Lesson12---模板

【本节目标】

- 1. 非类型模板参数
- 2. 类模板的特化
- 3. 模板的分离编译

1. 非类型模板参数

> 模板参数分类类型形参与非类型形参。
>
> 类型形参即：出现在模板参数列表中，跟在class或者typename之类的参数类型名称。
>
> 非类型形参，就是用一个常量作为类(函数)模板的一个参数，在类(函数)模板中可将该参数当成常量来使用。

```
namespace bite
{
    // 定义一个模板类型的静态数组
    template<class T, size_t N = 10>
    class array
    {
    public:
        T& operator[](size_t index){return _array[index];}
        const T& operator[](size_t index)const{return _array[index];}

        size_t size()const{return _size;}
        bool empty()const{return 0 == _size;}

    private:
        T _array[N];
        size_t _size;
    };
}

```

> 注意：
>
> 1. 浮点数、类对象以及字符串是不允许作为非类型模板参数的。
> 2. 非类型的模板参数必须在编译期就能确认结果。

2. 模板的特化

2.1 概念

通常情况下，**使用模板可以实现一些与类型无关的代码，但对于一些特殊类型的可能会得到一些错误的结果**，需要特殊处理，比如：实现了一个专门用来进行小于比较的函数模板

```
// 函数模板 -- 参数匹配
template<class T>
bool Less(T left, T right)
{
    return left < right;
}

int main()
{
    cout << Less(1, 2) << endl;   // 可以比较，结果正确

    Date d1(2022, 7, 7);
    Date d2(2022, 7, 8);
    cout << Less(d1, d2) << endl;  // 可以比较，结果正确

    Date* p1 = &d1;
    Date* p2 = &d2;
    cout << Less(p1, p2) << endl;  // 可以比较，结果错误

    return 0;
}
```

可以看到，Less绝对多数情况下都可以正常比较，但是在特殊场景下就得到错误的结果。上述示例中，p1指向的d1显然小于p2指向的d2对象，但是Less内部并没有比较p1和p2指向的对象内容，而比较的是p1和p2指针的地址，这就无法达到预期而错误。

此时，就**需要对模板进行特化。即：在原模板类的基础上，针对特殊类型所进行特殊化的实现方式。**模板特化中分为**函数模板特化**与**类模板特化。**

2.2 函数模板特化

函数模板的特化步骤：

1. 必须要先有一个基础的函数模板
2. 关键字template后面接一对空的尖括号<>
3. 函数名后跟一对尖括号，尖括号中指定需要特化的类型
4. 函数形参表: 必须要和模板函数的基础参数类型完全相同，如果不同编译器可能会报一些奇怪的错误。

```
// 函数模板 -- 参数匹配
template<class T>
bool Less(T left, T right)
{
    return left < right;
}

// 对Less函数模板进行特化
template<>
bool Less<Date*>(Date* left, Date* right)
{
```

```
    return *left < *right;
}

int main()
{
    cout << Less(1, 2) << endl;

    Date d1(2022, 7, 7);
    Date d2(2022, 7, 8);
    cout << Less(d1, d2) << endl;

    Date* p1 = &d1;
    Date* p2 = &d2;
    cout << Less(p1, p2) << endl;  // 调用特化之后的版本，而不走模板生成了
    return 0;
}
```

注意：一般情况下如果函数模板遇到不能处理或者处理有误的类型，为了实现简单通常都是将该函数直接给出。

```
bool Less(Date* left, Date* right)
{
    return *left < *right;
}
```

该种实现简单明了，代码的可读性高，容易书写，因为对于一些参数类型复杂的函数模板，特化时特别给出，因此函数模板不建议特化。

2.3 类模板特化

2.3.1 全特化

全特化即是将模板参数列表中所有的参数都确定化。

```
template<class T1, class T2>
class Data
{
public:
    Data() {cout<<"Data<T1, T2>" <<endl;}
private:
    T1 _d1;
    T2 _d2;
};

template<>
class Data<int, char>
{
public:
    Data() {cout<<"Data<int, char>" <<endl;}
private:
    int _d1;
    char _d2;
```

```
};

void TestVector()
{
    Data<int, int> d1;
    Data<int, char> d2;
}
```

2.3.2 偏特化

偏特化：任何针对模版参数进一步进行条件限制设计的特化版本。比如对于以下模板类：

```
template<class T1, class T2>
class Data
{
public:
    Data() {cout<<"Data<T1, T2>" <<endl;}
private:
    T1 _d1;
    T2 _d2;
};
```

偏特化有以下两种表现方式：

- 部分特化

  将模板参数类表中的一部分参数特化。

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

- 参数更进一步的限制

  偏特化并不仅仅是指特化部分参数，而是针对模板参数更进一步的条件限制所设计出来的一个特化版本。

  ```
  //两个参数偏特化为指针类型
  template <typename T1, typename T2>
  class Data <T1*, T2*>
  {
  public:
      Data() {cout<<"Data<T1*, T2*>" <<endl;}

  private:
  ```

```
    T1 _d1;
    T2 _d2;
};

//两个参数偏特化为引用类型
template <typename T1, typename T2>
class Data <T1&, T2&>
{
public:
    Data(const T1& d1, const T2& d2)
        : _d1(d1)
        , _d2(d2)
    {
            cout<<"Data<T1&, T2&>" <<endl;
    }

private:
    const T1 & _d1;
    const T2 & _d2;
 };

void test2 ()
{
     Data<double , int> d1;      // 调用特化的int版本
     Data<int , double> d2;      // 调用基础的模板
     Data<int *, int*> d3;       // 调用特化的指针版本
     Data<int&, int&> d4(1, 2);  // 调用特化的指针版本
}
```

2.3.3 类模板特化应用示例

有如下专门用来按照小于比较的类模板Less：

```
#include<vector>
#include <algorithm>

template<class T>
struct Less
{
    bool operator()(const T& x, const T& y) const
    {
        return x < y;
    }
};

int main()
{
    Date d1(2022, 7, 7);
    Date d2(2022, 7, 6);
    Date d3(2022, 7, 8);

    vector<Date> v1;
    v1.push_back(d1);
```

```
    v1.push_back(d2);
    v1.push_back(d3);
    // 可以直接排序，结果是日期升序
    sort(v1.begin(), v1.end(), Less<Date>());

    vector<Date*> v2;
    v2.push_back(&d1);
    v2.push_back(&d2);
    v2.push_back(&d3);

    // 可以直接排序，结果错误日期还不是升序，而v2中放的地址是升序
    // 此处需要在排序过程中，让sort比较v2中存放地址指向的日期对象
    // 但是走Less模板，sort在排序时实际比较的是v2中指针的地址，因此无法达到预期
    sort(v2.begin(), v2.end(), Less<Date*>());

    return 0;
}
```

通过观察上述程序的结果发现，对于日期对象可以直接排序，并且结果是正确的。但是如果待排序元素是指针，结果就不一定正确。因为：sort最终按照Less模板中方式比较，所以只会比较指针，而不是比较指针指向空间中内容，此时可以使用类版本特化来处理上述问题：

```
// 对Less类模板按照指针方式特化
template<>
struct Less<Date*>
{
    bool operator()(Date* x, Date* y) const
    {
        return *x < *y;
    }
};
```

特化之后，在运行上述代码，就可以得到正确的结果

3 模板分离编译

3.1 什么是分离编译

一个程序（项目）由若干个源文件共同实现，而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。

3.2 模板的分离编译

假如有以下场景，模板的声明与定义分离开，在头文件中进行声明，源文件中完成定义：

```
// a.h
template<class T>
T Add(const T& left, const T& right);


// a.cpp
```

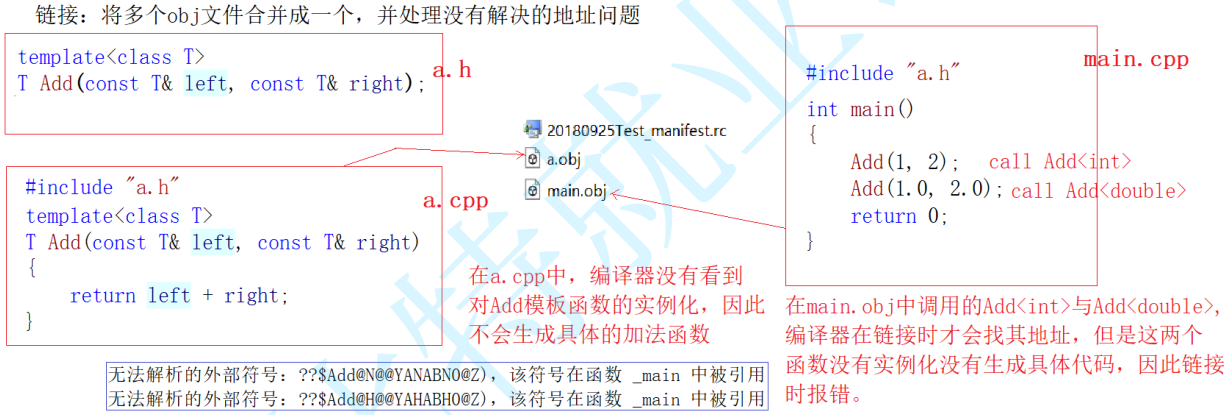
```cpp
template<class T>
T Add(const T& left, const T& right)
{
    return left + right;
}

// main.cpp
#include"a.h"
int main()
{
    Add(1, 2);
    Add(1.0, 2.0);

    return 0;
}
```

分析：

C/C++程序要运行，一般要经历一下步骤：
预处理 ---> 编译 ---> 汇编 ---> 链接

编译：对程序按照语言特性进行词法、语法、语义分析，错误检查无误后生成汇编代码
注意头文件不参与编译 编译器对工程中的多个源文件是分离开单独编译的。

链接：将多个obj文件合并成一个，并处理没有解决的地址问题



3.3 解决方法

1. **将声明和定义放到一个文件 "xxx.hpp" 里面或者xxx.h其实也是可以的。**推荐使用这种。
2. **模板定义的位置显式实例化。**这种方法不实用，不推荐使用。

【分离编译扩展阅读】 http://blog.csdn.net/pongba/article/details/19130

4. 模板总结

【优点】

1. **模板复用了代码，节省资源，更快的迭代开发，C++的标准模板库(STL)因此而产生**
2. **增强了代码的灵活性**

【缺陷】

1. **模板会导致代码膨胀问题，也会导致编译时间变长**
2. **出现模板编译错误时，错误信息非常凌乱，不易定位错误**