



第5章

模 板

内容安排



- 1 模板与泛型编程
- 2 函数模板
- 3 类模板
- 4 几个例子



1 模板与泛型编程

[引例]

```
int max( int x, int y)
{
    return (x>y) ? x: y;
}
double max( double x, double y)
{
    return (x>y) ? x: y;
}
char max( char x, char y)
{
    return (x>y) ? x: y;
}
```

可以看出，这些函数版本的功能都是相同的，只是参数类型和函数返回类型不同。

那么能否为这些函数只写出一套代码呢？

C++解决这个问题的一個方法就是使用模板。



1 模板与泛型编程

- 所谓**泛型编程**就是以独立于任何特定类型的方式编写代码。在使用时，再**提供**具体程序实例所操作的**类型和值**。
- **模板**是泛型编程的基础。
- STL（静态模板库）里面容器、迭代器和算法都是**泛型编程**。



1 模板与泛型编程

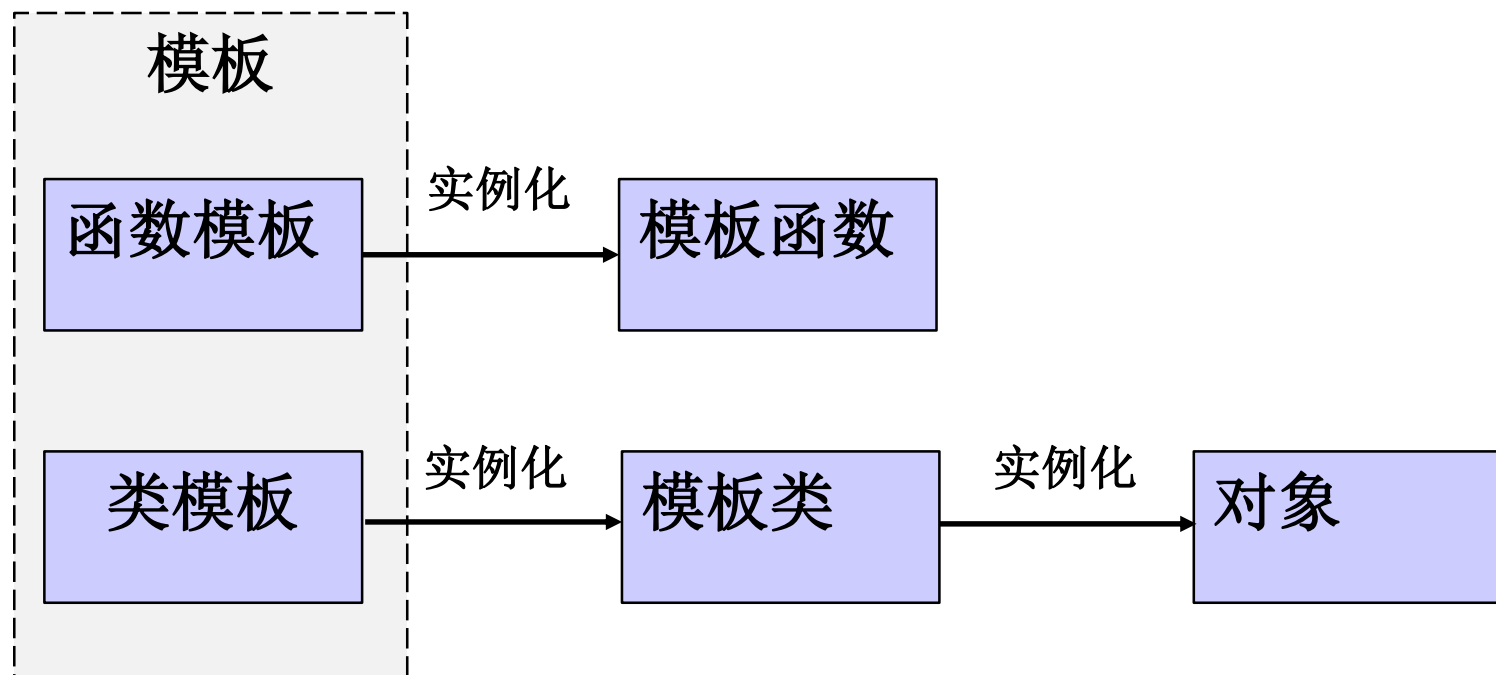


- 当对数据的处理方式相同，而参加运算的数据类型不同时，可定义为模板。
- 模板通过把类型定义为参数，实现类型参数化。“类型参数化设计”是在定义函数或类的时候，将其中某些形参、变量、返回值的类型用“模板参数”替代，形成函数模板或类模板。
- 使用模板可以建立起通用类型的函数库和类库，减少程序开发的重复及代码冗余，是实现代码重用机制的一种工具。
- 编译系统根据模板，可以生成能对不同类型数据操作的函数或类。
- C++中的模板分为类模板和函数模板。



1 模板与泛型编程

模板、模板函数、模板类和对象之间的关系





2 函数模板

- 2.1 函数模板和模板函数
- 2.2 函数模板定义
- 2.3 函数模板的使用
- 2.4 函数模板形参实参匹配问题
- 2.5 `typename` 和 `class` 的区别
- 2.6 非类型模板形参





2.1 函数模板和模板函数

● 函数模板

- 实质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（如：T）来代替，这个通用函数就称为函数模板。

● 模板函数

- 在定义了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数，该函数的定义与函数模板的函数定义体相同，称之为模板函数。



2.2 函数模板的定义



`template` <类型形式参数表>

函数返回值类型 函数模板名（形参表）

{ 函数体 }

} 两行函数首部，中间不能有其他语句，也可写成一行

○其中：类型形式参数表（或称类型参数表，模板形参）可写成如下形式：

<class(或typename) Type>

class、typename是关键字，表示定义类型形参；Type为类型形式参数名（或类型占位符），由用户命名，一般大写。

○如：`template <class T>`

`T max(T x, T y)`

`{`

`return x>y?x:y;`

`}`





2.2 函数模板的定义

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if(v1 < v2)
        return -1;
    if(v1 > v2)
        return 1;
    return 0;
}
```

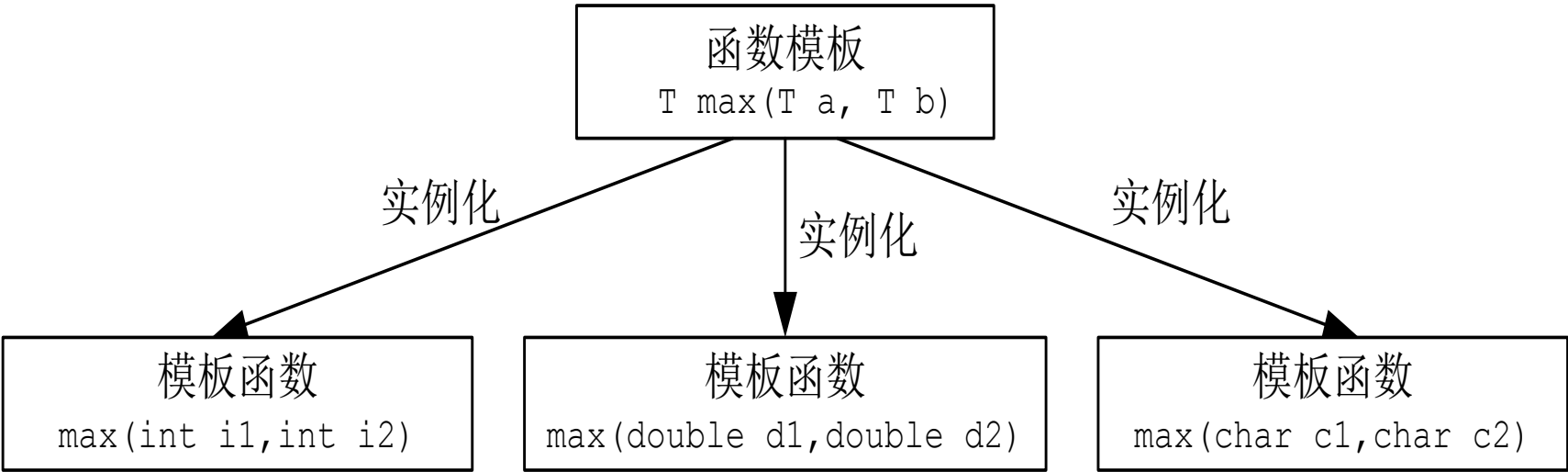
- 函数模板形参表不能为空
- 测试这一函数模板





2.3 函数模板的使用

- 【例1】 定义一个求任意两个具有相同类型的数据中较大值的函数模板，然后进行调用并完成相应的功能。





- **#include <iostream>**
- **using namespace std;**
- **template <typename T>**
- **T & max(T & a,T & b)**
- **{**
- **return a>b ? a : b;**
- **}**
- **int main()**
- **{**
- **int i1 = 30, i2 = 50;**
- **double d1 = 3.33, d2 = 4.44;**
- **char c1 = 'A', c2 = 'B';**
- **cout<<"较大的整数为: "<<max(i1,i2)<<"\n";**
- **cout<<"较大的浮点数为: "<<max(d1,d2)<<"\n";**
- **cout<<"较大的字符为: "<<max(c1,c2)<<endl;**
- **return 0;**
- **}**





2.4 函数模板形参实参匹配问题

```
#include <iostream>
```

```
using namespace std;
```

```
template <class T>    //定义函数模板min，类型形参为T，它表示某种类型
```

```
T min(T a, T b)      //函数值的类型为T，形参a，b的类型为T
```

```
{
```

```
    if (a<b)
```

```
        return a;
```

```
    else
```

```
        return b;
```

```
}
```





2.4 函数模板形参实参匹配问题

```
void main()
{
    int i1=10, i2=20;
    double d1=3.5, d2=-1.2;
    char c1='b', c2='x';
    cout<<min(i1,i2)<<endl;
    cout<<min(d1,d2)<<endl;
    cout<<min(c1,c2)<<endl;
    cout<<min(i1,c1)<<endl;
    cout<<min(i1,d1)<<endl;
}
```

凡是用同一模板参数T定义的各参数必须保持完全一致的类型，模板类型并不具有隐式的类型转换，如在int和char之间、float与int之间、float与double之间的隐式类型转换。

2.4 函数模板形参实参匹配问题

思考题：如何解决实参的常规转换？

解决办法：

(1) 采用强制类型转换。

[例] 将调用语句`min(i1,c1)`改写为：

`min(i1,(int)c1)`

(2) 显式给出模板实参，强制生成对特定实例的调用。具体地说，就是在调用格式中要插入一个模板实参表。

[例] 将调用语句`min(i1,c1)` 和 `min(i1,d1)` 分别改写为：

`min<int>(i1,c1)`

`min<double>(i1,d1)`





2.4 函数模板形参实参匹配问题

(3) 将函数模板中< >的类型参数定义为两个类型参数分别为T1和T2，分别接受不同的数据类型。函数模板的返回类型参数为T1或T2。

```
template <class T1,class T2>
T1 min(T1 a, T2 b)
{
    if (a<b)
        return a;
    else
        return b;
}
```





2.5 typename 和 class 的区别

- 在函数模板形参表中，typename和class声明类型形参时，功能相同，可以互换
- 此外，typename还有另外一个作用，当在模板声明中，类型形参含有子类型成员时，可以在成员名前加上关键字typename作为前缀，告诉编译器将成员当作类型对待。

```
template <class Parm, class U>
Parm fcn(Parm * array, U value)
{
    typename Parm::size_type * p; //ok: declares p to be a pointer
}
```

如果没有typename，则表示size_type为Parm类型的静态成员
实例化时，Parm必须有名为size_type的类型成员





2.6 非类型模板形参

● 模板形参不必都是类型。

```
template <class T, int N>
void array_init(T (&parm)[N]) //通过引用传递数组, parm是变量名
{
    for(int i = 0; i != N; i++)
        parm[i] = 0;
}
```

当调用array_init时, 编译器从数组实参计算非类型形参的值, 例如:

```
int x[42];
double y[10];
array_init(x); //N=42
array_init(y); //N=10
```





3 类模板

- 3.1 类模板简介
- 3.2 类模板的定义
- 3.3 类模板举例
- 3.4 类模板中的成员函数的定义
- 3.5 类模板的对象（即实例）的定义
- 3.6 类模板的使用
- 3.7 使用默认参数的类模板
- 3.8 类模板的派生





3.1 类模板简介

- 类是对一组对象的公共性质的抽象，而类模板则是对一组类的公共性质的抽象。
- 类模板是一系列相关类的模板，这些相关类的成员组成相同，成员函数的源代码形式也相同，不同的只是所针对的类型不同。
- 类模板为类声明了一种模式，使得类中的某些数据成员、成员函数的参数和成员函数的返回值能取任意类型（包括系统预定的和用户自定义的）。
- 类模板，也叫参数化类



3.2 类模板的定义

```
template <类型形参表>
class 类模板名
{
    private:
        私有成员定义
    protected:
        保护成员定义
    public:
        公有成员定义
};
```

利用类模板定义的只是对类的描述，本身不是一个实实在在的类，是类模板

- 说明：“类型形参表”中的参数，可看作是形式参数，各参数需要给出其参数名，参数的类型可由下列标识符修饰：
- ① “**class**”标识符：表示该参数可以接受一个类型参数。
- ② “类型说明符”标识符：表示该参数可以接受一个由“类型说明符”所规定类型的常量作为参数。
- 当“类型形参表”同时有多个参数时，各参数间需用逗号隔开。

3.3 类模板举例

```
template <class T, int N>
class Array
{
    private:
        T arr[N];
    public:
        Array();
        void set();
};
```





3.4 类模板中的成员函数的定义

成员函数既可在类模板的定义体中定义，也可在类模板的外部定义

- 在类模板外定义成员函数的格式：

template <类型形式参数表>

函数值的返回类型 类模板名<类型名表>::成员函数(形参)

{
 函数体
}

类模板中定义的名称

类模板定义中的类型形式参数表中的参数名





类模板外成员函数的实现

```
template<class T, int N>
Array<T, N>::Array()
{
    函数体
}
template<class T, int N>
void Array<T, N>::set()
{
    函数体
}
```





3.5 类模板的对象（即实例）的定义

- 用类模板定义对象格式：

类模板名<类型实参表> 对象名表；

- 其中：

- 类型实参表是由逗号分隔的若干类型标识符或常量表达式构成，它与类模板定义中的类型形参表中的参数的个数和类型要一一对应。
- 如果类模板定义中的类型形参表中的某个参数类型标识为class，则此处的实参应为某个类型标识符；
- 如果类模板定义中的类型形参表中的某个参数类型标识为某种具体类型标识符，则此处的实参应为该种类型的常量

- 举例：Array<int, 10> m;





3.6 类模板的使用


- 【例2】 定义类模板ABC，其内含成员函数set和get。用ABC生成对象abc1和abc2。它们的数组类型和元素数不同，显示的结果也不同。

成员函数定义体放于类模板内部定义


- 【例3】 定义类模板ABC，其内含成员函数set和get。用ABC生成对象abc1和abc2。它们的数组类型和元素数不同，显示的结果也不同。

成员函数定义体放于类模板外部定义





```
#include <iostream> //例2
using namespace std;
template <class T, int N>
class ABC
{
private:
    T array[N];
public:
    void set(T base)
    {
        for(int i = 0; i < N; i++)
            array[i] = base + i;
    }
    void get()
    {
        cout << "数组元素总数为: " << N << "\n";
        for(int i = 0; i < N; i++)
            cout << array[i] << " ";
        cout << endl;
    }
};
```



```
int main()
{
    ABC <int, 10> abc1;
    abc1.set(0);
    abc1.get();
    ABC <double, 20> abc2;
    abc2.set(100);
    abc2.get();
    return 0;
}
```

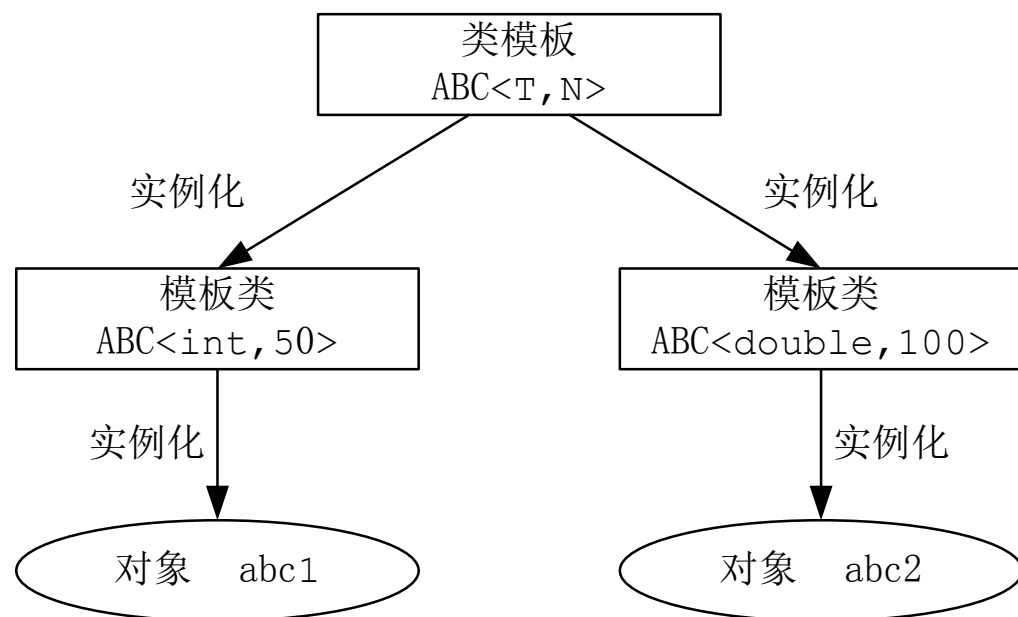
- **#include <iostream> //例3**
- **using namespace std;**
- **template <class T, int N>**
- **class ABC**
- **{**
- **private:**
- **T array[N];**
- **public:**
- **void set(T base);**
- **void get();**
- **};**
- **template <class T, int N>**
- **void ABC<T,N>::set(T base)**
- **{**
- **for(int i = 0;i<N;i++)**
- **array[i] = base + i;**
- **}**

- **template <class T, int N>**
- **void ABC<T,N>::get()**
- **{**
- **cout<<"数组元素总数为: "<<N<<"\n";**
- **for(int i = 0; i<N;i++)**
- **cout<<array[i]<<" ";**
- **cout<<endl;**
- **}**
- **int main()**
- **{**
- **ABC <int,10> abc1;**
- **abc1.set(0);**
- **abc1.get();**
- **ABC <double, 20> abc2;**
- **abc2.set(100);**
- **abc2.get();**
- **return 0;**
- **}**



类模板的使用方法

- (1) 给出类模板的定义体。
- (2) 在适当的位置创建一个类模板的实例，即一个实实在在的类定义，同时创建该模板类的对象。
- (3) 有了对象名，其以后的使用就和普通类对象的使用相一致。





3.7 使用默认参数的类模板

```
template <class T=int, int N=10>
class Array
{
    private:
        T arr[N];
    public:
        Array();
        void set();
};
```



使用默认参数的类模板

- `#include <iostream> // 例 4`
- `using namespace std;`
- `template <class T=int, int N=10>`
- `class ABC`
- `{`
- `private:`
- `T array[N];`
- `public:`
- `void set(T base);`
- `void get();`
- `};`
- `template <class T, int N>`
- `void ABC<T, N>::set(T base)`
- `{`
- `for (int i = 0; i<N; i++)`
- `array[i] = base + i;`
- `}`

- `template <class T, int N>`
- `void ABC<T, N>::get()`
- `{`
- `cout << "数组元素总数为: " << N << "\n";`
- `for (int i = 0; i<N; i++)`
- `cout << array[i] << " ";`
- `cout << endl;`
- `}`
- `int main()`
- `{`
- `ABC <double, 20> abc1; // double 型数组, 共 20 个元素`
- `abc1.set(0);`
- `abc1.get();`
- `ABC <double> abc2; // double 型数组, 元素个数默认`
- `abc2.set(100);`
- `abc2.get();`
- `ABC <> abc3; // 都取默认值, 即类型是 int, 元素个数 10 个`
- `abc3.set(10);`
- `abc3.get();`
- `return 0;`
- `}`





3.8 类模板的派生

- 类模板的派生有2种方式：
 - (1) 从类模板派生类模板
 - (2) 从类模板派生非模板类（普通类）



(1) 从类模板派生类模板



- 从一个已有的类模板派生出新的类模板，格式如下：
- `template <class T>`
- `class Base`
- `{`
- `...`
- `};`
- `template <class T>`
- `class Derived:public Base<T>`
- `{`
- `...`
- `};`
- 例5:类模板Rectangle是类模板Point的派生类，利用类模板Rectangle求矩形的位置、长度、宽度和面积。





- `#include <iostream>`
- `using namespace std;`
- `template <class T>`
- `class Point`
- `{`
- `public:`
- `Point(T i, T j) :x(i), y(j) {}`
- `T GetX() { return x; }`
- `T GetY() { return y; }`
- `private:`
- `T x, y;`
- `};`

- `template<class T>`
- `class Rectangle :public Point<T>`
- `{`
- `public:`
- `Rectangle(T x, T y, T w, T h) :Point(x, y)`
- `{`
- `width = w;`
- `height = h;`
- `}`
- `T GetW() { return width; }`
- `T GetH() { return height; }`
- `T Area() { return width*height; }`
- `private:`
- `T width, height;`
- `};`





```
int main()
{
    Rectangle<int> a(1, 2, 5, 8);
    cout << "Top is " << a.GetX() << endl;
    cout << "Left is " << a.GetY() << endl;
    cout << "Width is " << a.GetW() << endl;
    cout << "Height is " << a.GetH() << endl;
    cout << "Area is " << a.Area() << endl;
    return 0;
}
```

```
Top is 1
Left is 2
Width is 5
Height is 8
Area is 40
```



(2) 从类模板派生非模板类（普通类）

- 从一个已有的类模板派生出非模板类，格式如下：
- `template <class T>`
- `class Base`
- `{`
- `...`
- `};`
- `class Derived:public Base<int>`
- `{`
- `...`
- `};`
- 例6：类从类模板Point派生非模板类Rectangle，利用类Rectangle求矩形的位置、长度、宽度和面积。





- `#include <iostream>`
- `using namespace std;`
- `template <class T>`
- `class Point`
- `{`
- `public:`
- `Point(T i, T j) :x(i), y(j) {}`
- `T GetX() { return x; }`
- `T GetY() { return y; }`
- `private:`
- `T x, y;`
- `};`

- `class Rectangle :public Point<int>`
- `{`
- `public:`
- `Rectangle(int x, int y, int w, int h) :Point(x, y)`
- `{`
- `width = w;`
- `height = h;`
- `}`
- `int GetW() { return width; }`
- `int GetH() { return height; }`
- `int Area() { return width*height; }`
- `private:`
- `int width, height;`
- `};`





```
int main()
{
    Rectangle a(1, 2, 5, 8);
    cout << "Top is " << a.GetX() << endl;
    cout << "Left is " << a.GetY() << endl;
    cout << "Width is " << a.GetW() << endl;
    cout << "Height is " << a.GetH() << endl;
    cout << "Area is " << a.Area() << endl;
    return 0;
}
```

```
Top is 1
Left is 2
Width is 5
Height is 8
Area is 40
```





4 几个例子

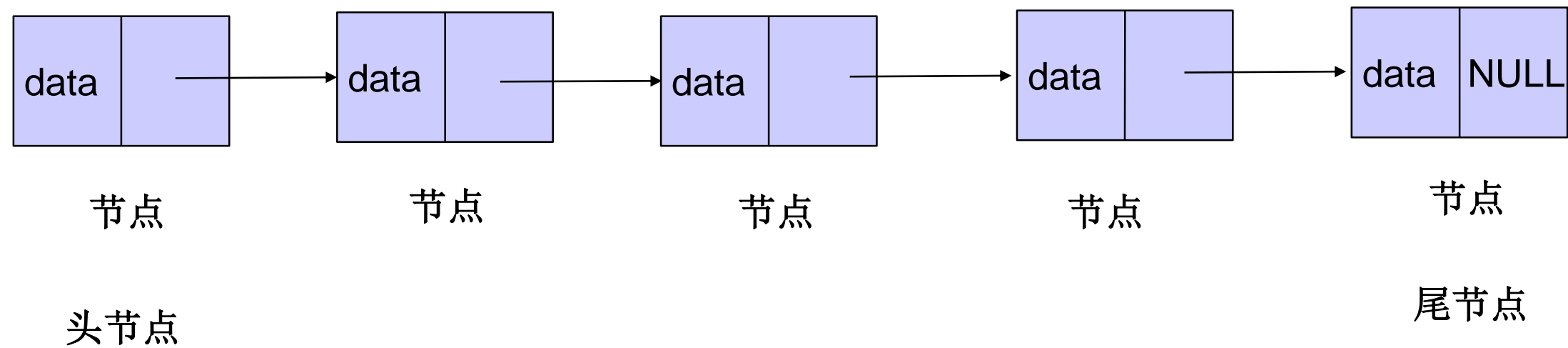
- 4.1 单向链表（指针）
- 4.2 综合实例1-单向链表类模板
- 4.3 综合实例2-函数模板
- 4.4 类模板Queue（做实验，选做）





4.1 单向链表（指针）

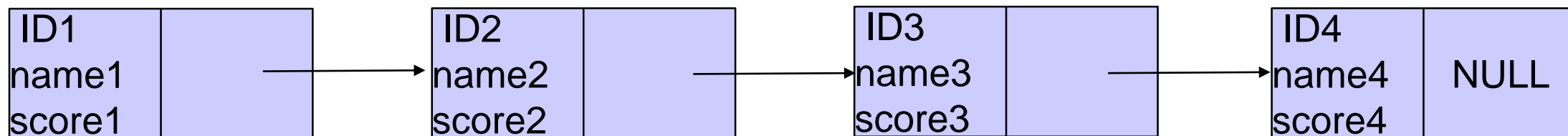
- 单向链表中，每个节点都有一个指针指向下一个节点，最后一个节点的指针值是NULL。
- 在程序中，用一个与链表节点的数据类型一致的指针指向链表的第一个节点，如果链表长度为0，这个指针指向NULL。



单向链表示例

- 输入一组学生的学号、姓名和课程成绩。当输入的成绩为负数时，输入结束。程序先将输入的数据存储在一个单向链表中，每个节点存储一个学生的数据，然后计算并输出全部学生的平均成绩。
- 每次访问链表进行节点的插入、删除、查找时，都需要使用这个指针

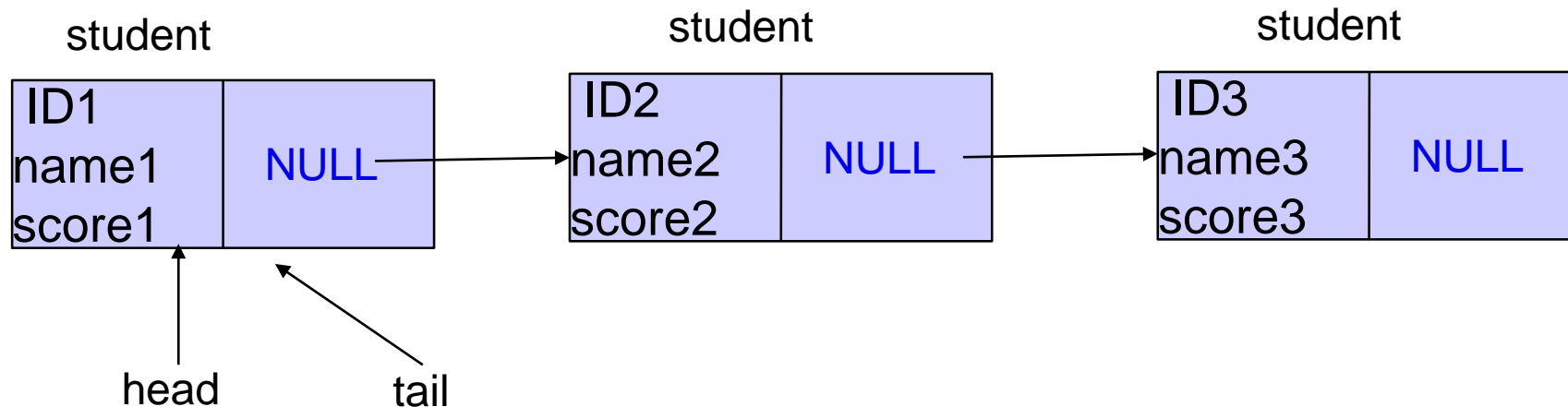
```
struct Student
{
    string ID;
    string name;
    double score;
    Student * next;
};
```



单向链表示例

建立链表

head=NULL tail=NULL

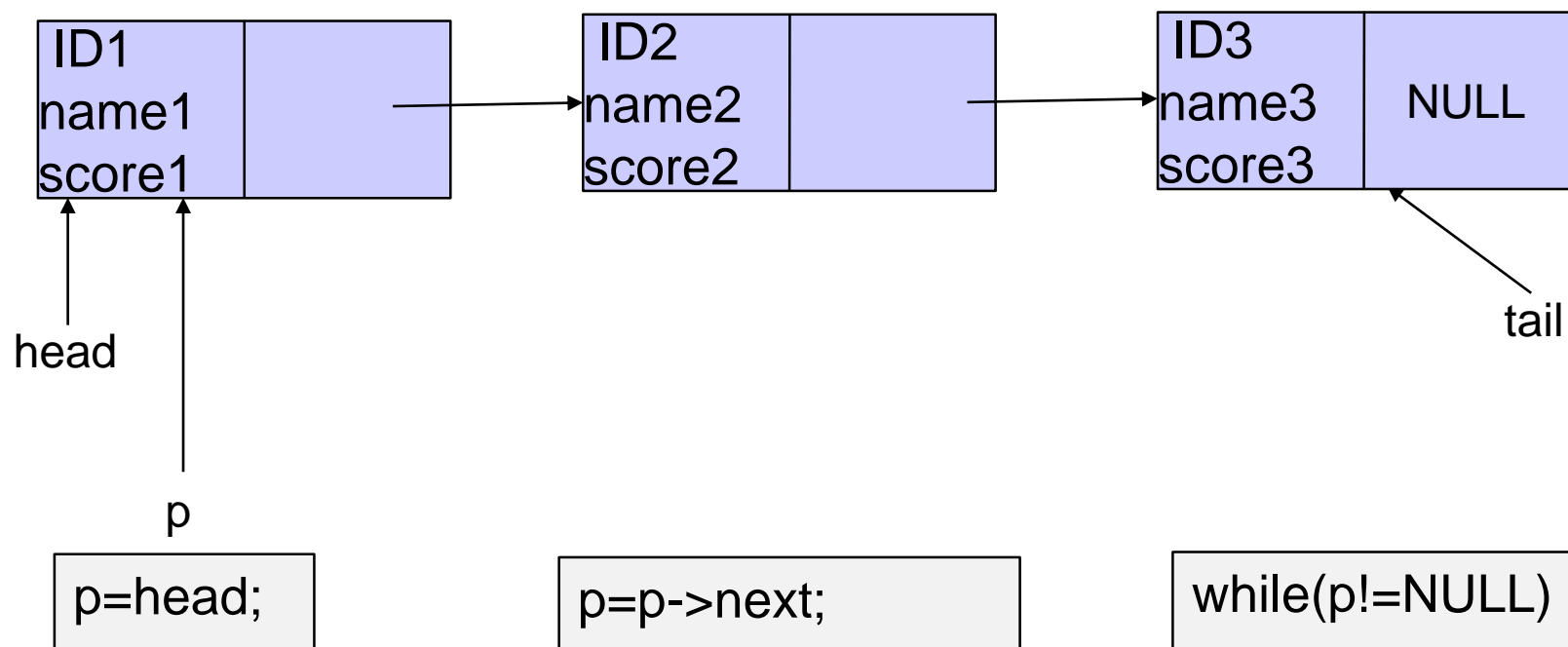


```
new student;  
输入student的内容;  
head=tail=student;
```

```
new student;  
输入student的内容;  
tail->next=student;  
tail=tail->next;
```

单向链表示例

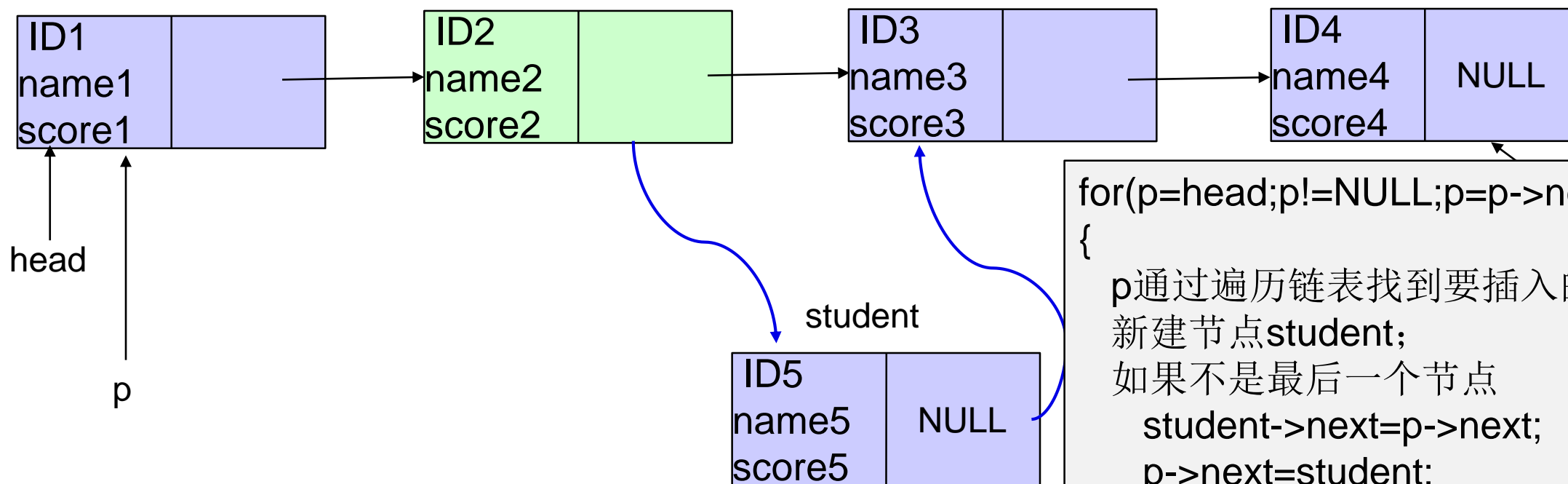
● 遍历链表



```
p=head;
while(p!=NULL)
{
    //处理语句，例如输出每个
    //节点的信息
    p=p->next;
}
```

单向链表示例

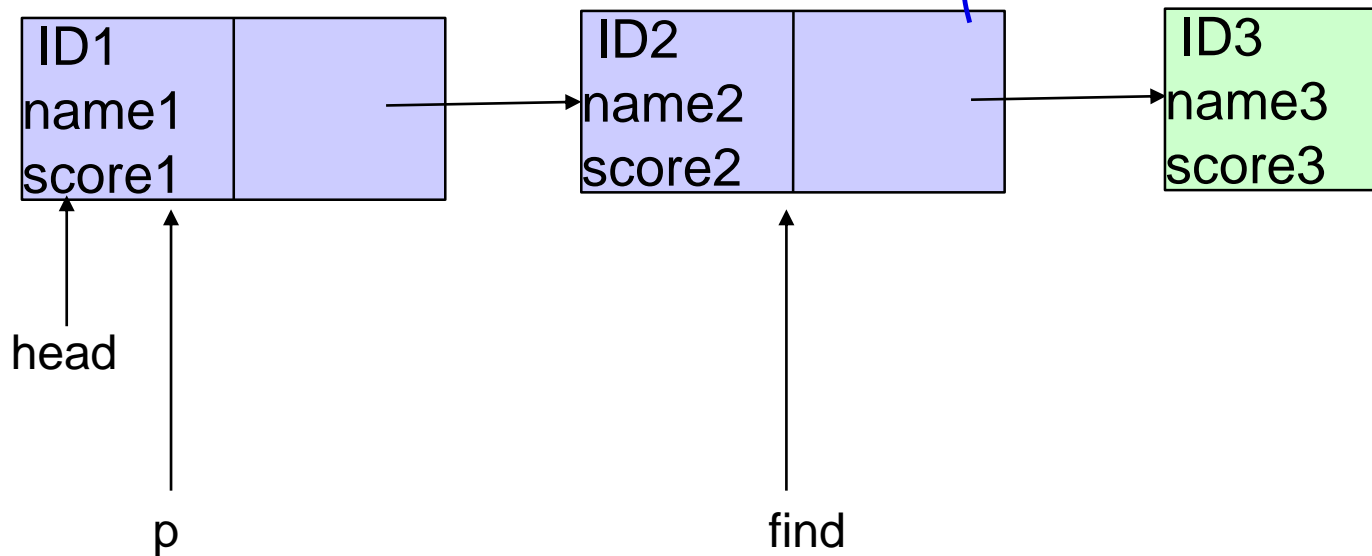
● 插入(在找到的节点后插入新节点)



```
for(p=head;p!=NULL;p=p->next)
{
    p通过遍历链表找到要插入的节点位置;
    新建节点student;
    如果不是最后一个节点
        student->next=p->next;
        p->next=student;
    如果是最后一个节点, 则修改tail
        student->next = NULL;
        p->next = student;
        tail=student;
}
```

单向链表示例

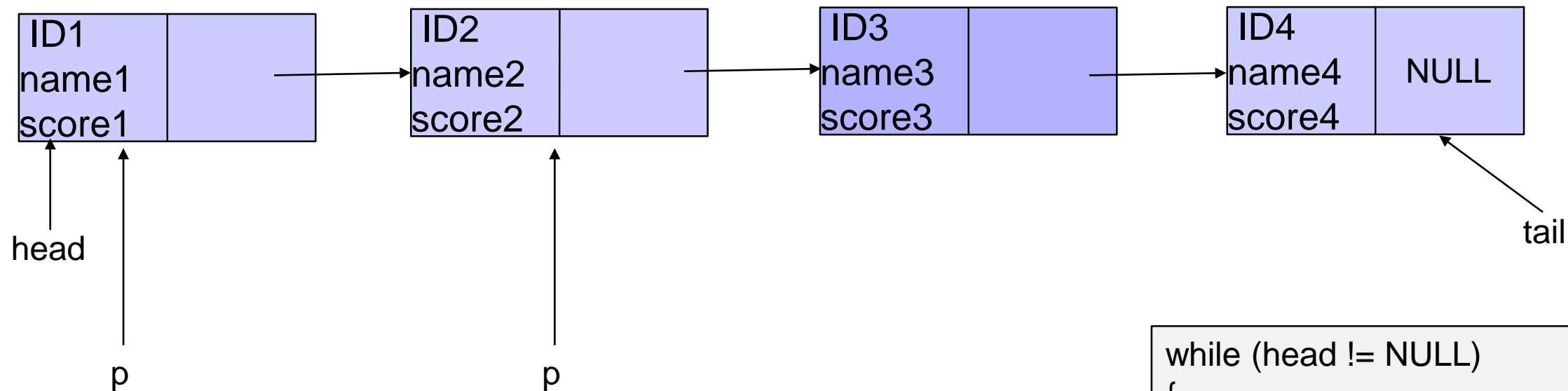
● 删除 (一个指定节点)



```
p=head;
for (find= head->next;find !=
NULL;p=find,find=find->next)
{
    如果find通过遍历链表找到要删除的节点
    如果不是最后一个节点
        p->next=find->next;
        delete find;
        break;
    如果是最后一个节点，则修改tail
        p->next = NULL;
        tail=p;
        delete find;
        break;
}
```

单向链表示例

● 删除 (整个链表)



```
while (head != NULL)
{
    p = head;
    head = head->next;
    delete p;
}
```



4.2 综合实例1

- 设计一个单向链表的类模板，使其能够完成单向链表的创建和显示等操作。





4.3 综合实例2

- 用函数模板的方式实现对不同数据类型的数组中的数据进行输入、从小到大排序和输出。然后用int整型数组和char型数组进行测试。
- 分析：
 - 根据本题的题意，需要设计3个函数模板input、sort、output分别完成对不同数据类型的数组的输入、排序、输出，将这3个函数模板中的形参的数组类型设置为T型，来代表某种数据类型；
 - 然后在主函数中分别用int型和char型将其实例化，最终完成3个函数模板的测试。





4.4 类模板Queue——实验选做

```
template <class Type> class Queue {  
public:  
    Queue();  
    virtual ~Queue();  
    Type & front();  
    const Type & front() const;  
    void push(const Type &);  
    void pop();  
    bool empty() const;  
private:  
    // ...  
};  
实例化Queue<int> qi; Queue<string> qs;
```



Queue的实例化



- 当编译`Queue<int> qi;`时, 编译器自动创建名为`Queue<int>`类, 生成:

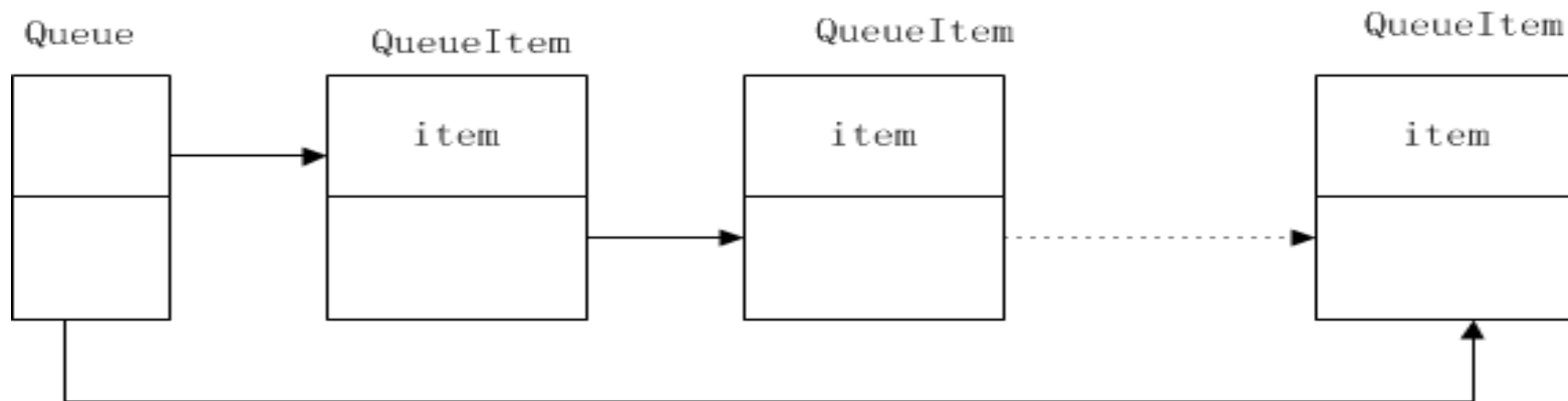
```
template <class Type> class Queue<int>
{
public:
    Queue();
    virtual ~Queue();
    int & front();           //Type & front();
    const int & front() const; //const Type & front() const;
    void push(const int &); //void push(const int &);
    void pop();
    bool empty() const;

private:
    // ...
};
```



Queue的实现

- 使用两个类: `QueueItem`和`Queue`
- `QueueItem`表示链表中的节点, 数据成员包括`item`和`next`。Queue中的每个元素保存在一个`QueueItem`对象中。
- `Queue`类也有两个数据成员:`head`和`tail`, 它们是`QueueItem`指针。





小结

- (1) 当在程序设计中遇到若干个程序结构有同一种模式时可以使用模板。
- (2) 模板是一种高度抽象的结构形式。
- (3) 类模板是对类的抽象，代表一类类，这些类具有相同的功能，但数据成员类型及成员函数返回类型和形参类型不同。
- (4) 函数模板是一类函数的抽象，代表了一类函数，这一类函数具有相同的功能。

