模板

一、为什么要定义模板?

1、简化程序，少写代码，维持结构的清晰，大大提高程序的效率。

2、解决强类型语言的**严格性**和**灵活性**之间的冲突。。

2.1、带参数的宏定义(原样替换)

2.2、函数重载(函数名字相同，参数不同)

2.3、模板(将数据类型作为参数)

3、强类型语言程序设计：C/C++/Java等，有严格的类型检查，如int a = 10，在编译时候明确变量的类型，如果有　　问题就可以在编译时发现错误，安全，但是不够灵活，C++引进auto其实就是借鉴弱类型语言的特征。

弱类型程序语言设计：js/python等，虽然也有类型，但是在使用的时候直接使用let/var number,不知道变量具

体类型，由编译器解释变量类型，属于解释型语言。如果有错，到运行时才发现，虽然灵活，但是不安全。。

二、模板的定义

template <class T,...>或者template <typename T,...>

注意：class与typename在此没有任何区别，完全一致。class出现的时间比较早，通用性更好一些，typename是后来才添加了，对于早期的编译器可能识别不了`ypename关键字。

三、模板的类型

函数模板与类模板。通过参数实例化构造出具体的函数或者类，称为模板函数或者模板类。

3.1、函数模板

template <模板参数表> //模板参数列表，**此处模板参数列表不能为空**

返回类型 函数名（参数列表）

{ //函数体 }

```
template <typename T>//模板参数列表
T add(T x, T y)
{
    cout << "T add(T, T)" << endl;
    return x + y;
}
```

模板参数推导(在实参传递的时候进行推导)

函数模板----------模板函数

实例化

3.1.1、实例化：隐式实例化与显示实例化

```
 cout << "add(ia, ib) = " << add(ia, ib) << endl;//隐式实例化，没有明确说明类型，靠编译器推导
 cout << "add(da, db) = " << add<double>(da, db) << endl;//显示实例化，编译器无序推导
```

3.1.2、函数模板、普通函数间的关系

1、函数模板与普通函数是可以进行重载的

2、普通函数优先于函数模板执行

3、函数模板与函数模板之间也是可以进行重载的

3.1.3、模板头文件与实现文件

模板**不能**写成头文件与实现文件形式(类型inline函数)，或者说不能将声明与实现分开，这样会导致编译报错。

分开可以编译，但是在链接的时候是有问题的。

3.1.4、模板的特化：偏特化与全特化

```
template <> //此处模板的参数只有一个，全部特化出来就是全特化
const char *add(const char *pstr1, const char *pstr2)
{
    size_t len = strlen(pstr1) + strlen(pstr2) + 1;
    char *ptmp = new char(len);
    strcpy(ptmp, pstr1);
    strcat(ptmp, pstr2);

    return ptmp;;
}
```

具体代码的例子如下(可以详细的看看):

```
template <typename T>
T add(T x, T y)
{
    cout << "T add(T, T)" << endl;
    return x + y;
}

template <typename T>
T add(T x, T y, T z)
{
    cout << "T add(T, T, T)" << endl;
    return x + y + z;
}

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

double add(double x, double y)
```

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

template <>
const char *add(const char *pstr1, const char *pstr2)
{
    size_t len = strlen(pstr1) + strlen(pstr2) + 1;
    char *ptmp = new char(len);
    strcpy(ptmp, pstr1);
    strcat(ptmp, pstr2);

    return ptmp;;
}

void test()
{
    int ia = 3, ib = 4, ic = 5;
    double da = 1.1, db = 5.5;
    char ca = 'a', cb = 1;

    string s1 = "hello";
    string s2 = "world";

    const char *pstr1 = "hubei";
    const char *pstr2 = ",wuhan";

    cout << "add(ia, ib) = " << add(ia, ib) << endl;//隐式实例化
    cout << "add(da, db) = " << add<double>(da, db) << endl;//显示实例化
    cout << "add(ca, cb) = " << add(ca, cb) << endl;
    cout << "add(s1, s2) = " << add(s1, s2) << endl;

    /* cout << "add(ia, db) = " << add(ia, db) << endl;//函数模板必须进行严格的推导，如果没有普通函数形式，这就话就error */
    cout << "add(ia, ib, ic) = " << add(ia, ib, ic) << endl;
    cout << "add(pstr1, pstr2) = " << add(pstr1, pstr2) << endl;
}
```

3.1.5、函数模板的参数类型

1、类型参数，class T 这种就是类型参数

2、非类型参数 常量表达式，整型：bool/char/short/int/long/size_t,注意：float/double这些就不是整型

```
template <typename T = int, short kMin = 10>
T multiply(T x, T y)
{
    return x * y * kMin;
}

void test()
{
    int ia = 3, ib = 4;
    double da = 3.3, db = 4.4;
    cout << "multiply(ia, ib) = " << multiply(ia, ib) << endl;
```

```
    cout << "multiply(ia, ib) = " << multiply<int, 4>(ia, ib) << endl;
    cout << "multiply(ia, ib) = " << multiply<double, 4>(da, db) << endl;
}
```

3.1.6、成员函数模板

就是类的成员函数也可以设置为模板，可以写例子看看。

```
class Point
{
public:
    //.............
    //成员函数模板,成员函数模板也是可以设置默认值
    template <typename T = int>
    T func()
    {
        return (T)_dx;
    }

private:
    double _dx;
    double _dy;
};

void test()
{
    Point pt(1.1, 2.2);
    cout << "pt.func() = " << pt.func<int>() << endl;
    cout << "pt.func() = " << pt.func<double>() << endl;
    cout << "pt.func() = " << pt.func() << endl;
}
```

3.2、类模板

使用与函数模板也差不多，只是要注意模板的嵌套(函数模板与类模板都可以嵌套，比如函数参数是模板，类模板里面还有类模板)，直接使用例子看类模板。

```
//类模板
template <typename T, size_t kSize = 10>//类型参数T与非类型参数kSize
class Stack
{
public:
    Stack()
    : _top(-1)
    , _data(new T[kSize]())
    {

    }
    ~Stack();
    bool empty() const;
    bool full() const;
    void push(const T &t);
    void pop();
    T top() const;
private:
    int _top;
    T *_data;
```

```
};

//类模板在类外面定义成员函数时候需要注意，模板是有类型的，需要使用参数加类型
template <typename T, size_t kSize>
Stack<T, kSize>::~Stack()
{
    if(_data)
    {
        delete [] _data;
        _data = nullptr;
    }
}

template <typename T, size_t kSize>
bool Stack<T, kSize>::empty() const
{
    return -1 == _top;//_top = -1
}

template <typename T, size_t kSize>
bool Stack<T, kSize>::full() const
{
    return _top == kSize - 1;
}

template <typename T, size_t kSize>
void Stack<T, kSize>::push(const T &t)
{
    if(!full())
    {
        _data[++_top] = t;
    }
    else
    {
        cout << "The Stack is full, cannot push any data" << endl;
    }
}

template <typename T, size_t kSize>
void Stack<T, kSize>::pop()
{
    if(!empty())
    {
        --_top;
    }
    else
    {
        cout << "The Stack is empty" << endl;
    }

}
template <typename T, size_t kSize>
T Stack<T, kSize>::top() const
{
    return _data[_top];
}

void test()
```

```
{
    Stack<int, 8> st;
}

void test1()
{
    Stack<string> st;
}
```

模板的嵌套，可以看个例子就ok。

```
template<class T>
class Outside
{
public:
    template <class R>
    class Inside
    {
    public:
        Inside(R x)
        {
            r = x;
        }

        void disp() {cout << "Inside: " << r << endl;}
    private:
        R r;
    };

    Outside(T x) : t(x)
    {}


    void disp()
    {
        cout<<"Outside:";
        t.disp();
    }

private:
    Inside<T> t;
};


void test()
{
    Outside<int>::Inside<double> obin(3.5);
    obin.disp();

    Outside<int> obout(2);
    obout.disp();

}
```

四、可变模板参数

是C++11新增的最强大的特性之一，它对参数进行了高度的泛化，它能表示0到任意个数、任意类型的参数。

4.1、模板参数包

```
template<typename… Args> class tuple;//tuple是元组的意思，其模板参数就是模板参数包
```

Args标识符的左侧使用了省略号,在C++11中Args被称为“模板参数包”,表示可以接受任意多个参数作为模板参数,编译器将多个模板参数**打包**成“单个”的模板参数包.

4.2、函数参数包

```
template<typename…T> void f(T…args);//args就是函数参数包
```

`args`被称为函数参数包,表示函数可以接受多个任意类型的参数.

在C++11标准中，要求**函数参数包必须唯一，且是函数的最后一个参数**; 模板参数包则没有。

当使用参数包时，省略号位于参数名称的右侧，表示立即展开该参数，这个过程也被称为**解包。**

4.3、可变模板参数的优势(有两条)

1、参数个数，那么对于模板来说，在模板推导的时候，就已经知道参数的个数了，也就是说在编译的时候就确定　　了，这样编译器就存在可能去优化代码

2、参数类型，推导的时候也已经确定了，模板函数就可以知道参数类型了。

```
template <typename... Args>
void print(Args... args)
{
    cout << "sizeof...(Agrs) = " << sizeof...(Args) << endl;
    cout << "sizeof...(agrs) = " << sizeof...(args) << endl;
}

void display()
{
    cout << endl;
}

template <typename T, typename... Args>
void display(T t, Args... args)
{
    cout << t << " ";
    display(args...);//当... 位于args右边的时候叫做解包
}

void test()
{
    string s1 = "hello";

    print();
    print(1, 2.2);
    print('a', true, s1);
    print(1, 2.2, 'b', "hello");
```

```
}
void test2()
{
    string s1 = "hello";

    display();
    display(1, 2.2);
    display('a', true, s1);
    display(1, 2.2, 'b', "hello");
}

template <class T>
T sum(T t)
{
    return t;
}

template <typename T, typename... Args>
T sum(T t, Args... args)
{
    return t + sum(args...);
}

void test3()
{
    cout << "sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) = "
         << sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) << endl;
}
```