模板初阶

泛型编程

泛型编程：编写与类型无关的通用代码，是代码复用的一种手段。模板(函数模板和类模板)是泛型编程的基础。

函数模板

函数模板代表了一个函数家族，该函数模板与类型无关，在使用时被参数化，根据实参类型产生函数的特定类型版本。

函数模板格式

```
template<typename T>
//关键字<模板参数(类型)>
void Swap(T& left,T& right)
{

}
//返回值类型 函数名(参数列表)
```

typename是用来定义模板参数关键字，也可以使用class

函数模板原理

我们写了模板，编译器通过模板实例化出对应的函数或类。

在编译器编译阶段(预处理阶段)，对于模板函数的使用，编译器需要根据传入的实参类型来推演生成对应类型的函数以供调用。

函数模板实例化

```
template<class T>
T Add(const T& left, const T& right)
{
    return left + right;
}
int main()
{
    int a1 = 10, a2 = 20;
    double d1 = 10.0, d2 = 20.0;
    //隐式实例化(T的类型是编译器自己推到的)
    Add(a1, a2);
    Add(d1, d2);

    //显示实例化(指定T的类型)
    Add<int>(a1,d2);

    return 0;
}
```

模板参数的匹配原则

- 一个非模板函数可以和一个同名的函数模板同时存在，而且该函数模板还可以被实例化为这个非模板函数。
- 对于非模板函数和同名函数模板，如果其他条件都相同，在调动时会优先调用非模板函数而不会从该模板产生出一个实例。如果模板可以产生一个具有更好匹配的函数， 那么将选择模板。

```
// 专门处理int的加法函数
int Add(int left, int right)
{
    return left + right;
}
// 通用模板加法函数
template<class T>
T Add(T left, T right)
{
    return left + right;
}
int main()
{
    Add(1, 2); // 与非模板函数匹配，编译器不需要特化
    Add<int>(1, 2); // 调用编译器特化的Add版本
}
```

类模板

类模板格式

```
template<class T1,class T2,……>
class 类模板名
{

}
// 注意：Vector不是具体的类，是编译器根据被实例化的类型生成具体类的模具
template<class T>
class Vector
{
public :
    Vector(size_t capacity = 10)
        : _pData(new T[capacity])
        , _size(0)
        , _capacity(capacity)
    {}
    // 在类中声明，在类外定义。
    ~Vector();
private:
    T* _pData;
    size_t _size;
    size_t _capacity;
};
// 类模板中函数放在类外进行定义时，需要加模板参数列表
template <class T>
Vector<T>::~Vector()
```

```
{
    if(_pData)
    delete[] _pData;
    _size = _capacity = 0;
}
```

类模板的实例化

类模板实例化需要在类模板名字后跟<>，然后将实例化的类型放在<>中即可，类模板名字不是真正的类，而实例化的结果才是真正的类。

```
// Vector类名，Vector<int>才是类型
Vector<int> s1;
Vector<double> s2;
```

非类型模板参数


```
#include<iostream>

//非类型模板参数
template<class T,int N>
//这里的N就是非类型模板参数
class Array
{
public:

private:
    T _arr[N];
};

int main()
{
    Array<int, 10> a1;
    Array<int, 5> a2;
    return 0;
}
```

- 浮点数、字符串、类对象不允许作为非类型模板参数；

模板的特化

在原模板的基础上，针对特殊类型所进行特殊化的实现方式。

函数模板特化

1. 首先有一个基础的函数模板
2. 关键字template后面接一对<>
3. 函数名后面带<>，括号中指定需要特化的类型
4. 函数参数列表，必须要和模板函数的基础参数完全相同

类模板特化

```
//类模板
template<class T1,class T2>
class Date
{
public:
    Date()
    {

    }
private:
    T1 _d1;
    T2 _d2;
};
```

- 全特化
    - 将模板参数列表中所有的参数都确定化

```
template<>
class Date<int,char>
{
public:
    Date()
    {
    }
private:
    T1 _d1;
    T2 _d2;
};
```

- 偏特化
    - 部分特化：将模板参数列表中的一部分参数特化

```
//将第二个参数特化为int
template<class T1>
class Date<T1,int>
{
public:
    Date()
    {
    }
private:
    T1 _d1;
    T2 _d2;
};
```

- 对参数的进一步限制

```
//两个参数特化为指针类型
template<class T1,class T2>
class Data<T1*,T2*>
{
public:
    Data()
    {
    }
private:
    T1 _d1;
    T2 _d2;
}
```

模板分离编译

分离编译：项目工程中一般将函数或者类的声明放到.h文件中，将函数或者类的定义放到.cpp文件中，每个源文件单独编译生成目标文件，最后将所有目标文件链接起来形成单一的可执行文件的过程称为分离编译模式。

- 同样是分离编译，普通函数/类可以，函数模板/类模板位为什么不行?
    1. 预处理——展开头文件、宏替换、条件编译、去掉注释
    2. 编译——检查语法、生成汇编代码
    3. 汇编——将汇编代码转成二进制的机器代码
    4. 链接——将目标文件整合到一起

模板的优缺点

- 优点
    - 模板复用了代码，节省资源，更快的迭代开发；
    - 增强了代码的灵活性；
- 缺点
    - 模板会导致代码膨胀问题，也会导致编译时间边长；
    - 出现模板错误时，错误信息十分凌乱，不易定位错误