Lesson13---模板进阶

【本节目标】

- 非类型模板参数
- 类模板的特化
- 类模板特化的应用之类型萃取
- 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。
>
> 请试试下面的OJ题目，现在要求变了要求O(1)的时间复杂度完成。
>
> 求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C

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**，比如：

```
template<class T>
bool IsEqual(T& left, T& right)
{
    return left == right;
}

void Test()
{
    char* p1 = "hello";
    char* p2 = "world";

    if(IsEqual(p1, p2))
        cout<<p1<<endl;
    else
        cout<<p2<<endl;
}
```

> 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。**

2.1 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。

```
template<>
bool IsEqual<char*>(char*& left, char*& right)
{
    if(strcmp(left, right) > 0)
        return true;

    return false;
}
```

> **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 IsEqual(char* left, char* right)
{
    if(strcmp(left, right) > 0)
        return true;

    return false;
}
```

2.2 类模板特化

2.2.1 全特化

> 全特化即是将模板参数类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:
    T1 _d1;
    T2 _d2;
};

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

2.2.2 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
}
```

3. 类模板特化应用之类型萃取(了解)

> **问题：如何实现一个通用的拷贝函数？下面的实现有问题吗？**

3.1 使用memcpy拷贝

```
template<class T>
void Copy(T* dst, const T* src, size_t size)
{
    memcpy(dst, src, sizeof(T)*size);
}

int main()
{
    // 试试下面的代码
    string strarr1[3] = {"11", "22", "33"};
    string strarr2[3];
    Copy(strarr2, strarr1, 3);
}
```

> 上述代码虽然对于任意类型的空间都可以进行拷贝，但是如果**拷贝自定义类型对象就可能会出错**，因为**自定义类型对象有可能会涉及到深拷贝(比如string)**，而**memcpy属于浅拷贝**。如果**对象中涉及到资源管理，就只能用赋值。**

3.2 使用赋值方式拷贝

```
template<class T>
void Copy(T* dst, const T* src, size_t size)
{
    for(size_t i = 0; i < size; ++i)
    {
        dst[i] = src[i];
    }
}
```

> 用**循环赋值的方式虽然可以，但是代码的效率比较低**，而C/C++程序最大的优势就是效率高。那能否将另种方式的优势结合起来呢？遇到内置类型就用memcpy来拷贝，遇到自定义类型就用循环赋值方式来做呢？

3.3 增加bool类型区分自定义与内置类型

```
template<class T>
void Copy(T* dst, const T* src, size_t size, bool IsPODType)
{
    if(IsPODType)
        memcpy(dst, src, sizeof(T)*size);
    else
    {
        for(size_t i = 0; i < size; ++i)
            dst[i] = src[i];
    }
}
```

> 通过多增加一个参数，就可将两种拷贝的优势体现结合起来。但缺陷是：**用户需要根据所拷贝元素的类型去传递第三个参数，那出错的可能性就增加**。那能否让函数自动去识别所拷贝类型是内置类型或者自定义类型呢？

3.4 使用函数区分内置于自定义类型

> 因为内置类型的个数是确定的，可以将所有内置类型集合在一起，如果能够将所拷贝对象的类型确定下来，在内置类型集合中查找其是否存在即可确定所拷贝类型是否为内置类型

```
//
// POD: plain old data 平凡类型（无关痛痒的类型）--基本类型
// 指在C++ 中与 C兼容的类型，可以按照 C 的方式处理。
//
// 此处只是举例，只列出个别类型
bool IsPODType(const char* strType)
{
    const char* arrType[] = {"char", "short", "int", "long", "long long", "float",
"double", "long double"};
    for(size_t i = 0; i < sizeof(array)/sizeof(array[0]); ++i)
    {
        if(0 == strcmp(strType, arrType[i]))
            return true;
    }

    return false;
}

template<class T>
void Copy(T* dst, const T* src, size_t size)
{
    if(IsPODType(typeid(T).name()))
        memcpy(dst, src, sizeof(T)*size);
    else
    {

        for(size_t i = 0; i < size; ++i)
```

```
            dst[i] = src[i];
    }
}
```

通过**typeid来确认所拷贝对象的实际类型**，然后再**在内置类型集合中枚举其是否出现过**，**既可确认**所拷贝元素的类型为内置类型或者自定义类型。但缺陷是：**枚举需要将所有类型遍历一遍**，每次比较都是字符串的比较，**效率比较低。**

3.5 类型萃取

为了将内置类型与自定义类型区分开，给出以下两个类分别代表内置类型与自定义类型。

```
// 代表内置类型
struct TrueType
{
    static bool Get(){return true;}
};

// 代表自定义类型
struct FalseType
{
     static bool Get(){return false;}
};
```

给出以下类模板，将来用户可以按照任意类型实例化该类模板。

```
template<class T>
struct TypeTraits
{ typedef FalseType   IsPODType; };
```

对上述的类模板进行以下方式的实例化：

```
template<>
struct TypeTraits<char>
{ typedef TrueType     IsPODType;};

template<>
struct TypeTraits<short>
{ typedef TrueType     IsPODType;};

template<>
struct TypeTraits<int>
{ typedef TrueType     IsPODType; };

template<>
struct TypeTraits<long>
{ typedef TrueType     IsPODType; };

// ... 所有内置类型都特化一下
```

> 通过对TypeTraits类模板重写改写方式四中的Copy函数模板，来确认所拷贝对象的实际类型。

```
/*
T为int：TypeTraits<int>已经特化过，程序运行时就会使用已经特化过的TypeTraits<int>，该类中的
        IsPODType刚好为类TrueType，而TrueType中Get函数返回true，内置类型使用memcpy方式拷贝

T为string：TypeTraits<string>没有特化过，程序运行时使用TypeTraits类模板，该类模板中的IsPODType
刚
           好为类FalseType，而FalseType中Get函数返回true，自定义类型使用赋值方式拷贝
*/
template<class T>
void Copy(T* dst, const T* src, size_t size)
{
    if(TypeTraits<T>::IsPODType::Get())
        memcpy(dst, src, sizeof(T)*size);
    else
    {
        for(size_t i = 0; i < size; ++i)
            dst[i] = src[i];
    }
}

int main()
{
    int a1[] = {1,2,3,4,5,6,7,8,9,0};
    int a2[10];
    Copy(a2, a1, 10);

    string s1[] = {"1111", "2222", "3333", "4444"};
    string s2[4];
    Copy(s2, s1, 4);
    return 0;
}
```

4 模板分离编译

4.1 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。

4.2 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
template<class T>

T Add(const T& left, const T& right)
```

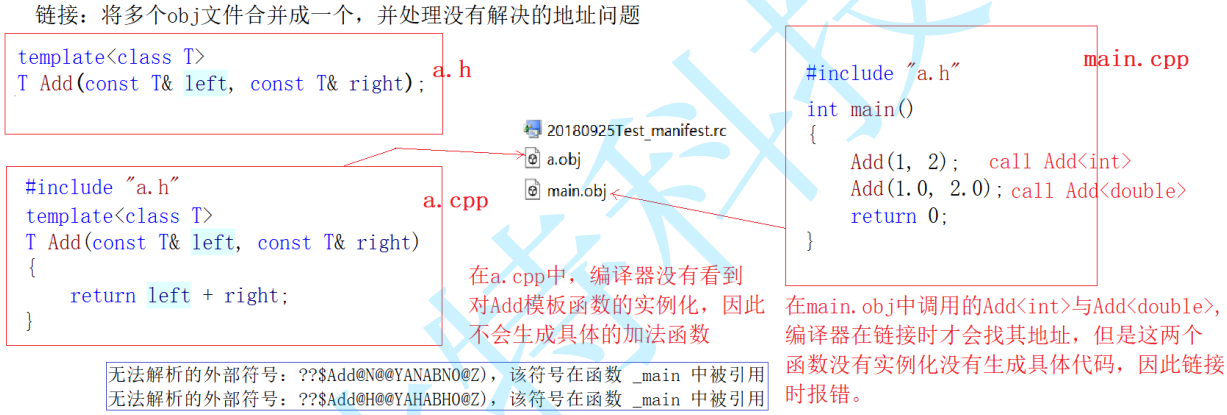
```
{
    return left + right;
}

// 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



4.3 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

5. 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**

【作业】

将本届内容写一篇博客进行总结。