第七周：输入输出和模板

第一节 输入输出流相关的类

1.与输入输出流操作相关的类

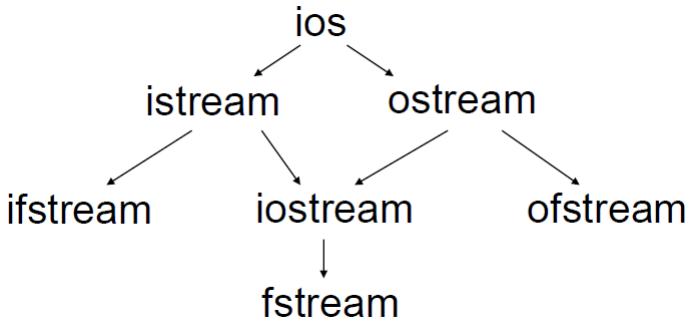

图 1.1 与输入输出流操作相关的类

istream 是用于输入的流类，cin 就是该类的对象；ostream 是用于输出的流类，cout 就是该类的对象；ifstream 是用于从文件读取数据的类；ofstream 是用于向文件写入数据的类；iostream 是既能用于输入，又能用于输出的类；fstream 是既能从文件读取数据，又能向文件写入数据的类。

2.标准流对象

（1）输入流对象：cin 与标准输入设备相连。

（2）输出流对象：cout 与标准输出设备相连；cerr，与标准错误输出设备相连；clog，与标准错误输出设备相连。缺省情况下，以下三者功能相同：

```
cerr << "Hello,world" << endl;
clog << "Hello,world" << endl;
cout << “Hello,world” << endl;
```

cin 对应于标准输入流，用于从键盘读取数据，也可以被重定向为从文件中读取数据。

- cout 对应于标准输出流，用于向屏幕输出数据，也可以被重定向为向文件写入数据。
- cerr 对应于标准错误输出流，用于向屏幕输出出错信息，
- clog 对应于标准错误输出流，用于向屏幕输出出错信息，
- cerr 和 clog 的区别在于 cerr **不使用缓冲区**,直接向显示器输出信息；而输出到 clog 中的信息**先会被存放在缓冲区,缓冲区**满或者刷新时才输出到屏幕。

3.输入重定向和输出重定向

（1）输入重定向

```
#include <iostream>
using namespace std;
int main(){
    double f; int n;
    freopen("t.txt","r",stdin);//cin 被改为从 t.txt 中读取数据
    cin>>f>>n;
    cout<<f<<","<<n<<endl;
    return 0;
}
```

（2）输出重定向

```
#include <iostream>
using namespace std;
int main(){
    int x,y;
    freopen("test.txt","w",stdout);//将标准输出重定向到 test.txt 文件
    if(y==0)
        cerr<<"error."<<endl;
    elsecout<<x/y;//结果输出到 test.txt
    return 0;
}
```

在上面这个程序里面，我们把标准输出重定向为输出到 test.txt 文件中去。这个时候，只要执行 cout 就会向 test.txt 中写文件。而如果我们这个时候出现错误需要向屏幕显示错误信息而不是向文件写入该怎么办呢？那就利用 cerr，因为 cerr 没有被重定向。

4.判断输入流结束

可以用如下方法判断输入流结束：

```
int x;
while(cin>>x){
…
}
return 0;
```

如果是从文件输入，比如前面有 freopen(“some.txt”,”r”,stdin);那么读到文件尾部，输入流就算结束。如果从键盘输入，则在单独一行输入 Ctrl+Z 代表输入流结束。

5.istream 类的成员函数

（1）istream & getline(char * buf, int bufSize);

从输入流中读取 bufSize-1 个字符到缓冲区 buf，或读到碰到‘\n’为止（哪个先到算哪个）。

（2）istream & getline(char * buf, int bufSize,char delim);

从输入流中读取 bufSize-1 个字符到缓冲区 buf，或读到碰到 delim 字符为止（哪个先到算哪个）。

两个函数都会自动在 buf 中读入数据的结尾添加\0’。,‘\n’或 delim 都不会被读入 buf，但会被从输入流中取走。

如果输入流中 ‘\n’或 delim 之前的字符个数达到或超过了 bufSize 个，就导致读入出错，其结果就是：虽然本次读入已经完成，但是之后的读入就都会失败了。

可以用 if(!cin.getline(…)) 判断输入是否结束。

bool eof(); 判断输入流是否结束

int peek(); 返回下一个字符,但不从流中去掉.

istream & putback(char c); 将字符 ch 放回输入流

istream & ignore(int nCount = 1, int delim = EOF); 从流中删掉最多 nCount 个字符，遇到 EOF 时结束。

例程：

```
#include<iostream>
using namespace std;
int main(){
    int x;
    char buf[100];
```

```
    cin >> x;
    cin.getline(buf,90);
    cout<<buf<<endl;
    return 0;
}
```

输入 12 abcd（回车），输出 abcd（即：空格+abcd）这是正常的。

但是输入 12（回车），程序立即结束，输出为空。这是为什么呢？因为 getline 读到留在流中的'\n'就会立即结束，然后返回。

第二节 用流操纵算子控制输出格式

1.流操纵算子

使用流操纵算子需要**#include <iomanip>**。

（1）整数流的基数：流操纵算子 dec，oct，hex，setbase（任何一个进制）

```
int n =10;
cout<<oct<<n<<endl;
```

设置了之后一直起作用，直到你设置了另一个流操纵算子。

（2）浮点数的精度（precision，setprecision）——对整型无影响

precision 是成员函数，其调用方式为 cout.precision(5);

setprecision 是流操作算子，其调用方式为 cout<<setprecision(5);//可以连续输出。

它们的功能相同。

指定输出浮点数的有效位数（非定点方式输出时）

指定输出浮点数的小数点后的有效位数（定点方式输出时）定点方式：小数点必须出现在个位数后面。

例程：

①非定点方式

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 1234567.89,y = 12.34567; int n = 1234567;
    int m = 12;
    cout << setprecision(6) << x << endl<< y << endl << n << endl << m;
}//默认为非定点
```

输出：1.23457e+006

12.3457

1234567

12

②设置定点方式如下：

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

```
{
    double x = 1234567.89,y = 12.34567; int n = 1234567;
    int m = 12;
    cout <<setiosflags(ios::fixed)<< setprecision(6) << x << endl<< y << endl << n << endl << m;
}
```

输出：1234567.890000　　12.345670　　1234567　　12

③取消定点

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 1234567.89;
    int m = 12;
    cout <<setiosflags(ios::fixed)<< setprecision(6) << x << endl<< resetiosflags(ios:fixed) << x;
}
```

输出：1234567.890000　　1.23457e+006

（3）设置域宽(setw，width)

两者功能相同，一个是成员函数（width），另一个是流操作算子（setw），调用方式不同：cin >> setw(4); 或者 cin.width(5); cout << setw(4); 或者 cout.width(5);

宽度设置有效性是一次性的，在每次读入和输出之前都要设置宽度。

例程：

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int n = 141;
//1) 分别以十六进制、十进制、八进制先后输出 n
    cout << "1) " << hex << n << " " << dec << n << " " << oct << n << endl; double x = 1234567.89,y = 12.34567;
//2) 保留 5 位有效数字
    cout << "2) " << setprecision(5) << x << " " << y << " " << endl;
//3) 保留小数点后面 5 位
    cout << "3) " << fixed << setprecision(5) <<    x << " " << y << endl ;
//4) 科学计数法输出，且保留小数点后面 5 位
    cout << "4) " << scientific << setprecision(5) <<x << " " << y << endl ;
//5) 非负数要显示正号，输出宽度为 12 字符，showpos 表示非负数要写出正号来。宽度不足则用'*'填补
    cout << "5) " << showpos << fixed << setw(12) << setfill('*') << 12.1
//6) 非负数不显示正号，输出宽度为 12 字符，宽度不足则右边用填充字符填充
    cout << "6) " << noshowpos << setw(12) << left << 12.1 << endl;
//7) 输出宽度为 12 字符，宽度不足则左边用填充字符填充
    cout << "7) " << setw(12) << right << 12.1 << endl;
```

```
//8) 宽度不足时，负号和数值分列左右，中间用填充字符填充
    cout << "8) " << setw(12) << internal << -12.1 << endl;
    cout << "9) " << 12.1 << endl;
    return 0;
}
```

输出结果如下：

```
1)  8d    141    215
2)  1.2346e+006    12.346
3)  1234567.89000    12.34567
4)  1.23457e+006    1.23457e+001
5)  ***+12.10000
6)  12.10000****
7)  ****12.10000
8)  -***12.10000
9)  12.10000
```

（4）用户自定义的流操纵算子等。

例程：

```
ostream &tab(ostream &output){
    return output<<'\t';
}
cout<<"aa"<<tab<<"bb"<<endl;
```

输出：aa　　bb

为什么能进行这样的操作呢？因为 iostream 里面对<<进行了重载（成员函数）：

```
ostream &operator<<(ostream&(*p)(osream&));
```

该函数内部会调用 p 所指向的函数，且以*this 作为参数。

第三节 文件读写（一）

1.创建文件

可以将顺序文件看作一个有限字符构成的顺序字符流，然后像对 cin，cout 一样的读写。

```
#include <fstream>//包含头文件
ofstream outFile("cliends.dat",ios::out|ios::binary);
```

在这其中，clients.dat 是要创建的文件的名字。ios::out 是文件打开方式，ios:out 输出到文件，删除原有内容。ios::app 输出到文件，保留原有内容，总是在尾部添加。ios::binary 以二进制文件格式打开文件。

也可以先创建 ofstream 对象，再用 open 函数打开：

```
ofstream fout;
fout.open("test.out",ios::out|ios:binary);
```

判断打开是否成功：

```
if(!fout){cout<<"File open error!"<<endl;}
```

文件名可以给出绝对路径，也可以给相对路径。没有交代路径信息，就是在当前文件夹下找文件。

2.文件名的绝对路径和相对路径

如图所示。

- 绝对路径:

```
"c:\\tmp\\mydir\\some.txt"
```

- 相对路径：

```
"\\tmp\\mydir\\some.txt"
```

当前盘符的根目录下的tmp\dir\some.txt

```
"tmp\\mydir\\some.txt"
```

当前文件夹的tmp子文件夹里面的.....

```
"..\\tmp\\mydir\\some.txt"
```

当前文件夹的父文件夹下面的tmp子文件夹里面的.....

```
"..\\..\\tmp\\mydir\\some.txt"
```

当前文件夹的父文件夹的父文件夹下面的tmp子文件夹里面的.....

图 3.1 文件路径

3.文件的读写指针

对于输入文件，有一个读指针；对于输出文件，有一个写指针；对于输入输出文件，有一个读写指针；标识文件操作的当前位置，该指针在哪里，读写操作就在哪里进行。

例程 1：

```
ofstream fout("a1.out",ios::app)//以添加方式打开
long location = fout.tellp(); //获取写指针的位置
location = 10;
fout.seekp(location);//将写指针移动到第 10 个字节处
fout.seekp(location,ios::beg); //从头数 location
location fout.seekp(location,ios::cur); //从当前位置数 location
location fout.seekp(location,ios::end); //从尾部数 location
```

例程 2：

```
ifstream fout("a1.in",ios::ate)
//打开文件，定位文件指针到文件尾
long location = fin.tellg();     //取得读指针的位置，获得文件的长度。
location = 10L;
fin.seekg(location); //  将读指针移动到第 10 个字节处
fin.seekg(location,ios::beg); //从头数 location
location fin.seekg(location,ios::cur); //从当前位置数 location
location fin.seekg(location,ios::end); //从尾部数 location
```

location 可以为负数！

4.显式关闭文件

读文件：

```
ifstream fin(“test.dat”,ios::in);
fin.close();
```

写文件：

```
ofstream fout(“test.dat”,ios::out);
fout.close();
```

5.字符文件读写

因为文件流也是流，所以流的成员函数和流操作算子也同样适用于文件流。写一个程序，将文件 in.txt 里面的整数排序后，输出到 out.txt。

程序如下：

```
#include <iostream>
```

```
#include <fstream>
#include <vector>
#include <algorithm> using namespace std;
int main()
{
    vector<int> v;
    ifstream srcFile("in.txt",ios::in);
    ofstream destFile("out.txt",ios::out);
    int x;
    while( srcFile >> x )
        v.push_back(x);
    sort(v.begin(),v.end());
    for( int i = 0;i < v.size();i ++ )
        destFile << v[i] << " "; destFile.close();
    srcFile.close();
    return 0;
}
```

第四节 文件读写（二）

1.二进制文件读写

（1）二进制读文件

ifstream 和 fstream 的成员函数：

istream& read (char* s, long n);

将文件读指针指向的地方的 n 个字节内容，读入到内存地址 s，然后将文件读指针向后移动 n 字节 (以 ios::in 方式打开文件时，文件读指针开始指向文件开头)。

（2）二进制写文件

ofstream 和 fstream 的成员函数：

istream& write(const char* s,long n);

将内存地址 s 处的 n 个字节内容，写入到文件中写指针指向的位置，然后将文件写指针向后移动 n 字节(以 ios::out 方式打开文件时，文件写指针开始指向文件开头, 以 ios::app 方式打开文件时，文件写指针开始指向文件尾部)。

例程：在文件中写入和读取一个整数

```
#include <iostream>
#include <fstream> using namespace std;
int main()  {
    ofstream fout("some.dat", ios::out | ios::binary); int x=120;
    fout.write( (const char *)(&x), sizeof(int) ); fout.close();
    ifstream fin("some.dat",ios::in | ios::binary); int y;
    fin.read((char * ) & y,sizeof(int));
    fin.close();
    cout << y <<endl;
    return 0;
}
```

例程 2：从键盘输入几个学生的姓名的成绩，并以二进制文件形式保存

```
#include <iostream>
#include <fstream>
using namespace std;
struct Student {
    char name[20];
    int score;
};
int main(){
    Student s;
    ofstream OutFile("c:\\tmp\\students.dat",ios::out|ios::binary);
    while(cin>>s.name>>s.core)
        OutFile.write((char*)&s,sizeof(s));
    OutFile.close();
    return 0;
}
```

例程 3：将 students.dat 文件的内容独出并显示

```
#include <iostream>
#include <fstream>
using namespace std;
struct Student {
    char name[20];
    int score;
};
int main(){
    Student s;
    ifstream inFile("students.dat",ios::in | ios::binary );
    if(!inFile) {
        cout << "error" <<endl;
        return 0;
    }
    while( inFile.read( (char* ) & s, sizeof(s) ) ) {
        int readedBytes = inFile.gcount();//看读了多少字节
        cout<<s.name<<" "<<s.score<<endl;
    }
    inFile.close();
    return 0;
}
```

例程 4：将 students.dat 文件的 Jane 的名字替换成 Mike

```
#include <iostream>
#include <fstream>
using namespace std;
struct Student {
    char name[20];
```

```
    int score;
};
int main(){
    Student s;
    fstream iofile( "c:\\tmp\\students.dat", ios::in|ios::out|ios::binary);
    if( !iofile) {
        cout << "error" ;
        return 0;
    }
    iofile.seekp( 2 * sizeof(s),ios::beg);   //定位写指针到第三个记录
    iofile.write("Mike",strlen("Mike")+1);
    iofile.seekg(0,ios::beg); //定位读指针到开头
    while( iofile.read( (char* ) & s, sizeof(s)) )
        cout << s.name << " " << s.score << endl; iofile.close();
    return 0;
}
```

例程 5：文件拷贝程序 mycopy 示例

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char * argv[]){
    if(argc!=3){
        cout<<"File name missing!"<<endl;
        return 0;
    }
    ifstream inFile(argv[1],ios::binary|ios::in);//打开文件用于读
    if(!inFile){
        cout<<"Source file open error"<<endl;
        return 0;
    }
    ofstream outFile(argv[2],ios::binary|ios::out);//打开文件用于写
    if(!inFile){
        cout<<"New file open error"<<endl;
        inFile.close();//务必把要读的文件关闭
        return 0;
    }
    char c;
    while(inFile.get(c))//每次读取一个字符，但是操作系统已经把硬盘上较大的空间读在内
存了
        outFile.put(c);//每次写入一个字符
    outFile.close();
    inFile.close();
    return 0;
}
```

2.二进制文件和文本文件的区别

Linux,Unix 下的换行符号：‘\n’（ASCII 码：0x0a）。

Windows 下的换行符号：‘\r\n’（ASCII 码： 0x0d0a），endl 就是 '\n'。

Mac OS 下的换行符号：‘\r’（ASCII 码：0x0d）。

导致 Linux，Mac OS 文本文件在 Windows 记事本中打开时不换行。

Unix/Linux 下打开文件，用不用 ios::binary 没区别。但是在 Windows 下，如果不用，则会出现以下情况：

（1）读取文件时，所有的'\r\n’会被当做一个字符'\n'处理，即少读了一个字符'\r'。

（2）写入文件时，写入单独的'\n'时，系统自动在前面加一个'\r'，即多写了一个'\r'

第五节 函数模板（泛型程序设计）

1.函数模板基础

格式如下：

```
template <class 类型参数 1, class 类型参数 2,.....>
返回值类型 模板名(形参表)
{
    函数体
};
```

举例如下：

```
template <class T>
void Swap(T & x, T& y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
int main()
{
    int n =1, m =2;
    Swap(n,m);//编译器自动生成 void Swap(int & , int &)
    double f = 1.2,g=2.3;
    Swap(f,g);//编译器自动生成 void Swap(double & , double &)
    return 0;
}
```

函数模板中可以不只有一个类型参数。例如：

```
template <class T1, class T2>
T2 print(T1 arg1, T2 arg2)
{
    cout<< arg1 << " "<< arg2<<endl;
    return arg2;
}
```

例程 1：求数组最大元素的 MaxElement 函数模板

```
template <class T>
```

```
T MaxElement(T a[], int size){
    T = tmpMax = a[0];
    for(int i = 0;i<size;i++)
    {
        if(tmpMax < a[i])
            tmpMax = a[i];
    }
    return tmpMax;
}
```

以上称为模板实例化，是通过参数实例化的。

例程 2：不通过参数实例化函数模板

```
#include <iostream>
using namespace std;
template <class T>
T Inc(T n)
{
    return 1 + n;
}
int main()
{
    cout << Inc<double>(4)/2;   //在 Inc 后面加一个<double>就是一个实例化。实例化后是 5，然后除以 2 是 2.5
    return 0;
}
```

2.函数模板的重载

函数模板可以重载，只要它们的**形参表或类型参数表**不同即可。

3.函数模板和函数的次序

在有多个函数和函数模板名字相同的情况下，编译器如下处理一条函数调用语句：

（1）先找参数完全匹配的普通函数(非由模板实例化而得的函数)。

（2）再找参数完全匹配的模板函数。（**匹配模板函数时，不进行类型自动转换!!!**）

（3）再找实参数经过自动类型转换后能够匹配的普通函数。

（4）上面的都找不到，则报错。

函数模板实例：Map

```
#include <iostream>
using namespace std;
template<class T,class Pred>
void Map(T s, T e, T x, Pred op){
    //往往 s 和 e 都是指针，s 是区间的开始位置，e 是区间的结束位置，x 也是指针，op 是函数指针
    for(;s!=e;++s,++x){
        *x = op(*s);//把 s 指向的东西（*s）通过 op 变换放到 x 指向的地方（*x）
    }
}
```

```
int Cube(int x){return x*x*x;}
double Square(double x){return x*x;}
int a[5] = {1,2,3,4,5}, b[5];//a 为源区间，b 为目标区间
double d[5] = { 1.1,2.1,3.1,4.1,5.1} , c[5]; //d 为源区间，c 为目标区间
int main() {
    Map(a,a+5,b,Square);
    for(int i =0;i<5;++i)
        cout<<b[i]<<",";
    cout<<endl;

    Map(a,a+5,b,Cube);
    for(int i = 0;i<5;++i)
        cout<<b[i]<<",";
    cout<<endl;

    Map(d,d+5,c,Square);
    for(int i=0;i<5;++i)
        cout<<c[i]<<",";
    cout<<endl;
    return 0;
}
```

输出：

```
1,4,9,16,25,
1,8,27,64,125,
1.21,4.41,9.61,16.81,26.01,
```

来分析以下，以第一个 Map 为例：

在调用的时候，Map 模板实际上变成了这样：

```
void Map(int * s, int * e, int * x, double ( *op)(double)) {
    for(;s!=e;++s,++x){
        *x = op(*s);
    }
}
```

第六节 类模板

为了多快好省地定义出一批相似的类,可以定义类模板,然后由类模板生成不同的类。

考虑一个可变长数组类，需要提供的基本操作

- len()：查看数组的长度
- getElement(int index)：获取其中的一个元素
- setElement(int index)：对其中的一个元素进行赋值

这些数组类，除了元素的类型不同之外，其他的完全相同。

类模板：在定义类的时候，加上一个/多个类型参数。在使用类模板时，指定类型参数应该如何替换成具体类型，编译器据此生成相应的模板类。

1.类模板的定义

```
template <class 类型参数 1，class 类型参数 2，……> //类型参数表
class 类模板名{
    //成员函数和成员变量
};
```

或

```
template <typename 类型参数 1，typename 类型参数 2，……>
class 类模板名{
    //成员函数和成员变量
};
```

类模板例成员函数的写法：

```
template <class 类型参数 1，class 类型参数 2，……> //类型参数表
返回值类型 类模板名<类型参数名列表>::成员函数名（参数表）{
//函数体
}
```

用类模板定义对象的写法：

```
类模板名 <真实类型参数表>  对象名(构造函数实参表);
```

类模板示例：Pair 类模板

```
template <class T1,class T2> class Pair
{
public:
    T1 key;   //关键字
    T2 value;        //值
    Pair(T1 k,T2 v):key(k),value(v) { };
    bool operator < ( const Pair<T1,T2> & p) const;
};
template<class T1,class T2> bool Pair<T1,T2>::operator <( const Pair<T1,T2> & p) const
//Pair 的成员函数  operator
{
    return key < p.key;
}
int main(){
    Pair<string,int> student("Tom",19);
//实例化出一个类  Pair<string,int>
    cout << student.key << " " << student.value;
    return 0;
}
```

输出：Tom 19

2.用类模板定义对象

编译器由类模板生成类的过程叫类模板的实例化。由类模板实例化得到的类，叫模板类。

同一个类模板的两个模板类是不兼容的

3.函数模板作为类模板成员

```
#include <iostream>
using namespace std;
template <class T>
class A
{
public:
    template<class T2>
    void Func(T2 t){cout<<t;}//成员函数模板
};
int main(){
    A<int> a;
    a.Func('K');//成员函数模板 Func 被实例化
    a.Func("hello");//成员函数模板再次被实例化
    return 0;
}
```

输出：Khello

4.类模板与非类型参数

类模板的“<类型参数表>”中可以出现非类型参数：

```
template <class T, int size>
class CArray{
    T    array[size];
public:
    void Print( ) {
        for( int i = 0;i < size; ++i)
            cout<<array[i]<<endl;
    }
};
CArray<double,40> a2;
CArray<int,50> a3;
//a2 和 a3 属于不同的类
```

第七节 类模板与派生、友元和静态成员变量

类模板的派生有以下四种情况：

（1）类模板从类模板派生；

（2）类模板从模板类派生；

（3）类模板从普通类派生；

（4）普通类从模板类派生。

1.类模板从类模板派生

例如：

```
template <class T1,class T2>
class A   {
    T1 v1; T2 v2;
};
```

```
template <class T1,class T2>
class B:public A<T2,T1>{
    T1 V3;T2 v4;
};
template<class T>
class C:public B<T,T>{
    T v5;
};
int main(){
    B<int,double> obj1;
    C<int> obj2;
    return 0;
}
```

通过 B<int, double> obj1;实例化出：

```
class B<int, double>:
    public A<double, int>
{
    int v3;double v4;
};
class A<double,int>
{
    double v1;int v2;
};
```

2.类模板从模板类派生

```
template <class T1,class T2>
class A {
T1 v1; T2 v2;
};
template <class T>
class B:public A<int,double>{
    T v;
};
int main(){
    B<char> obj1;//自动生成两个模板类：A<int,double>和 B<char>
    return 0;
}
```

3.类模板从普通类派生

```
class A{
    int v1;
};
template <class T>
class B:public A{//所有从 B 实例化得到的类，都以 A 为基类
    T v;
```

```
};
int main(){
    B<char> obj1;
    return 0;
}
```

4.普通类从模板类派生

```
template <class T>
class A {
    T v1;
    int n;
};

class B:public A<int>{
    double v;
};
int main(){
    B obj1;
    return 0;
}
```

5.类模板与友元

有以下类型：

①函数、类、类的成员函数作为类模板的友元；
②函数模板作为类模板的友元；
③函数模板作为类的友元；
④类模板作为类模板的友元。

（1）.函数、类、类的成员函数作为类模板的友元

```
void Func1() { }
class A {   };
class B
{
public:
    void Func(){}
};
template<class T>
class Tmpl
{
    friend void Func1();
    friend class A;
    friend void B::Func();
};
//任何从 Tmp1 实例化出来的类，都有以上三个友元
```

（2）.函数模板作为类模板的友元

```
#include <iostream>
#include <string>
using namespace std;
template <class T1,class T2>
class Pair
{
private:
	T1 key;		//关键字
	T2 value;	//值
public:
	Pair(T1 k,T2 v):key(k),value(v) { };
	bool operator < ( const Pair<T1,T2> & p) const;
	template <class T3,class T4>
	friend ostream & operator<< ( ostream & o, const Pair<T3,T4> & p);
};
template<class T1,class T2>
bool Pair<T1,T2>::operator <( const Pair<T1,T2> & p) const
{//小的意思就是关键字小
	return key < p.key;
}
template <class T1,class T2>
ostream & operator<< (ostream & o,const Pair<T1,T2> & p)
{
	o<< "(" << p.key << "," << p.value << ")" ;
	return o;
}
int main()
{
	Pair<string,int> student("Tom",29);
	Pair<int,double> obj(12,3.14);
	cout << student << " " << obj;
	return 0;
}
```

输出：

(Tom,29)(12,3.14)

任意从 template <class T1,class T2>

ostream & operator<< (ostream & o,const Pair<T1,T2> & p)生成的函数，都是任意 Pair 摸板类的友元

（3）.函数模板作为类的友元

```
#include <iostream>
using namespace std;
class A
{
    int v;
public:
    A(int n):v(n) { }
    template <class T>
    friend void Print(const T & p);
};
template <class T>
void Print(const T & p)
{
        cout << p.v;
}
```

```
int main()  {
    A a(4);
    Print(a);
    return 0;
}
```

输出：

4

所有从 `template <class T> void Print(const T & p)` 生成的函数，都成为 A 的友元

但是自己写的函数 `void Print(int a) { }` 不会成为A的友元

图 7.1 函数模板作为类的友元

（4）.类模板作为类模板的友元

```
#include <iostream>
using namespace std;
template <class T>
class B  {
    T v;
public:
    B(T n):v(n) { }
    template <class T2>
    friend class A;
};
template <class T>
class A  {
  public:
    void Func( )   {
       B<int> o(10);
       cout << o.v << endl;
    }
};
```

```
int main()
{
  A< double > a;
  a.Func ();
  return 0;
}
```

输出：

10

A< double>类，成了B<int>类的友元。任何从A模版实例化出来的类，都是任何B实例化出来的类的友元

图 7.2 类模板作为类模板的友元

6.类模板与静态成员变量

（1）类模板与 static 成员

类模板中可以定义静态成员，那么从该类模板实例化所得到的所有类，都包含同样的静态成员。

```
#include <iostream>
using namespace std;
template <class T> class A
{
private:
    static int count;
public:
    A(){count++;}
    ~A(){count--;}
    A(A&){count++;}
    static void PrintCount(){cout<<count<<endl;}
};
template<> int A<int>::count = 0;
template<> int A<double>::count = 0;
int main()
{
    A<int> ia;
```

```
    A<double> da;
    ia.PrintCount();
    da.PrintCount();
    return 0;
}
```

输出：

```
1
1
```