //将a1数组插入v向量  
for(int j=0;j<3;j++)  
    L.push_back(a2[j]);    //将a2数组插入L链表  
vector<int>::iterator pos;  
pos=search(v.begin(),v.end(),L.begin(),L.end());  
    //在v中查找L  
cout<<distance(v.begin(),pos)<<endl;  
    //distance计算找到元素在V中的下标  
}
```


7.5.6 算法

4. sort

sort可对指定容器区间内的元素进行排序，默认的排序方式是从小到大

- 其用法如下：

`sort(beg,end)`

`[beg, end]`是要排序的区间，**sort**将按从小到大的顺序对该区间的元素进行排序。

【例7-33】 利用sort算法对数组和向量进行排序。

//Eg7-33.cpp

```
#include<iostream>
```

```
#include<vector>
```

```
#include<algorithm>
```

```
using namespace std;
```

```
void main(){
```

```
    int a1[]={10,-20,30,4,50,13,7};
```

```
    int a2[]={-2,0,30,12,6,7,-9,56,32,78};
```

```
    sort(a1,a1+7);
```

//排序a1数组

```
    cout<<"a1[]: ";
```

```
    for(int i=0;i<7;i++)
```

```
        cout<<a1[i]<<"\t";
```

```
    cout<<endl;
```

```
    vector<int> L1;
```

```
    vector<int>::iterator pos;
```

```
    for(i=0;i<10;i++)
```

```
        L1.push_back(a2[i]);
```

//将a2数组插入到L1链表中

```
    sort(L1.begin(),L1.end());
```

//排序L1

```
    cout<<"L1: ";
```

```
    for(pos=L1.begin();pos!=L1.end();pos++)
```

```
        cout<<*pos<<"\t";
```

//输出L1链表中的值

```
    cout<<endl;
```

```
}
```

7.5.7 STL容器和算法处理自定义类的常见问题

■ 代码无问题却报出大量错误信息

- 初用STL库处理自定义类对象时，可以会遇到编译器一下报出了上百个错误。遇到这样的问题时不要慌，可能就一两个错误。
- 可以先排除程序本身的语法错误，再根据具体的错误信息检查是否是模板参数不匹配，或者某项内容与const参数不符合。
- 最常见的错误是应用STL中的set、map、sort、min、max等容器和算法处理自定义类对象时，没有重载小于或大于运算符函数，无法进行自定义类对象排序所产生的错误。原因是set和map容器是有序的（包括multiset和multimap等），在将元素插入这类容器时就要进行对象的大小比较，如果类对象没有提供大小比较函数就无法排序，程序就会产生错误。
- 如果自定义类没有定义大小比较方法，在用sort、min、max等算法处理这样的类对象时也会产生错误。解决的方法是为类提供比较大小的函数，最简单的方法是重载小于比较运算符函数，形式如下：

```
class A {  
    .....  
    // set、map要求存入其中的对象的比较函数为const函数  
    bool operator<(const A& o) const {...}  
    .....  
};
```

【例7-34】 定义简单的复数类，建立该复数的数组和set集合，并用STL中的sort算法对数组进行排序输出。

```
// Eg7-34.cpp
#include<iostream>
#include<set>
#include<algorithm>
using namespace std;

class Complex {
private:
    double i, r;
public:
    Complex(double ir, double rr):i(ir), r(rr) {}
    void set_r(double pr) { r = pr; }
    void set_i(double pi) { i = pi; }
    const double& real()const { return r; }
    double& image() { return i; }
    // bool operator<(const Complex &o)const { return r < o.r; } // L1
    void outdata() {
        if (i >= 0)
            cout<<r<<"+"<<i<<"i"<<endl;
        else
            cout<<r<<i<<"i"<<endl;
    }
};
```

```
int main() {
    Complex c1[] = {{1.0,1.0}, {6.0,2.0}, {3.0,3.0}};
    cout<<"-----sort array output-----"<<endl;
    sort(c1, c1+2);
    for(Complex c:c1)    c.outdata();
    cout<<"-----sort set output-----"<<endl;
    set<Complex>cset;
    for(Complex c:c1)
        cset.insert(c);
    Complex c2{4.0, 4.0}, c3{1.0, 1.0};
    cset.insert(c2);    //L2
    cset.insert(c3);    //L3
    for(Complex c:cset)
        c.outdata();
}
```

程序将产生编译错误，取消L1位置的注释符，程序就能够正常运行！

原因是L2、L3会默认调用小于比较运算符函数

小结: STL 与operator<运算符

- STL容器大多具有排序功能，这些容器在默认情况下，用operator<运算符对容器中的对象进行升序排序。
- 容器中的排序算法在默认情况下也用operator<运算符对容器中的对象进行大小比较，按升序排序。

如 list set/multiset map/multiset

- 因此，必须对 operator<进行重载实现自定义对象的大小比较，才能够将对象插入容器。
- 通用方法：以友元重载operator<

```
Class A{  
    bool operator<(const A&right);  
    friend bool operator<(const A &left,const A &right)  
}
```

7.6编程实作：模板和STL编程应用

- 现接着本书6.6节“编程实作二”继续完成comFinal、Account和Chemistry课程结构的程序设计。

【例7-35】 STL中的容器和算法不仅适用于内置数据类型，而且同样适用于用户自定义的数据类型。现在接着本书6.6节编程实作二继续完成comFinal、Account和Chemistry课程结构的程序设计。

问题简析：

- STL中的各种容器，如vector、list、stack、deque、set/multiset、map/multimap等都可以用来存取comFinal、Account及Chemistry类的对象。这里用list容器存取各个类的对象。
- 过程如下：

7.6编程实作：模板和STL编程应用

1. 建立项目并编写主程序

- 启动Visual C++2022，选择“文件 | 新建|项目”菜单命令，在C盘根目录下建立一个新项目，项目名为courseList。在建立项目时，**取消“安全开发生命周期（SDL）检查”**
- 进入项目的程序编辑窗口，修改该项目的源文件“courleList.cpp”，下面是其全部代码：

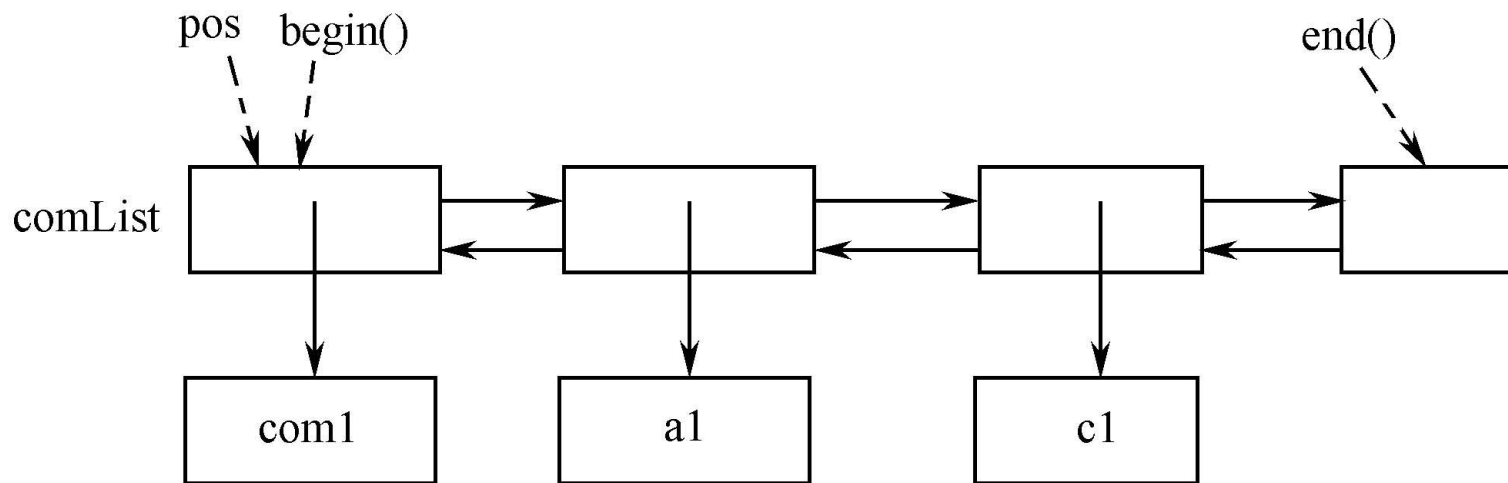


```
// courselist.cpp : 定义控制台应用程序的入口点。  
//
```

```
#include "stdafx.h"  
#include<iostream>  
#include<list>  
#include"Chemistry.h"  
#include"Account.h"  
using namespace std;  
void main() {  
    list<comFinal*> comList;//定义基类comFinal对象的指针链表  
    list<comFinal*>::iterator pos;  
    comFinal com1("阿曼", 76, 87, 90);  
    Account a1("张三星", 98, 90, 97, 90, 90);  
    Chemistry c1("光红顺", 89, 80, 80, 80, 80);  
    comList.push_back(&com1);//将基类comFinal对象的指针加入链表  
    comList.push_back(&a1);//将派生类Account的对象指针加入链表  
    comList.push_back(&c1);//将派生类Chemitry的对象指针加入链表  
    for (pos = comList.begin(); pos != comList.end(); pos++)  
        (*pos)->show();//遍历链表，输出各对象的数据成员  
}
```


7.6编程实作：模板和STL编程应用

- 上述主函数中建立了对象链表，如下图所示



7.6编程实作：模板和STL编程应用

2. 在项目中加入各个类的程序代码

- 将第6章建立的目录C:\course下的comFinal.h、Account.h、Chemistry.h以及comFinal.cpp、Account.cpp、Chemistry.cpp复制到courseList.cpp所在的目录中。
- 选择菜单“工程 | 添加工程 | files”，将上述各类的头文件和源码文件添加到当前工程中。

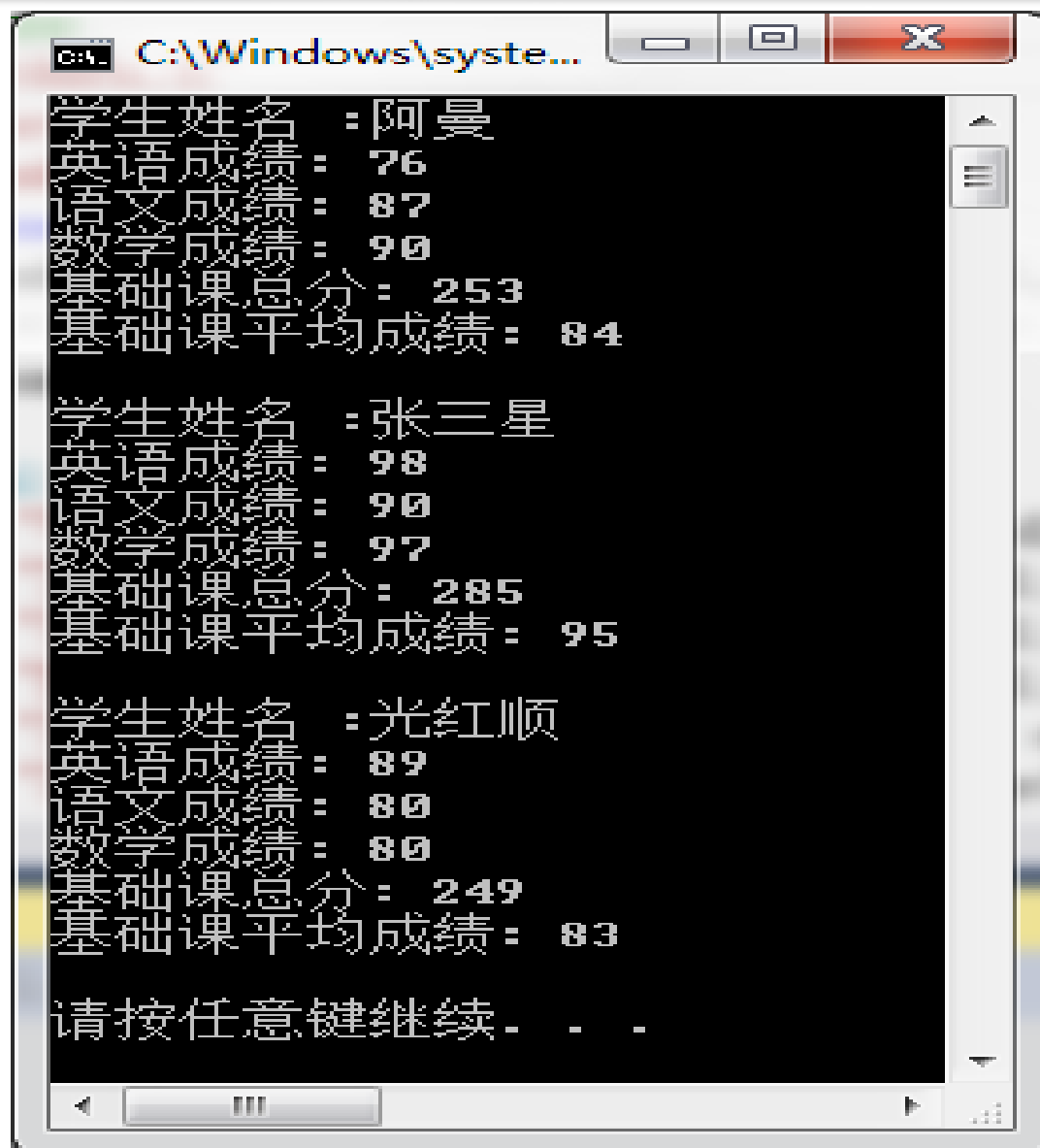
3. 验证程序

- 编译运行courseList程序。
- 若在编译时出现类似下面的错误
- 错误 **C1010** 在查找预编译头时遇到意外的文件结尾。是否忘记了向源中添加“`#include "stdafx.h"”`? `courseListc:\courselist\courselist\comfinal.cpp` **17**
- 则在comFinal.cpp、Account.cpp、Chemistry.cpp三个源码文件的开头添加下代的代码行：

```
#include "stdafx.h"
```

7.6编程实作：模板和STL编程应用

- 程序运行结果



```
C:\Windows\system...
学生姓名 : 阿曼
英语成绩 : 76
语文成绩 : 87
数学成绩 : 90
基础课总分 : 253
基础课平均成绩 : 84

学生姓名 : 张三星
英语成绩 : 98
语文成绩 : 90
数学成绩 : 97
基础课总分 : 285
基础课平均成绩 : 95

学生姓名 : 光红顺
英语成绩 : 89
语文成绩 : 80
数学成绩 : 80
基础课总分 : 249
基础课平均成绩 : 83

请按任意键继续. . .
```