

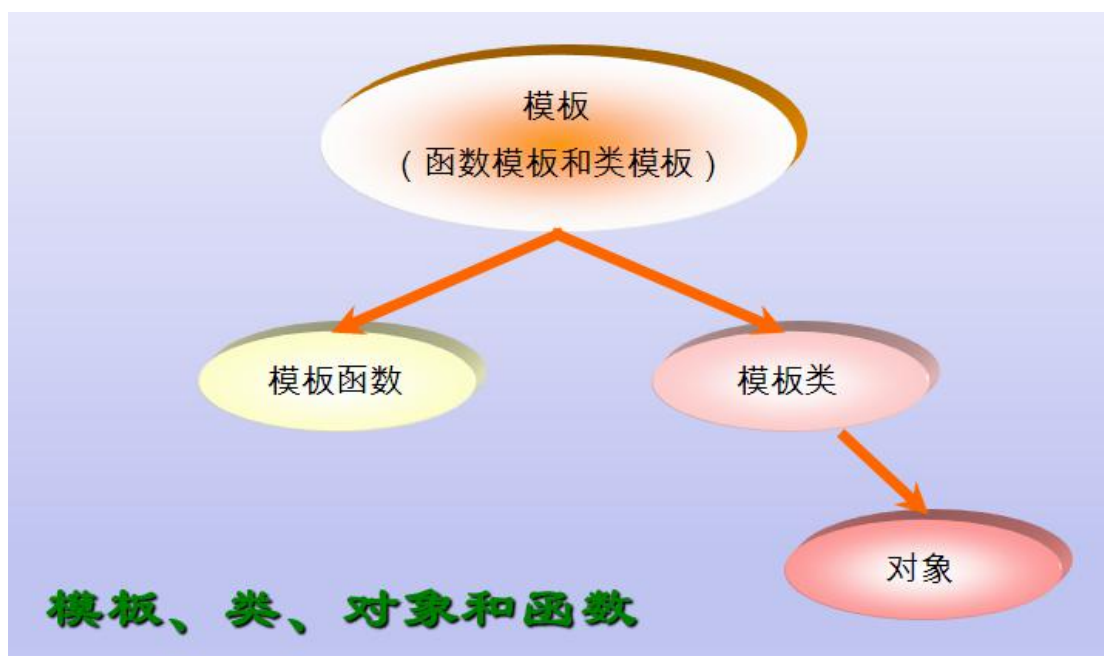
传智播客 C++ 进阶课程讲义

传智扫地僧

6、函数模板和类模板

前言

C++ 提供了函数模板(function template)。所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不具体指定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。凡是函数体相同的函数都可以用这个模板来代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现了不同函数的功能。



1) C++ 提供两种模板机制：函数模板、类模板

2) 类属 —— 类型参数化，又称参数模板

使得程序（算法）可以从逻辑功能上抽象，把被处理的对象（数据）类型作为参数传递。

总结：

- 模板把函数或类要处理的数据类型参数化，表现为参数的多态性，称为类属。
- 模板用于表达逻辑结构相同，但具体数据元素类型不同的数据对象的通用行为。

6.1 函数模板

6.1.1 为什么要有函数模板

需求：写 n 个函数，交换 char 类型、int 类型、double 类型变量的值。

案例：

```
#include <iostream>
using namespace std;
/*
void myswap(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
}

void myswap(char &a, char &b)
{
    char t = a;
    a = b;
    b = t;
}
*/

//template 关键字告诉 C++编译器 我要开始泛型了.你不要随便报错
//数据类型 T 参数化数据类型
template <typename T>
void myswap(T &a, T &b)
{
    T t;
    t = a;
    a = b;
    b = t;
}

void main()
{
    //char a = 'c';

    int x = 1;
    int y = 2;
    myswap(x, y); //自动数据类型 推导的方式
```

```
float a = 2.0;
float b = 3.0;

myswap(a, b); //自动数据类型 推导的方式
myswap<float>(a, b); //显示类型调用

cout<<"hello..."<<endl;
system("pause");
return ;
}
```

6.1.2 函数模板语法

函数模板定义形式

template < 类型形式参数表 >

类型形式参数的形式为：

typename T_1 , **typename** T_2 , , **typename** T_n
或 **class** T_1 , **class** T_2 , , **class** T_n

函数模板声明

```
template < 类型形式参数表>
类型 函数名 ( 形式参数表 )
{
    语句序列
}
```

- 函数模板定义由模板说明和函数定义组成
- 模板说明的类属参数必须在函数定义中至少出现一次
- 函数参数表中可以使用类属类型参数，也可以使用一般类型参数

函数模板调用

```
myswap<float>(a, b);    //显示类型调用
myswap(a, b);            //自动数据类型推导
```

6.1.3 函数模板和模板函数

The diagram illustrates the concept of function templates and template functions. It shows a C++ code snippet for a `max` function template, three specific function definitions (for `int`, `char`, and `double`), and a screenshot of the program's output.

Code Snippet:

```
#include <iostream.h>
template < typename T >
T max ( T a , T b )
{ return a > b ? a : b ; }

void main ()
{ cout << " max ( 3 , 5 ) is " << max ( 3 , 5 ) << endl ;
  cout << " max ( 'y' , 'e' ) is " << max ( 'y' , 'e' ) << endl ;
  cout << " max ( 9.3 , 0.5 ) is " << max ( 9.3 , 0.5 ) << endl ;
}
```

Compiler-generated template functions:

- `int max (int a , int b) { return a > b ? a : b ; }`
- `char max (char a , char b) { return a > b ? a : b ; }`
- `double max (double a , double b) { return a > b ? a : b ; }`

Program execution output:

```
max ( 3 , 5 ) is 5
max ( 'y' , 'e' ) is y
max ( 9.3 , 0.5 ) is 9.3
Press any key to continue
```

Annotations:

- 编译器生成的模板函数 (Compiler-generated template functions)
- 程序执行时匹配不同的版本 (Program execution matches different versions)

6.1.4 函数模板做函数参数

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

template<typename T, typename T2>
void sortArray(T *a, T2 num)
{
    T tmp ;
    int i, j ;
    for (i=0; i<num; i++)
    {
        for (j=i+1; j<num; j++)
        {
            if (a[i] < a[j])
            {
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
        }
    }
}
```

```
}

template<class T>
void pirntArray(T *a, int num)
{
    int i = 0;
    for (i=0; i<num; i++)
    {
        cout<<a[i]<<" ";
    }
}

void main()
{
    int num = 0;
    char a[] = "ddadeeetttt";
    num = strlen(a);

    printf("排序之前\n");
    pirntArray<char>(a, num);

    sortArray<char, int>(a, num); //显示类型调用 模板函数 <>
    printf("排序之后\n");
    pirntArray<char>(a, num);
    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

6.1.5 函数模板遇上函数重载

函数模板和普通函数区别结论：

/*

函数模板不允许自动类型转化

普通函数能够进行自动类型转换

*/

函数模板和普通函数在一起，调用规则：

/*

1 函数模板可以像普通函数一样被重载

2 C++编译器优先考虑普通函数

3 如果函数模板可以产生一个更好的匹配，那么选择模板

4 可以通过空模板实参列表的语法限定编译器只通过模板匹配

*/

案例 1:

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

template <typename T>
void myswap(T &a, T &b)
{
    T t;
    t = a;
    a = b;
    b = t;
    cout<<"myswap 模板函数 do"<<endl;
}

void myswap(char &a, int &b)
{
    int t;
    t = a;
    a = b;
    b = t;
    cout<<"myswap 普通函数 do"<<endl;
}

void main()
{
    char cData = 'a';
    int iData = 2;

    //myswap<int>(cData, iData); //结论 函数模板不提供隐式的数据类型转换 必须是严格的匹配

    myswap(cData, iData);
    //myswap(iData, cData);

    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

案例 2:

```
#include "iostream"
using namespace std;
```

```
int Max(int a, int b)
{
    cout<<"int Max(int a, int b)"<<endl;
    return a > b ? a : b;
}

template<typename T>
T Max(T a, T b)
{
    cout<<"T Max(T a, T b)"<<endl;
    return a > b ? a : b;
}

template<typename T>
T Max(T a, T b, T c)
{
    cout<<"T Max(T a, T b, T c)"<<endl;
    return Max(Max(a, b), c);
}

void main()
{
    int a = 1;
    int b = 2;

    cout<<Max(a, b)<<endl; //当函数模板和普通函数都符合调用时,优先选择普通函数
    cout<<Max<>(a, b)<<endl; //若显示使用函数模板,则使用<> 类型列表

    cout<<Max(3.0, 4.0)<<endl; //如果 函数模板产生更好的匹配 使用函数模板

    cout<<Max(5.0, 6.0, 7.0)<<endl; //重载

    cout<<Max('a', 100)<<endl; //调用普通函数 可以隐式类型转换
    system("pause");
    return ;
}
```

6.1.6 C++编译器模板机制剖析

思考：为什么函数模板可以和函数重载放在一块。C++编译器是如何提供函数模板机制的？

编译器编译原理

什么是 gcc

| |
|---|
| gcc (GNU C Compiler) 编译器的作者是 Richard Stallman, 也是 GNU 项目的奠基者。 |
| 什么是 gcc: gcc 是 GNU Compiler Collection 的缩写。最初是作为 C 语言的编译器 (GNU C Compiler), 现在已经支持多种语言了, 如 C、C++、Java、Pascal、Ada、COBOL 语言等。 |
| gcc 支持多种硬件平台, 甚至对 Don Knuth 设计的 MMIX 这类不常见的计算机都提供了完善的支持 |

gcc 主要特征

- 1) gcc 是一个可移植的编译器, 支持多种硬件平台
- 2) gcc 不仅仅是个本地编译器, 它还能跨平台交叉编译。
- 3) gcc 有多种语言前端, 用于解析不同的语言。
- 4) gcc 是按模块化设计的, 可以加入新语言和新 CPU 架构的支持
- 5) gcc 是自由软件

gcc 编译过程

| | |
|--|--|
| 预处理 (Pre-Processing) | |
| 编译 (Compiling) | |
| 汇编 (Assembling) | |
| 链接 (Linking) | |
| Gcc *.c -o lexe (总的编译步骤) | |
| Gcc -E 1.c -o 1.i //宏定义 宏展开 | |
| Gcc -S 1.i -o 1.s | |
| Gcc -c 1.s -o 1.o | |
| Gcc 1.o -o lexe | |
| 结论: gcc 编译工具是一个工具链。。。。 | |
| <pre>graph LR A[hello.c 源程序 (文本)] --> B[预处理器 (cpp)] B -- "hello.i 被修改的 源程序 (文本)" --> C[编译器 (cc1)] C -- "hello.s 汇编程序 (文本)" --> D[汇编器 (as)] D -- "hello.o 可重定位 目标程序 (二进制)" --> E[链接器 (ld)] F[printf.o] --> E E -- "hello 可执行 目标程序 (二进制)" --> G[]</pre> | |
| hello 程序是一个高级 C 语言程序, 这种形式容易被人读懂。为了在系统上运行 hello.c 程序, 每条 C 语句都必须转化为低级机器指令。然后将这些指令打包成可执行目标文件格式, 并以二进制形式存储于磁盘中。 | |

gcc 常用编译选项

| 选项 | 作用 |
|----|-------------------------------------|
| -o | 产生目标 (.i、.s、.o、可执行文件等) |
| -c | 通知 gcc 取消链接步骤, 即编译源码并在最后生成目标文件 |
| -E | 只运行 C 预编译器 |
| -S | 告诉编译器产生汇编语言文件后停止编译, 产生的汇编语言文件扩展名为.s |

| | |
|-------|-------------------------------|
| -Wall | 使 gcc 对源文件的代码有问题的地方发出警告 |
| -ldir | 将 dir 目录加入搜索头文件的目录路径 |
| -Ldir | 将 dir 目录加入搜索库的目录路径 |
| -llib | 链接 lib 库 |
| -g | 在目标文件中嵌入调试信息，以便 gdb 之类的调试程序调试 |

练习

| |
|---|
| <pre>gcc -E hello.c -o hello.i (预处理) gcc -S hello.i -o hello.s (编译) gcc -c hello.s -o hello.o (汇编) gcc hello.o -o hello (链接) 以上四个步骤，可合成一个步骤 gcc hello.c -o hello (直接编译链接成可执行目标文件) gcc -c hello.c 或 gcc -c hello.c -o hello.o (编译生成可重定位目标文件)</pre> |
| <p>建议初学都加这个选项。下面这个例子如果不加-Wall 选项编译器不报任何错误，但是得到的结果却不是预期的。</p> <pre>#include <stdio.h> int main(void) { printf("2+1 is %f", 3); return 0; }</pre> |
| Gcc 编译多个.c |
| <pre>hello_1.h hello_1.c main.c 一次性编译 gcc hello_1.c main.c -o newhello 独立编译 gcc -Wall -c main.c -o main.o gcc -Wall -c hello_1.c -o hello_fn.o gcc -Wall main.o hello_1.o -o newhello</pre> |

模板函数反汇编观察

命令：g++ -S 7.cpp -o 7.s

| |
|---|
| <pre>.file "7.cpp" .text .def __ZL6printfPKcz; .scl 3; .type 32; .endef __ZL6printfPKcz: LFB264: .cfi_startproc</pre> |
|---|

```
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    pushl    %ebx
    subl $36, %esp
    .cfi_offset 3, -12
    leal 12(%ebp), %eax
    movl     %eax, -12(%ebp)
    movl     -12(%ebp), %eax
    movl     %eax, 4(%esp)
    movl     8(%ebp), %eax
    movl     %eax, (%esp)
    call     __mingw_vprintf
    movl     %eax, %ebx
    movl     %ebx, %eax
    addl $36, %esp
    popl %ebx
    .cfi_restore 3
    popl %ebp
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc

LFE264:
.lcomm __ZStL8__joinit,1,1
    .def __main; .scl 2; .type 32; .endef
    .section .rdata,"dr"

LC0:
    .ascii "a:%d b:%d \12\0"

LC1:
    .ascii "c1:%c c2:%c \12\0"

LC2:
    .ascii "pause\0"
    .text
    .globl __main
    .def _main; .scl 2; .type 32; .endef
_main:
LFB1023:
    .cfi_startproc
    .cfi_personality 0, __gxx_personality_v0
    .cfi_lsda 0, LLSDA1023
    pushl    %ebp
```

```
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl    %esp, %ebp
.cfi_def_cfa_register 5
andl    $-16, %esp
subl    $32, %esp
call    __main
movl    $0, 28(%esp)
movl    $10, 24(%esp)
movb    $97, 23(%esp)
movb    $98, 22(%esp)
leal    24(%esp), %eax
movl    %eax, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)
call    __Z6myswapliEvRT_S1_ //66 ==>126
movl    24(%esp), %edx
movl    28(%esp), %eax
movl    %edx, 8(%esp)
movl    %eax, 4(%esp)
movl    $LC0, (%esp)
call    __ZL6printfPKcz
leal    22(%esp), %eax
movl    %eax, 4(%esp)
leal    23(%esp), %eax
movl    %eax, (%esp)
call    __Z6myswaplciEvRT_S1_ //77 ==>155
movzbl    22(%esp), %eax
movsbl    %al, %edx
movzbl    23(%esp), %eax
movsbl    %al, %eax
movl    %edx, 8(%esp)
movl    %eax, 4(%esp)
movl    $LC1, (%esp)
call    __ZL6printfPKcz
movl    $LC2, (%esp)
LEHB0:
    call    _system
LEHE0:
    movl    $0, %eax
    jmp     L7
L6:
    movl    %eax, (%esp)
LEHB1:
```

```
    call __Unwind_Resume
LEHE1:
L7:
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE1023:
    .def __gxx_personality_v0; .scl 2; .type 32; .endef
    .section .gcc_except_table,"w"
LLSDA1023:
    .byte 0xff
    .byte 0xff
    .byte 0x1
    .uleb128 LLSDACSE1023-LLSDACSB1023
LLSDACSB1023:
    .uleb128 LEHB0-LFB1023
    .uleb128 LEHE0-LEHB0
    .uleb128 L6-LFB1023
    .uleb128 0
    .uleb128 LEHB1-LFB1023
    .uleb128 LEHE1-LEHB1
    .uleb128 0
    .uleb128 0
LLSDACSE1023:
    .text
    .section .text$__Z6myswapliEvRT_S1_"x"
    .linkonce discard
    .globl __Z6myswapliEvRT_S1_
    .def __Z6myswapliEvRT_S1_; .scl 2; .type 32; .endef
__Z6myswapliEvRT_S1_: //126
LFB1024:
    .cfi_startproc
    pushl %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl %esp, %ebp
    .cfi_def_cfa_register 5
    subl $16, %esp
    movl 8(%ebp), %eax
    movl (%eax), %eax
    movl %eax, -4(%ebp)
    movl 12(%ebp), %eax
```

```
    movl    (%eax), %edx
    movl    8(%ebp), %eax
    movl    %edx, (%eax)
    movl    12(%ebp), %eax
    movl    -4(%ebp), %edx
    movl    %edx, (%eax)
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE1024:
    .section .text$__Z6myswaplcevRT_S1_"x"
    .linkonce discard
    .globl  __Z6myswaplcevRT_S1_
    .def __Z6myswaplcevRT_S1_; .scl 2; .type 32; .endef
__Z6myswaplcevRT_S1_://155
LFB1025:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    subl    $16, %esp
    movl    8(%ebp), %eax
    movzbl  (%eax), %eax
    movb    %al, -1(%ebp)
    movl    12(%ebp), %eax
    movzbl  (%eax), %edx
    movl    8(%ebp), %eax
    movb    %dl, (%eax)
    movl    12(%ebp), %eax
    movzbl  -1(%ebp), %edx
    movb    %dl, (%eax)
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE1025:
    .text
    .def __tcf_0; .scl 3; .type 32; .endef
__tcf_0:
```

LFB1027:

```
.cfi_startproc
pushl    %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl     %esp, %ebp
.cfi_def_cfa_register 5
subl     $8, %esp
movl     $__ZStL8__ioinit, %ecx
call     __ZNSt8ios_base4InitD1Ev
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
```

LFE1027:

```
.def __Z41__static_initialization_and_destruction_0ii; .scl 3; .type 32; .endif
__Z41__static_initialization_and_destruction_0ii:
```

LFB1026:

```
.cfi_startproc
pushl    %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl     %esp, %ebp
.cfi_def_cfa_register 5
subl     $24, %esp
cmpl     $1, 8(%ebp)
jne      L11
cmpl     $65535, 12(%ebp)
jne      L11
movl     $__ZStL8__ioinit, %ecx
call     __ZNSt8ios_base4InitC1Ev
movl     $__tcf_0, (%esp)
call     _atexit
```

L11:

```
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
```

LFE1026:

```
.def __GLOBAL__sub_I_main; .scl 3; .type 32; .endif
__GLOBAL__sub_I_main:
```

LFB1028:

```
.cfi_startproc
pushl    %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl     %esp, %ebp
.cfi_def_cfa_register 5
subl     $24, %esp
movl     $65535, 4(%esp)
movl     $1, (%esp)
call     __Z41__static_initialization_and_destruction_0ii
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc

LFE1028:
.section .ctors,"w"
.align 4
.long    __GLOBAL__sub_I_main
.ident   "GCC: (rev2, Built by MinGW-builds project) 4.8.0"
.def     __mingw_vprintf; .scl 2; .type 32; .endef
.def     _system; .scl 2; .type 32; .endef
.def     __Unwind_Resume; .scl 2; .type 32; .endef
.def     __ZNSt8ios_base4InitD1Ev; .scl 2; .type 32; .endef
.def     __ZNSt8ios_base4InitC1Ev; .scl 2; .type 32; .endef
.def     _atexit; .scl 2; .type 32; .endef
```

函数模板机制结论

编译器并不是把函数模板处理成能够处理任意类的函数

编译器从函数模板通过具体类型产生不同的函数

编译器会对函数模板进行**两次编译**

在声明的地方对模板代码本身进行编译；在调用的地方对参数替换后的代码进行编译。

6.2 类模板

6.2.1 为什么需要类模板

类模板与函数模板的定义和使用类似，我们已经进行了介绍。有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同，如下面语句声明了一个类：

类模板由模板说明和类说明构成

template <类型形式参数表>
类声明

例如

```
template< typename Type >
class TClass
{ // TClass的成员函数
private :
    Type DateMember;
    //...
};
```

类属参数必须至少在类说明中出现一次

- 类模板用于实现类所需数据的类型参数化
- 类模板在表示如数组、表、图等数据结构显得特别重要，这些数据结构的表示和算法不受所包含的元素类型的影响

6.2.2 单个类模板语法

//类的类型参数化 抽象的类

//单个类模板

```
template<typename T>
```

```
class A
```

```
{
```

```
public:
```

```
    A(T t)
```

```
    {
```

```
        this->t = t;
```

```
    }
```

```
    T &getT()
```

```
    {
```

```
        return t;
```

```
    }
```

```
protected:
```

```
public:
```

```
    T t;
```

```
};
```

```
void main()
```



```

{
    //模板了中如果使用了构造函数,则遵守以前的类的构造函数的调用规则
    A<int> a(100);
    a.getT();
    printAA(a);
    return ;
}

```

6.2.3 继承中的类模板语法

从类模板A派生普通类B

```

#include<iostream.h>
template< typename T >    //定义类模板
class A
{ public :
    A( T x ) { t = x ; }
    void out() { cout << t << endl ; }
protected :
    T t ;
};

```

实例化基类
抽象类型参数

```

void main()
{ A <int> a( 123 ) ;
  a.out() ;
  B b ( 789, 5.16 ) ;
  b.out() ;
}

```

```

class B: public A<int>    //派生一般类
{ public :
    B ( int a, double x ) : A <int> ( a ) { y = x ; }
    void out() { A <int> :: out() ; cout << y << endl ; }
protected :
    double y ;
};

```

//结论: 子类从模板类继承的时候,需要让编译器知道 父类的数据类型具体是什么(数据类型的本质:固定大小内存块的别名)A<int>

```

//
class B : public A<int>
{
public:
    B(int i) : A<int>(i)
    {

    }

    void printB()
    {
        cout<<"A:"<<t<<endl;
    }
}

```

```
    }
protected:
private:
};

//模板与上继承
//怎么样从基类继承
//若基类只有一个带参数的构造函数,子类是如何启动父类的构造函数
void printBB(B &b)
{
    b.printB();
}
void printAA(A<int> &a)  //类模板做函数参数
{
    //
    a.getT();
}

void main()
{
    A<int>  a(100); //模板了中如果使用了构造函数,则遵守以前的类的构造函数的调用规则
    a.getT();
    printAA(a);

    B b(10);
    b.printB();

    cout<<"hello..."<<endl;
    system("pause");
    return ;
}
```

6.2.4 类模板语法知识体系梳理

6.2.4.1 所有的类模板函数写在类的内部

6.2.4.2 所有的类模板函数写在类的外部，在一个 `cpp` 中

```
//构造函数 没有问题
//普通函数 没有问题
//友元函数：用友元函数重载 <<>>
// friend ostream& operator<< <T> (ostream &out, Complex<T> &c3);
//友元函数：友元函数不是实现函数重载（非 <<>>）
//1) 需要在类前增加 类的前置声明 函数的前置声明
template<typename T>
class Complex;
template<typename T>
Complex<T> mySub(Complex<T> &c1, Complex<T> &c2);

//2) 类的内部声明 必须写成：
friend Complex<T> mySub<T> (Complex<T> &c1, Complex<T> &c2);
//3) 友元函数实现 必须写成：
template<typename T>
Complex<T> mySub(Complex<T> &c1, Complex<T> &c2)
{
    Complex<T> tmp(c1.a - c2.a, c1.b-c2.b);
    return tmp;
}
//4) 友元函数调用 必须写成
Complex<int> c4 = mySub<int>(c1, c2);
cout<<c4;
```

结论：友元函数只用来进行 左移 友移操作符重载。

6.2.4.3 所有的类模板函数写在类的外部，在不同的.h 和.cpp 中，

也就是类模板函数说明和类模板实现分开

```
//类模板函数
构造函数
普通成员函数
友元函数
    用友元函数重载<<>>;
    用友元函数重载非<<>>
//要包含.cpp
```

6.2.4.4 总结

归纳以上的介绍，可以这样声明和使用类模板：

1) 先写出一个实际的类。由于其语义明确，含义清楚，一般不会出错。

2) 将此类中准备改变的类型名(如 `int` 要改变为 `float` 或 `char`)改用一个自己指定的虚拟类型名(如上例中的 `numtype`)。

3) 在类声明前面加入一行，格式为：

```
template <class 虚拟类型参数>
```

如：

```
template <class numtype> //注意本行末尾无分号
class Compare
{...}; //类体
```

4) 用类模板定义对象时用以下形式：

```
类模板名<实际类型名> 对象名;
```

```
类模板名<实际类型名> 对象名(实参表列);
```

如：

```
Compare<int> cmp;
Compare<int> cmp(3,7);
```

5) 如果在类模板外定义成员函数，应写成类模板形式：

```
template <class 虚拟类型参数>
```

```
函数类型 类模板名<虚拟类型参数>::成员函数名(函数形参表列) {...}
```

关于类模板的几点说明：

1) 类模板的类型参数可以有一个或多个，每个类型前面都必须加 `class`，如：

```
template <class T1,class T2>
class someclass
{...};
```

在定义对象时分别代入实际的类型名，如：

```
someclass<int,double> obj;
```

2) 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。

3) 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

6.2.5 类模板中的 `static` 关键字

- 从类模板实例化的每个模板类有自己的类模板数据成员，该模板类的所有对象共享一个 `static` 数据成员
- 和非模板类的 `static` 数据成员一样，模板类的 `static` 数据成员也应该在文件范围定义和初始化
- 每个模板类有自己的类模板的 `static` 数据成员副本

例9-6 为圆类模板定义静态成员

```
#include<iostream.h>
const double pi=3.14159;
template<typename T> class Circle
{ T radius;
    static int total; //类模板的静态数据成员
public:
    Circle(T r=0) { radius = r; total++; }
    void Set_Radius(T r) { radius = r; }
    double Get_Radius() { return radius; }
    double Get_Girth() { return 2 * pi * radius; }
    double Get_Area() { return pi * radius * radius; }
    static int ShowTotal(); //类模板的静态成员函数
};

template<typename T> int Circle<T>::total=0;
template<typename T>
int Circle<T>::ShowTotal() { return total; }
```

例9-6 为圆类模板定义静态成员

```
void main()
{ Circle<int> A, B;
    A.Set_Radius(16);
    cout << "A.Radius = " << A.Get_Radius() << endl;
    cout << "A.Girth = " << A.Get_Girth() << endl;
    cout << "A.Area = " << A.Get_Area() << endl;
    B.Set_Radius(105);
    cout << "B.radius = " << B.Get_Radius() << endl;
    cout << "B.Girth=" << B.Get_Girth() << endl;
    cout << "B.Area = " << B.Get_Area() << endl;
    cout<<"Total1="<<Circle<int>::ShowTotal()<<endl;

    cout << "X.Radius = " << X.Get_Radius() << endl;
    cout << "X.Girth = " << X.Get_Girth() << endl;
    cout << "X.Area = " << X.Get_Area() << endl;
    cout << "Y.radius = " << Y.Get_Radius() << endl;
    cout << "Y.Girth=" << Y.Get_Girth() << endl;
    cout << "Y.Area = " << Y.Get_Area() << endl;
    cout << "Z.Girth=" << Z.Get_Girth() << endl;
    cout << "Z.Area = " << Z.Get_Area() << endl;
    cout<<"Total2="<<Circle<double>::ShowTotal()<<endl;
}

//显示建立的对象数
```

A.ShowTotal()
或 B.ShowTotal()

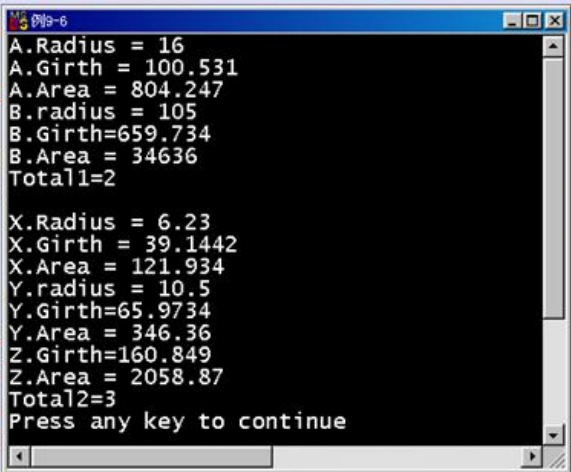
void main()
{ Circle<int> A, B; //建立了2个对象
A.Set_Radius(16);
cout << "A.Radius = " << A.Get_Radius() << endl;
cout << "A.Girth = " << A.Get_Girth() << endl;
cout << "A.Area = " << A.Get_Area() << endl;
B.Set_Radius(105);
cout << "B.radius = " << B.Get_Radius() << endl;
cout << "B.Girth=" << B.Get_Girth() << endl;
cout << "B.Area = " << B.Get_Area() << endl;
cout << "Total1=" << Circle<int>::ShowTotal() << endl; //显示建立的
Circle<double> X(6.23), Y(10.5), Z(25.6);
cout << "X.Radius = " << X.Get_Radius() << endl;
cout << "X.Girth = " << X.Get_Girth() << endl;
cout << "X.Area = " << X.Get_Area() << endl;
cout << "Y.radius = " << Y.Get_Radius() << endl;
cout << "Y.Girth=" << Y.Get_Girth() << endl;
cout << "Y.Area = " << Y.Get_Area() << endl;
cout << "Z.Girth=" << Z.Get_Girth() << endl;
cout << "Z.Area = " << Z.Get_Area() << endl;
cout << "Total2=" << Circle<double>::ShowTotal() << endl;
}

例9-6 为圆类模板定义静态成员

**X.ShowTotal()
或 Y.ShowTotal()**

void main()
{ Circle<int> A, B; //建立了2个对象
A.Set_Radius(16);
cout << "A.Radius = " << A.Get_Radius() << endl;
cout << "A.Girth = " << A.Get_Girth() << endl;
cout << "A.Area = " << A.Get_Area() << endl;
B.Set_Radius(105);
cout << "B.radius = " << B.Get_Radius() << endl;
cout << "B.Girth=" << B.Get_Girth() << endl;
cout << "B.Area = " << B.Get_Area() << endl;
cout << "Total1=" << Circle<int>::ShowTotal() << endl;
cout << endl;
Circle<double> X(6.23), Y(10.5), Z(25.6);
cout << "X.Radius = " << X.Get_Radius() << endl;
cout << "X.Girth = " << X.Get_Girth() << endl;
cout << "X.Area = " << X.Get_Area() << endl;
cout << "Y.radius = " << Y.Get_Radius() << endl;
cout << "Y.Girth=" << Y.Get_Girth() << endl;
cout << "Y.Area = " << Y.Get_Area() << endl;
cout << "Z.Girth=" << Z.Get_Girth() << endl;
cout << "Z.Area = " << Z.Get_Area() << endl;
cout << "Total2=" << Circle<double>::ShowTotal() << endl;
}

例9-6 为圆类模板定义静态成员



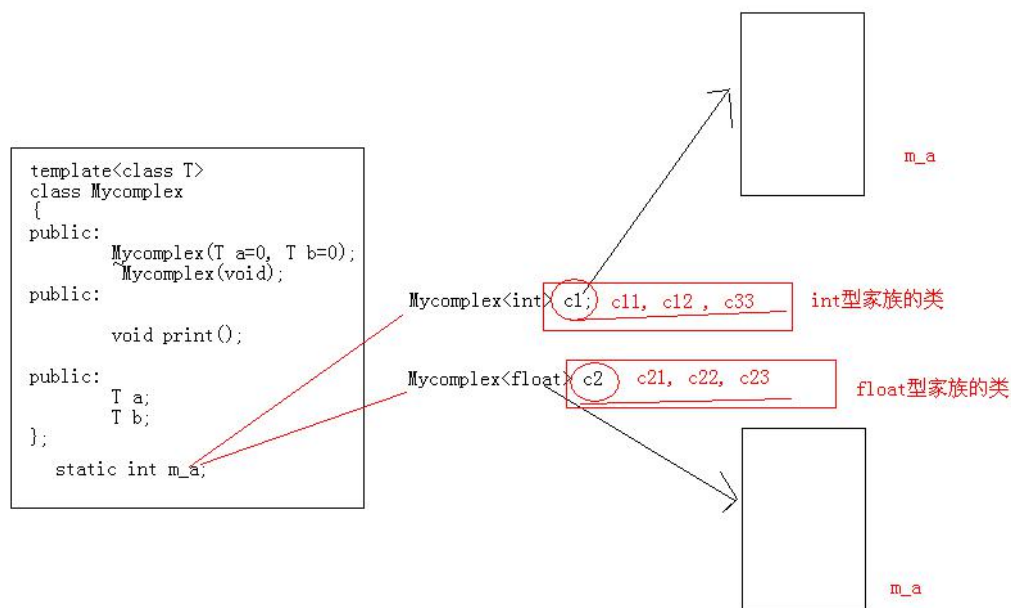
```

A.Radius = 16
A.Girth = 100.531
A.Area = 804.247
B.radius = 105
B.Girth=659.734
B.Area = 34636
Total1=2

X.Radius = 6.23
X.Girth = 39.1442
X.Area = 121.934
Y.radius = 10.5
Y.Girth=65.9734
Y.Area = 346.36
Z.Girth=160.849
Z.Area = 2058.87
Total2=3
Press any key to continue

```

原理图:



6.3 类模板在项目开发中的应用

小结

- 模板是 C++ 类型参数化的多态工具。C++ 提供函数模板和类模板。
- 模板定义以模板说明开始。类属参数必须在模板定义中至少出现一次。
- 同一个类属参数可以用于多个模板。
- 类属参数可用于函数的参数类型、返回类型和声明函数中的变量。
- 模板由编译器根据实际数据类型实例化，生成可执行代码。实例化的函数。模板称为模板函数；实例化的类模板称为模板类。
- 函数模板可以用多种方式重载。
- 类模板可以在类层次中使用。

训练题

- 1) 请设计一个数组模板类 (MyVector)，完成对 int、char、Teacher 类型元素的管理。

需求

设计:

类模板 构造函数 拷贝构造函数 <<[] 重载=操作符

a2=a1

实现

- 2) 请仔细思考:

- a) 如果数组模板类中的元素是 Teacher 元素时，需要 Teacher 类做什么工作
- b) 如果数组模板类中的元素是 Teacher 元素时，Teacher 类含有指针属性哪？

```
class Teacher
```

```
{
    friend ostream & operator<<(ostream &out, const Teacher &obj);
public:
    Teacher(char *name, int age)
    {
        this->age = age;
        strcpy(this->name, name);
    }

    Teacher()
    {
        this->age = 0;
        strcpy(this->name, "");
    }

private:
    int age;
    char name[32];
};
```

```
class Teacher
{
    friend ostream & operator<<(ostream &out, const Teacher &obj);
public:
    Teacher(char *name, int age)
    {
        this->age = age;
        strcpy(this->name, name);
    }

    Teacher()
    {
        this->age = 0;
        strcpy(this->name, "");
    }

private:
    int age;
    char *pname;
};
```

结论1: 如果把Teacher放入到MyVector数组中, 并且Teacher类的属性含有指针, 就是

出现深拷贝和浅拷贝的问题。

结论2：需要Teacher封装的函数有：

- 1) 重写拷贝构造函数
- 2) 重载等号操作符
- 3) 重载左移操作符。

理论提高：所有容器提供的都是值（value）语义，而非引用（reference）语义。**容器执行插入元素的操作时，内部实施拷贝动作。**所以 STL 容器内存储的元素必须**能够被拷贝**（必须提供拷贝构造函数）。

3) 请从数组模板中进行派生

```
//演示从模板类 派生 一般类
#include "MyVector.cpp"

class MyArray01 : public MyVector<double>
{
public:
    MyArray01(int len) : MyVector<double>(len)
    {
        ;
    }
protected:
private:
};

//演示从模板类 派生 模板类 //BoundArray
template <typename T>
class MyArray02 : public MyVector<T>
{
public:
    MyArray02(int len) : MyVector<double>(len)
    {
        ;
    }
protected:
private:
};
```

测试案例：

```
//演示 从模板类 继承 模板类
void main()
```

```
{  
    MyArray02<double> dArray2(10);  
    dArray2[1] = 3.15;  
}
```

//演示 从模板类 继承 一般类

```
void main11()  
{  
    MyArray01 d_array(10);  
  
    for (int i=0; i<d_array.getLen(); i++)  
    {  
        d_array[i] = 3.15;  
    }  
  
    for (int i=0; i<d_array.getLen(); i++)  
    {  
        cout << d_array[i] << " ";  
    }  
  
    cout<<"hello..."<<endl;  
    system("pause");  
    return ;  
}
```

6.4 作业

封装你自己的数组类：设计被存储的元素为类对象；

思考：类对象的类，应该实现的功能。

- //1 优化 Teacher 类, 属性变成 char *panme, 构造函数里面 分配内存
- //2 优化 Teacher 类,析构函数 释放 panme 指向的内存空间
- //3 优化 Teacher 类,避免浅拷贝 重载= 重写拷贝构造函数
- //4 优化 Teacher 类,在 Teacher 增加 <<
- //5 在模板数组类中,存 int char Teacher Teacher*(指针类型)
- //=====>stl 容器的概念

7、C++的类型转换

7.1 类型转换名称和语法

C 风格的强制类型转换(Type Cast)很简单，不管什么类型的转换统统是：

TYPE b = (TYPE)a

C++风格的类型转换提供了 4 种类型转换操作符来应对不同场合的应用。

`static_cast` 静态类型转换。如 `int` 转换成 `char`

`reinterpret_cast` 重新解释类型

`dynamic_cast` 命名上理解是动态类型转换。如子类 and 父类之间的多态类型转换。

`const_cast`，字面上理解就是去 `const` 属性。

4 种类型转换的格式：

TYPE B = static_cast<TYPE>(a)

7.2 类型转换一般性介绍

1) `static_cast<>()` 静态类型转换，编译的时 **c++编译器会做类型检查**；

基本类型能转换 但是不能转换指针类型

2) 若不同类型之间，进行**强制类型转换**，用 `reinterpret_cast<>()` 进行重新解释

3) 一般性结论：

C 语言中 能隐式类型转换的，在 c++中可用 `static_cast<>()`进行类型转换。因 C++ 编译器在编译检查一般都能通过；

C 语言中不能隐式类型转换的，在 c++中可以用 `reinterpret_cast<>()` 进行强行**类型解释**。总结：`static_cast<>()`和 `reinterpret_cast<>()` 基本上把 C 语言中的 强制类型转换给覆盖

`reinterpret_cast<>()`很难保证移植性。

4) `dynamic_cast<>()`，动态类型转换，安全的基类和子类之间转换；运行时类型检查

5) `const_cast<>()`，去除变量的只读属性

7.3 典型案例

7.3.1 static_cast 用法和 reinterpret_cast 用法

```
void main01()
```

```
{
```

```
    double dPi = 3.1415926;
```

```
    //1 静态的类型转换： 在编译的时 进行基本类型的转换 能替代 c 风格的类型转换 可以  
    进行一部分检查
```

```
int num1 = static_cast<int> (dPi); //c++的新式的类型转换运算符
int num2 = (int)dPi;              //c 语言的 旧式类型转换
int num3 = dPi;                  //隐士类型转换
cout << "num1:" << num1 << " num2:" << num2 << " num3:" << num3 << endl;

char *p1 = "hello wangbaoming ";
int *p2 = NULL;
p2 = (int *)p1;

//2 基本类型能转换 但是不能转换指针类型
//p2 = static_cast<int *> (p1); // “static_cast” : 无法从 “char *” 转换为 “int *”

//3 可以使用 reinterpret_cast 进行重新解释
p2 = reinterpret_cast<int *> (p1);
cout << "p1 " << p1 << endl;
cout << "p2 " << p2 << endl;

//4 一般性的结论: c 语言中 能隐式类型转换的 在 c++中可以用 static_cast<>() 进
行类型转换 //C++编译器在编译检查一般都能通过
//c 语言中不能隐式类型转换的,在 c++中可以用 reinterpret_cast<>() 进行强行类型 解
释

system("pause");
return ;
}
```

7.3.2 dynamic_cast 用法和 reinterpret_cast 用法

```
class Animal
{
public:
    virtual void cry() = 0;
};

class Dog : public Animal
{
public:
    virtual void cry()
    {
        cout << "wangwang " << endl;
    }
}
```

```
    }

    void doSwim()
    {
        cout << "我要狗爬" << endl;
    }
};

class Cat : public Animal
{
public:
    virtual void cry()
    {
        cout << "miaomiao " << endl;
    }
    void doTree()
    {
        cout << "我要爬树" << endl;
    }
};

class Book
{
public:
    void printP()
    {
        cout << price << endl;
    }

private:
    int price;
};

void ObjPlay(Animal *base)
{
    base->cry();
    Dog *pDog = dynamic_cast<Dog *>(base);
    if (pDog != NULL)
    {
        pDog->cry();
        pDog->doSwim();
    }
}
```

```
    }

    Cat *pCat = dynamic_cast<Cat *>(base);
    if (pCat != NULL)
    {
        pCat->cry();
        pCat->doTree();
    }
}

void main02()
{
    Animal *base = NULL;

    //1 可以把子类指针赋给 父类指针 但是反过来是不可以的 需要 如下转换
    //pdog = base;
    Dog *pDog = static_cast<Dog *>(base);

    //2 把 base 转换成其他 非动物相关的 err
    //Book *book= static_cast<Book *>(base);

    //3 reinterpret_cast //可以强制类型转换
    Book *book2= reinterpret_cast<Book *>(base);

    //4 dynamic_cast 用法
    ObjPlay(new Cat());

    system("pause");
}
```

7.3.3 const_cast 用法

```
//典型用法 把形参的只读属性去掉
void Opbuf(const char *p)
{
    cout << p << endl;
    char *p2 = const_cast<char*>(p);
    p2[0] = 'b';
    cout << p << endl;
}

void main()
```

```
{
    const char *p1 = "111111111111";

    char *p2 = "22222222";

    char *p3 = const_cast<char *>(p1);
    char buf[100] = "aaaaaaaaaaaa";

    Opbuf(buf);

    //要保证指针所执行的内存空间能修改才行 若不能修改 还是会引起程序异常
    //Opbuf("ddddddddddssssssssssssss");

    system("pause");
}
```

7.4 总结

结论 1: 程序员要清除的知道: 要转的变量, 类型转换前是什么类型, 类型转换后是什么类型。转换后有什么后果。

结论 2: 一般情况下, 不建议进行类型转换; 避免进行类型转换。

8、异常处理机制专题

前言

1) 异常是一种程序控制机制, 与函数机制独立和互补

函数是一种以栈结构展开的上下函数衔接的程序控制系统, 异常是另一种控制结构, 它依附于栈结构, 却可以同时设置多个异常类型作为网捕条件, 从而以类型匹配在栈机制中跳跃回馈。

2) 异常设计目的:

栈机制是一种高度节律性控制机制, 面向对象编程却要求对象之间有方向、有目的的控制传动, 从一开始, 异常就是冲着改变程序控制结构, 以适应面向对象程序更有效地工作这个主题, 而不是仅为了进行错误处理。

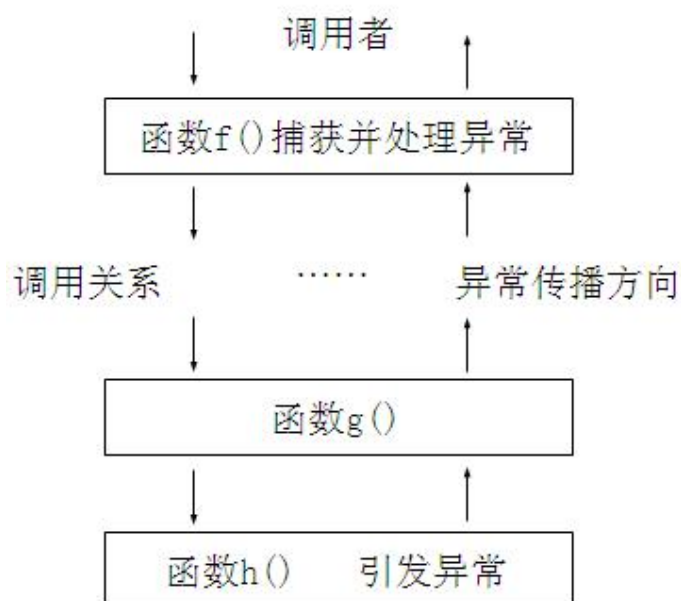
异常设计出来之后, 却发现在错误处理方面获得了最大的好处。

8.1 异常处理的基本思想

8.1.1 传统错误处理机制

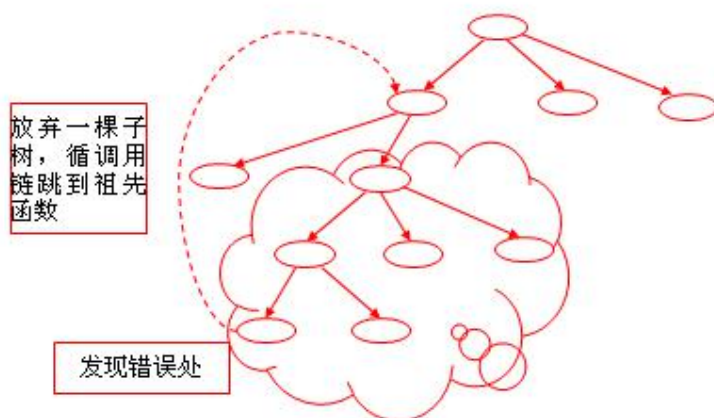
通过函数返回值来处理错误。

8.1.2 异常处理的基本思想



- 1) C++的异常处理机制使得**异常的引发**和**异常的处理**不必在同一个函数中，这样底层的函数可以着重解决具体问题，而不必过多的考虑异常的处理。上层调用者可以再适当的位置设计**对不同类型异常**的处理。
- 2) 异常是专门针对抽象编程中的一系列错误处理的，C++中不能借助函数机制，因为栈结构的本质是先进后出，依次访问，无法进行跳跃，但错误处理的特征却是遇到错误信息就想要转到若干级之上进行重新尝试，如图

❖ 错误处理示意：



- 3) 异常超脱于函数机制，决定了其对函数的跨越式回跳。
- 4) 异常跨越函数

8.2 C++异常处理的实现

8.2.1 异常基本语法

❖ 抛掷异常的程序段

```
void Fun ()
{
    .....
    throw    表达式;
    .....
}
```

❖ 捕获并处理异常的程序段

```
try {
    复合语句
}
catch (异常类型声明)
    复合语句
catch (类型    (形参))
    复合语句
...
```

- 1) 若有异常则通过 `throw` 操作 **创建一个异常对象** 并抛掷。
- 2) 将可能抛出异常的程序段嵌在 `try` 块之中。控制通过正常的顺序执行到达 `try` 语句，然后执行 `try` 块内的保护段。
- 3) 如果在保护段执行期间没有引起异常，那么跟在 `try` 块后的 `catch` 子句就不执行。程序从 `try` 块后跟随的最后一个 `catch` 子句后面的语句继续执行下去。
- 4) `catch` 子句按其在 `try` 块后出现的顺序被检查。匹配的 `catch` 子句将捕获并处理异常（或继续抛掷异常）。
- 5) 如果匹配的处理器未找到，则运行函数 `terminate` 将被自动调用，其缺省功能是调用 `abort` 终止程序。
- 6) 处理不了的异常，可以在 `catch` 的最后一个分支，使用 `throw` 语法，向上扔。

案例 1：被零整除案例

```
int divide(int x, int y)
{
    if (y == 0)
    {
        throw x;
    }
    return x/y;
}

void main41()
{
    try
    {
```

```
        cout << "8/2 = " << divide(8, 2) << endl;
        cout << "10/0 =" << divide(10, 0) << endl;
    }
    catch (int e)
    {
        cout << "e" << " is divided by zero!" << endl;
    }
    catch(...)
    {
        cout << "未知异常" << endl;
    }

    cout << "ok" << endl;
    system("pause");
    return ;
}
```

案例 2:

```
class A{};
void f(){
    if(...) throw A;
}
void g(){
    try{
        f();
    }catch(B){
        cout<<"exception B\n";
    }
}
int main(){
    g();
}
```

throw A 将穿透函数 f, g 和 main, 抵达系统的最后一道防线——激发 terminate 函数。

该函数调用引起运行终止的 abort 函数。

最后一道防线的函数可以由程序员设置。从而规定其终止前的行为。

修改系统默认行为:

- ◆ 可以通过 set_terminate 函数修改捕捉不住异常的默认处理器, 从而使得发生捉不住异常时, 被自定义函数处理:
- ◆ void myTerminate(){cout<<"HereIsMyTerminate\n";}
- ◆ set_terminate(myTerminate);
- ◆ set_terminate 函数在头文件 exception 中声明, 参数为函数指针 void(*)().

案例 3:

- ❖ 构造函数没有返回类型, 无法通过返回值来报告运行状态, 所以只通过一种非函数

机制的途径，即异常机制，来解决构造函数的出错问题。

7) 异常机制与函数机制互不干涉，但捕捉的方式是**基于类型匹配**。捕捉相当于函数返回**类型的匹配**，而不是**函数参数的匹配**，所以捕捉不用考虑一个抛掷中的多种数据类型匹配问题比如：

```
class A{};
class B{};

int main()
{
    try
    {
        int j = 0;
        double d = 2.3;
        char str[20] = "Hello";
        cout<<"Please input a exception number: ";
        int a;
        cin>>a;
        switch(a)
        {
            case 1:
                throw d;
            case 2:
                throw j;
            case 3:
                throw str;
            case 4:
                throw A();
            case 5:
                throw B();
            default:
                cout<<"No throws here.\n";
        }
    }
    catch(int)
    {
        cout<<"int exception.\n";
    }
    catch(double)
    {
        cout<<"double exception.\n";
    }
    catch(char*)
    {

```

```
        cout<<"char* exception.\n";
    }
    catch(A)
    {
        cout<<"class A exception.\n";
    }
    catch(B)
    {
        cout<<"class B exception.\n";
    }
    cout<<"That's ok.\n";
    system("pause");
} //=====
```

catch代码块必须出现在try后，并且在try块后可以出现多个catch代码块，以捕捉各种不同类型的抛掷。

异常机制是基于这样的原理：程序运行实质上是数据实体在做一些操作，因此发生异常现象的地方，一定是某个实体出了差错，该实体所对应的**数据类型便作为抛掷和捕捉的依据**。

8) 异常捕捉严格按照类型匹配

- ◆ 异常捕捉的类型匹配之苛刻程度可以和模板的类型匹配媲美,它不允许相容类型的隐式转换,比如,抛掷 char 类型用 int 型就捕捉不到。例如下列代码不会输出 “int exception.”，从而也不会输出 “That’s ok.” 因为出现异常后提示退出

```
int main(){
    try{
        throw 'H' ;
    }catch(int){
        cout<<"int exception.\n";
    }
    cout<<"That's ok.\n";
}
```

8.2.2 栈解旋(unwinding)

异常被抛出后，从进入 try 块起，到异常被抛掷前，这期间在栈上的构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反。这一过程称为栈的解旋(unwinding)。

```
class MyException {};
```



```
class Test
{
public:
```

```
Test(int a=0, int b=0)
{
    this->a = a;
    this->b = b;
    cout << "Test 构造函数执行" << "a:" << a << " b: " << b << endl;
}
void printT()
{
    cout << "a:" << a << " b: " << b << endl;
}
~Test()
{
    cout << "Test 析构函数执行" << "a:" << a << " b: " << b << endl;
}
private:
    int a;
    int b;
};

void myFunc() throw (MyException)
{
    Test t1;
    Test t2;

    cout << "定义了两个栈变量,异常抛出后测试栈变量的如何被析构" << endl;

    throw MyException();
}

void main()
{
    //异常被抛出后,从进入 try 块起,到异常被抛掷前,这期间在栈上的构造的所有对象,
    //都会被自动析构。析构的顺序与构造的顺序相反。
    //这一过程称为栈的解旋(unwinding)
    try
    {
        myFunc();
    }
    //catch(MyException &e) //这里不能访问异常对象
    catch(MyException ) //这里不能访问异常对象
    {
        cout << "接收到 MyException 类型异常" << endl;
    }
    catch(...)
```

```
{
    cout << "未知类型异常" << endl;
}

system("pause");
return ;
}
```

8.2.3 异常接口声明

- 1) 为了加强程序的可读性，可以在函数声明中列出可能抛出的所有异常类型，例如：
`void func() throw (A, B, C, D);` //这个函数 func（）能够且只能抛出类型 A B C D 及其子类型的异常。
- 2) 如果在函数声明中没有包含异常接口声明，则次函数可以抛掷任何类型的异常，例如：
`void func();`
- 3) 一个不抛掷任何类型异常的函数可以声明为：
`void func() throw();`
- 4) 如果一个函数抛出了它的异常接口声明所不允许抛出的异常，unexpected 函数会被调用，该函数默认行为调用 terminate 函数中止程序。

8.2.4 异常类型和异常变量的生命周期

- 1) throw 的异常是有类型的，可以使，数字、字符串、类对象。
- 2) throw 的异常是有类型的，catch 严格按照类型进行匹配。
- 3) 注意 异常对象的内存模型 。

8.2.2.1 传统处理错误

```
//文件的二进制 copy
int filecopy01(char *filename2, char *filename1 )
{
    FILE *fp1= NULL, *fp2 = NULL;

    fp1 = fopen(filename1, "rb");
    if (fp1 == NULL)
    {
        return 1;
    }

    fp2 = fopen(filename2, "wb");
    if (fp1 == NULL)
```

```
{
    return 2;
}

char buf[256];
int readlen, writelen;
while ( (readlen = fread(buf, 1, 256, fp1)) > 0 ) //如果读到数据，则大于 0
{
    writelen = fwrite(buf, 1, readlen, fp2);
    if (readlen != writelen)
    {
        return 3;
    }
}

fclose(fp1);
fclose(fp2);
return 0;
}
```

测试程序

```
void main11()
{
    int ret;
    ret = filecopy01("c:/1.txt", "c:/2.txt");
    if (ret != 0)
    {
        switch(ret)
        {
            case 1:
                printf("打开源文件时出错!\n");
                break;
            case 2:
                printf("打开目标文件时出错!\n");
                break;
            case 3:
                printf("拷贝文件时出错!\n");
                break;
            default:
                printf("发生未知错误!\n");
                break;
        }
    }
}
```

8.2.2.2 throw int 类型异常

```
/文件的二进制 copy
void filecopy02(char *filename2, char *filename1 )
{
    FILE *fp1= NULL,  *fp2 = NULL;

    fp1 = fopen(filename1, "rb");
    if (fp1 == NULL)
    {
        //return 1;
        throw 1;
    }

    fp2 = fopen(filename2, "wb");
    if (fp1 == NULL)
    {
        //return 2;
        throw 2;
    }

    char buf[256];
    int  readlen, writelen;
    while ( (readlen = fread(buf, 1, 256, fp1)) > 0 ) //如果读到数据，则大于 0
    {
        writelen = fwrite(buf, 1, readlen, fp2);
        if (readlen != writelen)
        {
            //return 3;
            throw 3;
        }
    }

    fclose(fp1);
    fclose(fp2);
    return ;
}
```

8.2.2.3 throw 字符类型异常

```
//文件的二进制 copy
void filecopy03(char *filename2, char *filename1 )
```



```
{
    FILE *fp1= NULL,  *fp2 = NULL;

    fp1 = fopen(filename1, "rb");
    if (fp1 == NULL)
    {
        throw "打开源文件时出错";
    }

    fp2 = fopen(filename2, "wb");
    if (fp1 == NULL)
    {
        throw "打开目标文件时出错";
    }

    char buf[256];
    int  readlen, writelen;
    while ( (readlen = fread(buf, 1, 256, fp1)) > 0 ) //如果读到数据，则大于 0
    {
        writelen = fwrite(buf, 1, readlen, fp2);
        if (readlen != writelen)
        {
            throw "拷贝文件过程中失败";
        }
    }

    fclose(fp1);
    fclose(fp2);
    return ;
}
```

8.2.2.4 throw 类对象类型异常

```
//throw int 类型变量
//throw 字符串类型
//throw 类类型
class BadSrcFile
{
public:
    BadSrcFile()
    {
        cout << "BadSrcFile 构造 do "<<endl;
```

```
}
~BadSrcFile()
{
    cout << "BadSrcFile 析构 do "<<endl;
}
BadSrcFile(BadSrcFile & obj)
{
    cout << "拷贝构造 do "<<endl;
}
void toString()
{
    cout << "aaaa" << endl;
}

};
class BadDestFile {};
class BadCpyFile {};

void filecopy04(char *filename2, char *filename1 )
{
    FILE *fp1= NULL, *fp2 = NULL;

    fp1 = fopen(filename1, "rb");
    if (fp1 == NULL)
    {
        //throw new BadSrcFile();
        throw BadSrcFile();
    }

    fp2 = fopen(filename2, "wb");
    if (fp1 == NULL)
    {
        throw BadDestFile();
    }

    char buf[256];
    int readlen, writelen;
    while ( (readlen = fread(buf, 1, 256, fp1)) > 0 ) //如果读到数据，则大于 0
    {
        writelen = fwrite(buf, 1, readlen, fp2);
        if (readlen != writelen)
        {
            throw BadCpyFile();
        }
    }
}
```

```
    }

    fclose(fp1);
    fclose(fp2);
    return ;
}
```

main 测试案例

//结论: //C++编译器通过 **throw** 来产生对象, C++编译器再执行对应的 **catch** 分支, 相当于一个函数调用, 把实参传递给形参。

```
void main11()
{
    try
    {
        //filecopy02("c:/1.txt","c:/2.txt");
        // filecopy03("c:/1.txt","c:/2.txt");
        filecopy04("c:/1.txt","c:/2.txt");
    }
    catch (int e)
    {
        printf("发生异常: %d \n", e);
    }
    catch (const char * e)
    {
        printf("发生异常: %s \n", e);
    }
    catch ( BadSrcFile *e)
    {
        e->toString();
        printf("发生异常: 打开源文件时出错!\n");
    }
    catch ( BadSrcFile &e)
    {
        e.toString();
        printf("发生异常: 打开源文件时出错!\n");
    }
    catch ( BadDestFile e)
    {
        printf("发生异常: 打开目标文件时出错!\n");
    }
    catch ( BadCpyFile e)
    {
        printf("发生异常: copy 时出错!\n");
    }
}
```

```
catch(...) //抓漏网之鱼
{
    printf("发生了未知异常! 抓漏网之鱼\n");
}
//class BadSrcFile {};
//class BadDestFile {};
//class BadCpyFile {};
```

8.2.5 异常的层次结构(继承在异常中的应用)

- ❖ 异常是类 – 创建自己的异常类
- ❖ 异常派生
- ❖ 异常中的数据：数据成员
- ❖ 按引用传递异常
 - 在异常中使用虚函数

案例：设计一个数组类 `MyArray`，重载 `[]` 操作，
数组初始化时，对数组的个数进行有效检查

- 1) `index < 0` 抛出异常 `eNegative`
- 2) `index = 0` 抛出异常 `eZero`
- 3) `index > 1000` 抛出异常 `eTooBig`
- 4) `index < 10` 抛出异常 `eTooSmall`
- 5) `eSize` 类是以上类的父类，实现有参数构造、并定义 `virtual void printErr()` 输出错误。

8.3 标准程序库异常

C++ 标准提供了一组标准异常类，这些类以基类 `Exception` 开始，标准程序库抛出的所有异常，都派生于该基类，这些类构成如图 12-2 所示的异常类的派生继承关系。该基类提供一个成员函数 `what()`，用于返回错误信息（返回类型为 `const char *`）。在 `Exception` 类中，`what()` 函数的声明如下：

```
virtual const char* what() const throw();
```

该函数可以在派生类中重定义。

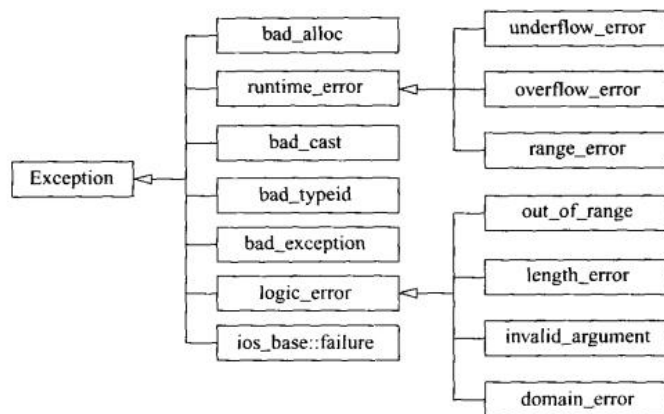


图 12-2 标准异常类的继承关系

表 12-1 列出了各个具体异常类的含义及定义它们的头文件。runtime_error 和 logic_error 是一些具体的异常类的基类，它们分别表示两大类异常。logic_error 表示那些可以在程序中被预先检测到的异常，也就是说如果小心地编写程序，这类异常能够避免；而 runtime_error 则表示那些难以被预先检测的异常。

C++ 标准库各种异常类所代表的异常

| 异常类 | 头文件 | 异常的含义 |
|-------------------|-----------|--|
| bad_alloc | exception | 用 new 动态分配空间失败 |
| bad_cast | new | 执行 dynamic_cast 失败(dynamic_cast 参见 8.7.2 节) |
| bad_typeid | typeinfo | 对某个空指针 p 执行 typeid(*p)(typeid 参见 8.7.2 节) |
| bad_exception | typeinfo | 当某个函数 fun()因在执行过程中抛出了异常声明所不允许的异常而调用 unexpected()函数时,若 unexpected()函数又一次抛出了 fun()的异常声明所不允许的异常,且 fun()的异常声明列表中有 bad_exception,则会有一个 bad_exception 异常在 fun()的调用点被抛出 |
| ios_base::failure | ios | 用来表示 C++ 的输入输出流执行过程中发生的错误 |
| underflow_error | stdexcept | 算术运算时向下溢出 |
| overflow_error | stdexcept | 算术运算时向上溢出 |
| range_error | stdexcept | 内部计算时发生作用域的错误 |
| out_of_range | stdexcept | 表示一个参数值不在允许的范围之内 |
| length_error | stdexcept | 尝试创建一个长度超过最大允许值的对象 |
| invalid_argument | stdexcept | 表示向函数传入无效参数 |
| domain_error | stdexcept | 执行一段程序所需要的先决条件不满足 |

习惯 一些编程语言规定只能抛掷某个类的派生类(例如 Java 中允许抛掷的类必须派生自 Exception 类),C++ 虽然没有这项强制的要求,但仍然可以这样实践。例如,在程序中可以使得所有抛出的异常皆派生自 Exception(或者直接抛出标准程序库提供的异常类型,或者从标准程序库提供的异常类派生出新的类),这样会带来很多方便。

logic_error 和 runtime_error 两个类及其派生类,都有一个接收 const string & 型参数的构造函数。在构造异常对象时需要将具体的错误信息传递给该函数,如果调用该对象的 what 函数,就可以得到构造时提供的错误信息。

案例 1:

```
// out_of_range
#include "iostream"
using namespace std;
#include <stdexcept>

class Teacher
{
public:
    Teacher(int age) //构造函数, 通过异常机制 处理错误
    {
        if (age > 100)
        {
            throw out_of_range("年龄太大");
        }
        this->age = age;
    }
protected:
private:
    int age;
};

void mainxx()
{
    try
    {
        Teacher t1(102);
    }
    catch (out_of_range e)
    {
        cout << e.what() << endl;
    }

    exception e;
    system("pause");
}
```

案例 2

```
class Dog
{
public:
```

```
Dog()
{
    parr = new int[1024*1024*100]; //4MB
}
private:
    int *parr;
};

int main31()
{
    Dog *pDog;
    try{
        for(int i=1; i<1024; i++) //40GB!
        {
            pDog = new Dog();
            cout << i << ": new Dog 成功." << endl;
        }
    }
    catch(bad_alloc err)
    {
        cout << "new Dog 失败: " << err.what() << endl;
    }

    return 0;
}
```

案例 3

例 12-3 三角形面积计算。

编写一个计算三角形面积的函数,函数的参数为三角形三边边长 a, b, c , 可以用 Heron 公式计算:

设 $p = \frac{a+b+c}{2}$, 则三角形面积 $S = \sqrt{p(p-a)(p-b)(p-c)}$

在计算三角形面积的函数中需要判断输入的参数 a, b, c 是否构成一个三角形, 若三个边长不能构成三角形, 则需要抛出异常。下面是源程序:

```
//12_3.cpp
#include<iostream>
#include<cmath>
#include<stdexcept>
using namespace std;

//给出三角形三边长,计算三角形面积
double area(double a, double b, double c) throw (invalid_argument) {
    //判断三角形边长是否为正
    if (a<=0||b<=0||c<=0)
        throw invalid_argument("the side length should be positive");
    //判断三边长是否满足三角不等式
    if (a+b<=c||b+c<=a||c+a<=b)
        throw invalid_argument("the side length should fit the triangle inequation");
    //由 Heron 公式计算三角形面积

    double s=(a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

int main() {
    double a, b, c;                //三角形三边长
    cout<<"Please input the side lengths of a triangle: ";
    cin>>a>>b>>c;
    try {
        double s=area(a, b, c);    //尝试计算三角形面积
```



```
        cout<<"Area: "<<s<<endl;
    } catch (exception &e) {
        cout<<"Error: "<<e.what()<<endl;
    }
    return 0;
}
```

程序运行结果 1:

```
Please input the side lengths of a triangle: 3 4 5
Area: 6
```

程序运行结果 2:

```
Please input the side lengths of a triangle: 0 5 5
Error: the side length should be positive
```

8.4 训练强化

9 C++输入和输出流

9.1 I/O 流的概念和流类库的结构

程序的输入指的是从输入文件将数据传送给程序,程序的输出指的是从程序将数据传送给输出文件。

C++输入输出包含以下三个方面的内容:

对系统指定的标准设备的输入和输出。即从键盘输入数据,输出到显示器屏幕。这种输入输出称为标准的输入输出,简称标准 I/O。

以外存磁盘文件为对象进行输入和输出,即从磁盘文件输入数据,数据输出到磁盘文件。以外存文件为对象的输入输出称为文件的输入输出,简称文件 I/O。

对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间(实际上可以利用该空间存储任何信息)。这种输入和输出称为字符串输入输出,简称串 I/O。

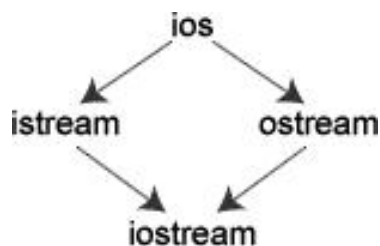
C++的 I/O 对 C 的发展--类型安全和可扩展性

在 C 语言中,用 printf 和 scanf 进行输入输出,往往不能保证所输入输出的数据是可靠的安全的。在 C++的输入输出中,编译系统对数据类型进行严格的检查,凡是类型不正确的数据都不可能通过编译。因此 C++的 I/O 操作是类型安全(type safe)的。C++的 I/O 操作是可扩展的,不仅可以用来输入输出标准类型的数据,也可以用于用户自定义类型的数据。

C++通过 I/O 类库来实现丰富的 I/O 功能。这样使 C++的输入输出明显地优于 C 语言中的 printf 和 scanf,但是也为之付出了代价,C++的 I/O 系统变得比较复杂,要掌握许多细节。

C++编译系统提供了用于输入输出的 iostream 类库。iostream 这个单词是由 3 个部 分组

成的，即 i-o-stream，意为输入输出流。在 iostream 类库中包含许多用于输入输出的类。常用的见表

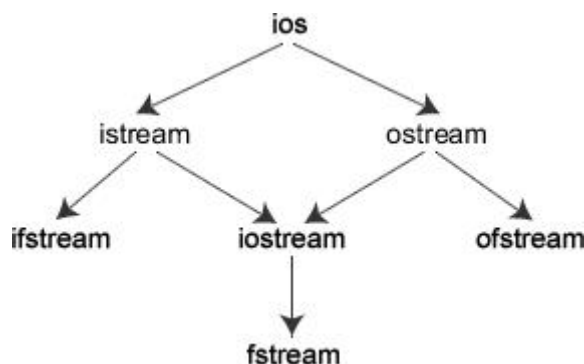


.I/O类库中的常用流类

| 类名 | 作用 | 在哪个头文件中声明 |
|------------|--------------------|-----------|
| ios | 抽象基类 | iostream |
| istream | 通用输入流和其他输入流的基类 | iostream |
| ostream | 通用输出流和其他输出流的基类 | iostream |
| iostream | 通用输入输出流和其他输入输出流的基类 | iostream |
| ifstream | 输入文件流类 | fstream |
| ofstream | 输出文件流类 | fstream |
| fstream | 输入输出文件流类 | fstream |
| istrstream | 输入字符串流类 | strstream |
| ostrstream | 输出字符串流类 | strstream |
| strstream | 输入输出字符串流类 | strstream |

ios 是抽象基类，由它派生出 istream 类和 ostream 类，两个类名中第 1 个字母 i 和 o 分别代表输入(input)和输出(output)。istream 类支持输入操作，ostream 类支持输出操作，iostream 类支持输入输出操作。iostream 类是从 istream 类和 ostream 类通过多重继承而派生的类。其继承层次见上图表示。

C++对文件的输入输出需要用 ifstream 和 ofstream 类，两个类名中第 1 个字母 i 和 o 分别代表输入和输出，第 2 个字母 f 代表文件 (file)。ifstream 支持对文件的输入操作，ofstream 支持对文件的输出操作。类 ifstream 继承了类 istream，类 ofstream 继承了类 ostream，类 fstream 继承了类 iostream。见图



I/O 类库中还有其他一些类，但是对于一般用户来说，以上这些已能满足需要了。

与 iostream 类库有关的头文件

iostream 类库中不同的类的声明被放在不同的头文件中，用户在自己的程序中使用#include 命令包含了有关的头文件就相当于在本程序中声明了所需要用到的类。可以换一种说法：头文件是程序与类库的接口，iostream 类库的接口分别由不同的头文件来实现。常用的有

- iostream 包含了对输入输出流进行操作所需的基本信息。
- fstream 用于用户管理的文件的 I/O 操作。
- strstream 用于字符串流 I/O。
- stdiostream 用于混合使用 C 和 C++ 的 I/O 机制时，例如想将 C 程序转变为 C++ 程序。
- iomanip 在使用格式化 I/O 时应包含此头文件。

在 iostream 头文件中定义的流对象

在 iostream 头文件中定义的类有 ios, istream, ostream, iostream, istream_withassign, ostream_withassign, iostream_withassign 等。

在 iostream 头文件中不仅定义了有关的类，还定义了 4 种流对象，

| 对象 | 含义 | 对应设备 | 对应的类 | c 语言中相应的标准文件 |
|------|-------|------|--------------------|--------------|
| cin | 标准输入流 | 键盘 | istream_withassign | stdin |
| cout | 标准输出流 | 屏幕 | ostream_withassign | stdout |
| cerr | 标准错误流 | 屏幕 | ostream_withassign | stderr |
| clog | 标准错误流 | 屏幕 | ostream_withassign | stderr |

在 iostream 头文件中定义以上 4 个流对象用以下的形式（以 cout 为例）：

```
ostream cout ( stdout);
```

在定义 cout 为 ostream 流类对象时，把标准输出设备 stdout 作为参数，这样它就与标准输出设备(显示器)联系起来，如果有

```
cout <<3;
```

就会在显示器的屏幕上输出 3。

在 iostream 头文件中重载运算符

“<<”和“>>”本来在 C++ 中是被定义为左位移运算符和右位移运算符的，由于在 iostream 头文件对它们进行了重载，使它们能用作标准类型数据的输入和输出运算符。所以，在它们的程序中必须用#include 命令把 iostream 包含到程序中。

```
#include <iostream>
```

- 1) >>a 表示将数据放入 a 对象中。
- 2) <<a 表示将 a 对象中存储的数据拿出。

9.2 标准 I/O 流

标准 I/O 对象:cin, cout, cerr, clog

cout 流对象

cout 是 console output 的缩写，意为在控制台（终端显示器）的输出。强调几点。

1) cout 不是 C++ 预定义的关键字，它是 ostream 流类的对象，在 ostream 中定义。顾名思义，流是流动的数据，cout 流是流向显示器的数据。cout 流中的数据是用流插入运算符“<<”顺序加入的。如果有

```
cout<<"I "<<"study C++ "<<"very hard. <<"wang bao ming ";
```

按顺序将字符串"I ", "study C++ ", "very hard."插入到 cout 流中，cout 就将它们送到显示器，在显示器上输出字符串"I study C++ very hard."。cout 流是容纳数据的载体，它并不是一个运算符。人们关心的是 cout 流中的内容，也就是向显示器输出什么。

2) 用“ccmt<<”输出基本类型的数据时，可以不必考虑数据是什么类型，系统会判断数据的类型，并根据其类型选择调用与之匹配的运算符重载函数。这个过程都是自动的，用户不必干预。如果在 C 语言中用 printf 函数输出不同类型的数据，必须分别指定相应的输出格式符，十分麻烦，而且容易出错。C++ 的 I/O 机制对用户来说，显然是方便而安全的。

3) cout 流在内存中对应开辟了一个缓冲区，用来存放流中的数据，当向 cout 流插入一个 endl 时，不论缓冲区是否已满，都立即输出流中所有数据，然后插入一个换行符，并刷新流（清空缓冲区）。注意如果插入一个换行符“\n”（如 cout<<a<<"\n"），则只输出和换行，而不刷新 cout 流（但并不是所有编译系统都体现出这一区别）。

4) 在 ostream 中只对“<<”和“>>”运算符用于标准类型数据的输入输出进行了重载，但未对用户声明的类型数据的输入输出进行重载。如果用户声明了新的类型，并希望用“<<”和“>>”运算符对其进行输入输出，按照重载运算符重载来做。

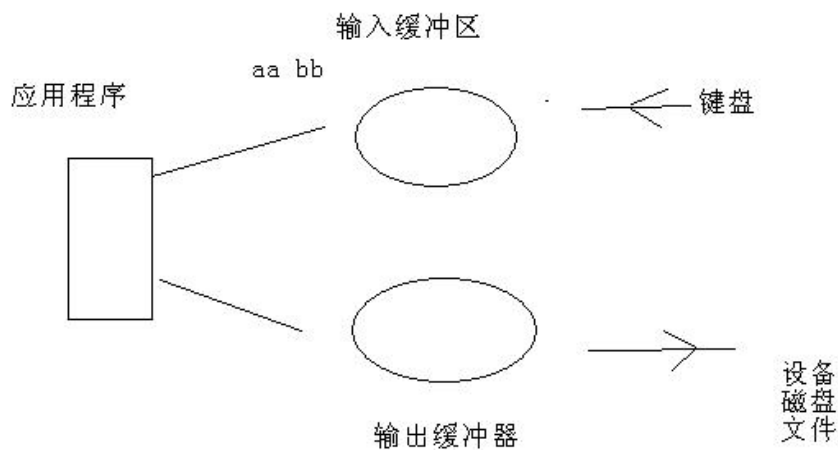
cerr 流对象

cerr 流对象是标准错误流，cerr 流已被指定为与显示器关联。cerr 的作用是向标准错误设备(standard error device)输出有关出错信息。cerr 与标准输出流 cout 的作用和用法差不多。但有一点不同：cout 流通常是传送到显示器输出，但也可以被重定向输出到磁盘文件，而 cerr 流中的信息只能在显示器输出。当调试程序时，往往不希望程序运行时的出错信息被送到其他文件，而要求在显示器上及时输出，这时应该用 cerr。cerr 流中的信息是用户根据需要指定的。

clog 流对象

clog 流对象也是标准错误流，它是 console log 的缩写。它的作用和 cerr 相同，都是在终端显示器上显示出错信息。区别：cerr 是不经过缓冲区，直接向显示器上输出有关信息，而 clog 中的信息存放在缓冲区中，缓冲区满后或遇 endl 时向显示器输出。

缓冲区的概念：



1 读和写是站在应用程序的角度来说的

9.2.1 标准输入流

标准输入流对象 `cin`，重点掌握的函数

```
cin.get() //一次只能读取一个字符
cin.get(一个参数) //读一个字符
cin.get(三个参数) //可以读字符串
cin.getline()
cin.ignore()
cin.peek()
cin.putback()
```

标准输入流常见 api 编程案例

```
//1 cin cout 能根据类型 获取数据 / 输出数据
//2 输入字符串 你 好 遇见空格,停止接受输入
void main01()
{
    char YourName[50];
    int myInt;
    long myLong;
    double myDouble;
    float myFloat;
    unsigned int myUnsigned;
```

```
    cout << "请输入一个 Int: ";
    cin >> myInt;
    cout << "请输入一个 Long: ";
    cin >> myLong;
    cout << "请输入一个 Double: ";
    cin >> myDouble;

    cout << "请输入你的姓名: ";
    cin >> YourName;

    cout << "\n\n 你输入的数是: " << endl;
    cout << "Int: \t" << myInt << endl;
    cout << "Long: \t" << myLong << endl;
    cout << "Double: \t" << myDouble << endl;
    cout << "姓名: \t" << YourName << endl;
    cout << endl << endl;
    system("pause");
    return ;
}

//1 输入英文 ok
//2 ctrl+z 会产生一个 EOF(-1)
int main02()
{
    char ch;
    while( (ch= cin.get())!= EOF)
    {
        std::cout << "字符: " << ch << std::endl;
    }
    std::cout << "\n 结束.\n";
    system("pause");
    return 0;
}

//演示:读一个字符 链式编程
void main03()
{
    char a, b, c;
    cin.get(a);
    cin.get(b);
    cin.get(c);
    cout << a << b << c << endl;
```

```
cout << "开始链式编程" << endl;
cout.flush();

cin.get(a).get(b).get(c);
cout << a << b << c << endl;
system("pause");
return ;
}

//演示 cin.getline() 可以接受空格
void main04()
{
    char buf1[256];
    char buf2[256];
    cout << "\n 请输入你的字符串 不超过 256";
    cin.getline(buf1, 256, '\n');
    cout << buf1 << endl;

    //
    cout << "注意: cin.getline() 和 cin >> buf2 的区别, 能不能带空格 " << endl;
    cin >> buf2; //流提取操作符 遇见空格 停止提取输入流
    cout << buf2 << endl;
    system("pause");
}

//缓冲区实验
/*
1 输入 "aa bb cc dd" 字符串入缓冲区
2 通过 cin >> buf1; 提走了 aa
3 不需要输入 可以通过 cin.getline() 把剩余的缓冲区数据提走
*/
void main05()
{
    char buf1[256];
    char buf2[256];

    cout << "请输入带有空格的字符串,测试缓冲区" << endl;
    cin >> buf1;
    cout << "buf1:" << buf1 << endl;

    cout << "请输入数据..." << endl;

    //缓冲区没有数据,就等待; 缓冲区如果有数据直接从缓冲区中拿走数据
```

```
cin.getline(buf2, 256);
cout << "buf2:" << buf2 << endl;
system("pause");
}

// ignore 和 peek
void main06()
{
    int  intchar;
    char buf1[256];
    char buf2[256];

    cout << "请输入带有空格的字符串,测试缓冲区 aa bb cc dd ee " << endl;
    cin >> buf1;
    cout << "buf1:" << buf1 << endl;

    cout << "请输入数据..." << endl;
    cin.ignore(2);
    //intchar = cin.peek();
    //cout << "缓冲区若有数据,返回第一个数据的 asc 码:" << intchar << endl;

    //缓冲区没有数据,就等待; 缓冲区如果有数据直接从缓冲区中拿走数据
    cin.getline(buf2, 256);
    cout << "buf2:" << buf2 << endl;

    intchar = cin.peek(); //没有缓冲区 默认是阻塞模式
    cout << "缓冲区若有数据,返回第一个数据的 asc 码:" << intchar << endl;
    system("pause");
}

//案例:输入的整数和字符串分开处理
int main07()
{
    cout << "Please, enter a number or a word: ";
    char c = std::cin.get();

    if ( (c >= '0') && (c <= '9') ) //输入的整数和字符串 分开处理
    {
        int n; //整数不可能 中间有空格 使用 cin >> n
        cin.putback(c);
        cin >> n;
        cout << "You entered a number: " << n << '\n';
    }
    else
```



```
{
    string str;
    cin.putback(c);
    getline(cin, str); // //字符串 中间可能有空格 使用 cin.getline();
    cout << "You entered a word: " << str << '\n';
}    system("pause");
return 0;
}
```

9.2.2 标准输出流

```
/*
标准输出流对象 cout
    cout.flush()
    cout.put()
    cout.write()
    cout.width()
    cout.fill()
    cout.setf(标记)
*/
/*
manipulator(操作符、控制符)
flush
endl
oct
dec
hex
setbase
setw
setfill
setprecision
...
*/
```

标准输出流常见 api 编程案例

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

void main81()
{
```

```
cout << "hello world" << endl;
cout.put('h').put('e').put('l').put('\n');
cout.write("hello world", 4); //输出的长度

char buf[] = "hello world";
printf("\n");
cout.write(buf, strlen(buf));

printf("\n");
cout.write(buf, strlen(buf) - 6);

printf("\n");
cout.write(buf, strlen(buf) + 6); //给的大于 buf 长度 不会帮我们检查 提高速度

printf("\n");

system("pause");
return ;
}

//使用 cout.setf()控制符
void main82()
{
    //使用类成员函数
    cout << "<start>";
    cout.width(30);
    cout.fill('*');
    cout.setf(ios::showbase); //include <iomanip>
    cout.setf(ios::internal); //设置
    cout << hex << 123 << "<End>\n";

    cout << endl;
    cout << endl;
    //manipulator(操作符、控制符)

    //使用控制阀
    cout << "<Start>"
        << setw(30)
        << setfill('*')
        << setiosflags(ios::showbase) //基数
        << setiosflags(ios::internal)
        << hex
        << 123
        << "<End>\n"
```

```
<< endl;

system("pause");
}
```

C++ 格式化输出，C++ 输出格式控制

在输出数据时，为简便起见，往往不指定输出的格式，由系统根据数据的类型采取默认的格式，但有时希望数据按指定的格式输出，如要求以十六进制或八进制形式 输出一个 整数，对输出的小数只保留两位小数等。有两种方法可以达到此目的。

- 1) 使用控制符的方法；
- 2) 使用流对象的有关成员函数。分别叙述如下。

使用控制符的方法

```
int main()
{
    int a;
    cout<<"input a:";
    cin>>a;
    cout<<"dec:"<<dec<<a<<endl; //以十进制形式输出整数
    cout<<"hex:"<<hex<<a<<endl; //以十六进制形式输出整数 a
    cout<<"oct:"<<setbase(8)<<a<<endl; //以八进制形式输出整数 a
    char *pt="China"; //pt 指向字符串"China"
    cout<<setw(10)<<pt<<endl; //指定域宽为,输出字符串
    cout<<setfill('*')<<setw(10)<<pt<<endl; //指定域宽,输出字符串,空白处以'*'填充
    double pi=22.0/7.0; //计算 pi 值
    //按指数形式输出,8 位小数
    cout<<setiosflags(ios::scientific)<<setprecision(8);
    cout<<"pi="<<pi<<endl; //输出 pi 值
    cout<<"pi="<<setprecision(4)<<pi<<endl; //改为位小数
    cout<<"pi="<<setiosflags(ios::fixed)<<pi<<endl; //改为小数形式输出
    system("pause");
    return 0;
}
```

运行结果如下:

```
input a:34↵(输入 a 的值)
dec:34          (十进制形式)
hex:22          (十六进制形式)
oct:42          (八进制形式)
      China     (域宽为)
*****China     (域宽为,空白处以'*'填充)
pi=3.14285714e+00 (指数形式输出,8 位小数)
```

| | |
|---------------|----------------|
| pi=3.1429e+00 | (指数形式输出,4 位小数) |
| pi=3.143 | (小数形式输出,精度仍为) |

人们在输入输出时有一些特殊的要求，如在输出实数时规定字段宽度，只保留两位小数，数
据向左或向右对齐等。C++提供了在输入输出流中使用的控制符(有的书中称为操纵符)

表 3.1 输入输出流的控制符

| 控制符 | 作用 |
|---------------------------------|--|
| dec | 设置数值的基数为10 |
| hex | 设置数值的基数为16 |
| oct | 设置数值的基数为8 |
| setfill(c) | 设置填充字符c，c可以是字符常量或字符变量 |
| setprecision(n) | 设置浮点数的精度为n位。在以一般十进制小数形式输出时，n代表有效数字。在以 fixed(固定小数位数)形式和 scientific(指数)形式输出时，n为小数位数 |
| setw(n) | 设置字段宽度为n位 |
| setiosflags(ios::fixed) | 设置浮点数以固定的小数位数显示 |
| setiosflags(ios::scientific) | 设置浮点数以科学记数法(即指数形式)显示 |
| setiosflags(ios::left) | 输出数据左对齐 |
| setiosflags(ios::right) | 输出数据右对齐 |
| setiosflags(ios::skipws) | 忽略前导的空格 |
| setiosflags(ios::uppercase) | 数据以十六进制形式输出时字母以大写表示 |
| setiosflags(ios::lowercase) | 数据以十六进制形式输出时字母以小写表示 |
| setiosflags(ios::showpos) | 输出正数时给出 “+” 号 |

需要注意的是：如果使用了控制符，在程序单位的开头除了要加iostream头文件外，还要加iomanip头文件。

举例，输出双精度数：

```
double a=123.456789012345; // 对 a 赋初值
```

- 1) cout<<a; 输出： 123.456
- 2) cout<<setprecision(9)<<a; 输出： 123.456789
- 3) cout<<setprecision(6); 恢复默认格式(精度为 6)
- 4) cout<< setiosflags(ios::fixed); 输出： 123.456789
- 5) cout<<setiosflags(ios::fixed)<<setprecision(8)<<a; 输出： 123.45678901
- 6) cout<<setiosflags(ios::scientific)<<a; 输出： 1.234568e+02
- 7) cout<<setiosflags(ios::scientific)<<setprecision(4)<<a; 输出： 1.2346e02

下面是整数输出的例子：

```
int b=123456; // 对 b 赋初值
```

- 1) cout<<b; 输出： 123456
- 2) cout<<hex<<b; 输出： 1e240

```
3) cout<<setiosflags(ios::uppercase)<<b; 输出:  1E240
4) cout<<setw(10)<<b<<', '<<b;  输出:  123456, 123456
5) cout<<setfill('*')<<setw(10)<<b; 输出:  **** 123456
6) cout<<setiosflags(ios::showpos)<<b; 输出:  +123456
```

如果在多个 `cout` 语句中使用相同的 `setw(n)`，并使用 `setiosflags(ios::right)`，可以实现各行数据右对齐，如果指定相同的精度，可以实现上下小数点对齐。

例如：各行小数点对齐。

```
int main( )
{
    double a=123.456,b=3.14159,c=-3214.67;
    cout<<setiosflags(ios::fixed)<<setiosflags(ios::right)<<setprecision(2);
    cout<<setw(10)<<a<<endl;
    cout<<setw(10)<<b<<endl;
    cout<<setw(10)<<c<<endl;
    system("pause");
    return 0;
}
```

输出如下：

```
123.46 (字段宽度为 10，右对齐，取两位小数)
3.14
-3214.67
```

先统一设置定点形式输出、取两位小数、右对齐。这些设置对其后的输出均有效(除非重新设置)，而 `setw` 只对其后一个输出项有效，因此必须在输出 `a`，`b`，`c` 之前都要写 `setw(10)`。

//

用流对象的成员函数控制输出格式

除了可以用控制符来控制输出格式外，还可以通过调用流对象 `cout` 中用于控制输出格式的成员函数来控制输出格式。用于控制输出格式的常用的成员函数如下：

表13.4 用于控输出格式的流成员函数

| 流成员函数 | 与之作用相同的控制符 | 作用 |
|--------------|-----------------|--|
| precision(n) | setprecision(n) | 设置实数的精度为n位 |
| width(n) | setw(n) | 设置字段宽度为n位 |
| fill(c) | setfill(c) | 设置填充字符c |
| setf() | setiosflags() | 设置输出格式状态，括号中应给出格式状态，内容与控制符setiosflags括号中的内容相同，如表13.5所示 |
| unsetf() | resetiosflags() | 终止已设置的输出格式状态，在括号中应指定内容 |

流成员函数 `setf` 和控制符 `setiosflags` 括号中的参数表示格式状态，它是通过格式标志来指定的。格式标志在类 `ios` 中被定义为枚举值。因此在引用这些格式标志时要在前面加上类名 `ios` 和域运算符“`::`”。格式标志见表 13.5。

表13.5 设置格式状态的格式标志

| 格式标志 | 作用 |
|-----------------|--------------------------------|
| ios::left | 输出数据在本域宽范围内向左对齐 |
| ios::right | 输出数据在本域宽范围内向右对齐 |
| ios::internal | 数值的符号位在域宽内左对齐，数值右对齐，中间由填充字符填充 |
| ios::dec | 设置整数的基数为10 |
| ios::oct | 设置整数的基数为8 |
| ios::hex | 设置整数的基数为16 |
| ios::showbase | 强制输出整数的基数(八进制数以0打头，十六进制数以0x打头) |
| ios::showpoint | 强制输出浮点数的小点和尾数0 |
| ios::uppercase | 在以科学记数法格式E和以十六进制输出字母时以大写表示 |
| ios::showpos | 对正数显示“+”号 |
| ios::scientific | 浮点数以科学记数法格式输出 |
| ios::fixed | 浮点数以定点格式(小数形式)输出 |
| ios::unitbuf | 每次输出之后刷新所有的流 |
| ios::stdio | 每次输出之后清除stdout, stderr |

例：用流控制成员函数输出数据。

```
int main( )
{
    int a=21;
    cout.setf(ios::showbase);//显示基数符号(0x 或)
    cout<<"dec:"<<a<<endl; //默认以十进制形式输出 a
    cout.unsetf(ios::dec); //终止十进制的格式设置
    cout.setf(ios::hex); //设置以十六进制输出的状态
    cout<<"hex:"<<a<<endl; //以十六进制形式输出 a
    cout.unsetf(ios::hex); //终止十六进制的格式设置
    cout.setf(ios::oct); //设置以八进制输出的状态
    cout<<"oct:"<<a<<endl; //以八进制形式输出 a
    cout.unsetf(ios::oct);
    char *pt="China"; //pt 指向字符串"China"
    cout.width(10); //指定域宽为
    cout<<pt<<endl; //输出字符串
    cout.width(10); //指定域宽为
    cout.fill('*'); //指定空白处以'*'填充
```

```
cout<<pt<<endl; //输出字符串
double pi=22.0/7.0; //输出 pi 值
cout.setf(ios::scientific); //指定用科学记数法输出
cout<<"pi="; //输出"pi="
cout.width(14); //指定域宽为
cout<<pi<<endl; //输出 pi 值
cout.unsetf(ios::scientific); //终止科学记数法状态
cout.setf(ios::fixed); //指定用定点形式输出
cout.width(12); //指定域宽为
cout.setf(ios::showpos); //正数输出 “+” 号
cout.setf(ios::internal); //数符出现在左侧
cout.precision(6); //保留位小数
cout<<pi<<endl; //输出 pi,注意数符 “+” 的位置
system("pause");
return 0;
}
```

运行情况如下:

```
dec:21(十进制形式)
hex:0x15      (十六进制形式,以 x 开头)
oct:025       (八进制形式,以开头)
      China   (域宽为)
****China    (域宽为,空白处以'*'填充)
pi=**3.142857e+00 (指数形式输出,域宽,默认位小数)
+***3.142857   (小数形式输出,精度为,最左侧输出数符“+”)
```

对程序的几点说明:

1) 成员函数 `width(n)` 和控制符 `setw(n)` 只对其后的第一个输出项有效。如:

```
cout.width(6);
cout<<20<<3.14<<endl;
```

输出结果为 203.14

在输出第一个输出项 20 时,域宽为 6,因此在 20 前面有 4 个空格,在输出 3.14 时, `width(6)` 已不起作用,此时按系统默认的域宽输出(按数据实际长度输出)。如果要求在输出数据时都按指定的同一域宽 `n` 输出,不能只调用一次 `width(n)`,而必须在输出每一项前都调用一次 `width(n)`,上面的程序中就是这样做的。

2) 在表 13.5 中的输出格式状态分为 5 组,每一组中同时只能选用一种(例如 `dec`、`hex` 和 `oct` 中只能选一,它们是互相排斥的)。在用成员函数 `setf` 和控制符 `setiosflags` 设置输出格式状态后,如果想改设置为同组的另一状态,应当调用成员函数 `unsetf`(对应于成员函数 `self`)或 `resetiosflags`(对应于控制符 `setiosflags`),先终止原来设置的状态。然后再设置其他状态,大家可以从本程序中看到这点。程序在开始虽然没有用成员函数 `self` 和控制符 `setiosflags` 设置用 `dec` 输出格式状态,但系统默认指定为 `dec`,因此要改变为 `hex` 或 `oct`,也应当先用 `unsetf` 函数终止原来设置。如果删去程序中的第 7 行和第 10 行,虽然在第 8 行和第 11 行中用成员函数 `setf` 设置了 `hex` 和 `oct` 格式,由于未终止 `dec` 格式,因此 `hex` 和 `oct` 的设置均不起作用,系统依然以十进制形式输出。

同理,程序倒数第 8 行的 `unsetf` 函数的调用也是不可缺少的。

3) 用 `setf` 函数设置格式状态时,可以包含两个或多个格式标志,由于这些格式标志在 `ios` 类中被定义为枚举值,每一个格式标志以一个二进位代表,因此可以用位或运算符“`|`”组合多个格式标志。如倒数第 5、第 6 行可以用下面一行代替:

```
cout.setf(ios::internal | ios::showpos); //包含两个状态标志,用"|"组合
```

3) 可以看到:对输出格式的控制,既可以用控制符(如例 13.2),也可以用 `cout` 流的有关成员函数(如例 13.3),二者的作用是相同的。控制符是在头文件 `iomanip` 中定义的,因此用控制符时,必须包含 `iomanip` 头文件。`cout` 流的成员函数是在头文件 `iostream` 中定义的,因此只需包含头文件 `iostream`,不必包含 `iomanip`。许多程序人员感到使用控制符方便简单,可以在一个 `cout` 输出语句中连续使用多种控制符。

9.3 文件 I/O

- ❖ 文件输入流 `ifstream`
- ❖ 文件输出流 `ofstream`
- ❖ 文件输入输出流 `fstream`
- ❖ 文件的打开方式
- ❖ 文件流的状态
- ❖ 文件流的定位:文件指针(输入指针、输出指针)
- ❖ 文本文件和二进制文件

9.3.1 文件流类和文件流对象

输入输出是以系统指定的标准设备(输入设备为键盘,输出设备为显示器)为对象的。在实际应用中,常以磁盘文件作为对象。即从磁盘文件读取数据,将数据输出到磁盘文件。和文件有关系的输入输出类主要在 `fstream.h` 这个头文件中被定义,在这个头文件中主要被定义了三个类,由这三个类控制对文件的各种输入输出操作,他们分别是 `ifstream`、`ofstream`、`fstream`,其中 `fstream` 类是由 `iostream` 类派生而来,他们之间的继承关系见下图所示。



由于文件设备并不像显示器屏幕与键盘那样是标准默认设备,所以它在 `fstream.h` 头文件中是没有像 `cout` 那样预先定义的全局对象,所以我们必须自己定义一个该类的对象。

`ifstream` 类,它是从 `istream` 类派生的,用来支持从磁盘文件的输入。

`ofstream` 类,它是从 `ostream` 类派生的,用来支持向磁盘文件的输出。

`fstream` 类,它是从 `iostream` 类派生的,用来支持对磁盘文件的输入输出。

9.3.2 C++ 文件的打开与关闭

打开文件

所谓打开(open)文件是一种形象的说法,如同打开房门就可以进入房间活动一样。打开文件是指在文件读写之前做必要的准备工作,包括:

- 1) 为文件流对象和指定的磁盘文件建立关联,以便使文件流流向指定的磁盘文件。
- 2) 指定文件的工作方式,如,该文件是作为输入文件还是输出文件,是 ASCII 文件还是二进制文件等。

以上工作可以通过两种不同的方法实现。

- 1) 调用文件流的成员函数 open。如

```
ofstream outfile; //定义 ofstream 类(输出文件流类)对象 outfile
outfile.open("f1.dat",ios::out); //使文件流与 f1.dat 文件建立关联
```

第 2 行是调用输出文件流的成员函数 open 打开磁盘文件 f1.dat,并指定它为输出文件,文件流对象 outfile 将向磁盘文件 f1.dat 输出数据。ios::out 是 I/O 模式的一种,表示以输出方式打开一个文件。或者简单地说,此时 f1.dat 是一个输出文件,接收从内存输出的数据。

调用成员函数 open 的一般形式为:

文件流对象.open(磁盘文件名,输入输出方式);

磁盘文件名可以包括路径,如"c:\new\f1.dat",如缺省路径,则默认为当前目录下的文件。

- 2) 在定义文件流对象时指定参数

在声明文件流类时定义了带参数的构造函数,其中包含了打开磁盘文件的功能。因此,可以在定义文件流对象时指定参数,调用文件流类的构造函数来实现打开文件的功能。如

```
ostream outfile("f1.dat",ios::out); 一般多用此形式,比较方便。作用与 open 函数相同。
```

输入输出方式是在 ios 类中定义的,它们是枚举常量,有多种选择,见表 13.6。

表13.6 文件输入输出方式设置值

| 方式 | 作用 |
|-------------------------------------|---|
| <code>ios::in</code> | 以输入方式打开文件 |
| <code>ios::out</code> | 以输出方式打开文件（这是默认方式），如果已有此名字的文件，则将其原有内容全部清除 |
| <code>ios::app</code> | 以输出方式打开文件，写入的数据添加在文件末尾 |
| <code>ios::ate</code> | 打开一个已有的文件，文件指针指向文件末尾 |
| <code>ios::trunc</code> | 打开一个文件，如果文件已存在，则删除其中全部数据，如文件不存在，则建立新文件。如已指定了 <code>ios::out</code> 方式，而未指定 <code>ios::app</code> ， <code>ios::ate</code> ， <code>ios::in</code> ，则同时默认此方式 |
| <code>ios::binary</code> | 以二进制方式打开一个文件，如不指定此方式则默认为ASCII方式 |
| <code>ios::nocreate</code> | 打开一个已有的文件，如文件不存在，则打开失败。 <code>nocreate</code> 的意思是不建立新文件 |
| <code>ios::noreplace</code> | 如果文件不存在则建立新文件，如果文件已存在则操作失败， <code>replace</code> 的意思是不更新原有文件 |
| <code>ios::in ios::out</code> | 以输入和输出方式打开文件，文件可读可写 |
| <code>ios::out ios::binary</code> | 以二进制方式打开一个输出文件 |
| <code>ios::in ios::binary</code> | 以二进制方式打开一个输入文件 |

几点说明：

1) 新版本的 I/O 类库中不提供 `ios::nocreate` 和 `ios::noreplace`。

2) 每一个打开的文件都有一个文件指针，该指针的初始位置由 I/O 方式指定，每次读写都从文件指针的当前位置开始。每读入一个字节，指针就后移一个字节。当文件指针移到最后，就会遇到文件结束 EOF（文件结束符也占一个字节，其值为-1），此时流对象的成员函数 `eof` 的值为非 0 值（一般设为 1），表示文件结束了。

3) 可以用“位或”运算符“|”对输入输出方式进行组合，如表 13.6 中最后 3 行所示那样。还可以举出下面一些例子：

`ios::in | ios::noreplace` //打开一个输入文件，若文件不存在则返回打开失败的信息

`ios::app | ios::nocreate` //打开一个输出文件，在文件尾接着写数据，若文件不存在，则返回打开失败的信息

`ios::out | ios::noreplace` //打开一个新文件作为输出文件，如果文件已存在则返回打开失败的信息

`ios::in | ios::out | ios::binary` //打开一个二进制文件，可读可写

但不能组合互相排斥的方式，如 `ios::nocreate | ios::noreplace`。

4) 如果打开操作失败，`open` 函数的返回值为 0（假），如果是用调用构造函数的方式打开文件的，则流对象的值为 0。可以据此测试打开是否成功。如

```
if(outfile.open("f1.bat", ios::app) == 0)
    cout << "open error";
```

或

```
if( !outfile.open("f1.bat", ios::app) )
    cout << "open error";
```

关闭文件

在对已打开的磁盘文件的读写操作完成后，应关闭该文件。关闭文件用成员函数 `close`。如

```
outfile.close( ); //将输出文件流所关联的磁盘文件关闭
```

所谓关闭，实际上是解除该磁盘文件与文件流的关联，原来设置的工作方式也失效，这样，就不能再通过文件流对该文件进行输入或输出。此时可以将文件流与其他磁盘文件建立关联，通过文件流对新的文件进行输入或输出。如

```
outfile.open("f2.dat",ios::app|ios::nocreate);
```

此时文件流 `outfile` 与 `f2.dat` 建立关联，并指定了 `f2.dat` 的工作方式。

9.3.3C++对 ASCII 文件的读写操作

如果文件的每一个字节中均以 ASCII 代码形式存放数据,即一个字节存放一个字符,这个文件就是 ASCII 文件(或称字符文件)。程序可以从 ASCII 文件中读入若干个字符,也可以向它输出一些字符。

- 1) 用流插入运算符“<<”和流提取运算符“>>”输入输出标准类型的数据。“<<”和“>>”都已在 `iostream` 中被重载为能用于 `ostream` 和 `istream` 类对象的标准类型的输入输出。由于 `ifstream` 和 `ofstream` 分别是 `ostream` 和 `istream` 类的派生类；因此它们从 `ostream` 和 `istream` 类继承了公用的重载函数，所以在对磁盘文件的操作中，可以通过文件流对象和流插入运算符“<<”及流提取运算符“>>”实现对磁盘文件的读写，如同用 `cin`、`cout` 和 `<<`、`>>` 对标准设备进行读写一样。
- 2) 用文件流的 `put`、`get`、`getline` 等成员函数进行字符的输入输出，：用 C++ 流成员函数 `put` 输出单个字符、C++ `get()` 函数读入一个字符和 C++ `getline()` 函数读入一行字符。

案例 1：写文件，然后读文件

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

int main92()
{
    char fileName[80];
    char buffer[255];

    cout << "请输入一个文件名: ";
    cin >> fileName;

    ofstream fout(fileName, ios::app);
```

```
fout << "11111111111111111111\n";
fout << "22222222222222222222\n";
//cin.ignore(1,'\n');
cin.getline(buffer,255); //从键盘输入
fout << buffer << "\n";
fout.close();

ifstream fin(fileName);
cout << "Here's the the content of the file: \n";
char ch;
while(fin.get(ch))
    cout << ch;

cout << "\n***End of file contents.***\n";
fin.close();
system("pause");
return 0;
}
```

案例 2（自学扩展思路）

ofstream 类的默认构造函数原形为：

```
ofstream::ofstream(constchar *filename, int mode = ios::out,
    int penprot = filebuf::openprot);
```

- filename: 要打开的文件名
- mode: 要打开文件的方式
- prot: 打开文件的属性

其中 mode 和 openprot 这两个参数的可选项表见下表：

| mode 属性表 | |
|-------------|----------------------------------|
| ios::app | 以追加的方式打开文件 |
| ios::ate | 文件打开后定位到文件尾，ios::app 就包含有此属性 |
| ios::binary | 以二进制方式打开文件，缺省的方式是文本方式。两种方式的区别见前文 |
| ios::in | 文件以输入方式打开 |
| ios::out | 文件以输出方式打开 |
| ios::trunc | 如果文件存在，把文件长度设为 0 |

可以用“|”把以上属性连接起来，如 ios::out|ios::binary。

| openprot 属性表 | |
|--------------|-----------|
| 属性 | 含义 |
| 0 | 普通文件，打开访问 |
| 1 | 只读文件 |
| 2 | 隐含文件 |
| 4 | 系统文件 |

可以用“或”或者“+”把以上属性连接起来，如 3 或 1|2 就是以只读和隐含属性打开文件。

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

int main()
{
    ofstream myfile("c:\\1.txt", ios::out | ios::trunc, 0);
    myfile << "传智播客" << endl << "网址: " << "www.itcast.cn";
    myfile.close();
    system("pause");
}
```

文件使用完后可以使用 close 成员函数关闭文件。

ios::app 为追加模式，在使用追加模式的时候同时进行文件状态的判断是一个比较好的习惯。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream myfile("c:\\1.txt", ios::app, 0);
    if (!myfile) // 或者写成 myfile.fail()
    {
        cout << "文件打开失败，目标文件状态可能为只读！";
        system("pause");
        exit(1);
    }
    myfile << "传智播客" << endl << "网址: " << " www.itcast.cn " << endl;
    myfile.close();
}
```

在定义 ifstream 和 ofstream 类对象的时候，我们也可以不指定文件。以后可以通过成员函数 open() 显式的把一个文件连接到一个类对象上。

例如：

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream myfile;
    myfile.open("c:\\1.txt", ios::out | ios::app, 0);
    if(!myfile) // 或者写成 myfile.fail()
    {
        cout << "文件创建失败, 磁盘不可写或者文件为只读!";
        system("pause");
        exit(1);
    }
    myfile << "传智播客" << endl << "网址: " << "www.itcast.cn" << endl;
    myfile.close();
}
```

下面我们来看一下是如何利用 `ifstream` 类对象，将文件中的数据读取出来，然后再输出到标准设备中的例子。

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
    ifstream myfile;
    myfile.open("c:\\1.txt", ios::in, 0);
    if(!myfile)
    {
        cout << "文件读错误";
        system("pause");
        exit(1);
    }
    char ch;
    string content;
    while(myfile.get(ch))
    {
        content += ch;
        cout.put(ch); // cout << ch; 这么写也是可以的
    }
    myfile.close();
}
```

```
    cout<<content;
    system("pause");
}
```

上例中，我们利用成员函数 `get()`，逐一的读取文件中的有效字符，再利用 `put()` 成员函数，将文件中的数据通过循环逐一输出到标准设备(屏幕)上，`get()` 成员函数会在文件读到默尾的时候返回假值，所以我们可以利用它的这个特性作为 `while` 循环的终止条件，我们同时也在上例中引入了 C++ 风格的字符串类型 `string`，在循环读取的时候逐一保存到 `content` 中，要使用 `string` 类型，必须包含 `string.h` 的头文件。

我们在简单介绍过 `ofstream` 类和 `ifstream` 类后，我们再来看一下 `fstream` 类，`fstream` 类是由 `iostream` 派生而来，`fstream` 类对象可以同对文件进行读写操作。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream myfile;
    myfile.open("c:\\1.txt", ios::out | ios::app, 0);
    if(!myfile)
    {
        cout<<"文件写错误,文件属性可能为只读!"<<endl;
        system("pause");
        exit(1);
    }
    myfile<<"传智播客"<<endl<<"网址: "<<"www.itcast.cn"<<endl;
    myfile.close();

    myfile.open("c:\\1.txt", ios::in, 0);
    if(!myfile)
    {
        cout<<"文件读错误,文件可能丢失!"<<endl;
        system("pause");
        exit(1);
    }
    char ch;
    while(myfile.get(ch))
    {
        cout.put(ch);
    }
    myfile.close();
    system("pause");
}
```

由于 `fstream` 类可以对文件同时进行读写操作，所以对它的对象进行初始化的时候一定要显式的指定 `mode` 和 `openprot` 参数。

9.3.4 C++对二进制文件的读写操作

二进制文件不是以 `ASCII` 代码存放数据的，它将内存中数据存储形式不加转换地传送到磁盘文件，因此它又称为**内存数据的映像文件**。因为文件中的信息不是字符数据，而是字节中的二进制形式的信息，因此它又称为**字节文件**。

对二进制文件的操作也需要先打开文件，用完后要关闭文件。在打开时要用 `ios::binary` 指定为以二进制形式传送和存储。二进制文件除了可以作为输入文件或输出文件外，还可以是既能输入又能输出的文件。这是和 `ASCII` 文件不同的地方。

用成员函数 `read` 和 `write` 读写二进制文件

对二进制文件的读写主要用 `istream` 类的成员函数 `read` 和 `write` 来实现。这两个成员函数的原型为

```
istream& read(char *buffer,int len);
```

```
ostream& write(const char * buffer,int len);
```

字符指针 `buffer` 指向内存中一段存储空间。`len` 是读写的字节数。调用的方式为：

a. `write(p1,50);`

b. `read(p2,30);`

上面第一行中的 `a` 是输出文件流对象，`write` 函数将字符指针 `p1` 所给出的地址开始的 50 个字节的内容不加转换地写到磁盘文件中。在第二行中，`b` 是输入文件流对象，`read` 函数从 `b` 所关联的磁盘文件中，读入 30 个字节(或遇 `EOF` 结束)，存放在字符指针 `p2` 所指的一段空间内。

案例 1

```
//二进制
int main()
{
    char fileName[255] = "c:/teacher.dat";
    ofstream fout(fileName,ios::binary);
    if(!fout)
    {
        cout << "Unable to open " << fileName << " for writing.\n";
        return(1);
    }
}
```



```
}

Teacher t1(31, "31");
Teacher t2(32, "32");
fout.write((char *)&t1, sizeof Teacher);
fout.write((char *)&t2, sizeof Teacher);
fout.close();

cout << "保存对象到二进制文件里成功!" << endl;

ifstream fin(fileName, ios::binary);
if(!fin)
{
    cout << "Unable to open " << fileName << " for reading.\n";
    return (1);
}
Teacher tmp(100, "100");

fin.read((char *)&tmp, sizeof Teacher);
tmp.printT();
fin.read((char *)&tmp, sizeof Teacher);
tmp.printT();
system("pause");

return 0;
}
```

9.4 作业练习

1 编程实现以下数据输入/输出：

- (1) 以左对齐方式输出整数,域宽为 12。
- (2) 以八进制、十进制、十六进制输入/输出整数。
- (3) 实现浮点数的指数格式和定点格式的输入/输出,并指定精度。
- (4) 把字符串读入字符型数组变量中,从键盘输入,要求输入串的空格也全部读入,以回车符结束。
- (5) 将以上要求用流成员函数和操作符各做一遍。

2 编写一程序, 将两个文件合并成一个文件。

3 编写一程序, 统计一篇英文文章中单词的个数与行数。

4 编写一程序, 将 C++ 源程序每行前加上行号与一个空格。

4.5 编写一程序, 输出 ASCII 码值从 20 到 127 的 ASCII 码字符表, 格式为每行 10 个。

参考答案：

第一题

ios 类成员函数实现

```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
    long a=234;
    double b=2345.67890;
    char c[100];
    cout.fill('*');
    cout.flags(ios_base::left);
    cout.width(12);
    cout<<a<<endl;
    cout.fill('*');
    cout.flags(ios::right);
    cout.width(12);
    cout<<a<<endl;
    cout.flags(ios.hex);
    cout<<234<<'\\t';
    cout.flags(ios.dec);
    cout<<234<<'\\t';
    cout.flags(ios.oct);
    cout<<234<<endl;
    cout.flags(ios::scientific);
    cout<<b<<'\\t';
    cout.flags(ios::fixed);
    cout<<b<<endl;
    cin.get(c,99);
    cout<<c<<endl;
    return 0;
}
```

操作符实现

```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
    long a=234;
    double b=2345.67890;
    char c[100];
    cout<<setfill('*');
    cout<<left<<setw(12)<<a<<endl;
    cout<<right<<setw(12)<<a<<endl;
    cout<<hex<<a<<'\\t'<<dec<<a<<'\\t'<<oct<<a<<endl;
    cout<<scientific<<b<<'\\t'<<fixed<<b<<endl;
    return 0;
}
```

```
}
```

第二题:

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    int i=1;
    char c[1000];
    ifstream ifile1("D:\\1.cpp");
    ifstream ifile2("D:\\2.cpp");
    ofstream ofile("D:\\3.cpp");
    while(!ifile1.eof()){
        ifile1.getline(c,999);
        ofile<<c<<endl;
    }
    while(!ifile2.eof()){
        ifile2.getline(c,999);
        ofile<<c<<endl;
    }
    ifile1.close();
    ifile2.close();
    ofile.close();
    return 0;
}
```

第三题

```
#include<iostream>
#include<fstream>
using namespace std;
bool isalph(char);
int main(){
    ifstream ifile("C:\\daily.doc");
    char text[1000];
    bool inword=false;
    int rows=0,words=0;
    int i;
    while(!ifile.eof()){
        ifile.getline(text,999);
        rows++;
        i=0;
        while(text[i]!=0){
```

```
        if(!isalph(text[i]))
            inword=false;
        else if(isalph(text[i]) && inword==false){
            words++;
            inword=true;
        }
        i++;
    }
}
cout<<"rows= "<<rows<<endl;
cout<<"words= "<<words<<endl;
ifile.close ();
return 0;
}
bool isalph(char c){
    return ((c>='A' && c<='Z') || (c>='a' && c<='z'));
}
```

第四题

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    int i=1;
    char c[1000];
    ifstream ifile("D:\\1.cpp");
    ofstream ofile("D:\\2.cpp");
    while(!ifile.eof()){
        ofile<<i++<<" ";
        ifile.getline(c,999);
        ofile<<c<<endl;
    }
    ifile.close();
    ofile.close();
    return 0;
}
```

第五题

```
#include<iostream>
using namespace std;
int main(){
```

```
int i,l;  
for(i=32;i<127;i++){  
    cout<<char(i)<<" ";  
    l++;  
    if(l%10==0)cout<<endl;  
}  
cout<<endl;  
return 0;  
}
```

10、STL 实用技术专题

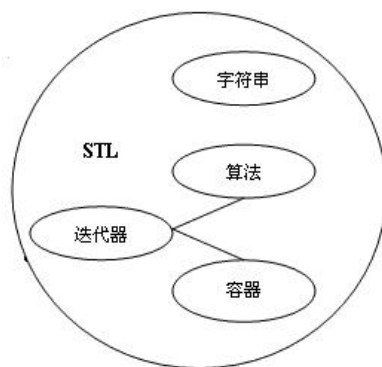
10.1 STL(标准模板库)理论基础

10.1.1 基本概念

STL (**Standard Template Library**, 标准模板库)是惠普实验室开发的一系列软件的统称。现然主要出现在 C++中,但在被引入 C++之前该技术就已经存在了很长的一段时间。

STL 的从广义上讲分为三类: **algorithm** (算法)、**container** (容器) 和 **iterator** (迭代器), 容器和算法通过迭代器可以进行无缝地连接。几乎所有的代码都采用了模板类和模板函数的方式, 这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。在 C++标准中, STL 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>。



STL 详细的说六大组件

- 容器 (Container)
- 算法 (Algorithm)
- 迭代器 (Iterator)
- 仿函数 (Function object)
- 适配器 (Adaptor)
- 空间配制器 (allocator)

使用 STL 的好处

1) STL 是 C++ 的一部分，因此不用额外安装什么，它被内建在你的编译器之内。

2) **STL 的一个重要特点是数据结构和算法的分离**。尽管这是个简单的概念，但是这种分离确实使得 STL 变得非常通用。

例如，在 STL 的 `vector` 容器中，可以放入元素、基础数据类型变量、元素的地址；

STL 的 `sort()` 函数可以用来操作 `vector`, `list` 等容器。

3) 程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。

4) STL 具有高可重用性，高性能，高移植性，跨平台的优点。

高可重用性：STL 中几乎所有的代码都采用了模板类和模板函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。关于模板的知识，已经给大家介绍了。

高性能：如 `map` 可以高效地从十万条记录里面查找出指定的记录，因为 `map` 是采用红黑树的变体实现的。(红黑树是平衡二叉树的一种)

高移植性：如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。

跨平台：如用 windows 的 Visual Studio 编写的代码可以在 Mac OS 的 XCode 上直接编译。

5) 程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。

6) 了解到 STL 的这些好处，我们知道 STL 无疑是最值得 C++ 程序员骄傲的一部分。每一个 C++ 程序员都应该好好学习 STL。**只有能够熟练使用 STL 的程序员，才是好的 C++ 程序员。**

7) 总之：招聘工作中，经常遇到 C++ 程序员对 STL 不是非常了解。大多是有一个大致的映像，而对于在什么情况下应该使用哪个容器和算法都感到比较茫然。**STL 是 C++ 程序员的一项不可或缺的基本技能**，掌握它对提升 C++ 编程大有裨益。

10.1.2 容器

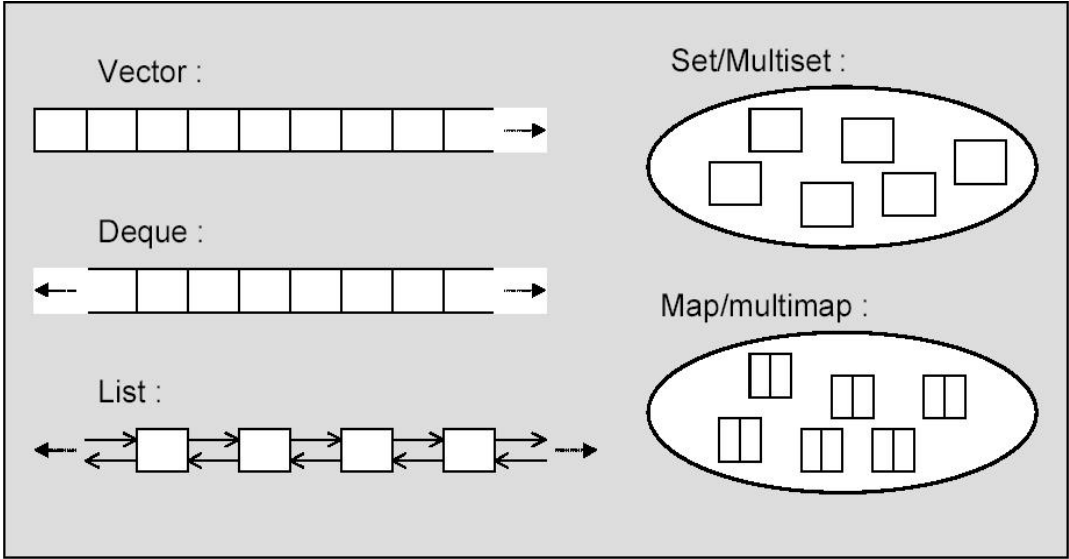
在实际的开发过程中，数据结构本身的重要性不会逊于操作于数据结构的算法的重要性，当程序中存在对时间要求很高的部分时，数据结构的选择就显得更加重要。

经典的数据结构数量有限，但是我们常常重复着一些为了实现向量、链表等结构而编写的代码，这些代码都十分相似，只是为了适应不同数据的变化而在细节上有所出入。STL 容器就为我们提供了这样的方便，它允许我们重复利用已有的实现构造自己的特定类型下的数据结构，通过设置一些模板，STL 容器对最常用的数据结构提供了支持，这些模板的参数允许我们指定容器中元素的数据类型，可以将我们许多重复而乏味的工作简化。

容器部分主要由头文件 `<vector>`, `<list>`, `<deque>`, `<set>`, `<map>`, `<stack>` 和 `<queue>` 组成。对于常用的一些容器和容器适配器（可以看作由其它容器实现的容器），可以通过下表总结一下它们和相应头文件的对应关系。

10.1.2.1 容器的概念

用来管理一组元素



10.1.2.2 容器的分类

- 序列式容器 (Sequence containers)
- 每个元素都有固定位置——取决于插入时机和地点，和元素值无关。
 - vector*、*deque*、*list*
- 关联式容器 (Associated containers)
- 元素位置取决于特定的排序准则，和插入顺序无关
 - set*、*multiset*、*map*、*multimap*

| 数据结构 | 描述 | 实现头文件 |
|----------------------|--|----------|
| 向量(vector) | 连续存储的元素 | <vector> |
| 列表(list) | 由节点组成的双向链表，每个结点包含着一个元素 | <list> |
| 双队列(deque) | 连续存储的指向不同元素的指针所组成的数组 | <deque> |
| 集合(set) | 由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序 | <set> |
| 多重集合(multiset) | 允许存在两个次序相等的元素的集合 | <set> |
| 栈(stack) | 后进先出的值的排列 | <stack> |
| 队列(queue) | 先进先出的值的排列 | <queue> |
| 优先队列(priority_queue) | 元素的次序是由作用于所存储的值对上的某种谓词决定的一种队列 | <queue> |
| 映射(map) | 由{键, 值}对组成的集合，以某种作用于键对上的谓词排列 | <map> |
| 多重映射(multimap) | 允许键对有相等的次序的映射 | <map> |

10.1.3 迭代器

迭代器从作用上来说是最基本的部分，可是理解起来比前两者都要费力一些。软件设计有一个基本原则，所有的问题都可以通过引进一个间接层来简化，这种简化在 STL 中就是用迭代器来完成的。概括来说，迭代器在 STL 中用来将算法和容器联系起来，起着一种黏和剂的作用。几乎 STL 提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器，用以存取容器中的元素。

迭代器部分主要由头文件 `<utility>`、`<iterator>` 和 `<memory>` 组成。`<utility>` 是一个很小的头文件，它包括了贯穿使用在 STL 中的几个模板的声明，`<iterator>` 中提供了迭代器使用的许多方法，而对于 `<memory>` 的描述则十分的困难，它以不同寻常的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生的临时对象提供机制，`<memory>` 中的主要部分是模板类 `allocator`，它负责产生所有容器中的默认分配器。

10.1.4 算法

函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类型要高。而 C++ 通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL 就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——可以将所有的类型划分为少数的几类，然后就可以在模板的参数中使用一种类型替换掉同一种类中的其他类型。

STL 提供了大约 100 个实现算法的模板函数，比如算法 `for_each` 将为指定序列中的每一个元素调用指定的函数，`stable_sort` 以你所指定的规则对序列进行稳定性排序等等。这样一来，只要熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

算法部分主要由头文件 `<algorithm>`、`<numeric>` 和 `<functional>` 组成。`<algorithm>` 是所有 STL 头文件中最大的一个（尽管它很好理解），它是由一大堆模板函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。`<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。`<functional>` 中则定义了一些模板类，用以声明函数对象。

10.1.5 C++ 标准库

C++ 强大的功能来源于其丰富的类库及库函数资源。C++ 标准库的内容总共在 50 个标准头文件中定义。在 C++ 开发中，要尽可能地利用标准库完成。这样做的直接好处包括：（1）成本：已经作为标准提供，何苦再花费时间、人力重新开发呢；（2）质量：标准库的都是经过严格测试的，正确性有保证；（3）效率：关于人的效率已经体现在成本中了，关于代码的执行效率要相信实现标准库的大牛们的水平；（4）良好的编程风格：采用行业中普遍的做法进行开发。

在 C++ 程序设计课程中，尤其是作为第一门程序设计课程，我们注重了语法、语言的机制等方面的内容。程序设计能力的培养有个过程，跨过基本的原理性知识直接进入工程

中的普遍做法，由于跨度决定了其难度。再者，在掌握了基本原理的基础上，在认识标准库的问题上完全可以凭借实践，逐步地掌握。标准库的学习不需要认认真真地读书，需要的是在了解概貌的情况下，在实践中深入。

这个任务就是要知道 C++ 程序设计课程中不讲，但对程序设计又很重要的这部分内容。至少我们要能先回答出“有什么”的问题。

C++ 标准库的内容分为 10 类，分别是（建议在阅读中，将你已经用过或听说过的头文件划出来）：

C1. 标准库中与语言支持功能相关的头文件

| 头文件 | 描 述 |
|-------------|--|
| <cstdint> | 定义宏 NULL 和 offsetof，以及其他标准类型 size_t 和 ptrdiff_t。与对应的标准 C 头文件的区别是，NULL 是 C++ 空指针常量的补充定义，宏 offsetof 接受结构或者联合类型参数，只要他们没有成员指针类型的非静态成员即可。 |
| <limits> | 提供与基本数据类型相关的定义。例如，对于每个数值数据类型，它定义了可以表示出来的最大值和最小值以及二进制数字的位数。 |
| <climits> | 提供与基本整数数据类型相关的 C 样式定义。这些信息的 C++ 样式定义在 <limits> 中 |
| <cfloat> | 提供与基本浮点型数据类型相关的 C 样式定义。这些信息的 C++ 样式定义在 <limits> 中 |
| <cstdlib> | 提供支持程序启动和终止的宏和函数。这个头文件还声明了许多其他杂项函数，例如搜索和排序函数，从字符串转换为数值等函数。它与对应的标准 C 头文件 stdlib.h 不同，定义了 abort(void)。abort() 函数还有额外的功能，它不为静态或自动对象调用析构函数，也不调用传给 atexit() 函数的函数。它还定义了 exit() 函数的额外功能，可以释放静态对象，以注册的逆序调用用 atexit() 注册的函数。清除并关闭所有 打开的 C 流，把控制权返回给主机环境。 |
| <new> | 支持动态内存分配 |
| <typeinfo> | 支持变量在运行期间的类型标识 |
| <exception> | 支持异常处理，这是处理程序中可能发生的错误的一种方式 |
| <cstdarg> | 支持接受数量可变的参数的函数。即在调用函数时，可以给函数传送数量不等的参数项。它定义了宏 va_arg、va_end、va_start 以及 va_list 类型 |
| <setjmp> | 为 C 样式的非本地跳跃提供函数。这些函数在 C++ 中不常用 |
| <csignal> | 为中断处理提供 C 样式支持 |

C2. 支持流输入/输出的头文件

| 头文件 | 描 述 |
|------------|--|
| <iostream> | 支持标准流 cin、cout、cerr 和 clog 的输入和输出，它还支持多字节字符标准流 wcin、wcout、wcerr 和 wclog。 |
| <iomanip> | 提供操纵程序，允许改变流的状态，从而改变输出的格式。 |
| <ios> | 定义 iostream 的基类 |
| <istream> | 为管理输入流缓存区的输入定义模板类 |
| <ostream> | 为管理输出流缓存区的输出定义模板类 |
| <sstream> | 支持字符串的流输入输出 |
| <fstream> | 支持文件的流输入输出 |
| <iosfwd> | 为输入输出对象提供向前的声明 |

| | |
|-------------|-------------------|
| <streambuf> | 支持流输入和输出的缓存 |
| <stdio> | 为标准流提供 C 样式的输入和输出 |
| <wchar> | 支持多字节字符的 C 样式输入输出 |

C3. 与诊断功能相关的头文件

| 头文件 | 描 述 |
|-------------|-------------------|
| <stdexcept> | 定义标准异常。异常是处理错误的方式 |
| <cassert> | 定义断言宏，用于检查运行期间的情形 |
| <cerrno> | 支持 C 样式的错误信息 |

C4. 定义工具函数的头文件

| 头文件 | 描 述 |
|--------------|--|
| <utility> | 定义重载的关系运算符，简化关系运算符的写入，它还定义了 pair 类型，该类型是一种模板类型，可以存储一对值。这些功能在库的其他地方使用 |
| <functional> | 定义了许多函数对象类型和支持函数对象的功能，函数对象是支持 operator() () 函数调用运算符的任意对象 |
| <memory> | 给容器、管理内存的函数和 auto_ptr 模板类定义标准内存分配器 |
| <ctime> | 支持系统时钟函数 |

C5. 支持字符串处理的头文件

| 头文件 | 描 述 |
|-----------|--|
| <string> | 为字符串类型提供支持和定义，包括单字节字符串 (由 char 组成) 的 string 和多字节字符串 (由 wchar_t 组成) |
| <cctype> | 单字节字符类别 |
| <cwctype> | 多字节字符类别 |
| <cstring> | 为处理非空字节序列和内存块提供函数。这不同于对应的标准 C 库头文件，几个 C 样式字符串的一般 C 库函数被返回值为 const 和非 const 的函数对替代了 |
| <wchar> | 为处理、执行 I/O 和转换多字节字符序列提供函数，这不同于对应的标准 C 库头文件，几个多字节 C 样式字符串操作的一般 C 库函数被返回值为 const 和非 const 的函数对替代了。 |
| <stdlib> | 为把单字节字符串转换为数值、在多字节字符和多字节字符串之间转换提供函数 |

C6. 定义容器类的模板的头文件

| | |
|----------|--|
| <vector> | 定义 vector 序列模板，这是一个大小可以重新设置的数组类型，比普通数组更安全、更灵活 |
| <list> | 定义 list 序列模板，这是一个序列的链表，常常在任意位置插入和删除元素 |
| <deque> | 定义 deque 序列模板，支持在开始和结尾的高效插入和删除操作 |
| <queue> | 为队列 (先进先出) 数据结构定义序列适配器 queue 和 priority_queue |
| <stack> | 为堆栈 (后进先出) 数据结构定义序列适配器 stack |
| <map> | map 是一个关联容器类型，允许根据键值是唯一的，且按照升序存储。multimap 类似于 map，但键不是唯一的。 |
| <set> | set 是一个关联容器类型，用于以升序方式存储唯一值。multiset 类似于 set，但是值不必是唯一的。 |
| <bitset> | 为固定长度的位序列定义 bitset 模板，它可以看作固定长度的紧凑型 bool 数组 |

C7. 支持迭代器的头文件

| 头文件 | 描 述 |
|------------|-------------|
| <iterator> | 给迭代器提供定义和支持 |

C8. 有关算法的头文件

| 头文件 | 描 述 |
|-------------|--|
| <algorithm> | 提供一组基于算法的函数，包括置换、排序、合并和搜索 |
| <cstdlib> | 声明 C 标准库函数 bsearch() 和 qsort()，进行搜索和排序 |
| <ciso646> | 允许在代码中使用 and 代替&& |

C9. 有关数值操作的头文件

| 头文件 | 描 述 |
|------------|--|
| <complex> | 支持复杂数值的定义和操作 |
| <valarray> | 支持数值矢量的操作 |
| <numeric> | 在数值序列上定义一组一般数学操作，例如 accumulate 和 inner_product |
| <cmath> | 这是 C 数学库，其中还附加了重载函数，以支持 C++ 约定 |
| <cstdlib> | 提供的函数可以提取整数的绝对值，对整数进行取余数操作 |

C10. 有关本地化的头文件

| 头文件 | 描 述 |
|-----------|-----------------------------|
| <locale> | 提供的本地化包括字符类别、排序序列以及货币和日期表示。 |
| <clocale> | 对本地化提供 C 样式支持 |

C++ 标准库的所有头文件都没有扩展名。C++ 标准库以 <cname> 形式的标准头文件提供。在 <cname> 形式标准的头文件中，与宏相关的名称在全局作用域中定义，其他名称在 std 命名空间中声明。在 C++ 中还可以使用 name.h 形式的标准 C 库头文件名

10.1.6 模板简要回顾

- ◆ 模板是实现代码重用机制的一种工具，实质就是实现类型参数化，即把类型定义为参数。
- ◆ C++ 提供两种模板：函数模板，类模板

函数模板的简介

- ◆ 函数模板就是建立一个通用的函数，其函数返回类型和形参类型不具体指定，而是用虚拟的类型来代表。
- ◆ 凡是函数体相同的函数都可以用函数模板来代替，不必定义多个函数，只需在模板中定义一次即可。
- ◆ 在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现了不同函数的功能。

类模板的简介

- ◆ 我们先来看一下下面这个类，求最大值的类
- ◆ 和函数模板一样，类模板就是建立一个通用类，其数据成员的类型、成员函数的返回类型和参数类型都可以不具体指定，而用虚拟的类型来代表。
- ◆ 当使用类模板建立对象时，系统会根据实参的类型取代类模板中的虚拟类型，从而实现不同类的功能。

10.2 容器

10.2.1 STL 的 string

1String 概念

- ◆ string 是 STL 的字符串类型，通常用来表示字符串。而在使用 string 之前，字符串通常是用 char* 表示的。string 与 char* 都可以用来表示字符串，那么二者有什么区别呢。

string 和 char* 的比较

- ◆ string 是一个类，char* 是一个指向字符的指针。
string 封装了 char*，管理这个字符串，是一个 char* 型的容器。
- ◆ string 不用考虑内存释放和越界。
string 管理 char* 所分配的内存。每一次 string 的复制，取值都由 string 类负责维护，不用担心复制越界和取值越界等。
- ◆ string 提供了一系列的字符串操作函数（这个等下会详讲）
查找 find，拷贝 copy，删除 erase，替换 replace，插入 insert

2string 的构造函数

- ◆ 默认构造函数：
string(); //构造一个空的字符串 string s1。
- ◆ 拷贝构造函数：
string(const string &str); //构造一个与 str 一样的 string。如 string s1(s2)。
- ◆ 带参数的构造函数
string(const char *s); //用字符串 s 初始化
string(int n, char c); //用 n 个字符 c 初始化

3string 的存取字符操作

- ◆ string 类的字符操作：
const char &operator[] (int n) const;
const char &at(int n) const;
char &operator[] (int n);

```
char &at(int n);
```

◆ `operator[]`和 `at()`均返回当前字符串中第 `n` 个字符，但二者是有区别的。

主要区别在于 `at()`在越界时会抛出异常，`[]`在刚好越界时会返回 `(char)0`，再继续越界时，编译器直接出错。如果你的程序希望可以通过 `try,catch` 捕获异常，建议采用 `at()`。

4 从 `string` 取得 `const char*`的操作

◆ `const char *c_str() const;` //返回一个以 `'\0'`结尾的字符串的首地址

5 把 `string` 拷贝到 `char*`指向的内存空间的操作

◆ `int copy(char *s, int n, int pos=0) const;`

把当前串中以 `pos` 开始的 `n` 个字符拷贝到以 `s` 为起始位置的字符数组中，返回实际拷贝的数目。注意要保证 `s` 所指向的空间足够大以容纳当前字符串，不然会越界。

6 `string` 的长度

```
int length() const; //返回当前字符串的长度。长度不包括字符串结尾的'\0'。
```

```
bool empty() const; //当前字符串是否为空
```

7 `string` 的赋值

```
string &operator=(const string &s); //把字符串 s 赋给当前的字符串
```

```
string &assign(const char *s); //把字符串 s 赋给当前的字符串
```

```
string &assign(const char *s, int n); //把字符串 s 的前 n 个字符赋给当前的字符串
```

```
string &assign(const string &s); //把字符串 s 赋给当前字符串
```

```
string &assign(int n, char c); //用 n 个字符 c 赋给当前字符串
```

```
string &assign(const string &s, int start, int n); //把字符串 s 中从 start 开始的 n 个字符赋给当前字符串
```

8 `string` 字符串连接

```
string &operator+=(const string &s); //把字符串 s 连接到当前字符串结尾
```

```
string &operator+=(const char *s); //把字符串 s 连接到当前字符串结尾
```

```
string &append(const char *s); //把字符串 s 连接到当前字符串结尾
```

```
string &append(const char *s, int n); //把字符串 s 的前 n 个字符连接到当前字符串结尾
```

```
string &append(const string &s); //同 operator+=()
```

```
string &append(const string &s, int pos, int n); //把字符串 s 中从 pos 开始的 n 个字符连接到当前字符串结尾
```

```
string &append(int n, char c); //在当前字符串结尾添加 n 个字符 c
```

9string 的比较

```
int compare(const string &s) const; //与字符串 s 比较
int compare(const char *s) const;  //与字符串 s 比较
```

compare 函数在>时返回 1, <时返回 -1, ==时返回 0。比较区分大小写, 比较时参考字典顺序, 排越前面的越小。大写的 A 比小写的 a 小。

10string 的子串

```
string substr(int pos=0, int n=npos) const;    //返回由 pos 开始的 n 个字符组成的子字符串
```

11string 的查找 和 替换

查找

```
int find(char c,int pos=0) const; //从 pos 开始查找字符 c 在当前字符串的位置
int find(const char *s, int pos=0) const; //从 pos 开始查找字符串 s 在当前字符串的位置
int find(const string &s, int pos=0) const; //从 pos 开始查找字符串 s 在当前字符串中的位置
find 函数如果查找不到, 就返回-1
int rfind(char c, int pos=npos) const; //从 pos 开始从后向前查找字符 c 在当前字符串中的位置
int rfind(const char *s, int pos=npos) const;
int rfind(const string &s, int pos=npos) const;
//rfind 是反向查找的意思, 如果查找不到, 返回-1
```

替换

```
string &replace(int pos, int n, const char *s);//删除从 pos 开始的 n 个字符, 然后在 pos 处插入串 s
string &replace(int pos, int n, const string &s); //删除从 pos 开始的 n 个字符, 然后在 pos 处插入串 s
void swap(string &s2); //交换当前字符串与 s2 的值
```

//4 字符串的查找和替换

```
void main25()
{
    string s1 = "wbm hello wbm 111 wbm 222 wbm 333";
    size_t index = s1.find("wbm", 0);
    cout << "index: " << index;

    //求 itcast 出现的次数
    size_t offindex = s1.find("wbm", 0);
    while (offindex != string::npos)
```

```
{
    cout << "在下标 index: " << offindex << "找到 wbm\n";
    offindex = offindex + 1;
    offindex = s1.find("wbm", offindex);
}

//替换
string s2 = "wbm hello wbm 111 wbm 222 wbm 333";
s2.replace(0, 3, "wbm");
cout << s2 << endl;

//求 itcast 出现的次数
offindex = s2.find("wbm", 0);
while (offindex != string::npos)
{
    cout << "在下标 index: " << offindex << "找到 wbm\n";
    s2.replace(offindex, 3, "WBM");
    offindex = offindex + 1;
    offindex = s1.find("wbm", offindex);
}
cout << "替换以后的 s2: " << s2 << endl;
}
```

12String 的区间删除和插入

```
string &insert(int pos, const char *s);
string &insert(int pos, const string &s);
//前两个函数在 pos 位置插入字符串 s
string &insert(int pos, int n, char c); //在 pos 位置 插入 n 个字符 c
```

```
string &erase(int pos=0, int n=npos); //删除 pos 开始的 n 个字符，返回修改后的字符串
```

13string 算法相关

```
void main27()
{
    string s2 = "AAAbbb";
    transform(s2.begin(), s2.end(), s2.begin(), toupper);
    cout << s2 << endl;

    string s3 = "AAAbbb";
    transform(s3.begin(), s3.end(), s3.begin(), tolower);
}
```

```
    cout << s3 << endl;
}
```

10.2.2 Vector 容器

1 Vector 容器简介

- ◆ vector 是将元素置于一个动态数组中加以管理的容器。
- ◆ vector 可以随机存取元素（支持索引值直接存取，用[]操作符或 at()方法，这个等下会详讲）。
- vector 尾部添加或移除元素非常快速。但是在中部或头部插入元素或移除元素比较费时

2 vector 对象的默认构造

vector 采用模板类实现，vector 对象的默认构造形式

```
vector<T> vecT;
```

```
vector<int> vecInt;           //一个存放 int 的 vector 容器。
vector<float> vecFloat;      //一个存放 float 的 vector 容器。
vector<string> vecString;    //一个存放 string 的 vector 容器。
...                          //尖括号内还可以设置指针类型或自定义类型。
Class CA{};
vector<CA*> vecpCA;          //用于存放 CA 对象的指针的 vector 容器。
vector<CA> vecCA;            //用于存放 CA 对象的 vector 容器。由于容器元素的存放是按值复制的方式进行的，所以此时 CA 必须提供 CA 的拷贝构造函数，以保证 CA 对象间拷贝正常。
```

3 vector 对象的带参数构造

理论知识

- ◆ vector(begin,end); //构造函数将[begin, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- ◆ vector(n,elem); //构造函数将 n 个 elem 拷贝给本身。
- ◆ vector(const vector &vec); //拷贝构造函数

```
int iArray[] = {0,1,2,3,4};
vector<int> vecIntA( iArray, iArray+5 );
```

```
vector<int> vecIntB ( vecIntA.begin(), vecIntA.end() ); //用构造函数初始化容器 vecIntB
vector<int> vecIntB ( vecIntA.begin(), vecIntA.begin()+3 );
vector<int> vecIntC(3,9); //此代码运行后，容器 vecIntB 就存放 3 个元素，每个元素的值是 9。
```



```
vector<int> vecIntD(vecIntA);
```

4vector 的赋值

理论知识

- ◆ `vector.assign(beg,end);` //将`[beg, end)`区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- ◆ `vector.assign(n,elem);` //将 `n` 个 `elem` 拷贝赋值给本身。
- ◆ `vector& operator=(const vector &vec);` //重载等号操作符
- ◆ `vector.swap(vec);` // 将 `vec` 与本身的元素互换。

```
vector<int> vecIntA, vecIntB, vecIntC, vecIntD;
```

```
int iArray[] = {0,1,2,3,4};
```

```
vecIntA.assign(iArray,iArray+5);
```

```
vecIntB.assign( vecIntA.begin(), vecIntA.end() ); //用其它容器的迭代器作参数。
```

```
vecIntC.assign(3,9);
```

```
vector<int> vecIntD;
```

```
vecIntD = vecIntA;
```

```
vecIntA.swap(vecIntD);
```

5vector 的大小

理论知识

- ◆ `vector.size();` //返回容器中元素的个数
- ◆ `vector.empty();` //判断容器是否为空
- ◆ `vector.resize(num);` //重新指定容器的长度为 `num`，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
- ◆ `vector.resize(num, elem);` //重新指定容器的长度为 `num`，若容器变长，则以 `elem` 值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

例如 `vecInt` 是 `vector<int>` 声明的容器，现已包含 1,2,3 元素。

```
int iSize = vecInt.size(); //iSize == 3;
```

```
bool bEmpty = vecInt.empty(); // bEmpty == false;
```

执行 `vecInt.resize(5);` //此时里面包含 1,2,3,0,0 元素。

再执行 `vecInt.resize(8,3);` //此时里面包含 1,2,3,0,0,3,3,3 元素。

再执行 `vecInt.resize(2);` //此时里面包含 1,2 元素。

6vector 末尾的添加移除操作

```
vector<int> vecInt;
vecInt.push_back(1); //在容器尾部加入一个元素
vecInt.push_back(3); //移除容器中最后一个元素
vecInt.push_back(5);
vecInt.push_back(7);
vecInt.push_back(9);
vecInt.pop_back();
vecInt.pop_back();
//{5,7,9}
```

7vector 的数据存取

理论知识

```
vec.at(idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range 异常。
vec[idx];    //返回索引 idx 所指的数据，越界时，运行直接报错
```

```
vector<int> vecInt; //假设包含 1,3,5,7,9
vecInt.at(2) == vecInt[2]; //5
vecInt.at(2) = 8; 或 vecInt[2] = 8;
vecInt 就包含 1,3,8,7,9 值
```

```
int iF = vector.front(); //iF==1
int iB = vector.back();  //iB==9
vector.front() = 11; //vecInt 包含{11,3,8,7,9}
vector.back() = 19; //vecInt 包含{11,3,8,7,19}
```

8 迭代器基本原理

- ◆ 迭代器是一个“可遍历 STL 容器内全部或部分元素”的对象。
- ◆ 迭代器指出容器中的一个特定位置。
- ◆ 迭代器就如同一个指针。
- ◆ 迭代器提供对一个容器中的对象的访问方法，并且可以定义了容器中对象的范围。
- ◆ 这里大概介绍一下迭代器的类别。

输入迭代器：也有叫法称之为“只读迭代器”，它从容器中读取元素，只能一次读入一个元素向前移动，只支持一遍算法，同一个输入迭代器不能两遍遍历一个序列。

输出迭代器：也有叫法称之为“只写迭代器”，它往容器中写入元素，只能一次写入一个元素向前移动，只支持一遍算法，同一个输出迭代器不能两遍遍历一个序列。

正向迭代器：组合输入迭代器和输出迭代器的功能，还可以多次解析一个迭代器指定的位置，可以对一个值进行多次读/写。

双向迭代器：组合正向迭代器的功能，还可以通过--操作符向后移动位置。

随机访问迭代器：组合双向迭代器的功能，还可以向前向后跳过任意个位置，可以直接访问容器中任何位置的元素。

- ◆ 目前本系列教程所用到的容器，都支持双向迭代器或随机访问迭代器，下面将会详细介绍这两个类别的迭代器。

9 双向迭代器与随机访问迭代器

双向迭代器支持的操作：

```
it++, ++it,    it--,  --it, *it,  itA = itB,
itA == itB, itA != itB
```

其中 list, set, multiset, map, multimap 支持双向迭代器。

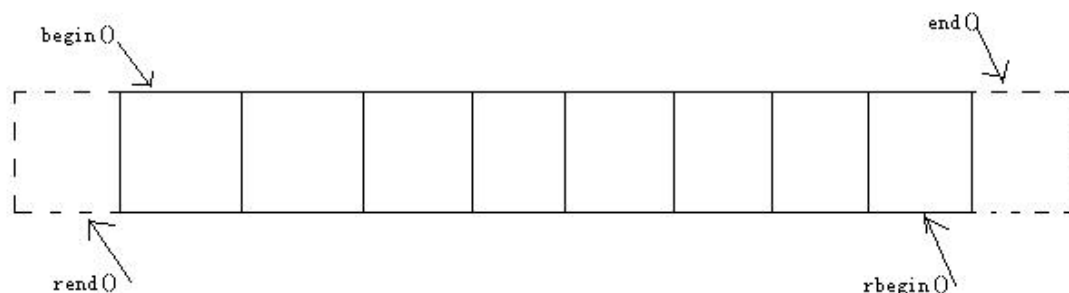
随机访问迭代器支持的操作：

在双向迭代器的操作基础上添加

```
it+=i,  it-=i,  it+i(或 it=it+i), it[i],
itA<itB, itA<=itB, itA>itB, itA>=itB 的功能。
```

其中 vector, deque 支持随机访问迭代器。

10 vector 与迭代器的配合使用



```
vector<int>  vecInt; //假设包含 1,3,5,7,9 元素
vector<int>::iterator it;    //声明容器 vector<int>的迭代器。
it = vecInt.begin();    // *it == 1
++it;                  //或者 it++;  *it == 3 ，前++的效率比后++的效率，前++返回引用，后++返回值。
it += 2;               // *it == 7
it = it+1;             // *it == 9
++it;                  // it == vecInt.end(); 此时不能再执行 *it,会出错!
```

正向遍历：

```
for(vector<int>::iterator it=vecInt.begin(); it!=vecInt.end(); ++it)
{
    int item = *it;
    cout << item;    //或直接使用    cout << *it;
}
```

这样子便打印出 1 3 5 7 9

逆向遍历:

```
for(vector<int>::reverse_iterator rit=vecInt.rbegin(); rit!=vecInt.rend(); ++rit)    //注意,小括号内仍是++rit
```

```
{
    int item = *rit;
    cout << item;    //或直接使用 cout << *rit;
}
```

此时将打印出 9,7,5,3,1

注意,这里迭代器的声明采用 `vector<int>::reverse_iterator`, 而非 `vector<int>::iterator`。

迭代器还有其它两种声明方法:

`vector<int>::const_iterator` 与 `vector<int>::const_reverse_iterator`

以上两种分别是 `vector<int>::iterator` 与 `vector<int>::reverse_iterator` 的只读形式,使用这两种迭代器时,不会修改到容器中的值。

备注: 不过容器中的 `insert` 和 `erase` 方法仅接受这四种类型中的 `iterator`, 其它三种不支持。

《Effective STL》建议我们尽量使用 `iterator` 取代 `const_iterator`、`reverse_iterator` 和 `const_reverse_iterator`。

11vector 的插入

理论知识

- ◆ `vector.insert(pos,elem);` //在 `pos` 位置插入一个 `elem` 元素的拷贝,返回新数据的位置。
- ◆ `vector.insert(pos,n,elem);` //在 `pos` 位置插入 `n` 个 `elem` 数据,无返回值。
- ◆ `vector.insert(pos,beg,end);` //在 `pos` 位置插入`[beg,end)`区间的数据,无返回值

简单案例

```
vector<int> vecA;
vector<int> vecB;

vecA.push_back(1);
vecA.push_back(3);
vecA.push_back(5);
vecA.push_back(7);
vecA.push_back(9);

vecB.push_back(2);
vecB.push_back(4);
vecB.push_back(6);
```

```
vecB.push_back(8);

vecA.insert(vecA.begin(), 11);    //{11, 1, 3, 5, 7, 9}
vecA.insert(vecA.begin()+1,2,33);    //{11,33,33,1,3,5,7,9}
vecA.insert(vecA.begin() , vecB.begin() , vecB.end() ); //{2,4,6,8,11,33,33,1,3,5,7,9}
```

12vector 的删除

理论知识

- ◆ `vector.clear();` //移除容器的所有数据
- ◆ `vec.erase(beg,end);` //删除**[beg,end)**区间的数据，返回下一个数据的位置。
- ◆ `vec.erase(pos);` //删除 `pos` 位置的数据，返回下一个数据的位置。

简单案例:

删除区间内的元素

`vecInt` 是用 `vector<int>` 声明的容器，现已包含按顺序的 1,3,5,6,9 元素。

```
vector<int>::iterator itBegin=vecInt.begin()+1;
```

```
vector<int>::iterator itEnd=vecInt.begin()+2;
```

```
vecInt.erase(itBegin,itEnd);
```

//此时容器 `vecInt` 包含按顺序的 1,6,9 三个元素。

假设 `vecInt` 包含 1,3,2,3,3,3,4,3,5,3，删除容器中等于 3 的元素

```
for(vector<int>::iterator it=vecInt.begin(); it!=vecInt.end(); )    //小括号里不需写 ++it
```

```
{
    if(*it == 3)
    {
        it = vecInt.erase(it);    //以迭代器为参数，删除元素 3，并把数据删除后的
        //下一个元素位置返回给迭代器。
        //此时，不执行 ++it;
    }
    else
    {
        ++it;
    }
}
```

//删除 `vecInt` 的所有元素

```
vecInt.clear();    //容器为空
```

13vector 小结

这一讲，主要讲解如下要点：

容器的简介，容器的分类，各个容器的数据结构

vector, deque, list, set, multiset, map, multimap

容器 vector 的具体用法（包括迭代器的具体用法）。

vector 简介，vector 使用之前的准备，vector 对象的默认构造，vector 末尾的添加移除操作，vector 的数据存取，迭代器的简介，双向迭代器与随机访问迭代器

vector 与迭代器的配合使用，vector 对象的带参数构造，vector 的赋值，vector 的大小，vector 的插入，vector 的删除。

10.2.3 Deque 容器

Deque 简介

- ◆ deque 是“double-ended queue”的缩写，和 vector 一样都是 STL 的容器，deque 是双端数组，而 vector 是单端的。
- ◆ deque 在接口上和 vector 非常相似，在许多操作的地方可以直接替换。
- ◆ deque 可以随机存取元素（支持索引值直接存取，用[]操作符或 at()方法，这个等下会详讲）。
- ◆ deque 头部和尾部添加或移除元素都非常快速。但是在中部安插元素或移除元素比较费时。
- ◆ #include <deque>

deque 对象的默认构造

deque 采用模板类实现，deque 对象的默认构造形式：deque<T> deqT;

deque<int> deqInt; //一个存放 int 的 deque 容器。

deque<float> deqFloat; //一个存放 float 的 deque 容器。

deque<string> deqString; //一个存放 string 的 deque 容器。

...

//尖括号内还可以设置指针类型或自定义类型。

deque 末尾的添加移除操作

理论知识:

- ◆ deque.push_back(elem); //在容器尾部添加一个数据
- ◆ deque.push_front(elem); //在容器头部插入一个数据
- ◆ deque.pop_back(); //删除容器最后一个数据
- ◆ deque.pop_front(); //删除容器第一个数据

```
deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
```

```
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);
deqInt.pop_front();
deqInt.pop_front();
deqInt.push_front(11);
deqInt.push_front(13);
deqInt.pop_back();
deqInt.pop_back();
//deqInt { 13,11,5}
```

deque 的数据存取

理论知识:

- ◆ deque.at(idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range。
- ◆ deque[idx]; //返回索引 idx 所指的数据，如果 idx 越界，不抛出异常，直接出错。
- ◆ deque.front(); //返回第一个数据。
- ◆ deque.back(); //返回最后一个数据

```
deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);
```

```
int iA = deqInt.at(0);    //1
int iB = deqInt[1];       //3
deqInt.at(0) = 99;        //99
deqInt[1] = 88;           //88
```

```
int iFront = deqInt.front(); //99
int iBack = deqInt.back();   //9
deqInt.front() = 77;         //77
deqInt.back() = 66;          //66
```

deque 与迭代器

理论知识

- ◆ deque.begin(); //返回容器中第一个元素的迭代器。
- ◆ deque.end(); //返回容器中最后一个元素之后的迭代器。
- ◆ deque.rbegin(); //返回容器中倒数第一个元素的迭代器。
- ◆ deque.rend(); //返回容器中倒数最后一个元素之后的迭代器。

```

deque<int> deqInt;
deqInt.push_back(1);
deqInt.push_back(3);
deqInt.push_back(5);
deqInt.push_back(7);
deqInt.push_back(9);

for (deque<int>::iterator it=deqInt.begin(); it!=deqInt.end(); ++it)
{
    cout << *it;
    cout << " ";
}
// 1 3 5 7 9

for (deque<int>::reverse_iterator rit=deqInt.rbegin(); rit!=deqInt.rend(); ++rit)
{
    cout << *rit;
    cout << " ";
}
// 9 7 5 3 1

```

deque 对象的带参数构造

理论知识

- ◆ `deque(beg,end);` //构造函数将`[beg, end)`区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- ◆ `deque(n,elem);` //构造函数将 `n` 个 `elem` 拷贝给本身。
- ◆ `deque(const deque &deq);` //拷贝构造函数。

```

deque<int> deqIntA;
deqIntA.push_back(1);
deqIntA.push_back(3);
deqIntA.push_back(5);
deqIntA.push_back(7);
deqIntA.push_back(9);

deque<int> deqIntB(deqIntA.begin(),deqIntA.end());    //1 3 5 7 9
deque<int> deqIntC(5,8);                               //8 8 8 8 8
deque<int> deqIntD(deqIntA);                           //1 3 5 7 9

```


deque 的赋值

理论知识

- ◆ `deque.assign(beg,end);` //将`[beg, end)`区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- ◆ `deque.assign(n,elem);` //将 `n` 个 `elem` 拷贝赋值给本身。
- ◆ `deque& operator=(const deque &deq);` //重载等号操作符
- ◆ `deque.swap(deq);` // 将 `vec` 与本身的元素互换

```
deque<int> deqIntA,deqIntB,deqIntC,deqIntD;
deqIntA.push_back(1);
deqIntA.push_back(3);
deqIntA.push_back(5);
deqIntA.push_back(7);
deqIntA.push_back(9);

deqIntB.assign(deqIntA.begin(),deqIntA.end()); // 1 3 5 7 9

deqIntC.assign(5,8);                          //8 8 8 8 8

deqIntD = deqIntA;                             //1 3 5 7 9

deqIntC.swap(deqIntD);                         //互换
```

deque 的大小

理论知识

- ◆ `deque.size();` //返回容器中元素的个数
- ◆ `deque.empty();` //判断容器是否为空
- ◆ `deque.resize(num);` //重新指定容器的长度为 `num`，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
- ◆ `deque.resize(num, elem);` //重新指定容器的长度为 `num`，若容器变长，则以 `elem` 值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

```
deque<int> deqIntA;
deqIntA.push_back(1);
deqIntA.push_back(3);
deqIntA.push_back(5);

int iSize = deqIntA.size(); //3

if (!deqIntA.empty())
{
```

```
    deqIntA.resize(5);        //1 3 5 0 0
    deqIntA.resize(7,1);      //1 3 5 0 0 1 1
    deqIntA.resize(2);        //1 3
}
```

deque 的插入

理论知识

- ◆ `deque.insert(pos,elem);` //在 `pos` 位置插入一个 `elem` 元素的拷贝，返回新数据的位置。
- ◆ `deque.insert(pos,n,elem);` //在 `pos` 位置插入 `n` 个 `elem` 数据，无返回值。
- ◆ `deque.insert(pos,beg,end);` //在 `pos` 位置插入`[beg,end)`区间的数据，无返回值。

```
deque<int> deqA;
deque<int> deqB;
```

```
deqA.push_back(1);
deqA.push_back(3);
deqA.push_back(5);
deqA.push_back(7);
deqA.push_back(9);
```

```
deqB.push_back(2);
deqB.push_back(4);
deqB.push_back(6);
deqB.push_back(8);
```

```
deqA.insert(deqA.begin(), 11);        //{11, 1, 3, 5, 7, 9}
deqA.insert(deqA.begin()+1,2,33);     //{11,33,33,1,3,5,7,9}
deqA.insert(deqA.begin(), deqB.begin(), deqB.end()); //{2,4,6,8,11,33,33,1,3,5,7,9}
```

deque 的删除

理论知识

- ◆ `deque.clear();` //移除容器的所有数据
- ◆ `deque.erase(beg,end);` //删除`[beg,end)`区间的数据，返回下一个数据的位置。
- ◆ `deque.erase(pos);` //删除 `pos` 位置的数据，返回下一个数据的位置。

删除区间内的元素

`deqInt` 是用 `deque<int>` 声明的容器，现已包含按顺序的 1,3,5,6,9 元素。

```
deque<int>::iterator itBegin=deqInt.begin()+1;
deque<int>::iterator itEnd=deqInt.begin()+3;
deqInt.erase(itBegin,itEnd);
```

//此时容器 `deqInt` 包含按顺序的 1,6,9 三个元素。

假设 `deqInt` 包含 1,3,2,3,3,3,4,3,5,3，删除容器中等于 3 的元素

```
for(deque<int>::iterator it=deqInt.begin(); it!=deqInt.end(); )    //小括号里不需写 ++it
{
    if(*it == 3)
    {
        it = deqInt.erase(it);    //以迭代器为参数，删除元素 3，并把数据删除后的
        //此时，不执行 ++it;
        //下一个元素位置返回给迭代器。
    }
    else
    {
        ++it;
    }
}

//删除 deqInt 的所有元素
deqInt.clear();    //容器为空
```

10.2.4 stack 容器

Stack 简介

- ◆ `stack` 是堆栈容器，是一种“先进后出”的容器。
- ◆ `stack` 是简单地装饰 `deque` 容器而成为另外的一种容器。
- ◆ `#include <stack>`

stack 对象的默认构造

`stack` 采用模板类实现，`stack` 对象的默认构造形式：`stack <T> stkT;`

`stack <int> stkInt;` //一个存放 `int` 的 `stack` 容器。

`stack <float> stkFloat;` //一个存放 `float` 的 `stack` 容器。

`stack <string> stkString;` //一个存放 `string` 的 `stack` 容器。

...

//尖括号内还可以设置指针类型或自定义类型。

stack 的 `push()` 与 `pop()` 方法

`stack.push(elem);` //往栈头添加元素

```
stack.pop();    //从栈头移除第一个元素
```

```
stack<int> stkInt;  
stkInt.push(1);stkInt.push(3);stkInt.pop();  
stkInt.push(5);stkInt.push(7);  
stkInt.push(9);stkInt.pop();  
stkInt.pop();  
此时 stkInt 存放的元素是 1,5
```

stack 对象的拷贝构造与赋值

```
stack(const stack &stk);           //拷贝构造函数  
stack& operator=(const stack &stk); //重载等号操作符
```

```
stack<int> stkIntA;  
stkIntA.push(1);  
stkIntA.push(3);  
stkIntA.push(5);  
stkIntA.push(7);  
stkIntA.push(9);  
  
stack<int> stkIntB(stkIntA);        //拷贝构造  
stack<int> stkIntC;  
stkIntC = stkIntA;                  //赋值
```

stack 的数据存取

```
◆ stack.top();           //返回最后一个压入栈元素  
stack<int> stkIntA;  
stkIntA.push(1);  
stkIntA.push(3);  
stkIntA.push(5);  
stkIntA.push(7);  
stkIntA.push(9);  
  
int iTop = stkIntA.top();    //9  
stkIntA.top() = 19;         //19
```

stack 的大小

```
◆ stack.empty();         //判断堆栈是否为空  
◆ stack.size();          //返回堆栈的大小
```

```
stack<int> stkIntA;
stkIntA.push(1);
stkIntA.push(3);
stkIntA.push(5);
stkIntA.push(7);
stkIntA.push(9);

if (!stkIntA.empty())
{
    int iSize = stkIntA.size();    //5
}
```

10.2.5 Queue 容器

Queue 简介

- ◆ queue 是队列容器，是一种“先进先出”的容器。
- ◆ queue 是简单地装饰 deque 容器而成为另外一种容器。
- ◆ #include <queue>

queue 对象的默认构造

queue 采用模板类实现，queue 对象的默认构造形式：queue<T> queT; 如：

```
queue<int> queInt;           //一个存放 int 的 queue 容器。
queue<float> queFloat;      //一个存放 float 的 queue 容器。
queue<string> queString;    //一个存放 string 的 queue 容器。
```

...

//尖括号内还可以设置指针类型或自定义类型。

queue 的 push()与 pop()方法

```
queue.push(elem);    //往队尾添加元素
queue.pop();         //从队头移除第一个元素
```

```
queue<int> queInt;
queInt.push(1);queInt.push(3);
queInt.push(5);queInt.push(7);
queInt.push(9);queInt.pop();
```

```
queInt.pop();
```

此时 queInt 存放的元素是 5,7,9

queue 对象的拷贝构造与赋值

```
queue(const queue &que);           //拷贝构造函数  
queue& operator=(const queue &que); //重载等号操作符
```

```
queue<int> queIntA;  
queIntA.push(1);  
queIntA.push(3);  
queIntA.push(5);  
queIntA.push(7);  
queIntA.push(9);
```

```
queue<int> queIntB(queIntA);    //拷贝构造  
queue<int> queIntC;  
queIntC = queIntA;             //赋值
```

queue 的数据存取

- ◆ queue.back(); //返回最后一个元素
- ◆ queue.front(); //返回第一个元素

```
queue<int> queIntA;  
queIntA.push(1);  
queIntA.push(3);  
queIntA.push(5);  
queIntA.push(7);  
queIntA.push(9);
```

```
int iFront = queIntA.front(); //1  
int iBack = queIntA.back();   //9
```

```
queIntA.front() = 11;         //11  
queIntA.back() = 19;         //19
```

queue 的大小

- ◆ queue.empty(); //判断队列是否为空
 - ◆ queue.size(); //返回队列的大小
- ```
queue<int> queIntA;
```

```
queIntA.push(1);
queIntA.push(3);
queIntA.push(5);
queIntA.push(7);
queIntA.push(9);

if (!queIntA.empty())
{
 int iSize = queIntA.size(); //5
}
```

## 10.2.6 List 容器

### List 简介

- ◆ list 是一个**双向链表容器**，可高效地进行插入删除元素。
- ◆ list 不可以随机存取元素，所以不支持 `at(pos)` 函数与 `[]` 操作符。It++(ok) **it+5(err)**
- ◆ `#include <list>`

### list 对象的默认构造

list 采用模板类实现,对象的默认构造形式: `list<T> lstT;` 如:

```
list<int> lstInt; //定义一个存放 int 的 list 容器。
list<float> lstFloat; //定义一个存放 float 的 list 容器。
list<string> lstString; //定义一个存放 string 的 list 容器。
...
//尖括号内还可以设置指针类型或自定义类型。
```

### list 头尾的添加移除操作

- ◆ `list.push_back(elem);` //在容器尾部加入一个元素
- ◆ `list.pop_back();` //删除容器中最后一个元素
- ◆ `list.push_front(elem);` //在容器开头插入一个元素
- ◆ `list.pop_front();` //从容器开头移除第一个元素

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
```

```
lstInt.push_back(9);
lstInt.pop_front();
lstInt.pop_front();
lstInt.push_front(11);
lstInt.push_front(13);
lstInt.pop_back();
lstInt.pop_back();
// lstInt {13,11,5}
```

## list 的数据存取

- ◆ list.front(); //返回第一个元素。
- ◆ list.back(); //返回最后一个元素。

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
lstInt.push_back(9);

int iFront = lstInt.front();//1
int iBack = lstInt.back(); //9
lstInt.front() = 11; //11
lstInt.back() = 19; //19
```

## list 与迭代器

- ◆ list.begin(); //返回容器中第一个元素的迭代器。
- ◆ list.end(); //返回容器中最后一个元素之后的迭代器。
- ◆ list.rbegin(); //返回容器中倒数第一个元素的迭代器。
- ◆ list.rend(); //返回容器中倒数最后一个元素的后面的迭代器。

```
list<int> lstInt;
lstInt.push_back(1);
lstInt.push_back(3);
lstInt.push_back(5);
lstInt.push_back(7);
lstInt.push_back(9);

for (list<int>::iterator it=lstInt.begin(); it!=lstInt.end(); ++it)
{
 cout << *it;
```



```
 cout << " ";
 }

 for (list<int>::reverse_iterator rit=lstInt.rbegin(); rit!=lstInt.rend(); ++rit)
 {
 cout << *rit;
 cout << " ";
 }
```

## list 对象的带参数构造

- ◆ list(beg,end); //构造函数将[beg, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。
- ◆ list(n,elem); //构造函数将 n 个 elem 拷贝给本身。
- ◆ list(const list &lst); //拷贝构造函数。

```
list<int> lstIntA;
lstIntA.push_back(1);
lstIntA.push_back(3);
lstIntA.push_back(5);
lstIntA.push_back(7);
lstIntA.push_back(9);

list<int> lstIntB(lstIntA.begin(),lstIntA.end()); //1 3 5 7 9
list<int> lstIntC(5,8); //8 8 8 8 8
list<int> lstIntD(lstIntA); //1 3 5 7 9
```

## list 的赋值

- ◆ list.assign(beg,end); //将[beg, end)区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。
- ◆ list.assign(n,elem); //将 n 个 elem 拷贝赋值给本身。
- ◆ list& operator=(const list &lst); //重载等号操作符
- ◆ list.swap(lst); // 将 lst 与本身的元素互换。

```
list<int> lstIntA,lstIntB,lstIntC,lstIntD;
lstIntA.push_back(1);
lstIntA.push_back(3);
lstIntA.push_back(5);
lstIntA.push_back(7);
lstIntA.push_back(9);

lstIntB.assign(lstIntA.begin(),lstIntA.end()); //1 3 5 7 9
```

```
lstIntC.assign(5,8); //8 8 8 8 8
lstIntD = lstIntA; //1 3 5 7 9
lstIntC.swap(lstIntD); //互换
```

## list 的大小

- ◆ `list.size();` //返回容器中元素的个数
- ◆ `list.empty();` //判断容器是否为空
- ◆ `list.resize(num);` //重新指定容器的长度为 `num`，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
- ◆ `list.resize(num, elem);` //重新指定容器的长度为 `num`，若容器变长，则以 `elem` 值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

```
list<int> lstIntA;
lstIntA.push_back(1);
lstIntA.push_back(3);
lstIntA.push_back(5);

if (!lstIntA.empty())
{
 int iSize = lstIntA.size(); //3
 lstIntA.resize(5); //1 3 5 0 0
 lstIntA.resize(7,1); //1 3 5 0 0 1 1
 lstIntA.resize(2); //1 3
}
```

## list 的插入

- ◆ `list.insert(pos,elem);` //在 `pos` 位置插入一个 `elem` 元素的拷贝，返回新数据的位置。
- ◆ `list.insert(pos,n,elem);` //在 `pos` 位置插入 `n` 个 `elem` 数据，无返回值。
- ◆ `list.insert(pos,beg,end);` //在 `pos` 位置插入 `[beg,end)` 区间的数据，无返回值。

```
list<int> lstA;
list<int> lstB;

lstA.push_back(1);
lstA.push_back(3);
lstA.push_back(5);
lstA.push_back(7);
lstA.push_back(9);

lstB.push_back(2);
lstB.push_back(4);
```

```

lstB.push_back(6);
lstB.push_back(8);

lstA.insert(lstA.begin(), 11); //{11, 1, 3, 5, 7, 9}
lstA.insert(++lstA.begin(),2,33); //{11,33,33,1,3,5,7,9}
lstA.insert(lstA.begin(), lstB.begin(), lstB.end()); //{2,4,6,8,11,33,33,1,3,5,7,9}

```

## list 的删除

- ◆ `list.clear();`     //移除容器的所有数据
- ◆ `list.erase(beg,end);`   //删除**[beg,end)**区间的数据，返回下一个数据的位置。
- ◆ `list.erase(pos);`     //删除 `pos` 位置的数据，返回下一个数据的位置。
- ◆ `lst.remove(elem);`     //删除容器中所有与 `elem` 值匹配的元素。

### 删除区间内的元素

`lstInt` 是用 `list<int>` 声明的容器，现已包含按顺序的 1,3,5,6,9 元素。

```

list<int>::iterator itBegin=lstInt.begin();
++ itBegin;
list<int>::iterator itEnd=lstInt.begin();
++ itEnd;
++ itEnd;
++ itEnd;
lstInt.erase(itBegin,itEnd);
//此时容器 lstInt 包含按顺序的 1,6,9 三个元素。

```

假设 `lstInt` 包含 1,3,2,3,3,3,4,3,5,3，删除容器中等于 3 的元素的方法一

```

for(list<int>::iterator it=lstInt.begin(); it!=lstInt.end();) //小括号里不需写 ++it
{
 if(*it == 3)
 {
 it = lstInt.erase(it); //以迭代器为参数，删除元素 3，并把数据删除后的
 下一个元素位置返回给迭代器。
 //此时，不执行 ++it;
 }
 else
 {
 ++it;
 }
}

```

### 删除容器中等于 3 的元素的方法二

```
lstInt.remove(3);
```

删除 `lstInt` 的所有元素

```
lstInt.clear(); //容器为空
```

## list 的反序排列

- ◆ `lst.reverse();` //反转链表，比如 `lst` 包含 1,3,5 元素，运行此方法后，`lst` 就包含 5,3,1 元素。

```
list<int> lstA;
```

```
lstA.push_back(1);
```

```
lstA.push_back(3);
```

```
lstA.push_back(5);
```

```
lstA.push_back(7);
```

```
lstA.push_back(9);
```

```
lstA.reverse(); //9 7 5 3 1
```

## 小结:

- ◆ 一、容器 `deque` 的使用方法  
适合 在头尾添加移除元素。使用方法与 `vector` 类似。
- ◆ 二、容器 `queue`, `stack` 的使用方法  
适合队列，堆栈的操作方式。
- ◆ 三、容器 `list` 的使用方法  
适合在任意位置快速插入移除元素

## 10.2.7 优先级队列 `priority_queue`

- ❖ 最大值优先级队列、最小值优先级队列
- ❖ 优先级队列适配器 STL `priority_queue`
- ❖ 用来开发一些特殊的应用,请对 `stl` 的类库,多做扩展性学习

```
priority_queue<int, deque<int>> pq;
```

```
priority_queue<int, vector<int>> pq;
```

```
pq.empty()
```

```
pq.size()
```

```
pq.top()
```

```
pq.pop()
```

```
pq.push(item)
```

```
#include <iostream>
using namespace std;
#include "queue"
void main81()
{
 priority_queue<int> p1; //默认是 最大值优先级队列
 //priority_queue<int, vector<int>, less<int> > p1; //相当于这样写
 priority_queue<int, vector<int>, greater<int>> p2; //最小值优先级队列

 p1.push(33);
 p1.push(11);
 p1.push(55);
 p1.push(22);
 cout << "队列大小" << p1.size() << endl;
 cout << "队头" << p1.top() << endl;

 while (p1.size() > 0)
 {
 cout << p1.top() << " ";
 p1.pop();
 }
 cout << endl;

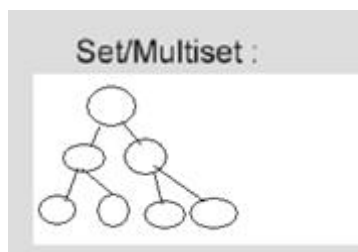
 cout << "测试 最小值优先级队列" << endl;
 p2.push(33);
 p2.push(11);
 p2.push(55);
 p2.push(22);
 while (p2.size() > 0)
 {
 cout << p2.top() << " ";
 p2.pop();
 }
}
```

## 10.2.8 Set 和 multiset 容器

### set/multiset 的简介

- ◆ set 是一个集合容器, 其中所包含的元素是唯一的, 集合中的元素按一定的顺序排列。元素插入过程是按排序规则插入, 所以不能指定插入位置。

- ◆ set 采用**红黑树**变体的数据结构实现，红黑树属于平衡二叉树。在插入操作和删除操作上比 vector 快。
- ◆ set 不可以直接存取元素。（不可以使用 at.(pos)与[]操作符）。
- ◆ multiset 与 set 的区别：set 支持唯一键值，每个元素值只能出现一次；而 multiset 中**同一值可以出现多次**。
- ◆ 不可以直接修改 set 或 multiset 容器中的元素值，因为该类容器是自动排序的。如果希望修改一个元素值，必须先删除原有的元素，再插入新的元素。
- ◆ #include <set>



## set/multiset 对象的默认构造

```
set<int> setInt; //一个存放 int 的 set 容器。
set<float> setFloat; //一个存放 float 的 set 容器。
set<string> setString; //一个存放 string 的 set 容器。
multiset<int> mulsetInt; //一个存放 int 的 multi set 容器。
multi set<float> multisetFloat; //一个存放 float 的 multi set 容器。
multi set<string> multisetString; //一个存放 string 的 multi set 容器。
```

## set 的插入与迭代器

- ◆ set.insert(elem); //在容器中插入元素。
- ◆ set.begin(); //返回容器中第一个数据的迭代器。
- ◆ set.end(); //返回容器中最后一个数据之后的迭代器。
- ◆ set.rbegin(); //返回容器中倒数第一个元素的迭代器。
- ◆ set.rend(); //返回容器中倒数最后一个元素的后面的迭代器。

```
set<int> setInt;
setInt.insert(3); setInt.insert(1); setInt.insert(5); setInt.insert(2);
for(set<int>::iterator it=setInt.begin(); it!=setInt.end(); ++it)
{
 int iltem = *it;
 cout << iltem; //或直接使用 cout << *it
}
//这样子便顺序输出 1 2 3 5。
```

set.rbegin()与 set.rend()。略。

## Set 集合的元素排序

- ◆ `set<int,less<int>> setIntA;` //该容器是按升序方式排列元素。
- ◆ `set<int,greater<int>> setIntB;` //该容器是按降序方式排列元素。
- ◆ `set<int>` 相当于 `set<int,less<int>>`。
- ◆ `less<int>`与 `greater<int>`中的 `int` 可以改成其它类型，该类型主要要跟 `set` 容纳的数据类型一致。
- ◆ 疑问 1: `less<>`与 `greater<>`是什么？
- ◆ 疑问 2: 如果 `set<>`不包含 `int` 类型，而是包含自定义类型，`set` 容器如何排序？
- ◆ 要解决如上两个问题，需要了解容器的函数对象，也叫伪函数，英文名叫 `functor`。
- ◆ 下面将讲解什么是 `functor`，`functor` 的用法。

使用 `stl` 提供的函数对象

```
set<int,greater<int>> setIntB;
setIntB.insert(3);
setIntB.insert(1);
setIntB.insert(5);
setIntB.insert(2);
此时容器 setIntB 就包含了按顺序的 5,3,2,1 元素
```

## 函数对象 functor 的用法

- ◆ 尽管函数指针被广泛用于实现函数回调，但 C++ 还提供了一个重要的实现回调函数的方法，那就是函数对象。
- ◆ `functor`，翻译成函数对象，伪函数，算符，是重载了 “`()`” 操作符的普通类对象。从语法上讲，它与普通函数行为类似。
- ◆ `greater<>`与 `less<>`就是函数对象。
- ◆ 下面举出 `greater<int>`的简易实现原理。

下面举出 `greater<int>`的简易实现原理。

```
struct greater
{
 bool operator() (const int& iLeft, const int& iRight)
 {
 return (iLeft>iRight); //如果是实现 less<int>的话，这边是写 return (iLeft<iRight);
 }
}
```

容器就是调用函数对象的 `operator()`方法去比较两个值的大小。

**题目：**学生包含学号，姓名属性，现要求任意插入几个学生对象到 `set` 容器中，使得容器中

的学生按学号的升序排序。

解：

//学生类

class CStudent

```
{
 public:
 CStudent(int iID, string strName)
 {
 m_iID = iID;
 m_strName = strName;
 }
 int m_iID; //学号
 string m_strName; //姓名
}
```

//为保持主题鲜明，本类不写拷贝构造函数，不类也不需要写拷贝构造函数。但大家仍要有考虑拷贝构造函数的习惯。

//函数对象

struct StuFunctor

```
{
 bool operator() (const CStudent &stu1, const CStudent &stu2)
 {
 return (stu1.m_iID<stu2.m_iID);
 }
}
```

//main 函数

void main()

```
{
 set<CStudent, StuFunctor> setStu;
 setStu.insert(CStudent(3,"小张"));
 setStu.insert(CStudent(1,"小李"));
 setStu.insert(CStudent(5,"小王"));
 setStu.insert(CStudent(2,"小刘"));
 //此时容器 setStu 包含了四个学生对象，分别是按姓名顺序的“小李”，“小刘”，
 “小张”，“小王”
}
```

## set 对象的拷贝构造与赋值

```
set(const set &st); //拷贝构造函数
```



```
set& operator=(const set &st); //重载等号操作符
set.swap(st); //交换两个集合容器
```

```
set<int> setIntA;
setIntA.insert(3);
setIntA.insert(1);
setIntA.insert(7);
setIntA.insert(5);
setIntA.insert(9);
```

```
set<int> setIntB(setIntA); //1 3 5 7 9
```

```
set<int> setIntC;
setIntC = setIntA; //1 3 5 7 9
```

```
setIntC.insert(6);
setIntC.swap(setIntA); //交换
```

## set 的大小

- ◆ set.size(); //返回容器中元素的数目
- ◆ set.empty();//判断容器是否为空

```
set<int> setIntA;
setIntA.insert(3);
setIntA.insert(1);
setIntA.insert(7);
setIntA.insert(5);
setIntA.insert(9);
```

```
if (!setIntA.empty())
{
 int iSize = setIntA.size(); //5
}
```

## set 的删除

- ◆ set.clear(); //清除所有元素
- ◆ set.erase(pos); //删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
- ◆ set.erase(beg,end); //删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
- ◆ set.erase(elem); //删除容器中值为 elem 的元素。

删除区间内的元素

setInt 是用 set<int> 声明的容器，现已包含按顺序的 1,3,5,6,9,11 元素。

```
set<int>::iterator itBegin=setInt.begin();
++ itBegin;
set<int>::iterator itEnd=setInt.begin();
++ itEnd;
++ itEnd;
++ itEnd;
setInt.erase(itBegin,itEnd);
//此时容器 setInt 包含按顺序的 1,6,9,11 四个元素。
```

删除容器中第一个元素

```
setInt.erase(setInt.begin()); //6,9,11
```

删除容器中值为 9 的元素

```
set.erase(9);
```

删除 setInt 的所有元素

```
setInt.clear(); //容器为空
```

## set 的查找

- ◆ set.find(elem); //查找 elem 元素，返回指向 elem 元素的迭代器。
- ◆ set.count(elem); //返回容器中值为 elem 的元素个数。对 set 来说，要么是 0，要么是 1。对 multiset 来说，值可能大于 1。
- ◆ set.lower\_bound(elem); //返回第一个  $\geq$  elem 元素的迭代器。
- ◆ set.upper\_bound(elem); // 返回第一个  $>$  elem 元素的迭代器。
- ◆ set.equal\_range(elem); //返回容器中与 elem 相等的上下限的两个迭代器。上限是闭区间，下限是开区间，如 [beg,end)。
- ◆
- ◆ 以上函数返回两个迭代器，而这两个迭代器被封装在 pair 中。
- ◆ 以下讲解 pair 的含义与使用方法。
- ◆

```
set<int> setInt;
setInt.insert(3);
setInt.insert(1);
setInt.insert(7);
setInt.insert(5);
setInt.insert(9);
```

```
set<int>::iterator itA = setInt.find(5);
int iA = *itA; //iA == 5
```

```
int iCount = setInt.count(5); //iCount == 1

set<int>::iterator itB = setInt.lower_bound(5);
set<int>::iterator itC = setInt.upper_bound(5);
int iB = *itB; //iB == 5
int iC = *itC; //iC == 7

pair< set<int>::iterator, set<int>::iterator > pairIt = setInt.equal_range(5); //pair 是什么?
```

## pair 的使用

- ◆ pair 译为对组，可以将两个值视为一个单元。
- ◆ pair<T1,T2>存放的两个值的类型，可以不一样，如 T1 为 int，T2 为 float。T1,T2 也可以是自定义类型。
- ◆ pair.first 是 pair 里面的第一个值，是 T1 类型。
- ◆ pair.second 是 pair 里面的第二个值，是 T2 类型。

```
set<int> setInt;
... //往 setInt 容器插入元素 1,3,5,7,9
pair< set<int>::iterator, set<int>::iterator > pairIt = setInt.equal_range(5);
set<int>::iterator itBeg = pairIt.first;
set<int>::iterator itEnd = pairIt.second;
//此时 *itBeg==5 而 *itEnd == 7
```

## 小结

- ◆ 一、容器 set/multiset 的使用方法；  
红黑树的变体，查找效率高，插入不能指定位置，插入时自动排序。
- ◆ 二、functor 的使用方法；  
类似于函数的功能，可用来自定义一些规则，如元素比较规则。
- ◆ 三、pair 的使用方法。  
对组，一个整体的单元，存放两个类型(T1,T2，T1 可与 T2 一样)的两个元素。

案例:

```
int x;
scanf("%ld",&x);
multiset<int> h;//建立一个multiset类型，变量名是h，h序列里面存的是int类型,初始h
为空
while(x!=0){
 h.insert(x);//将x插入h中
 scanf("%ld",&x);
}
```

```

pair< multiset<int>::iterator , multiset<int>::iterator > pairIt = h.equal_range(22);
multiset<int>::iterator itBeg = pairIt.first;
multiset<int>::iterator itEnd = pairIt.second;

int nBeg = *itBeg;
int nEnd = *itEnd;

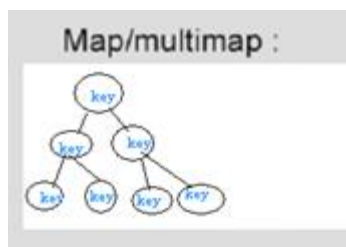
while(!h.empty()){// 序列非空h.empty()==true时表示h已经空了
 multiset<int>::iterator c = h.begin();//c指向h序列中第一个元素的地址，第一个
 元素是最小的元素
 printf("%ld ",*c);//将地址c存的数据输出
 h.erase(c);//从h序列中将c指向的元素删除
}

```

## 10.2.9 Map 和 multimap 容器

### map/multimap 的简介

- ◆ map 是标准的**关联式**容器，一个 map 是一个键值对序列，即(key,value)对。它提供基于 key 的快速检索能力。
- ◆ map 中 **key 值是唯一的**。集合中的元素按一定的**顺序**排列。元素插入过程是按排序规则插入，所以不能指定插入位置。
- ◆ **map 的具体实现采用红黑树变体的平衡二叉树的数据结构**。在插入操作和删除操作上比 vector 快。
- ◆ map 可以直接存取 key 所对应的 value，支持[]操作符，如 map[key]=value。
- ◆ multimap 与 map 的区别：map 支持唯一键值，每个键只能出现一次；而 multimap 中相同键可以出现多次。multimap 不支持[]操作符。
- ◆ #include <map>



### map/multimap 对象的默认构造

map/multimap 采用模板类实现，对象的默认构造形式：

```

map<T1,T2> mapTT;
multimap<T1,T2> multimapTT;

```

如：

```

map<int, char> mapA;

```

```
map<string,float> mapB;
```

```
//其中 T1,T2 还可以用各种指针类型或自定义类型
```

## map 的插入与迭代器

- ◆ `map.insert(...);` //往容器插入元素，返回 `pair<iterator,bool>`
- ◆ 在 `map` 中插入元素的三种方式：  
假设 `map<int, string> mapStu;`
- ◆ 一、通过 `pair` 的方式插入对象  
`mapStu.insert( pair<int,string>(3,"小张") );`
- ◆ 二、通过 `pair` 的方式插入对象  
`mapStu.inset(make_pair(-1, "校长-1"));`
- ◆ 三、通过 `value_type` 的方式插入对象  
`mapStu.insert( map<int,string>::value_type(1,"小李") );`
- ◆ 四、通过数组的方式插入值  
`mapStu[3] = "小刘";`  
`mapStu[5] = "小王";`
- ◆ 前三种方法，采用的是 `insert()` 方法，该方法返回值为 `pair<iterator,bool>`
- ◆ 第四种方法非常直观，但存在一个性能的问题。插入 3 时，先在 `mapStu` 中查找主键为 3 的项，若没发现，则将一个键为 3，值为初始化值的对组插入到 `mapStu` 中，然后再将值修改成“小刘”。若发现已存在 3 这个键，则修改这个键对应的 `value`。
- ◆ `string strName = mapStu[2];` //取操作或插入操作
- ◆ 只有当 `mapStu` 存在 2 这个键时才是正确的取操作，否则会自动插入一个实例，键为 2，值为初始化值。

```
假设 map<int, string> mapA;
```

```
pair< map<int,string>::iterator, bool > pairResult = mapA.insert(pair<int,string>(3,"小张"));
//插入方式一
```

```
int iFirstFirst = (pairResult.first)->first; //iFirst == 3;
string strFirstSecond = (pairResult.first)->second; //strFirstSecond 为"小张"
bool bSecond = pairResult.second; //bSecond == true;
```

```
mapA.insert(map<int,string>::value_type(1,"小李")); //插入方式二
```

```
mapA[3] = "小刘"; //修改 value
mapA[5] = "小王"; //插入方式三
```

```
string str1 = mapA[2]; //执行插入 string() 操作，返回的 str1 的字符串内容为空。
string str2 = mapA[3]; //取得 value，str2 为"小刘"
```

```
//迭代器遍历
```

```
for (map<int,string>::iterator it=mapA.begin(); it!=mapA.end(); ++it)
{
 pair<int, string> pr = *it;
 int iKey = pr.first;
 string strValue = pr.second;
}
map.rbegin()与 map.rend() 略。
```

- ◆ `map<T1,T2,less<T1>> mapA;` //该容器是按键的升序方式排列元素。未指定函数对象，默认采用 `less<T1>`函数对象。
- ◆ `map<T1,T2,greater<T1>> mapB;` //该容器是按键的降序方式排列元素。
- ◆ `less<T1>`与 `greater<T1>` 可以替换成其它的函数对象 `functor`。
- ◆ 可编写自定义函数对象以进行自定义类型的比较，使用方法与 `set` 构造时所用的函数对象一样。
- ◆ `map.begin();` //返回容器中第一个数据的迭代器。
- ◆ `map.end();` //返回容器中最后一个数据之后的迭代器。
- ◆ `map.rbegin();` //返回容器中倒数第一个元素的迭代器。
- ◆ `map.rend();` //返回容器中倒数最后一个元素的后面的迭代器。

## map 对象的拷贝构造与赋值

```
map(const map &mp); //拷贝构造函数
map& operator=(const map &mp); //重载等号操作符
map.swap(mp); //交换两个集合容器
```

例如:

```
map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));
```

```
map<int ,string> mapB(mapA); //拷贝构造
```

```
map<int, string> mapC;
mapC = mapA; //赋值
```

```
mapC[3] = "老张";
mapC.swap(mapA); //交换
```

## map 的大小

```
◆ map.size(); //返回容器中元素的数目
◆ map.empty(); //判断容器是否为空

map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));

if (mapA.empty())
{
 int iSize = mapA.size(); //iSize == 4
}
```

## map 的删除

```
◆ map.clear(); //删除所有元素
◆ map.erase(pos); //删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
◆ map.erase(beg,end); //删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
◆ map.erase(keyElem); //删除容器中 key 为 keyElem 的对组。

map<int, string> mapA;
mapA.insert(pair<int,string>(3,"小张"));
mapA.insert(pair<int,string>(1,"小杨"));
mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));

//删除区间内的元素
map<int,string>::iterator itBegin=mapA.begin();
++ itBegin;
++ itBegin;
map<int,string>::iterator itEnd=mapA.end();
mapA.erase(itBegin,itEnd); //此时容器 mapA 包含按顺序的{1,"小杨"}{3,"
小张"}两个元素。

mapA.insert(pair<int,string>(7,"小赵"));
mapA.insert(pair<int,string>(5,"小王"));

//删除容器中第一个元素
mapA.erase(mapA.begin()); //此时容器 mapA 包含了按顺序的{3,"小张"}{5,"小
王"}{7,"小赵"}三个元素

//删除容器中 key 为 5 的元素
```

```
mapA.erase(5);
```

```
//删除 mapA 的所有元素
```

```
mapA.clear(); //容器为空
```

## map 的查找

- ◆ `map.find(key);` 查找键 `key` 是否存在，若存在，返回该键的元素的迭代器；若不存在，返回 `map.end()`;
- ◆ `map.count(keyElem);` //返回容器中 `key` 为 `keyElem` 的对组个数。对 `map` 来说，要么是 0，要么是 1。对 `multimap` 来说，值可能大于 1。

```
map<int,string>::iterator it=mapStu.find(3);
if(it == mapStu.end())
{
 //没找到
}
else
{
 //找到了
 pair<int, string> pairStu = *it;
 int iID = pairStu.first; //或 int iID = it->first;
 string strName = pairStu.second; //或 string strName = it->second;
}
```

- ◆ `map.lower_bound(keyElem);` //返回第一个 `key>=keyElem` 元素的迭代器。

- ◆ `map.upper_bound(keyElem);` // 返回第一个 `key>keyElem` 元素的迭代器。

例如：`mapStu` 是用 `map<int,string>` 声明的容器，已包含{1,"小李"}{3,"小张"}{5,"小王"}{7,"小赵"}{9,"小陈"}元素。`map<int,string>::iterator it;`

```
it = mapStu.lower_bound(5); //it->first==5 it->second=="小王"
it = mapStu.upper_bound(5); //it->first==7 it->second=="小赵"
it = mapStu.lower_bound(6); //it->first==7 it->second=="小赵"
it = mapStu.upper_bound(6); //it->first==7 it->second=="小赵"
```

- ◆ `map.equal_range(keyElem);` //返回容器中 `key` 与 `keyElem` 相等的上下限的两个迭代器。上限是闭区间，下限是开区间，如[`beg,end`)。

以上函数返回两个迭代器，而这两个迭代器被封装在 `pair` 中。

例如 `map<int,string> mapStu;`

... //往 `mapStu` 容器插入元素{1,"小李"}{3,"小张"}{5,"小王"}{7,"小赵"}{9,"小陈"}



```

pair< map<int,string>::iterator , map<int,string>::iterator > pairIt = mapStu.equal_range(5);
map<int, string>::iterator itBeg = pairIt.first;
map<int, string>::iterator itEnd = pairIt.second;
//此时 itBeg->first==5 , itEnd->first == 7,
itBeg->second=="小王", itEnd->second=="小赵"

```

Multimap 案例:

//1个key值可以对应多个valude =>分组

//公司有销售部 sale (员工2名)、技术研发部 development (1人)、财务部 Financial (2人)

//人员信息有: 姓名, 年龄, 电话、工资等组成

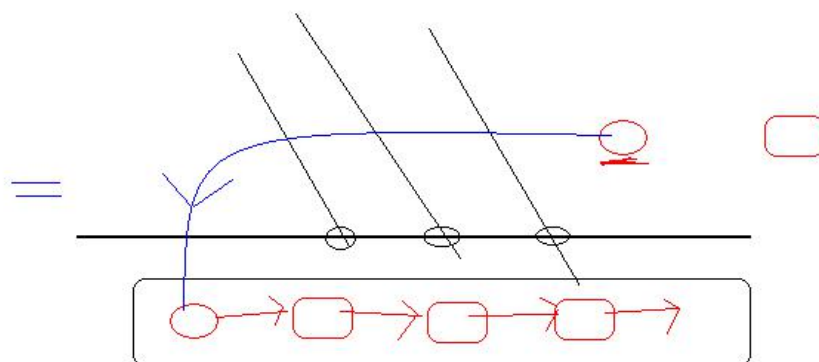
//通过 multimap进行 信息的插入、保存、显示

//分部门显示员工信息

## 10.2.10 容器共性机制研究

### 10.2.9.1 容器的共通能力

C++模板是容器的概念。



- 1 容器中缓存了 用户的结点
- 2 结点的类, 要保证结点能够插入到容器中  
一般结点类, 需要提供  
无参构造函数  
拷贝构造函数  
重载=操作符

**理论提高:** 所有容器提供的都是值 (value) 语义, 而非引用 (reference) 语义。容器执行插入元素的操作时, 内部实施拷贝动作。所以 STL 容器内存储的元素必须能够被拷贝 (必须提供拷贝构造函数)。

◆ 除了 queue 与 stack 外, 每个容器都提供可返回迭代器的函数, 运用返回的迭代器

就可以访问元素。

- ◆ 通常 STL 不会抛出异常。要求使用者确保传入正确的参数。
- ◆ 每个容器都提供了一个默认构造函数跟一个默认拷贝构造函数。
- ◆ 如已有容器 `vecIntA`。
- ◆ `vector<int> vecIntB(vecIntA);` //调用拷贝构造函数，复制 `vecIntA` 到 `vecIntB` 中。
- ◆ 与大小相关的操作方法(c 代表容器):
  - `c.size();` //返回容器中元素的个数
  - `c.empty();` //判断容器是否为空
- ◆ 比较操作(c1,c2 代表容器):
  - `c1 == c2` 判断 `c1` 是否等于 `c2`
  - `c1 != c2` 判断 `c1` 是否不等于 `c2`
  - `c1 = c2` 把 `c2` 的所有元素指派给 `c1`

### 10.2.9.2 各个容器的使用时机

|   | A      | B      | C     | D    | E   | F        | G       | H        |
|---|--------|--------|-------|------|-----|----------|---------|----------|
| 1 |        | vector | deque | list | set | multiset | map     | multimap |
| 2 | 典型内存结构 | 单端数组   | 双端数组  | 双向链表 | 二叉树 | 二叉树      | 二叉树     | 二叉树      |
| 3 | 可随机存取  | 是      | 是     | 否    | 否   | 否        | 对key而言是 | 否        |
| 4 | 元素搜寻速度 | 慢      | 慢     | 非常慢  | 快   | 快        | 对key而言快 | 对key而言快  |
| 5 | 快速安插移除 | 尾端     | 头尾两端  | 任何位置 | -   | -        | -       | -        |

- ◆ `deque` 的使用场景：比如**排队购票系统**，对排队者的存储可以采用 `deque`，支持头端的快速移除，尾端的快速添加。如果采用 `vector`，则头端移除时，会移动大量的数据，速度慢。
- ◆ `vector` 与 `deque` 的比较：
  - ◆ 一：`vector.at()`比 `deque.at()`效率高，比如 `vector.at(0)`是固定的，`deque` 的开始位置却是不固定的。
  - ◆ 二：如果有大量释放操作的话，`vector` 花的时间更少，这跟二者的内部实现有关。
  - ◆ 三：`deque` 支持头部的快速插入与快速移除，这是 `deque` 的优点。
- ◆ `list` 的使用场景：比如公交车乘客的存储，随时可能有乘客下车，**支持频繁的不确定位置元素的移除插入**。
- ◆ `set` 的使用场景：比如对手机游戏的**个人得分记录的存储**，存储要求从高分到低分顺序排列。
- ◆ `map` 的使用场景：比如按 **ID 号存储十万个用户**，想要快速要通过 ID 查找对应的用户。二叉树的查找效率，这时就体现出来了。如果是 `vector` 容器，最坏的情况下可能要遍历完整个容器才能找到该用户。

### 10.2.11 其他

## 10.3 算法

### 10.3.1 算法基础

#### 10.3.1.1 算法概述

- ◆ 算法部分主要由头文件`<algorithm>`，`<numeric>`和`<functional>`组成。
- ◆ `<algorithm>`是所有 STL 头文件中最大的一个，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、反转、排序、合并等等。
- ◆ `<numeric>`体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。
- ◆ `<functional>`中则定义了一些模板类，用以声明函数对象。
- ◆ STL 提供了大量实现算法的模版函数，只要我们熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能，从而大大地提升效率。
- ◆ `#include <algorithm>`
- ◆ `#include <numeric>`
- ◆ `#include <functional>`

#### 10.3.1.2 STL 中算法分类

- 操作对象
  - 直接改变容器的内容
  - 将原容器的内容复制一份，修改其副本，然后传回该副本
- 功能：
  - 非可变序列算法 指不直接修改其所操作的容器内容的算法
    - 计数算法 `count`、`count_if`
    - 搜索算法 `search`、`find`、`find_if`、`find_first_of`、...
    - 比较算法 `equal`、`mismatch`、`lexicographical_compare`
  - 可变序列算法 指可以修改它们所操作的容器内容的算法
    - 删除算法 `remove`、`remove_if`、`remove_copy`、...
    - 修改算法 `for_each`、`transform`
    - 排序算法 `sort`、`stable_sort`、`partial_sort`、
  - 排序算法 包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作
  - 数值算法 对容器内容进行数值计算

### 10.3.1.3 查找算法(13 个): 判断容器中是否包含某个值

| 函数名           | 头文件         | 函数功能                                                                                                                          |
|---------------|-------------|-------------------------------------------------------------------------------------------------------------------------------|
| adjacent_find | <algorithm> | 在 iterator 对标识元素范围内, 查找一对相邻重复元素, 找到则返回指向这对元素的第一个元素的 ForwardIterator。否则返回 last. 重载版本使用输入的二元操作符代替相等的判断                          |
|               | 函数原形        | template<class FwdIt><br>FwdIt adjacent_find(FwdIt first, FwdIt last);                                                        |
|               |             | template<class FwdIt, class Pred><br>FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);                                   |
| binary_search | <algorithm> | 在有序序列中查找 value, 找到返回 true. 重载的版本实用指定的比较函数对象或函数指针来判断相等                                                                         |
|               | 函数原形        | template<class FwdIt, class T><br>bool binary_search(FwdIt first, FwdIt last, const T& val);                                  |
|               |             | template<class FwdIt, class T, class Pred><br>bool binary_search(FwdIt first, FwdIt last, const T& val, Pred pr);             |
| count         | <algorithm> | 利用等于操作符, 把标志范围内的元素与输入值比较, 返回相等元素个数                                                                                            |
|               | 函数原形        | template<class InIt, class Dist><br>size_t count(InIt first, InIt last, const T& val, Dist& n);                               |
| count_if      | <algorithm> | 利用输入的操作符, 对标志范围内的元素进行操作, 返回结果为 true 的个数                                                                                       |
|               | 函数原形        | template<class InIt, class Pred, class Dist><br>size_t count_if(InIt first, InIt last, Pred pr);                              |
| equal_range   | <algorithm> | 功能类似 equal, 返回一对 iterator, 第一个表示 lower_bound, 第二个表示 upper_bound                                                               |
|               | 函数原形        | template<class FwdIt, class T><br>pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val);                      |
|               |             | template<class FwdIt, class T, class Pred><br>pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val, Pred pr); |
| find          | <algorithm> | 利用底层元素的等于操作符, 对指定范围内的元素与输入值进行比较. 当匹配时, 结束搜索, 返回该元素的一个 InputIterator                                                           |
|               | 函数原形        | template<class InIt, class T><br>InIt find(InIt first, InIt last, const T& val);                                              |
| find_end      | <algorithm> | 在指定范围内查找“由输入的另外一对 iterator 标志的第二个序列”的最后一次出                                                                                    |

|               |             |                                                                                                                                                                                                                                                                                        |
|---------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|               | hm>         | 现. 找到则返回最后一对的第一个 ForwardIterator, 否则返回输入的“另外一对”的第一个 ForwardIterator. 重载版本使用用户输入的操作符代替等于操作                                                                                                                                                                                              |
|               | 函数原形        | <pre>template&lt;class FwdIt1, class FwdIt2&gt; FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2);  template&lt;class FwdIt1, class FwdIt2, class Pred&gt; FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);</pre>           |
| find_first_of | <algorithm> | 在指定范围内查找“由输入的另外一对 iterator 标志的第二个序列”中任意一个元素的第一次出现。重载版本中使用了用户自定义操作符                                                                                                                                                                                                                     |
|               | 函数原形        | <pre>template&lt;class FwdIt1, class FwdIt2&gt; FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2);  template&lt;class FwdIt1, class FwdIt2, class Pred&gt; FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);</pre> |
| find_if       | <algorithm> | 使用输入的函数代替等于操作符执行 find                                                                                                                                                                                                                                                                  |
|               |             | <pre>template&lt;class InIt, class Pred&gt; InIt find_if(InIt first, InIt last, Pred pr);</pre>                                                                                                                                                                                        |
| lower_bound   | <algorithm> | 返回一个 ForwardIterator, 指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置. 重载函数使用自定义比较操作                                                                                                                                                                                                                  |
|               | 函数原形        | <pre>template&lt;class FwdIt, class T&gt; FwdIt lower_bound(FwdIt first, FwdIt last, const T&amp; val);  template&lt;class FwdIt, class T, class Pred&gt; FwdIt lower_bound(FwdIt first, FwdIt last, const T&amp; val, Pred pr);</pre>                                                 |
| upper_bound   | <algorithm> | 返回一个 ForwardIterator, 指向在有序序列范围内插入 value 而不破坏容器顺序的最后一个位置, 该位置标志一个大于 value 的值. 重载函数使用自定义比较操作                                                                                                                                                                                            |
|               | 函数原形        | <pre>template&lt;class FwdIt, class T&gt; FwdIt upper_bound(FwdIt first, FwdIt last, const T&amp; val);  template&lt;class FwdIt, class T, class Pred&gt; FwdIt upper_bound(FwdIt first, FwdIt last, const T&amp; val, Pred pr);</pre>                                                 |
| search        | <algorithm> | 给出两个范围, 返回一个 ForwardIterator, 查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置, 查找失败指向 last1, 重载版本使用自定义的比较操作                                                                                                                                                                                             |
|               | 函数原形        | <pre>template&lt;class FwdIt1, class FwdIt2&gt; FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2);</pre>                                                                                                                                                         |

|          |             |                                                                                                                                                      |
|----------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
|          |             | <pre>template&lt;class FwdIt1, class FwdIt2, class Pred&gt; FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);</pre>  |
| search_n | <algorithm> | 在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作                                                                                                               |
|          | 函数原形        | <pre>template&lt;class FwdIt, class Dist, class T&gt; FwdIt search_n(FwdIt first, FwdIt last, Dist n, const T&amp; val);</pre>                       |
|          |             | <pre>template&lt;class FwdIt, class Dist, class T, class Pred&gt; FwdIt search_n(FwdIt first, FwdIt last, Dist n, const T&amp; val, P red pr);</pre> |

### 10.3.1.4 堆算法(4 个)

| 函数名       | 头文件         | 函数功能                                                                                                              |
|-----------|-------------|-------------------------------------------------------------------------------------------------------------------|
| make_heap | <algorithm> | 把指定范围内的元素生成一个堆。重载版本使用自定义比较操作                                                                                      |
|           | 函数原形        | <pre>template&lt;class RanIt&gt; void make_heap(RanIt first, RanIt last);</pre>                                   |
|           |             | <pre>template&lt;class RanIt, class Pred&gt; void make_heap(RanIt first, RanIt last, Pred pr);</pre>              |
| pop_heap  | <algorithm> | 并不真正把最大元素从堆中弹出，而是重新排序堆。它把 first 和 last-1 交换，然后重新生成一个堆。可使用容器的 back 来访问被“弹出”的元素或者使用 pop_back 进行真正的删除。重载版本使用自定义的比较操作 |
|           | 函数原形        | <pre>template&lt;class RanIt&gt; void pop_heap(RanIt first, RanIt last);</pre>                                    |
|           |             | <pre>template&lt;class RanIt, class Pred&gt; void pop_heap(RanIt first, RanIt last, Pred pr);</pre>               |
| push_heap | <algorithm> | 假设 first 到 last-1 是一个有效堆，要被加入到堆的元素存放在位置 last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作                            |
|           | 函数原形        | <pre>template&lt;class RanIt&gt; void push_heap(RanIt first, RanIt last);</pre>                                   |
|           |             | <pre>template&lt;class RanIt, class Pred&gt; void push_heap(RanIt first, RanIt last, Pred pr);</pre>              |
| sort_heap | <algorithm> | 对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作                                                                           |
|           | 函数原形        | <pre>template&lt;class RanIt&gt; void sort_heap(RanIt first, RanIt last);</pre>                                   |
|           |             | <pre>template&lt;class RanIt, class Pred&gt; void sort_heap(RanIt first, RanIt last, Pred pr);</pre>              |

### 10.3.1.5 关系算法(8个)

| 函数名                     | 头文件         | 函数功能                                                                                                                                           |
|-------------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| equal                   | <algorithm> | 如果两个序列在标志范围内元素都相等, 返回 true。重载版本使用输入的操作符代替默认的等于操作符                                                                                              |
|                         | 函数原形        | template<class InIt1, class InIt2><br>bool equal(InIt1 first, InIt1 last, InIt2 x);                                                            |
|                         |             | template<class InIt1, class InIt2, class Pred><br>bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);                                       |
| includes                | <algorithm> | 判断第一个指定范围内的所有元素是否都被第二个范围包含, 使用底层元素的<操作符, 成功返回 true。重载版本使用用户输入的函数                                                                               |
|                         | 函数原形        | template<class InIt1, class InIt2><br>bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2);                                     |
|                         |             | template<class InIt1, class InIt2, class Pred><br>bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);                |
| lexicographical_compare | <algorithm> | 比较两个序列。重载版本使用用户自定义比较操作                                                                                                                         |
|                         | 函数原形        | template<class InIt1, class InIt2><br>bool lexicographical_compare(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2);                      |
|                         |             | template<class InIt1, class InIt2, class Pred><br>bool lexicographical_compare(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr); |
| max                     | <algorithm> | 返回两个元素中较大一个。重载版本使用自定义比较操作                                                                                                                      |
|                         | 函数原形        | template<class T> const T& max(const T& x, const T& y);                                                                                        |
|                         |             | template<class T, class Pred><br>const T& max(const T& x, const T& y, Pred pr);                                                                |
| max_element             | <algorithm> | 返回一个 ForwardIterator, 指出序列中最大的元素。重载版本使用自定义比较操作                                                                                                 |
|                         | 函数原形        | template<class FwdIt> FwdIt max_element(FwdIt first, FwdIt last);                                                                              |
|                         |             | template<class FwdIt, class Pred><br>FwdIt max_element(FwdIt first, FwdIt last, Pred pr);                                                      |
| min                     | <algorithm> | 返回两个元素中较小一个。重载版本使用自定义比较操作                                                                                                                      |
|                         | 函数原形        | template<class T> const T& min(const T& x, const T& y);                                                                                        |

|             |             |                                                                                                                                                                                                                                                             |
|-------------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             |             | <pre>template&lt;class T, class Pred&gt; const T&amp; min(const T&amp; x, const T&amp; y, Pred pr);</pre>                                                                                                                                                   |
| min_element | <algorithm> | 返回一个 ForwardIterator，指出序列中最小的元素。重载版本使用自定义比较操作                                                                                                                                                                                                               |
|             | 函数原形        | <pre>template&lt;class FwdIt&gt; FwdIt min_element(FwdIt first, FwdIt last); template&lt;class FwdIt, class Pred&gt; FwdIt min_element(FwdIt first, FwdIt last, Pred pr);</pre>                                                                             |
| mismatch    | <algorithm> | 并行比较两个序列，指出第一个不匹配的位置，返回一对 iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的 last。重载版本使用自定义的比较操作                                                                                                                                                                           |
|             | 函数原形        | <pre>template&lt;class InIt1, class InIt2&gt; pair&lt;InIt1, InIt2&gt; mismatch(InIt1 first, InIt1 last, InIt2 x); template&lt;class InIt1, class InIt2, class Pred&gt; pair&lt;InIt1, InIt2&gt; mismatch(InIt1 first, InIt1 last, InIt2 x, Pred pr);</pre> |

### 10.3.1.6 集合算法(4个)

| 函数名              | 头文件         | 函数功能                                                                                                                                                                                                                                                                                                                      |
|------------------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| set_union        | <algorithm> | 构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作                                                                                                                                                                                                                                                                                   |
|                  | 函数原形        | <pre>template&lt;class InIt1, class InIt2, class OutIt&gt; OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x); template&lt;class InIt1, class InIt2, class OutIt, class Pred&gt; OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);</pre>               |
| set_intersection | <algorithm> | 构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作                                                                                                                                                                                                                                                                                     |
|                  | 函数原形        | <pre>template&lt;class InIt1, class InIt2, class OutIt&gt; OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x); template&lt;class InIt1, class InIt2, class OutIt, class Pred&gt; OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);</pre> |
| set_difference   | <algorithm> | 构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存                                                                                                                                                                                                                                                                                           |



|                          |             |                                                                                                                                                                                                                                                                                                                                                              |
|--------------------------|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| e                        |             | 在的元素。重载版本使用自定义的比较操作                                                                                                                                                                                                                                                                                                                                          |
|                          | 函数原形        | <div>template&lt;class InIt1, class InIt2, class OutIt&gt;<br/>OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);</div> <div>template&lt;class InIt1, class InIt2, class OutIt, class Pred&gt;<br/>OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);</div>                     |
| set_symmetric_difference | <algorithm> | 构造一个有序序列，该序列取两个序列的对称差集(并集-交集)                                                                                                                                                                                                                                                                                                                                |
|                          | 函数原形        | <div>template&lt;class InIt1, class InIt2, class OutIt&gt;<br/>OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);</div> <div>template&lt;class InIt1, class InIt2, class OutIt, class Pred&gt;<br/>OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);</div> |

10.3.1.6 列组合算法(2 个)

提供计算给定集合按一定顺序的所有可能排列组合

| 函数名              | 头文件         | 函数功能                                                                                          |
|------------------|-------------|-----------------------------------------------------------------------------------------------|
| next_permutation | <algorithm> | 取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作                                                         |
|                  | 函数原形        | template<class BidIt><br>bool next_permutation(BidIt first, BidIt last);                      |
|                  |             | template<class BidIt, class Pred><br>bool next_permutation(BidIt first, BidIt last, Pred pr); |
| prev_permutation | <algorithm> | 取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 false。重载版本使用自定义的比较操作                                    |
|                  | 函数原形        | template<class BidIt><br>bool prev_permutation(BidIt first, BidIt last);                      |
|                  |             | template<class BidIt, class Pred><br>bool prev_permutation(BidIt first, BidIt last, Pred pr); |

10.3.1.7 排序和通用算法(14 个)：提供元素排序策略

| 函数名 | 头文件 | 函数功能 |
|-----|-----|------|
|-----|-----|------|

|                   |             |                                                                                                                                                                                                                                                                                                      |
|-------------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| inplace_merge     | <algorithm> | 合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序                                                                                                                                                                                                                                                                  |
|                   | 函数原形        | <pre>template&lt;class BidIt&gt; void inplace_merge(BidIt first, BidIt middle, BidIt last);  template&lt;class BidIt, class Pred&gt; void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);</pre>                                                                                       |
| merge             | <algorithm> | 合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较                                                                                                                                                                                                                                                                       |
|                   | 函数原形        | <pre>template&lt;class InIt1, class InIt2, class OutIt&gt; OutIt merge(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);  template&lt;class InIt1, class InIt2, class OutIt, class Pred&gt; OutIt merge(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);</pre> |
| nth_element       | <algorithm> | 将范围内的序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作                                                                                                                                                                                                                                        |
|                   | 函数原形        | <pre>template&lt;class RanIt&gt; void nth_element(RanIt first, RanIt nth, RanIt last);  template&lt;class RanIt, class Pred&gt; void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);</pre>                                                                                                 |
| partial_sort      | <algorithm> | 对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作                                                                                                                                                                                                                                                            |
|                   | 函数原形        | <pre>template&lt;class RanIt&gt; void partial_sort(RanIt first, RanIt middle, RanIt last);  template&lt;class RanIt, class Pred&gt; void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);</pre>                                                                                         |
| partial_sort_copy | <algorithm> | 与 partial_sort 类似，不过将经过排序的序列复制到另一个容器                                                                                                                                                                                                                                                                 |
|                   | 函数原形        | <pre>template&lt;class InIt, class RanIt&gt; RanIt partial_sort_copy(InIt first1, InIt last1, InIt first2, InIt last2);  template&lt;class InIt, class RanIt, class Pred&gt; RanIt partial_sort_copy(InIt first1, InIt last1, InIt first2, InIt last2, Pred pr);</pre>                               |
| partition         | <algorithm> | 对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前                                                                                                                                                                                                                                                  |

|                  |             |                                                                                                          |
|------------------|-------------|----------------------------------------------------------------------------------------------------------|
|                  | 函数原形        | template<class BidIt, class Pred><br>BidIt partition(BidIt first, BidIt last, Pred pr);                  |
| random_shuffle   | <algorithm> | 对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作                                                                          |
|                  | 函数原形        | template<class RanIt><br>void random_shuffle(RanIt first, RanIt last);                                   |
|                  |             | template<class RanIt, class Fun><br>void random_shuffle(RanIt first, RanIt last, Fun& f);                |
| reverse          | <algorithm> | 将指定范围内元素重新反序排序                                                                                           |
|                  | 函数原形        | template<class BidIt> void reverse(BidIt first, BidIt last);                                             |
| reverse_copy     | <algorithm> | 与 reverse 类似，不过将结果写入另一个容器                                                                                |
|                  | 函数原形        | template<class BidIt, class OutIt><br>OutIt reverse_copy(BidIt first, BidIt last, OutIt x);              |
| rotate           | <algorithm> | 将指定范围内元素移到容器末尾，由 middle 指向的元素成为容器第一个元素                                                                   |
|                  | 函数原形        | template<class FwdIt><br>void rotate(FwdIt first, FwdIt middle, FwdIt last);                             |
| rotate_copy      | <algorithm> | 与 rotate 类似，不过将结果写入另一个容器                                                                                 |
|                  | 函数原形        | template<class FwdIt, class OutIt><br>OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x); |
| sort             | <algorithm> | 以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作                                                                           |
|                  | 函数原形        | template<class RanIt> void sort(RanIt first, RanIt last);                                                |
|                  |             | template<class RanIt, class Pred><br>void sort(RanIt first, RanIt last, Pred pr);                        |
| stable_sort      | <algorithm> | 与 sort 类似，不过保留相等元素之间的顺序关系                                                                                |
|                  | 函数原形        | template<class BidIt><br>void stable_sort(BidIt first, BidIt last);                                      |
|                  |             | template<class BidIt, class Pred><br>void stable_sort(BidIt first, BidIt last, Pred pr);                 |
| stable_partition | <algorithm> | 与 partition 类似，不过不保证保留容器中的相对顺序                                                                           |
|                  | 函数原形        | template<class FwdIt, class Pred>                                                                        |

|  |  |                                                           |
|--|--|-----------------------------------------------------------|
|  |  | FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr); |
|--|--|-----------------------------------------------------------|

### 10.3.1.8 删除和替换算法(15 个)

| 函数名            | 头文件         | 函数功能                                                                                                            |
|----------------|-------------|-----------------------------------------------------------------------------------------------------------------|
| copy           | <algorithm> | 复制序列                                                                                                            |
|                | 函数原形        | template<class InIt, class OutIt><br>OutIt copy(InIt first, InIt last, OutIt x);                                |
| copy_backward  | <algorithm> | 与 copy 相同，不过元素是以相反顺序被拷贝                                                                                         |
|                | 函数原形        | template<class BidIt1, class BidIt2><br>BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);              |
| iter_swap      | <algorithm> | 交换两个 ForwardIterator 的值                                                                                         |
|                | 函数原形        | template<class FwdIt1, class FwdIt2><br>void iter_swap(FwdIt1 x, FwdIt2 y);                                     |
| remove         | <algorithm> | 删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用 remove 和 remove_if 函数                                               |
|                | 函数原形        | template<class FwdIt, class T><br>FwdIt remove(FwdIt first, FwdIt last, const T& val);                          |
| remove_copy    | <algorithm> | 将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置                                                             |
|                | 函数原形        | template<class InIt, class OutIt, class T><br>OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);  |
| remove_if      | <algorithm> | 删除指定范围内输入操作结果为 true 的所有元素                                                                                       |
|                | 函数原形        | template<class FwdIt, class Pred><br>FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);                         |
| remove_copy_if | <algorithm> | 将所有不匹配元素拷贝到一个指定容器                                                                                               |
|                | 函数原形        | template<class InIt, class OutIt, class Pred><br>OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr); |
| replace        | <algorithm> | 将指定范围内所有等于 vold 的元素都用 vnew 代替                                                                                   |

|                 |             |                                                                                                                                                                                                                                  |
|-----------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                 | 函数原形        | <pre>template&lt;class FwdIt, class T&gt; void replace(FwdIt first, FwdIt last, const T&amp; vold, const T&amp; vnew);</pre>                                                                                                     |
|                 | <algorithm> | 与 replace 类似，不过将结果写入另一个容器                                                                                                                                                                                                        |
| replace_copy    | 函数原形        | <pre>template&lt;class InIt, class OutIt, class T&gt; OutIt replace_copy(InIt first, InIt last, OutIt x, const T&amp; vold, const T&amp; vnew);</pre>                                                                            |
|                 | <algorithm> | 将指定范围内所有操作结果为 true 的元素用新值代替                                                                                                                                                                                                      |
| replace_if      | 函数原形        | <pre>template&lt;class FwdIt, class Pred, class T&gt; void replace_if(FwdIt first, FwdIt last, Pred pr, const T&amp; val);</pre>                                                                                                 |
|                 | <algorithm> | 与 replace_if，不过将结果写入另一个容器                                                                                                                                                                                                        |
| replace_copy_if | 函数原形        | <pre>template&lt;class InIt, class OutIt, class Pred, class T&gt; OutIt replace_copy_if(InIt first, InIt last, OutIt x, Pred pr, const T&amp; val);</pre>                                                                        |
|                 | <algorithm> | 交换存储在两个对象中的值                                                                                                                                                                                                                     |
| swap            | 函数原形        | <pre>template&lt;class T&gt; void swap(T&amp; x, T&amp; y);</pre>                                                                                                                                                                |
|                 | <algorithm> | 将指定范围内的元素与另一个序列元素值进行交换                                                                                                                                                                                                           |
| swap_range      | 函数原形        | <pre>template&lt;class FwdIt1, class FwdIt2&gt; FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);</pre>                                                                                                                   |
|                 | <algorithm> | 清除序列中重复元素，和 remove 类似，它也不能真正删除元素。重载版本使用自定义比较操作                                                                                                                                                                                   |
| unique          | 函数原形        | <pre>template&lt;class FwdIt&gt; FwdIt unique(FwdIt first, FwdIt last);</pre> <pre>template&lt;class FwdIt, class Pred&gt; FwdIt unique(FwdIt first, FwdIt last, Pred pr);</pre>                                                 |
|                 | <algorithm> | 与 unique 类似，不过把结果输出到另一个容器                                                                                                                                                                                                        |
| unique_copy     | 函数原形        | <pre>template&lt;class InIt, class OutIt&gt; OutIt unique_copy(InIt first, InIt last, OutIt x);</pre> <pre>template&lt;class InIt, class OutIt, class Pred&gt; OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);</pre> |

### 10.3.1.9 生成和变异算法(6 个)

| 函数名        | 头文件         | 函数功能                                                                                                                                          |
|------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| fill       | <algorithm> | 将输入值赋给标志范围内的所有元素                                                                                                                              |
|            | 函数原形        | template<class FwdIt, class T><br>void fill(FwdIt first, FwdIt last, const T& x);                                                             |
| fill_n     | <algorithm> | 将输入值赋给 first 到 first+n 范围内的所有元素                                                                                                               |
|            | 函数原形        | template<class OutIt, class Size, class T><br>void fill_n(OutIt first, Size n, const T& x);                                                   |
| for_each   | <algorithm> | 用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素                                                                                              |
|            | 函数原形        | template<class InIt, class Fun><br>Fun for_each(InIt first, InIt last, Fun f);                                                                |
| generate   | <algorithm> | 连续调用输入的函数来填充指定的范围                                                                                                                             |
|            | 函数原形        | template<class FwdIt, class Gen><br>void generate(FwdIt first, FwdIt last, Gen g);                                                            |
| generate_n | <algorithm> | 与 generate 函数类似，填充从指定 iterator 开始的 n 个元素                                                                                                      |
|            | 函数原形        | template<class OutIt, class Pred, class Gen><br>void generate_n(OutIt first, Dist n, Gen g);                                                  |
| transform  | <algorithm> | 将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器                                                                     |
|            | 函数原形        | template<class InIt, class OutIt, class Unop><br>OutIt transform(InIt first, InIt last, OutIt x, Unop uop);                                   |
|            |             | template<class InIt1, class InIt2, class OutIt, class Binop><br>OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2, OutIt x, Binop bop); |

### 10.3.1.10 算数算法(4 个)

| 函数名        | 头文件       | 函数功能                                                                |
|------------|-----------|---------------------------------------------------------------------|
| accumulate | <numeric> | iterator 对标识的序列段元素之和，加到一个由 val 指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上 |

|                     |           |                                                                                                                                                                                                                                                                            |
|---------------------|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                     | 函数原形      | template<class InIt, class T><br>T accumulate(InIt first, InIt last, T val);                                                                                                                                                                                               |
|                     |           | template<class InIt, class T, class Pred><br>T accumulate(InIt first, InIt last, T val, Pred pr);                                                                                                                                                                          |
| partial_sum         | <numeric> | 创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法                                                                                                                                                                                                                           |
|                     | 函数原形      | template<class InIt, class OutIt><br>OutIt partial_sum(InIt first, InIt last, OutIt result);<br><br>template<class InIt, class OutIt, class Pred><br>OutIt partial_sum(InIt first, InIt last, OutIt result, Pred pr);                                                      |
| product             | <numeric> | 对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。重载版本使用用户定义的操作                                                                                                                                                                                                                          |
|                     | 函数原形      | template<class InIt1, class InIt2, class T><br>T product(InIt1 first1, InIt1 last1, InIt2 first2, T val);<br><br>template<class InIt1, class InIt2, class T, class Pred1, class Pred2><br>T product(InIt1 first1, InIt1 last1, InIt2 first2, T val, Pred1 pr1, Pred2 pr2); |
| adjacent_difference | <numeric> | 创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差                                                                                                                                                                                                                         |
|                     | 函数原形      | template<class InIt, class OutIt><br>OutIt adjacent_difference(InIt first, InIt last, OutIt result);<br><br>template<class InIt, class OutIt, class Pred><br>OutIt adjacent_difference(InIt first, InIt last, OutIt result, Pred pr);                                      |

### 10.3.1.11 常用算法汇总

- ◆ 常用的查找算法：  
adjacent\_find()（ adjacent 是邻近的意思）,binary\_search(),count(),count\_if(),equal\_range(),find(),find\_if()。
- ◆ 常用的排序算法：  
merge(),sort(),random\_shuffle()（shuffle 是洗牌的意思）,reverse()。
- ◆ 常用的拷贝和替换算法：  
copy(), replace(), replace\_if(),swap()
- ◆ 常用的算术和生成算法：

- accumulate() (accumulate 是求和的意思), fill(),。
- ◆ 常用的集合算法:
  - set\_union(), set\_intersection(),
  - set\_difference()。
- ◆ 常用的遍历算法:
  - for\_each(), transform() (transform 是变换的意思)

## 10.3.2 STL 算法中函数对象和谓词

### 10.3.2.1 函数对象和谓词定义

函数对象:

重载函数调用操作符的类，其对象常称为函数对象 (function object)，即它们是行为类似函数的对象。一个类对象，表现出一个函数的特征，就是通过“对象名+(参数列表)”的方式使用一个类对象，如果没有上下文，完全可以把它看作一个函数对待。

这是通过重载类的 operator() 来实现的。

“在标准库中，函数对象被广泛地使用以获得弹性”，标准库中的很多算法都可以使用函数对象或者函数来作为自定的回调行为；

谓词:

一元函数对象：函数参数 1 个；

二元函数对象：函数参数 2 个；

一元谓词 函数参数 1 个，函数返回值是 bool 类型，可以作为一个判断式

谓词可以使一个仿函数，也可以是一个回调函数。

二元谓词 函数参数 2 个，函数返回值是 bool 类型

一元谓词函数举例如下

1, 判断给出的 string 对象的长度是否小于 6

```
bool GT6(const string &s)
{
 return s.size() >= 6;
}
```

2, 判断给出的 int 是否在 3 到 8 之间

```
bool Compare(int i)
{
 return (i >= 3 && i <= 8);
}
```

二元谓词举例如下

1, 比较两个 string 对象，返回一个 bool 值，指出第一个 string 是否比第二个短

```
bool isShorter(const string &s1, const string &s2)
{
 return s1.size() < s2.size();
}
```



### 10.3.2.2 一元函数对象案例

```
//1 普通类 重载 函数调用操作符
template <typename T>
void FuncShowElemt(T &t) //普通函数 不能像 仿函数那样记录状态
{
 cout << t << " ";
};

void showChar(char &t)
{
 cout << t << " ";
}

//函数模板 重载 函数调用操作符
template <typename T>
class ShowElemt
{
public:
 ShowElemt()
 {
 n = 0;
 }
 void operator()(T &t)
 {
 n++;
 cout << t << " ";
 }
 void printCount()
 {
 cout << n << endl;
 }
public:
 int n;
};

//1 函数对象 基本使用
void main11()
{
 int a = 100;
 FuncShowElemt<int>(a); //普通的函数调用

 ShowElemt<int> showElemt; //函数对象
```

```
 showElemt(a); //函数对象调用
}
```

### 10.3.2.3 一元谓词案例

//1 元谓词 例子

```
template <typename T>
```

```
class Isdiv
```

```
{
```

```
public:
```

```
 Isdiv(const T &divisor) //
```

```
 {
```

```
 this->divisor = divisor;
```

```
 }
```

```
 bool operator()(T &t)
```

```
 {
```

```
 return (t%divisor == 0);
```

```
 }
```

```
protected:
```

```
private:
```

```
 T divisor;
```

```
};
```

```
void main13()
```

```
{
```

```
 vector<int> v2;
```

```
 for (int i=10; i<33; i++)
```

```
 {
```

```
 v2.push_back(i);
```

```
 }
```

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

```
 int a = 4;
```

```
 Isdiv<int> mydiv(a);
```

```
 // _InIt find_if(_InIt _First, _InIt _Last, _Pr _Pred) //返回的是迭代器
```

```
 it = find_if(v2.begin(), v2.end(), Isdiv<int>(4));
```

```
 if (it != v2.end())
```

```
 {
```

```
 cout << "第一个被 4 整除的数是: " << *it << endl;
```

```
 }
```

```
}
```

### 10.3.2.4 二元函数对象案例

```
template <typename T>
struct SumAdd
{
 T operator()(T &t1, T &t2)
 {
 return t1 + t2;
 }
};

template <typename T>
void printE(T &t)
{
 for (vector<int>::iterator it = t.begin(); it!=t.end(); it++)
 {
 cout << *it << " ";
 }
}

void printVector(vector<int> &v)
{
 for (vector<int>::iterator it = v.begin(); it!=v.end(); it++)
 {
 cout << *it << " ";
 }
}

void main14()
{
 vector<int> v1, v2 ;
 vector<int> v3;
 v1.push_back(1);
 v1.push_back(2);
 v1.push_back(3);

 v2.push_back(4);
 v2.push_back(5);
 v2.push_back(6);

 v3.resize(10);

 //transform(v1.begin(), v1.end(), v2.begin(),v3.begin(), SumAdd<int>());
```

```
/*
template<class _InIt1,
class _InIt2,
class _OutIt,
class _Fn2> inline
 _OutIt transform(_InIt1 _First1, _InIt1 _Last1,
 _InIt2 _First2, _OutIt _Dest, _Fn2 _Func)
*/
vector<int>::iterator it = transform(v1.begin(), v1.end(), v2.begin(), v3.begin(),
SumAdd<int>());
cout << *it << endl;
printE(v3);
}
```

### 10.3.2.5 二元谓词案例

```
void current(int &v)
{
 cout << v << " ";
}

bool MyCompare(const int &a, const int &b)
{
 return a < b;
}

void main15()
{
 vector<int> v(10);

 for (int i=0; i<10; i++)
 {
 v[i] = rand() % 100;
 }

 for_each(v.begin(), v.end(), current);
 printf("\n");
 sort(v.begin(), v.end(), MyCompare);

 printf("\n");
 for (int i=0; i<10; i++)
 {
 printf("%d ", v[i]);
 }
}
```

```
 }
 printf("\n");
}
```

### 10.3.2.6 预定义函数对象和函数适配器

1) 预定义函数对象基本概念：标准模板库 STL 提前定义了很多预定义函数对象，`#include <functional>` 必须包含。

//1 使用预定义函数对象：

//类模板 `plus<>` 的实现了： 不同类型的数据进行加法运算

```
void main41()
{
 plus<int> intAdd;
 int x = 10;
 int y = 20;
 int z = intAdd(x, y); //等价于 x + y
 cout << z << endl;

 plus<string> stringAdd;
 string myc = stringAdd("aaa", "bbb");
 cout << myc << endl;

 vector<string> v1;
 v1.push_back("bbb");
 v1.push_back("aaa");
 v1.push_back("ccc");
 v1.push_back("zzzz");

 //缺省情况下，sort()用底层元素类型的小于操作符以升序排列容器的元素。
 //为了降序，可以传递预定义的类型模板 greater,它调用底层元素类型的大于操作符：
 cout << "sort()函数排序" << endl;;
 sort(v1.begin(), v1.end(), greater<string>()); //从大到小
 for (vector<string>::iterator it=v1.begin(); it!=v1.end(); it++)
 {
 cout << *it << endl;
 }
}
```

#### 2) 算术函数对象

预定义的函数对象支持加、减、乘、除、求余和取反。调用的操作符是与 `type` 相关联的实例

加法： `plus<Types>`

```
plus<string> stringAdd;
sres = stringAdd(sva1,sva2);
```

减法: minus<Types>  
乘法: multiplies<Types>  
除法 divides<Tpye>  
求余: modulus<Tpye>  
取反: negate<Type>  
negate<int> intNegate;  
ires = intNegate(ires);  
lres= UnaryFunc(negate<int>(),lval1);

### 3) 关系函数对象

等于 equal\_to<Tpye>  
    equal\_to<string> stringEqual;  
    sres = stringEqual(sval1,sval2);  
不等于 not\_equal\_to<Type>  
大于 greater<Type>  
大于等于 greater\_equal<Type>  
小于 less<Type>  
小于等于 less\_equal<Type>

```
void main42()
{
 vector<string> v1;
 v1.push_back("bbb");
 v1.push_back("aaa");
 v1.push_back("ccc");
 v1.push_back("zzzz");
 v1.push_back("ccc");
 string s1 = "ccc";
 //int num = count_if(v1.begin(),v1.end(), equal_to<string>(),s1);
 int num = count_if(v1.begin(),v1.end(),bind2nd(equal_to<string>(), s1));
 cout << num << endl;
}
```

### 4) 逻辑函数对象

逻辑与 logical\_and<Type>  
logical\_and<int> indAnd;  
    ires = intAnd(ival1,ival2);  
    dres=BinaryFunc( logical\_and<double>(),dval1,dval2);  
逻辑或 logical\_or<Type>  
逻辑非 logical\_not<Type>  
    logical\_not<int> IntNot;  
    lres = IntNot(ival1);  
    Dres=UnaryFunc( logical\_not<double>,dval1);

### 10.3.2.7 函数适配器

#### 1) 函数适配器的理论知识

STL 中已经定义了大量的函数对象,但是有时候需要对函数返回值进行进一步的简单计算,或者填上多余的参数,不能直接代入算法。函数适配器实现了这一功能,将一种函数对象转化为另一种符合要求的函数对象。函数适配器可以分为 4 大类:绑定适配器(bind adaptor)、组合适配器(composite adaptor)、指针函数适配器(pointer function adaptor)和成员函数适配器(member function adaptor)。

STL 中所有的函数适配器由表 10-6 列出。

表 10-6 STL 标准库中的函数适配器

| STL 函数适配器                  | 类 型     | 功 能 说 明                                 |
|----------------------------|---------|-----------------------------------------|
| binder1st                  | 绑定适配器   | 将数值绑定到二元函数的第一个参数,适配成一元函数                |
| binder2nd                  | 绑定适配器   | 将数值绑定到二元函数的第二个参数,适配成一元函数                |
| unary_negate               | 组合适配器   | 将一元谓词的返回值适配成其逻辑反                        |
| binary_negate              | 组合适配器   | 将二元谓词的返回值适配成其逻辑反                        |
| pointer_to_unary_function  | 指针函数适配器 | 将普通一元函数指针适配成 unary_function             |
| pointer_to_binary_function | 指针函数适配器 | 将普通二元函数指针适配成 binary_function            |
| mem_fun_t                  | 成员函数适配器 | 将无参数类成员函数适配成为一元函数对象,第 1 个参数为该类的指针类型     |
| mem_fun_ref_t              | 成员函数适配器 | 将无参数类成员函数适配成为一元函数对象,第 1 个参数为该类的引用类型     |
| mem_fun1_t                 | 成员函数适配器 | 将单参数类成员函数适配成为二元函数对象,第 1 个参数为该类的指针类型     |
| mem_fun1_ref_t             | 成员函数适配器 | 将单参数类成员函数适配成为二元函数对象,第 1 个参数为该类的引用类型     |
| const_mem_fun_t            | 成员函数适配器 | 将无参数类常量成员函数适配成为一元函数对象,第 1 个参数为该类的常量指针类型 |
| const_mem_fun_ref_t        | 成员函数适配器 | 将无参数类常量成员函数适配成为一元函数对象,第 1 个参数为该类的常量引用类型 |
| const_mem_fun1_t           | 成员函数适配器 | 将单参数类常量成员函数适配成为二元函数对象,第 1 个参数为该类的常量指针类型 |
| const_mem_fun1_ref_t       | 成员函数适配器 | 将单参数类常量成员函数适配成为二元函数对象,第 1 个参数为该类的常量引用类型 |

直接构造 STL 中的函数适配器通常会导致冗长的类型声明。为简化函数适配器的构造,STL 还提供了函数适配器辅助函数(如表 10-7 所示),借助于泛型自动推断技术,无须显式的类型声明便可实现函数适配器的构造。

表 10-7 STL 标准库中的函数适配器辅助函数

| 适配器辅助函数     | 功能说明                                                                     |
|-------------|--------------------------------------------------------------------------|
| bind1st     | 辅助构造 binder1st 适配器实例, 绑定固定值到二元函数的第一个参数位置                                 |
| bind2nd     | 辅助构造 binder2nd 适配器实例, 绑定固定值到二元函数的第二个参数位置                                 |
| not1        | 辅助构造 unary_negate 适配器实例, 生成一元函数的逻辑反函数                                    |
| not2        | 辅助构造 binary_negate 适配器实例, 生成二元函数的逻辑反函数                                   |
| ptr_fun     | 辅助构造一般函数指针的 pointer_to_unary_function 或 pointer_to_binary_function 适配器实例 |
| mem_fun     | 辅助构造 mem_fun_t 等成员函数适配器实例, 返回一元或二元函数对象                                   |
| mem_fun_ref | 辅助构造 mem_fun_ref_t 等成员函数适配器实例, 返回一元或二元函数对象                               |

## 2) 常用函数函数适配器

标准库提供一组函数适配器, 用来特殊化或者扩展一元和二元函数对象。常用适配器是:

1 绑定器 (binder): binder 通过把二元函数对象的一个实参绑定到一个特殊的值上, 将其转换成一元函数对象。C++ 标准库提供两种预定义的 binder 适配器: bind1st 和 bind2nd, 前者把值绑定到二元函数对象的第一个实参上, 后者绑定在第二个实参上。

2 取反器(negator): negator 是一个将函数对象的值翻转的函数适配器。标准库提供两个预定义的 ngeator 适配器: not1 翻转一元预定义函数对象的真值, 而 not2 翻转二元谓词函数的真值。

常用函数适配器列表如下:

```
bind1st(op, value)
bind2nd(op, value)
not1(op)
not2(op)
mem_fun_ref(op)
mem_fun(op)
ptr_fun(op)
```

## 3) 常用函数适配器案例

```
////////////////////////////////////
class IsGreat
{
public:
 IsGreat(int i)
 {
 m_num = i;
 }
 bool operator()(int &num)
 {
 if (num > m_num)
 {
 return true;
 }
 }
}
```



```
 return false;
 }
protected:
private:
 int m_num;
};

void main43()
{
 vector<int> v1;
 for (int i=0; i<5; i++)
 {
 v1.push_back(i+1);
 }

 for (vector<int>::iterator it = v1.begin(); it!=v1.end(); it++)
 {
 cout << *it << " ";
 }

 int num1 = count(v1.begin(), v1.end(), 3);
 cout << "num1:" << num1 << endl;

 //通过谓词求大于 2 的个数
 int num2 = count_if(v1.begin(), v1.end(), IsGreat(2));
 cout << "num2:" << num2 << endl;

 //通过预定义函数对象求大于 2 的个数 greater<int>() 有 2 个参数
 // param > 2
 int num3 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 2));
 cout << "num3:" << num3 << endl;

 //取模 能被 2 整除的数 求奇数
 int num4 = count_if(v1.begin(), v1.end(), bind2nd(modulus <int>(), 2));
 cout << "奇数 num4:" << num4 << endl;

 int num5 = count_if(v1.begin(), v1.end(), not1(bind2nd(modulus <int>(), 2)));
 cout << "偶数 num5:" << num5 << endl;
 return ;
}
```

### 10.3.2.8 STL 的容器算法迭代器的设计理念

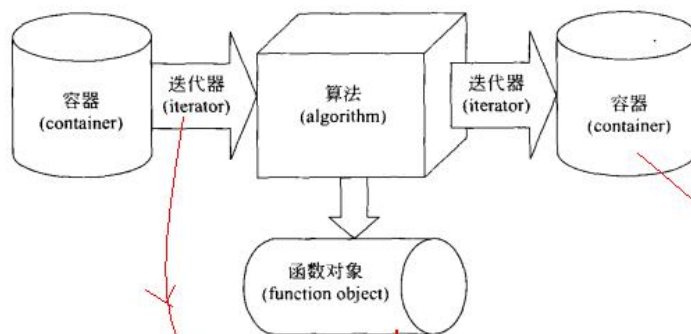


图 10-1 STL 组件之间的关系

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator
result, UnaryFunction op) {
 for (;first!=last; ++first, ++result)
 *result=op(*first);
 return result;
}
```

1 算法的重要特性，就是统一性；可以实现自定义数据类型和标准数据类型的运算。

- 1) STL 的容器通过**类模板**技术，实现数据类型和容器模型的分离。
- 2) STL 的**迭代器技术**实现了**遍历容器**的统一方法；也为 STL 的算法提供了统一性
- 3) STL 的**函数对象**实现了**自定义数据类型**的算法运算。（算法和）
- 4) 具体例子：transform 算法的输入，通过迭代器 first 和 last 指向的元算作为输入；通过 result 作为输出；通过函数对象来做自定义数据类型的运算。

### 10.3.3 常用的查找算法

#### adjacent\_find()

在 iterator 对标识元素范围内，查找一对相邻重复元素，找到则返回指向这对元素的第一个元素的迭代器。否则返回 past-the-end。

```
vector<int> vecInt;
vecInt.push_back(1);
vecInt.push_back(2);
vecInt.push_back(2);
vecInt.push_back(4);
vecInt.push_back(5);
vecInt.push_back(5);
```

```
vector<int>::iterator it = adjacent_find(vecInt.begin(), vecInt.end()); /*it == 2
```

## binary\_search

在有序序列中查找 value,找到则返回 true。注意：在无序序列中，不可使用。

```
set<int> setInt;
setInt.insert(3);
setInt.insert(1);
setInt.insert(7);
setInt.insert(5);
setInt.insert(9);

bool bFind = binary_search(setInt.begin(),setInt.end(),5);
```

## count()

利用等于操作符，把标志范围内的元素与输入值比较，返回相等的个数。

```
vector<int> vecInt;
vecInt.push_back(1);
vecInt.push_back(2);
vecInt.push_back(2);
vecInt.push_back(4);
vecInt.push_back(2);
vecInt.push_back(5);
int iCount = count(vecInt.begin(),vecInt.end(),2); //iCount==3
```

## count\_if()

假设 vector<int> vecIntA， vecIntA 包含 1,3,5,7,9 元素

//先定义比较函数

```
bool GreaterThree(int iNum)
{
 if(iNum>=3)
 {
 return true;
 }
 else
 {
```

```
 return false;
 }
}

int iCount = count_if(vecIntA.begin(), vecIntA.end(), GreaterThree);
//此时 iCount == 4
```

## find()

- ◆ find: 利用底层元素的等于操作符，对指定范围内的元素与输入值进行比较。当匹配时，结束搜索，返回该元素的迭代器。
- ◆ equal\_range: 返回一对 iterator，第一个表示 lower\_bound, 第二个表示 upper\_bound。

```
vector<int> vecInt;
vecInt.push_back(1);
vecInt.push_back(3);
vecInt.push_back(5);
vecInt.push_back(7);
vecInt.push_back(9);

vector<int>::iterator it = find(vecInt.begin(), vecInt.end(), 5); /*it == 5
```

## find\_if()

find\_if: 使用输入的函数代替等于操作符执行 find。返回被找到的元素的迭代器。  
假设 vector<int> vecIntA, vecIntA 包含 1,3,5,3,9 元素  
vector<int>::it = find\_if(vecIntA.begin(),vecIntA.end(),GreaterThree);  
此时 \*it==3, \*(it+1)==5, \*(it+2)==3, \*(it+3)==9

## 10.3.4 常用的排序算法

### merge()

- ◆ 以下是排序和通用算法：提供元素排序策略
- ◆ merge: 合并两个有序序列，存放到另一个序列。

例如: vecIntA,vecIntB,vecIntC 是用 vector<int>声明的容器，vecIntA 已包含 1,3,5,7,9 元素，  
vecIntB 已包含 2,4,6,8 元素  
vecIntC.resize(9); //扩大容量  
merge(vecIntA.begin(),vecIntA.end(),vecIntB.begin(),vecIntB.end(),vecIntC.begin());

此时 vecIntC 就存放了按顺序的 1,2,3,4,5,6,7,8,9 九个元素

## sort()

- ◆ sort: 以默认升序的方式重新排列指定范围内的元素。若要改排序规则，可以输入比较函数。

//学生类

Class CStudent:

```
{
public:
 CStudent(int iID, string strName)
 {
 m_iID=iID;
 m_strName=strName;
 }
public:
 int m_iID;
 string m_strName;
}
```

//学号比较函数

```
bool Compare(const CStudent &stuA,const CStudent &stuB)
{
 return (stuA.m_iID<stuB.m_iID);
}
```

void main()

```
{
 vector<CStudent> vecStu;
 vecStu.push_back(CStudent(2,"老二"));
 vecStu.push_back(CStudent(1,"老大"));
 vecStu.push_back(CStudent(3,"老三"));
 vecStu.push_back(CStudent(4,"老四"));

 sort(vecStu.begin(),vecStu.end(),Compare);

 // 此时，vecStu 容器包含了按顺序的"老大对象","老二对象","老三对象","老四对象"
}
```

## random\_shuffle()

- ◆ random\_shuffle: 对指定范围内的元素随机调整次序。  
srand(time(0)); //设置随机种子

```
vector<int> vecInt;
vecInt.push_back(1);
vecInt.push_back(3);
vecInt.push_back(5);
vecInt.push_back(7);
vecInt.push_back(9);

string str("itcastitcast ");

random_shuffle(vecInt.begin(), vecInt.end()); //随机排序, 结果比如: 9,7,1,5,3
random_shuffle(str.begin(), str.end()); //随机排序, 结果比如: " itstcasticat "
```

## reverse()

```
vector<int> vecInt;
vecInt.push_back(1);
vecInt.push_back(3);
vecInt.push_back(5);
vecInt.push_back(7);
vecInt.push_back(9);

reverse(vecInt.begin(), vecInt.end()); //{9,7,5,3,1}
```

## 10.3.5 常用的拷贝和替换算法

### copy()

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
vecIntA.push_back(7);
vecIntA.push_back(9);

vector<int> vecIntB;
vecIntB.resize(5); //扩大空间

copy(vecIntA.begin(), vecIntA.end(), vecIntB.begin()); //vecIntB: {1,3,5,7,9}
```

## replace()

- ◆ `replace(begin, end, oldValue, newValue)`: 将指定范围内的所有等于 `oldValue` 的元素替换成 `newValue`。

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
vecIntA.push_back(3);
vecIntA.push_back(9);
```

```
replace(vecIntA.begin(), vecIntA.end(), 3, 8); //{1,8,5,8,9}
```

## replace\_if()

- ◆ `replace_if`: 将指定范围内所有操作结果为 `true` 的元素用新值替换。

用法举例:

```
replace_if(vecIntA.begin(), vecIntA.end(), GreaterThree, newVal)
```

其中 `vecIntA` 是用 `vector<int>` 声明的容器

`GreaterThree` 函数的原型是 `bool GreaterThree(int iNum)`

//把大于等于 3 的元素替换成 8

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
vecIntA.push_back(3);
vecIntA.push_back(9);
```

```
replace_if(vecIntA.begin(), vecIntA.end(), GreaterThree, 8); // GreaterThree 的定义在上面。
```

## swap()

- ◆ `swap`: 交换两个容器的元素

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
```

```
vector<int> vecIntB;
vecIntB.push_back(2);
```

```
vecIntB.push_back(4);

swap(vecIntA, vecIntB); //交换
```

### 10.3.6 常用的算术和生成算法

#### accumulate()

◆ accumulate: 对指定范围内的元素求和，然后结果再加上一个由 **val** 指定的初始值。

◆ `#include<numeric>`

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
vecIntA.push_back(7);
vecIntA.push_back(9);
int iSum = accumulate(vecIntA.begin(), vecIntA.end(), 100); //iSum==125
```

#### fill()

◆ fill: 将输入值赋给标志范围内的所有元素。

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
vecIntA.push_back(5);
vecIntA.push_back(7);
vecIntA.push_back(9);
fill(vecIntA.begin(), vecIntA.end(), 8); //8, 8, 8, 8, 8
```

### 10.3.7 常用的集合算法

#### set\_union(),set\_intersection(),set\_difference()

◆ set\_union: 构造一个有序序列，包含两个有序序列的并集。

◆ set\_intersection: 构造一个有序序列，包含两个有序序列的交集。

◆ set\_difference: 构造一个有序序列，该序列保留第一个有序序列中存在而第二个有序序列中不存在的元素。

```
vector<int> vecIntA;
vecIntA.push_back(1);
vecIntA.push_back(3);
```



```
 vecIntA.push_back(5);
 vecIntA.push_back(7);
 vecIntA.push_back(9);

 vector<int> vecIntB;
 vecIntB.push_back(1);
 vecIntB.push_back(3);
 vecIntB.push_back(5);
 vecIntB.push_back(6);
 vecIntB.push_back(8);

 vector<int> vecIntC;
 vecIntC.resize(10);

 //并集
 set_union(vecIntA.begin(), vecIntA.end(), vecIntB.begin(), vecIntB.end(),
vecIntC.begin()); //vecIntC : {1,3,5,6,7,8,9,0,0,0}

 //交集
 fill(vecIntC.begin(),vecIntC.end(),0);
 set_intersection(vecIntA.begin(), vecIntA.end(), vecIntB.begin(), vecIntB.end(),
vecIntC.begin()); //vecIntC: {1,3,5,0,0,0,0,0,0,0}

 //差集
 fill(vecIntC.begin(),vecIntC.end(),0);
 set_difference(vecIntA.begin(), vecIntA.end(), vecIntB.begin(), vecIntB.end(),
vecIntC.begin()); //vecIntC: {7,9,0,0,0,0,0,0,0,0}
```

### 10.3.8 常用的遍历算法

#### for\_each()

- ◆ **for\_each**: 用指定函数依次对指定范围内所有元素进行迭代访问。该函数不得修改序列中的元素。

```
void show(const int &item)
{
 cout << item;
}

main()
{
 int iArray[] = {0,1,2,3,4};
 vector<int> vecInt(iArray,iArray+sizeof(iArray)/sizeof(iArray[0]));
 for_each(vecInt.begin(), vecInt.end(), show);
```

```
//结果打印出 0 1 2 3 4
}
```

## transform()

◆ transform: 与 for\_each 类似，遍历所有元素，但可对容器的元素进行修改

```
int increase (int i)
{
 return i+1;
}

main()
{
 vector<int> vecIntA;
 vecIntA.push_back(1);
 vecIntA.push_back(3);
 vecIntA.push_back(5);
 vecIntA.push_back(7);
 vecIntA.push_back(9);

 transform(vecIntA.begin(),vecIntA.end(),vecIntA.begin(),increase);
 //vecIntA : {2,4,6,8,10}
}
```

## 10.4 STL 综合案例

### 案例：学校演讲比赛

1) 某市举行一场演讲比赛，共有 24 个人参加，按参加顺序设置参赛号。比赛共三轮，前两轮为淘汰赛，第三轮为决赛。

2) 比赛方式：分组比赛

第一轮分为 4 个小组，根据参赛号顺序依次划分，比如 100-105 为一组，106-111 为第二组，依次类推，每组 6 个人，每人分别按参赛号顺序演讲。当小组演讲完后，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第二轮分为 2 个小组，每组 6 人，每个人分别按参赛号顺序演讲。当小组完后，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第三轮只剩下 6 个人，本轮为决赛，选出前三名。

选手每次要随机分组，进行比赛。

4) 比赛评分：10 个评委打分，去除最低、最高分，求平均分

每个选手演讲完由 10 个评委分别打分。该选手的最终得分是去掉一个最高分和一个最低分，求得剩下的 8 个成绩的平均分。选手的名次按得分降序排列，若得分一样，按参赛号升序排名。

用 STL 编程，求解一下问题

- 1) 请打印出所有选手的名字与参赛号，并以参赛号的升序排列。
- 2) 打印每一轮比赛前，分组情况
- 3) 打印每一轮比赛后，小组晋级名单
- 4) 打印决赛前三名，选手名称、成绩。

## 案例：足球比赛