

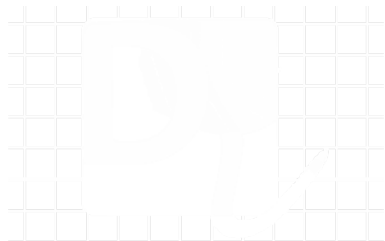
# 高级语言C++程序设计

## Lecture 11 类模板

南开大学 计算机学院

2022

# 函数模板



# 函数重载

---

**函数重载可以支持多种数据类型，但是冗余度高**

```
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;  
}
```

---

# 函数模板

---

**函数模板**：“提取”出一个可变化的类型参数，定义一组函数

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

# 函数模板

---

## 函数模板的完整定义

template 关键字

模板参数，用尖括号括起来，可以是一个或多个，用逗号分隔

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

模板参数类型

# 函数模板实例化

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

```
void main() {  
    int i1=-11, i2=0;  
    max(i1,i2);  
}
```

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

编译器编译到max(i1,i2)时，会根据模板实参生成一个具体函数，叫做函数模板实例化！

实例化在编译阶段完成，在函数调用处发生！

# 函数模板

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

```
void main() {  
    int i1=-11, i2=0;  
    double d1, d2;  
    cout<<max(i1,i2)<<endl; //OK,形参T对应于int  
    cout<<max(23,-56)<<endl;  
    cout<<max('f', 'k')<<endl; //OK,T对应char  
    cin>>d1>>d2;  
    cout<<max(d1,d2)<<endl; //OK,T对应double  
    cout<<max(23,-5.6)<<endl; // 错误!不进行实参到形  
                                参类型的自动转换  
}
```

# 函数模板

---

```
template <typename T, typename U>
    bool if_max (T a, U b){
        if(a > b) return true;
        else return false;
    }
```

```
void main() {
    int i1=-11;
    double d1=0.12;
    cout<<max(i1, d1)<<endl;
    //OK,形参T对应于int, U对应于double
}
```



# 函数模板与函数重载

调用顺序：首先检查是否存在重载函数，若匹配成功则调用该函数，否则再去匹配函数模板

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

```
void main() {  
    cout<<min(3,-10)<<endl; //使函数模板  
    cout<<min(2.5,99.5)<<endl; //使用函数模板  
    cout<<min('m','c')<<endl; //使用函数重载  
}
```

# 函数模板重载

定义两个函数模板，都叫做sum，都使用了一个类型参数T，但两者的形参个数不同

```
template <typename T> T sum(T a[], int size ){
    T total=0;
    for (int i=0;i<size;i++)
        total+=a[i];
    return total;
}
```

```
template <typename T>T sum(T a1[], T a2[], int
size ){
    T total=0;
    for (int i=0;i<size;i++)
        total+=(a1[i]+a2[i]);
    return total;
}
```

# 显式指定模板参数类型

```
template <typename T>
    T max1 (T a, T b){
        return a > b ? a : b;
    }
```

```
void main() {
    max1(3, 5); //模板类型参数自动匹配到int

    max1(2.4, 3.5); //模板类型参数自动匹配到double

    max1<double>(3, 5); //显式指定模板类型参数为
                        double

    max1<int>(2.4, 3.5); //显式指定模板类型参数为int
}
```

# 函数模板返回类型

```
template <class T1, class T2, class T3>
    T3 max1 (T1 a, T2 b){
    return a > b ? a : b;
}
```

```
void main() {
    max1(3, 5); //错误！ 无法推断返回类型参数T3

    max1(2.4, 3); //错误！ 无法推断返回类型参数T3

    max1<int, int, int>(3, 5); //显式指定T1, T2, T3

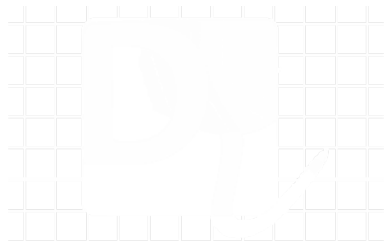
    max1<int>(2.4, 3.5); //错误！ 只指定了T1, T3仍然无法确定
}
```

# 函数模板返回类型

```
template <class T1, class T2, class T3>
    T3 max1 (T1 a, T2 b){
    return a > b ? a : b;
}
```

```
void main() {
    max1<int, double>(3, 5); //错误！ 只指定了T1和T2
                             , T3仍然无法无法推断
    max1<int, ,int>(2.4, 3); //错误！ 不能跳跃指定
    max1<int, double, char>(3, 5); //OK!
}
```

# 类模板



# 类模板

---

普通类定义只能支持一种数据类型

```
class TestClass {  
public:  
    int buffer[10]; // 仅支持int型数组  
    int getData(int j);  
};  
  
int TestClass::getData(int j) {  
    return *(buffer+j);  
};
```

# 类模板

类模板可以用来描述一个与数据类型无关的类

```
template <typename T>
class TestClass {
public:
    T buffer[10];

    T getData(int j);
};
```

T 代表任意类型

```
template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```



# 类模板定义

## 类模板定义格式

```
template <typename T>
```

```
class TestClass {
```

```
public:
```

```
    T buffer[10];
```

```
    T getData(int j);
```

```
};
```

```
template <typename T>
```

```
T TestClass<T>::getData(int j) {
```

```
    return *(buffer+j);
```

```
};
```

template 关键字，  
指明是函数或类模板

# 类模板定义

## 类模板定义格式

```
template <typename T>
class TestClass {
public:
    T buffer[10];

    T getData(int j);
};
```

模板形参表：用来说明一个或多个类型形参和普通形参；多个模板参数用逗号分隔

```
template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

# 类模板定义

## 类模板定义格式

```
template <typename T>
```

```
class TestClass {
```

```
public:
```

```
    T buffer[10];
```

```
    T getData(int j);
```

```
};
```

class关键字

```
template <typename T>
```

```
T TestClass<T>::getData(int j) {
```

```
    return *(buffer+j);
```

```
};
```


# 类模板定义

## 类模板定义格式

```
template <typename T>
class TestClass {
public:
    T buffer[10];

    T getData(int j);
};
```

类模板名，与类的命名规则相同



```
template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

# 类模板定义

## 类模板定义格式

```
template <typename T>
class TestClass {
public:
    T buffer[10];

    T getData(int j);
};
```

类的定义



```
template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

# 类模板实例化

```
template <typename T> class TestClass;
```

```
void f(){
```

```
    TestClass<char> ClassInstA;
```

```
}
```

```
class TestClass {  
public:  
    char buffer[10];  
  
    char getData(int j);  
};
```

编译器编译  
TestClass<char>时  
，根据类模板和模  
板实参生成一个类  
定义的过程，叫**类  
的实例化**

# 类模板实例化

---

```
void main() {  
    TestClass<char> ClassInstA;  
    //char取代 T，类模板实例化，编译器生成一个新类(数  
    组为char型)，ClassInstA是这个类的一个对象  
  
    TestClass<double> ClassInstB;  
    //double取代 T，类模板实例化，编译器生成一个新类(  
    数组为double型)，ClassInstB是这个类的一个对象  
  
    TestClass<int> ClassInstC;  
    //int取代 T，类模板实例化，编译器生成一个新类(数  
    组为int型)，ClassInstC是这个类的一个对象  
}
```

# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

函数声明，不需要类定义，不需要实例化



# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

定义引用，不需要类  
定义，不需要实例化

# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

定义类对象，需要类定义，需要实例化

# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

定义指针，不需要类  
定义，不需要实例化

# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

访问指针指向的类对象，需要类定义，需要实例化

# 类模板实例化

类模板实例化只在需要类定义的时候才会发生！

```
template<typename Type> class Graphics{};
void f1(Graphics<char>);
class Rect {
    Graphics<double>& rsd;
    Graphics<int> si;
}
int main(){
    Graphics<char>* sc;
    f1(*sc);
    int iobj=sizeof(Graphics<string>);
}
```

sizeof需要有类定义  
才能知道对象大小，  
需要类定义，需要实例化

# 类模板的不同形式

---

## 仅使用类型参数的类模板

```
template <typename T>
class TestClass {
public:
    T buffer[10];

    T getData(int j);
};

template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

# 类模板的不同形式

## 仅使用类型参数的类模板

```
TestClass<double> ClassInstF;
```

//double取代 T，类模板实例化为一个具体的类(数组为double型)，ClassInstF是这个类的一个对象

```
double fArr[6]={12.1, 23.2, 34.3, 45.4,
56.5, 67.6};
for(i=0; i<6; i++)
    ClassInstF.buffer[i]=fArr[i]-10;
for(i=0; i<6; i++) {
    double res=ClassInstF.getData(i);
    cout<<res<<" ";
}
}
```

# 类模板的不同形式

---

类模板也可以支持普通参数(非类型参数)

```
template <typename T, int i>
class TestClass {
public:
    T buffer[i]; //数组的大小为i, 相当于动态数组

    T getData(int j);
};

template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```



# 类模板的不同形式

类模板也可以支持普通参数(非类型参数)

```
template <typename T, int i>
class TestClass {
public:
    T buffer[i]; //数组的大小为i, 相当于动态数组

    T getData(int j);
};

template <typename T>
T TestClass<T>::getData(int j) {
    return *(buffer+j);
};
```

i是普通参数

# 类模板的不同形式

## 类模板的非类型参数

```
void main() {  
    TestClass<char, 5> ClassInstA;  
    //类模板实例化为一个具体的类，数组为char型，大小为5  
  
    char cArr[6]="abcde";  
    for(int i=0; i<5; i++)  
        ClassInstA.buffer[i]=cArr[i];  
  
    for(i=0; i<5; i++) {  
        char res=ClassInstA.getData(i);  
        cout<<res<<" ";  
    }  
}
```

# 类模板的不同形式

## 类模板的非类型参数

```
TestClass<double, 6> ClassInstF;
```

//类模板实例化为一个具体的类，数组为double型，大小为6

```
double fArr[6]={12.1, 23.2, 34.3, 45.4,
56.5, 67.6};
for(i=0; i<6; i++)
    ClassInstF.buffer[i]=fArr[i]-10;
for(i=0; i<6; i++) {
    double res=ClassInstF.getData(i);
    cout<<res<<" ";
}
}
```

# 类模板的静态成员

类模板的静态成员是模板实例化类的静态成员，对于每一个实例化类，其所有的对象共享其静态成员

```
template<typename T>class TA {  
    static int m_t1; //静态成员，类型为int  
    static T m_t2;   //静态成员，类型为 T  
};  
  
//静态成员初始化，类外进行  
template<typename T> int TA<T>::m_t1 = 100;  
template<typename T> T TA<T>::m_t2 = 0;
```

# 类模板的静态成员

---

```
TA <int> iobj1, iobj2;
```

// 实例化模板类TA<int>的两个对象iobj1和iobj2将共享TA<int>的静态成员int m\_t1、int m\_t2

```
TA <double> dobj1, dobj2;
```

// 实例化模板类TA<double>的两个对象dobj1和dobj2将共享TA<double>的静态成员int m\_t1、double m\_t2

# 类模板的静态成员

```
int main() {  
    TA <int> iobj1, iobj2;  
    TA <double> dobj1, dobj2;
```

```
    iobj1.m_t1 ++; iobj1.m_t2 ++;  
    iobj2.m_t1 ++; iobj2.m_t2 ++;
```

```
    dobj1.m_t1 ++; dobj1.m_t2 ++;  
    dobj2.m_t1 ++; dobj2.m_t2 ++;
```

```
    cout<<TA<int>::m_t1<<" "<<TA<int>::m_t2<<endl;  
    cout<<dobj1.m_t1<<"<<dobj2.m_t2<<endl;  
    return 0;  
}
```

类TA<int>和类TA<double>可以认为是两个不同的类，他们的类对象不共享静态变量m\_t1！

结果：

102 2

102 2

# 类模板的成员函数

## 类模板的所有成员函数都是函数模板

```
template<typename T> class MyStack {  
    public:  
        void create();  
        void push(const T item);  
};  
template<typename T> void MyStack<T>::create(){};  
template<typename T> void MyStack<T>::push( const  
T item ){};
```

//虽然没有用到模板类型 T, 但  
create()仍然是函数模板

```
MyStack<int> s1;  
s1.create();
```

//类模板实例化时成员函数并不自动被  
实例化, 只有函数调用时才被实例化

# 类模板的成员函数

## 类模板的所有成员函数都是函数模板

```
template<typename T> class MyStack {  
    public:  
        void create();  
        void push(const T item);  
};  
template<typename T> void MyStack<T>::create(){};  
template<typename T> void MyStack<T>::push( const  
T item ){};
```

//push函数的参数是  
模板类型 T，所以也是  
函数模板

```
MyStack<int> s1;  
s1.push(100);
```

//类模板实例化为int类型，函数push的  
参数也自动实例化为int类型



# 类模板的成员函数

类模板的成员函数也可以是另外一个函数模板

```
template<typename T> class MyStack {  
    public:  
        template<typename U> void create(U a);  
};  
    //create是函数模板，参数类型为 U，MyStack是类  
    模板，参数类型为 T  
template<typename T> template<typename U> void  
MyStack<T>::create(U a) {}
```

```
MyStack<int> s1; //类模板实例化为int类型，函数create  
s1.create(2.5); 实例化为double 类型
```

# 类模板的成员函数

普通类的成员函数也可以是函数模板

```
class MyStack {  
    public:  
        template<typename U> void create(U a);  
};  
    //MyStack是普通类，但成员函数create是函数模板  
template<typename U> void MyStack::create(U a)  
{}
```

```
MyStack s1;  
s1.create(2.5); //create实例化为double 类型
```

# 类模板与友元函数

如果类模板的友元函数是普通函数，则友元函数是该类模板任意类实例的友元

```
void create(){} //create是普通函数，是类模板MyStack  
                的友元函数  
template<typename T> class MyStack {  
    public:  
        friend void create();  
};
```

```
MyStack<int> s1; //create 是 类 MyStack<int> 和  
MyStack<double> s2; MyStack<double>的友元函数
```

# 类模板与友元函数

如果友元函数是与类模板无关的模板函数，则友元函数的任意函数实例是任意类实例的友元

```
template<typename U> void create(U x){};

template<typename T> class MyStack {
    public:
        friend template<typename U> void create(U x);
}; //create是函数模板(参数为U), MyStack是类模板(参数为T)
```

例如：函数模板的实例 `void create(int)` 和 `void create(char)` 都是类模板实例 `MyStack<int>`, `MyStack<double>` 的友元函数

# 类模板与友元函数

友元函数是与类模板有关的模板函数，则友元函数只是该类模板特定类实例的友元

```
template<typename U> void create(U x){};

template<typename T> class MyStack {
    public:
        friend void create<T>(T x);
};
```

//定义友元的时候将函数模板的参数写成与类模板参数一样!

例如：void create(int)只是 MyStack<int>的友元，不是其他类模板实例的友元；void create(char)只是 MyStack<char>的友元函数，不是其他类模板实例的友元

# 类型参数检测与特例版本

类模板的类型参数往往在实例化时不允许用任意的类（类型）作为“实参”

```
template <typename T>
class stack {
    T data [20];
    int top {0};
public:
    void showtop(void);
};

template <typename T> void
stack<T>::showtop(void) {
    cout<<"Top_Member:"<<data[top]<<endl;
} //输出数组下标为top的元素
```

# 类型参数检测与特例版本

类模板的类型参数往往在实例化时不允许用任意的类（类型）作为“实参”

```
class complex { //自定义类，表示复数
public:
    double real, image;
};
```

```
void main() {
    stack<int>i1; //OK，实例化为int型
    stack<char>c1; //OK，实例化为char型
    stack<float>f1; //OK，实例化为double型
    stack<comple>cp1;
    //ERROR! 实例化为complex型，但showtop函数里的输出
    操作符<<并不支持complex类型
}
```

# 类型参数检测与特例版本

解决方案一：对complex类进行<<运算符重载

解决方案二：在stack类模板里填加showtop函数的特例版本专门支持complex类

此函数的特例版本专门针对complex类



```
void stack<complex>::showtop() {  
    cout<<"Top_Member:"<<data[top].real<<"  
"<<data[top].image<<endl;  
    //分别输出实部real和虚部image  
}
```



# 类型参数检测与特例版本

---

```
void main() {  
    stack<int> s1;  
    s1.showtop(); //调用模板类中通用的showtop  
    stack<char> s2;  
    s2.showtop(); //调用模板类中通用的showtop  
    stack<complex> s1c;  
    s1c.showtop(); //调用showtop特例版本  
}
```

# 类模板的继承与派生

一般类（其中不使用类型参数的类）作基类，派生出类模板（其中要使用类型参数）

```
class CB { //基类CB 为一般类（其中不使用类型参数）
    ...
};

//派生类CA为类模板，使用了类型参数T
template <typename T> class CA:public CB {
    T t; //派生类新加的成员
public:
    ...
};
```

# 类模板的继承与派生

类模板作基类，派生出新的类模板，但仅基类中用到类型参数T，而派生的类模板中不使用T

```
template <typename T> class CB { //CB是类模板
    T t;
public:
    T gett(){return t;}
};
template <typename T> class CA : public CB<T>{
    double t1; //派生类的新成员并不使用 T
};
```

基类的名字应为实例化后的“CB<T>”而非仅使用“CB”，在派生类说明中，要对基类进行指定时必须使用“CB<T>”而不可只使用“CB”

# 类模板的继承与派生

类模板作基类，派生出新的类模板，且基类与派生类中均使用同一个类型参数T

```
template <typename T> class CB { //基类CB是类模板
    T t;
public:
    T gett(){
        return t;
    }
};

template <typename T> class CA : public CB<T>{
    T t1; //派生类新填加的成员也使用了基类的模板类型T
};
```

# 类模板的继承与派生

类模板作基类，派生出新的类模板，但基类和派生类使用的类型参数不同

```
template <typename T2> class CB { //基类CB是类模板
    T2 t2; //数据类型为T2
public:
    ...
};

//派生类也是类模板，继承基类的部分使用基类的模板类型T2
//新成员使用类型T1
template <typename T1, typename T2> class CA :
    public CB<T2> {
    T1 t1; //数据类型为T1
};
```

**END**

