第九章 模板

现在的C++编译器实现了一项新的特性：模板（ Template ）

模板是一种通用的描述机制，使用模板允许使用通用类型来定义函数或类。在使用时，通用类型可被具体的类型，如 int、double 甚至是用户自定义的类型来代替。模板引入一种全新的编程思维方式，称为“**泛型编程**”或“**通用编程**”。

为什么要定义模板

像C/C++/Java等语言，是**编译型语言**，先编译后运行。它们都有一个强大的类型系统，也被称为**强类型语言**，希望在程序执行之前，尽可能地发现错误，防止错误被延迟到运行时。所以会对语言本身的使用造成一些限制，称之为**静态语言**。与之对应的，还有**动态语言**，也就是**解释型语言**。如javascript/python/Go，在使用的过程中，一个变量可以表达多种类型，也称为**弱类型语言**。因为没有编译的过程，所以相对更难以调试。

强类型程序设计中，参与运算的所有对象的类型在编译时即确定下来，并且编译程序将进行严格的类型检查。为了解决强类型的严格性和灵活性的冲突，也就是在严格的语法要求下尽可能提高灵活性，有以下3种方式解决：

- 带参数宏定义（原样替换）
- 重载函数（函数名相同，函数参数不同）
- **模板（将数据类型作为参数）**

```
int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

long add(long x, long y)
{
    return x + y;
}
```

```
15
16  string add(string x, string y)
17  {
18      return x + y;
19  }
20
21  //希望将类型参数化
22  //使用class关键字或typename关键字都可以
23  template <class/typename T>
24  T add(T x, T y)
25  {
26      return x + y;
27  }
```

模板作为实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现了真正的代码可重用性。模板可以分为两类，一个是**函数模版**，另外一个是**类模板**。通过参数实例化构造出具体的函数或类，称为**模板函数**或**模板类**。

```
#if 0
//编译器在走到模板被使用的语句时
//会生成一个模板函数（实例化）
short add(short x,short y){
    return x + y;
}
#endif
```

模板的定义

模板发生的时机是在编译时

模板本质上就是一个代码生成器，它的作用就是让编译器根据实际调用来生成代码。

编译器去处理时，实际上由函数模板生成了多个模板函数，或者由类模板生成了多个模板类。

形式： template <typename/class T1, typename T2,... > 模板形参列表

称为类型参数或者模板参数

函数模板

由函数模板到模板函数的过程称之为**实例化**

函数模板 --》 生成相应的模板函数 --》编译 ---》链接 --》可执行文件

下图中实际上可以理解为生成了四个模板函数

```
template <class T>
T add(T t1,T t2)
{ return t1 + t2; }

void test0(){
    short s1 = 1, s2 = 2;
    int i1 = 3, i2 = 4;
    long l1 = 5, l2 = 6;
    double d1 = 1.1, d2 = 2.2;

    cout << "add(s1,s2): " << add(s1,s2) << endl;
    cout << "add(i1,i2): " << add(i1,i2) << endl;
    cout << "add(l1,l2): " << add(l1,l2) << endl;
    cout << "add(d1,d2): " << add(d1,d2) << endl;
}
```

上述代码中在进行模板实例化时，并没有指明任何类型，函数模板在生成模板函数时通过传入的参数类型确定出模板类型，这种做法称为**隐式实例化**。

我们在使用函数模板时还可以在函数名之后直接写上模板的类型参数列表，指定类型，这种用法称为**显式实例化**。

```
template <class T>
T add(T t1,T t2)
{ return t1 + t2; }

void test0(){
    int i1 = 3, i2 = 4;
    cout << "add(i1,i2): " << add<int>(i1,i2) << endl;
}
```

函数模板的重载

1. **函数模板可以与函数模板进行重载**
   如果在使用函数模板时传入两个不同类型的参数,会出错，此时就需要进行显式实例化。
   如下，指定了类型T为int型，虽然s1是short型数据，但是会发生类型转换。这个转换没有问题，因为int肯定能存放short型数据的所有内容。

```
template <class T>
T add(T t1,T t2)
{ return t1 + t2; }

void test0(){
    short s1 = 1;
    int i2 = 4;

    cout << "add(s1,s2): " << add(s1,i2) << endl;//error
    cout << "add(s1,s2): " << add<int>(s1,i2) << endl;//ok
}
```

但如果是以下这种转换，实际上就会损失数据精度。此时的d2会转换成int型。

```
int i1 = 4;
double d2 = 5.3;
cout << "add(i1,d2): " << add<int>(i1,d2) << endl;
```

如果一个函数模板无法实例化出合适的模板函数（去进行显式实例化也有一些问题）的时候，可以再给出另一个函数模板

```
    //函数模板与函数模板重载
    //模板参数个数不同,ok
    template <class T> //模板一
    T add(T t1,T t2)
    { return t1 + t2; }


    template <class T1, class T2>  //模板二
    T1 add(T1 t1, T2 t2)
    {
        return t1 + t2;
    }


    double x = 9.1;
    int y = 10;
```

```
    cout << add(x,y) << endl;    //会调用模板二生成的模板函数，不会损失精度

    //试一试
    cout << add(y,x) << endl; //返回值是一个int数据
```

如果仍然采用显式实例化

可以传入两个类型参数，那么一定会调用模板二生成的模板函数。传入的两个类型参数会作为T1、T2的实例化参数。

也可以传入一个类型参数，那么这个参数会作为模板参数列表中的第一个类型参数进行实例化。

如果仍然需要进行类型转换，那么就会使用第一个函数模板进行实例化，如果不需要进行类型转换，就会使用第二个函数模板进行实例化。

```
int x = 10;
double y = 9.2;
cout << add<int,int>(x,y) << endl; //模板二
cout << add<int>(x,y) << endl; //模板二
cout << add<int>(y,x) << endl; //模板一
```

```
int x = 10;
double y = 9.2;
//有了模板二之后，显式实例化时可以指定两个类型参数
cout << add<int,int>(x,y) << endl; //模板二
cout << add<int,double>(x,y) << endl; //模板二

//指定了T1类型为int，没有指定T2类型
cout << add<int>(x,y) << endl;
//能够隐式实例化尽量使用隐式实例化
cout << add(x,y) << endl;
cout << add(y,x) << endl;
```

函数模板与函数模板重载的条件：

（1）名称相同（这是必须的）

（2）模板参数列表中的模板参数在函数中所处位置不同 —— **但是强烈不建议进行这样的重载**。

这样进行重载时，要注意，隐式实例化可能造成冲突，需要显式实例化。（如果能够通过类型转换去匹配上两个函数模板的时候，即使是显式实例化也很难避免冲突）

```cpp
template <class T1, class T2>
T1 add(T1 t1, T2 t2)
{
    cout << "T1,T2, return T1" << endl;
    return t1 + t2;
}

template <class T1, class T2>
T2 add(T1 t1, T2 t2)
{
    cout << "T1,T2, return T2" << endl;
    return t1 + t2;
}
```

（3）模板参数的个数不一样时，可以构成重载（相对常见）

```cpp
template <class T2, class T1>
T1 add(T2 t2, T1 t1)
{
    return t1 + t2;
}

template <class T1, class T2, class T3>
T1 add(T1 t1, T2 t2, T3 t3)
{
    return t1 + t2 + t3;
}
```

```cpp
void test0(){
    double x = 1.2;
    double y = 2.9;
    int z = 11;
    cout << add(x,y) << endl;
    cout << add(x,z) << endl;
    cout << add(z,x) << endl;
    cout << add(x,y,z) << endl;
    cout << add(z,x,y) << endl;
}
```

```
T class
4.1
T1,T2
12.2
T1,T2
12
T1,T2,T3
15.1
T1,T2,T3
15
```

2. 函数模板与普通函数重载

普通函数优先于函数模板执行——因为普通函数更快

（编译器扫描到函数模板的实现时并没有生成函数，只有扫描到下面调用add函数的语句时，给add传参，知道了参数的类型，这才生成一个相应类型的模板函数——模板参数推导。所以使用函数模板一定会增加编译的时间。此处，就直接调用了普通函数，而不去采用函数模板）

```cpp
//函数模板与普通函数重载
template <class T1, class T2>
T1 add(T1 t1, T2 t2)
{
    return t1 + t2;
}

short add(short s1, short s2){
    cout << "add(short,short)" << endl;
    return s1 + s2;
}

void test1(){
    short s1 = 1, s2 = 2;
    cout << add(s1,s2) << endl;
}
```

```cpp
//函数模板与普通函数重载
template <class T1, class T2>
T1 add(T1 t1, T2 t2)
{
    cout << "T1,T2" << endl;
    return t1 + t2;
}

short add(short x,short y){
    cout << "short add(short,short)" << endl;
    return x + y;
}

void test0(){
    short s1 = 7, s2 = 8;
    //可以调用函数模板
    //T1和T2推导出的结果——参数类型可以相同
    //当普通函数和函数模板重载时
    //优先使用普通函数，因为效率更高（更直接）
    cout << add(s1,s2) << endl;
}
```

头文件与实现文件形式（重要）

为什么C++标准头文件没有所谓的.h后缀?

在一个源文件中，函数模板的声明与定义分离是可以的，即使把函数模板的实现放在调用之下也是ok的，与普通函数一致。

```cpp
//函数模板的声明
template <class T>
T add(T t1, T t2);


void test1(){
    int i1 = 1, i2 = 2;
    cout << add(i1,i2) << endl;
}

//函数模板的实现
template <class T>
T add(T t1, T t2)
{
    return t1 + t2;
}
```

如果在不同文件中进行分离

如果像普通函数一样去写出了头文件、实现文件、测试文件，编译报错

```cpp
//add.h
template <class T>
T add(T t1, T t2);

//add.cc
#include "add.h"
template <class T>
T add(T t1, T t2)
{
 return t1 + t2;
}

//testAdd.cc
#include "add.h"
void test0(){
 int i1 = 1, i2 = 2;
 cout << add(i1,i2) << endl;
}
```

```
ray@ubuntu:~/HaiBao/53th/day15/add$ g++ testAdd.cc add.cc
/tmp/cc7myPlz.o: In function `test0()':
testAdd.cc:(.text+0x21): undefined reference to `int add<int>(int, int
)'
collect2: error: ld returned 1 exit status
```

- 单独编译“实现文件”，使之生成目标文件，查看目标文件，会发现没有生成任何与add相关的内容。

```
ray@ubuntu:~/HaiBao/53th/day15/add$ g++ -c add.cc
ray@ubuntu:~/HaiBao/53th/day15/add$ ls
add.cc  add.h  add.o  testAdd.cc
ray@ubuntu:~/HaiBao/53th/day15/add$ nm add.o
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
0000000000000049 t _GLOBAL__sub_I_add.cc
0000000000000000 t _Z41__static_initialization_and_destruction_0ii
                 U _ZNSt8ios_base4InitC1Ev
                 U _ZNSt8ios_base4InitD1Ev
0000000000000000 r _ZStL19piecewise_construct
0000000000000000 b _ZStL8__ioinit
ray@ubuntu:~/HaiBao/53th/day15/add$
```

- 单独编译测试文件，发现有与add名称相关的函数，但是没有地址，这就表示只有声明。

```
ray@ubuntu:~/HaiBao/53th/day15/add$ nm testAdd.o
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
00000000000000a4 t _GLOBAL__sub_I__Z5test0v
000000000000004b T main
                 U _Z3addIiET_S0_S0_
000000000000005b t _Z41__static_initialization_and_destruction_0ii
0000000000000000 T _Z5test0v
                 U _ZNSolsEi
```

在”实现文件”中要进行调用，因为有了调用才有推导，才能由函数模板生成需要的函数

```
1  template <class T>
2  T add(T t1, T t2)
3  {
4   return t1 + t2;
5  }
6
7  //在这个文件中如果只是写出了函数模板的实现
8  //并没有调用的话，就不会实例化出模板函数
9  void test1(){
10  cout << add(1,2) << endl;
11 }
```

此时单独编译实现文件，发现生成了对应的函数

```
ray@ubuntu:~/HaiBao/53th/day15/add$ g++ -c add.cc
ray@ubuntu:~/HaiBao/53th/day15/add$ ls
add.cc  add.h  add.o  a.out  testAdd.cc  testAdd.o
ray@ubuntu:~/HaiBao/53th/day15/add$ nm add.o
                 U __cxa_atexit
                 U __dso_handle
                 U _GLOBAL_OFFSET_TABLE_
0000000000000082 t _GLOBAL__sub_I__Z5test1v
0000000000000000 W _Z3addIiET_S0_S0_
0000000000000039 t _Z41__static_initialization_and_destruction_0ii
0000000000000000 T _Z5test1v
                 U _ZNSt8ios_base4Init
```

但是在"实现文件"中对函数模板进行了调用，这种做法不优雅。

设想：如果在测试文件调用时，推导的过程中，**看到的是完整的模板的代码**，那么应该可以解决问题

```
1  //add.h
2  template <class T>
3  T add(T t1, T t2);
4
5  #include "add.cc"
```

在头文件中加上#include "add.cc"，即使实现文件中没有调用函数模板，单独编译testAdd.cc，也可以发现问题已经解决。

因为本质上相当于把函数模板的定义写到了头文件中。

总结：

对模板的使用，必须要拿到模板的全部实现，如果只有一部分，那么推导也只能推导出一部分，无法满足需求。

换句话说，就是模板的使用过程中，其实没有了头文件和实现文件的区别，在头文件中也需要获取模板的完整代码，不能只有一部分。

C++的标准库都是由模板开发的，所以经过标准委员的商讨，**将这些头文件取消了后缀名，与C的头文件形成了区分；这些实现文件的后缀名设为了tcc**

模板的特化

在函数模板的使用中，有时候会有一些通用模板处理不了的情况，我们可以定义普通函数或特化模板来解决。虽然普通函数的优先级更高，但有些场景下是必须使用特化模板的。它的形式是固定的：

1. template后直接跟 <> ，里面不写类型
2. 在函数名后跟 <> ，其中写要特化的类型

比如，add函数模板在处理字符串相加时遇到问题

```
//特化模板
//这里就是告诉编译器这里是一个模板
template <>
const char * add<const char *>(const char * p1,const char * p2){
    //先开空间
    char * ptmp = new char[strlen(p1) + strlen(p2) + 1]();
    strcpy(ptmp,p1);
    strcat(ptmp,p2);
    return ptmp;
}
```

注意：

使用模板特化时，必须要先有基础的函数模板

如果没有模板的通用形式，无法定义模板的特化形式。

特化模板是为了解决通用模板无法处理的特殊类型的操作。

```
//特化模板
template <>
const char * add<const char *>(const char * p1,const char * p2){
    char * ptemp = new char[strlen(p1) + strlen(p2) + 1]();
    strcpy(ptemp,p1);
    strcat(ptemp,p2);
    return ptemp;
}
```

特化版本的函数名、参数列表要和原基础的模板函数相同，避免不必要的错误。

使用模板的规则（重要）

1. **在一个模块中定义多个通用模板的写法应该谨慎使用；**
2. **调用函数模板时尽量使用隐式调用，让编译器推导出类型；**
3. **无法使用隐式调用的场景只指定必须要指定的类型；**
4. **需要使用特化模板的场景就根据特化模板将类型指定清楚。**

模板的参数类型

1. 类型参数

   之前的T/T1/T2等等成为模板参数，也称为类型参数，类型参数T可以写成任何类型

2. 非类型参数

   需要是整型数据， char/short/int/long/size_t等

   不能是浮点型，float/double不可以

定义模板时，在模板参数列表中除了类型参数还可以加入非类型参数。如下，调用模板时需要传入非类型参数的值

```
template <class T,int kBase>
T multiply(T x, T y){
    return x * y * kBase;
}

void test0(){
    int i1 = 3,i2 = 4;
    cout << multiply<int,10>(i1,i2) << endl;
}
```

可以给非类型参数赋默认值，有了默认值后调用模板时就可以不用传入这个非类型参数的值

```cpp
template <class T,int kBase = 10>
T multiply(T x, T y){
    return x * y * kBase;
}

void test0(){
    int i1 = 3,i2 = 4;
    cout << multiply<int,10>(i1,i2) << endl;
    cout << multiply<int>(i1,i2) << endl;
    cout << multiply(i1,i2) << endl;
}
```

```cpp
template <class T, int kBase = 10>
T multiply(T x, T y){
    return x * y * kBase;
}

void test0(){
    double x = 1.2, y = 1.2;
    //模板的非类型参数赋予默认值后
    //调用模板时可以隐式实例化
    cout << multiply(x,y) << endl;
    cout << multiply<double>(x,y) << endl;
    cout << multiply<double,100>(x,y) << endl;
}
```

函数模板的模板参数赋默认值与普通函数相似，从右到左，右边的非类型参数赋了默认值，左边的类型参数也可以赋默认值

```cpp
template <class T = int,int kBase = 10>
T multiply(T x, T y){
    return x * y * kBase;
}

void test0(){
    double d1 = 1.2, d2 = 1.2;
    cout << multiply(d1,d2) << endl;
    cout << multiply<int>(d1,d2) << endl;
}
```

```
template <class T = int, int kBase = 10>
T multiply(T x, T y){
    return x * y * kBase;
}

void test0(){
    double x = 1.2, y = 1.2;
    //模板的非类型参数赋予默认值后
    //调用模板时可以隐式实例化
    cout << multiply(x,y) << endl;
    cout << multiply<double>(x,y) << endl;
    cout << multiply<double,100>(x,y) << endl;
}
```

```
14.4
14.4
144
```

优先级：指定的类型 > 推导出的类型 > 类型的默认参数

模板参数的默认值（不管是类型参数还是非类型参数）只有在没有足够的信息用于推导时起作用。当存在足够的信息时，编译器会按照实际参数的类型去调用，不会受到默认值的影响。

成员函数模板

在一个普通类中也可以定义成员函数模板

```
class Point
{
public:
Point(double x,double y)
: _x(x)
, _y(y)
{}

//定义一个成员函数模板
//将_x转换成目标类型
template <class T>
T convert()
{
return (T)_x;
}
private:
double _x;
double _y;
};

```

```
21
22  void test0(){
23  Point pt(1.1,2.2);
24  cout << pt.convert<int>() << endl;
25  cout << pt.convert() << endl;  //error
26  }
```

——此时调用这个成员函数模板，不能采用隐式实例化的方式，不知道要将pt._x转换成什么类型

```
1  //定义一个成员函数模板
2  //将_x转换成目标类型
3  template <class T = int>
4  T convert()
5  {
6  return (T)_x;
7  }
8
9  cout << pt.convert() << endl;//ok
```

——可以给成员函数模板中类型参数赋默认值，有了默认值后才可以进行隐式实例化

```
    //定义一个成员函数模板
    //将_x转换成目标类型
    template <class T = int>
        T convert(){
            return (T)_x;
        }

    template <class T>
        T add(T t1){
            return _x + _y + t1;
        }
private:
    double _x;
    double _y;
};


void test0(){
    Point pt(1.1,2.2);
    //使用了模板的类型参数的默认值
    cout << pt.convert() << endl;
    //指定方式
    cout << pt.convert<int>() << endl;
    //推导方式
    cout << pt.add(3.2) << endl;
```

—— 如果要将成员函数模板在类之外进行实现

```
    template <class T>
        T add(T t1);
private:
    double _x;
    double _y;
};

    template <class T>
        T Point::add(T t1){
            return _x + _y + t1;
        }
```

类模板

一个类模板允许用户为类定义个一种模式，使得类中的某些数据成员、默认成员函数的参数，某些成员函数的返回值，能够取任意类型(包括内置类型和自定义类型)。

如果一个类中的数据成员的数据类型不能确定，或者是某个成员函数的参数或返回值的类型不能确定，就需要将此类声明为模板，它的存在不是代表一个具体的、实际的类，而是代表一类类。

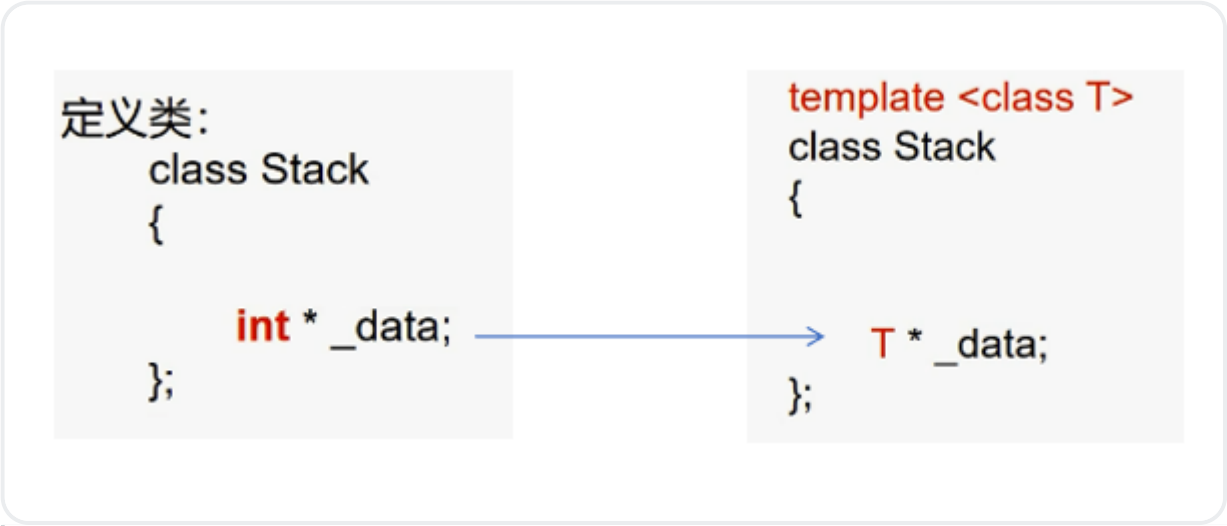
类模板的定义形式如下：

```
template <class/typename T, ...>
class 类名{
//类定义. . . . . .
};
```

类模板定义

示例，用类模板的方式实现一个Stack类，可以存放任意类型的数据

——使用函数模板实例化模板函数使用类模板实例化模板类




```cpp
template <class T, int kCapacity = 10>
class Stack
{
public:
    Stack()
    : _top(-1)
    , _data(new T[kCapacity]())
    {
        cout << "Stack()" << endl;
    }
    ~Stack(){
        if(_data){
            delete [] _data;
            _data = nullptr;
        }
        cout << "~Stack()" << endl;
    }
    bool empty() const;
    bool full() const;
    void push(const T &);
    void pop();
    T top();
private:
    int _top;
    T * _data;
};
```

类模板的成员函数如果放在类模板定义之外进行实现，需要注意

（1）需要带上template模板形参列表（如果有默认参数，此处不要写）

（2）在添加作用域限定时需要写上完整的类名和模板实参列表

```cpp
template <class T, int kCapacity>
bool Stack<T,kCapacity>::empty() const{
    return _top == -1;
}
```

定义了这样一个类模板后，就可以去创建存放各种类型的栈

```cpp
    Stack<int,20> stack2;
    Stack<double,30> stack3;

    Stack<string> stack;
```

可变模板参数

可变模板参数(variadic templates)是 C++11 新增的最强大的特性之一，它对参数进行了高度泛化，它能表示0到任意个数、任意类型的参数。由于可变模版参数比较抽象，使用起来需要一定的技巧，所以它也是 C++11 中最难理解和掌握的特性之一。

可变参数模板和普通模板的语义是一样的，只是写法上稍有区别，声明可变参数模板时需要在typename 或 class 后面带上省略号“...”，省略号写在右边，代表打包

```cpp
template <class... Args>
void func(Args... args);
```

Args叫做模板参数包，args叫做函数参数包。

类比于C语言中的printf函数的参数个数可能有很多个，用...表示，参数的个数、类型、顺序可以随意，可以写0到任意个参数。

```cpp
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

我们在定义一个函数时，可能有很多个不同类型的参数，不适合一一写出，所以提供了可变模板参数的方法。

定义一个可变模板参数

Args里面打包了 T1/T2/T3...这样的一些类型

args里面打包了函数的参数

...在左边就是打包的含义

利用可变参数模板输出参数包中参数的个数

```
template <class ...Args>//Args 模板参数包
void display(Args ...args)//args 函数参数包
{
    //输出模板参数包中类型参数个数
    cout << "sizeof...(Args) = " << sizeof...(Args) << endl;
    //输出函数参数包中参数的个数
    cout << "sizeof...(args) = " << sizeof...(args) << endl;
}

void test0(){
    display();
    display(1,"hello",3.3,true);
}
```

——试验：希望打印出传入的参数的内容

就需要对参数包进行解包。每次解出第一个参数，然后递归调用函数模板，直到**递归出口**

```
//递归的出口
void print(){
    cout << endl;
}

void print(int x){
    cout << x << endl;
}

//重新定义一个可变参数模板，至少得有一个参数
template <class T,class ...Args>
void print(T x, Args ...args)
{
    cout << x << " ";
    print(args...);
}
```

如下所示，各种调用的步骤：

```
void test1(){
    //调用普通函数
    //不会调用函数模板，因为函数模板至少有一个参数
    print();

```

```
    //cout << 2.3 << " ";
    //cout << endl;
    print(2.3);

    //cout << 1 << " ";
    //print("hello",3.6,true);
    //  cout << "hello" << " ";
    //  print(3.6,true);
    //    ...
    print(1,"hello",3.6,true);


    //在剩下一个参数时结束递归
    print(1,"hello",3.6,true,100);
}
```

——想要输出类型

```
void print(){
    cout << endl;
}

void print(int x){
    cout << x << endl;
}

//重新定义一个可变参数模板，至少得有一个参数
template <class T,class ...Args>
void print(T x, Args ...args)
{
    cout << typeid(x).name() << " ";
    print(args...);
}

print(1,"hello",3.6,true,100);
```

只剩下一个int型参数的时候，也没有使用函数模板，而是通过普通函数结束了递归。

关于递归的出口，可以使用普通函数或者普通的函数模板，但是规范操作是使用普通函数。

（1）尽量避免函数模板之间的重载；

（2）普通函数的优先级一定高于函数模板，更不容易出错。