2020/05/26_C++_第15课_泛型编程（模板）

笔记本： C++
创建时间： 2020/5/26 星期二 15:35
作者： ileemi

函数模板

解决问题：解决功能相似只有类型不同、重复、冗余的代码
示例：

```
int Add(int val1, int val2)
{
    return val1 + val2;
}
float Add(float val1, float val2)
{
    return val1 + val2;
}
double Add(double val1, double val2)
{
    return val1 + val2;
}
```

宏的缺点：

- 没有语法、类型检查
- 调试的时候观察不到变量的值

C++提供了一种解决上述问题的方法：**模板**

隐式实例化

函数模板定义的一般形式如下所示：

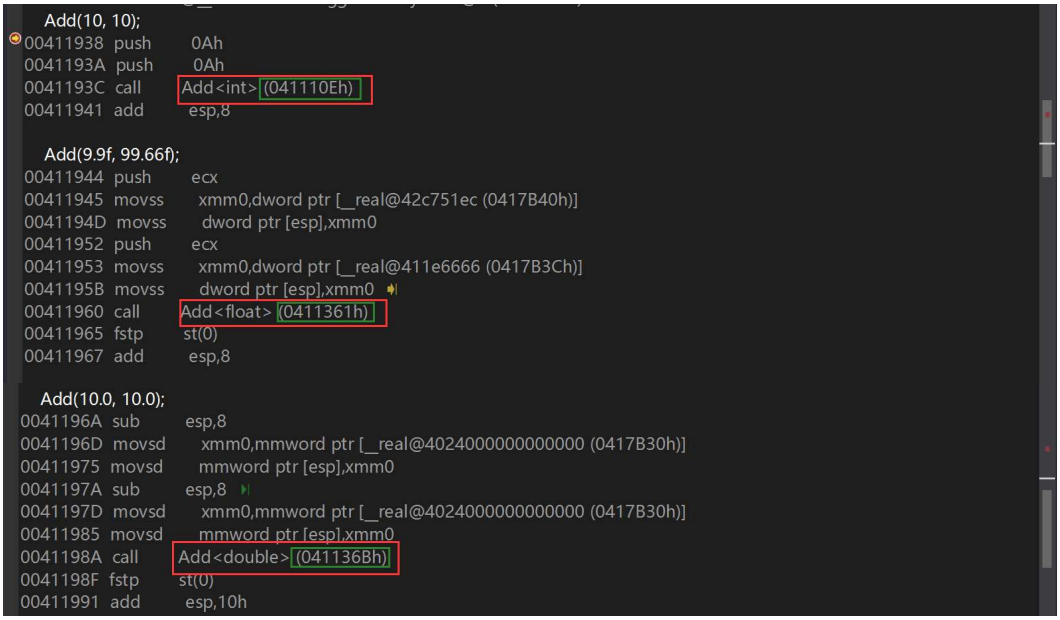
```
template <typename type>
return-type func-name(parameter list)
{
    // 函数的主体
}
```

代码示例：

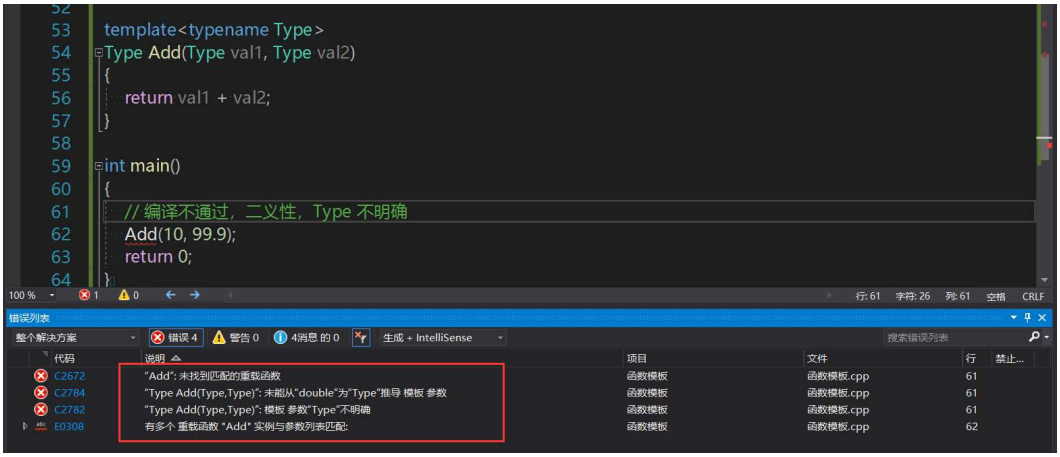
```
//说明：
//Type --> 占位符，  一般情况下写成 T(Type)

template<typename Type>
Type Add(Type nVal1, Type nVal2)
{
    return nVal1 + nVal2;
}
int main()
{
    // 模板实例化 -隐式实例化 编译器自动推导类型参数

    /*
    template<typename Type>
    Type Add(Type nVal1, Type nVal2)
    {
    return nVal1 + nVal2;
    }

    Type = int --> Type（参数类型）

    int Add(int nVal1, int nVal2)
    {
    return nVal1 + nVal2;
    }
    */
    // 通过函数找到模板，根据参数类型、通过类型替换Type，生成对应函数代
码
    Add(2, 4);
```

```
    // 隐式实例化，同上
    // Type --> float
    Add(3.3f, 9.9f);

    // 同上
    // Type --> double
    Add(66.6, 99.9);

    // 编译器生成对应的函数代码，通过反汇编代码可以查看，编译期生成

    return 0;
}
```

通过上面的代码可以看出，只定义一个函数模板，就可以通过调用该模板，实现不同类型的数值进行对用的操作，这里编译器其实是为其生成了对应的代码。生成对应的代码时在编译期间生成的，通过编译预处理是观察不到其对应的代码的，通过反汇编代码可以看出其生成代码对应的地址是不同的，如下图所示。



在main函数内调用模板函数，当参数的类型是多个的时候，程序运行如下：



显示实例化

解决上述代码 通过隐式实例化存在二义性的问题：

- 再添加一个占位符（占位符可以是多个，用来解决不同参数类型）
- 使用显示实例化（明确参数类型）

```
Type --> 占位符，  一般情况下写成 T(Type)
template<typename T1, typename T2>
T1 Add(T1 nVal1, T2 nVal2)
{
    return nVal1 + nVal2;
}
int main()
{
    // 显示实例化（推荐使用），明确参数类型
    Add<int, double>(999, 66.66);
    return 0;
}
```

模板支持自定义类型

显示实例化是否支持自定义类型?

显示实例化 同时还可以用于自定义类型（类），注意：需要在类中添加重载运算符

```
class CInteger
{
public:
    CInteger(int nVal):m_nVal(nVal)
    {
    }
    // 运算符 + 重载
    int operator+(int nVal)
    {
        return m_nVal + nVal;
    }
private:
    int m_nVal;
};

template<typename T1, typename T2>
T1 Add(T1 val1, T2 val2)
{
    // 调用运算符重载
    return val1 + val2;
```

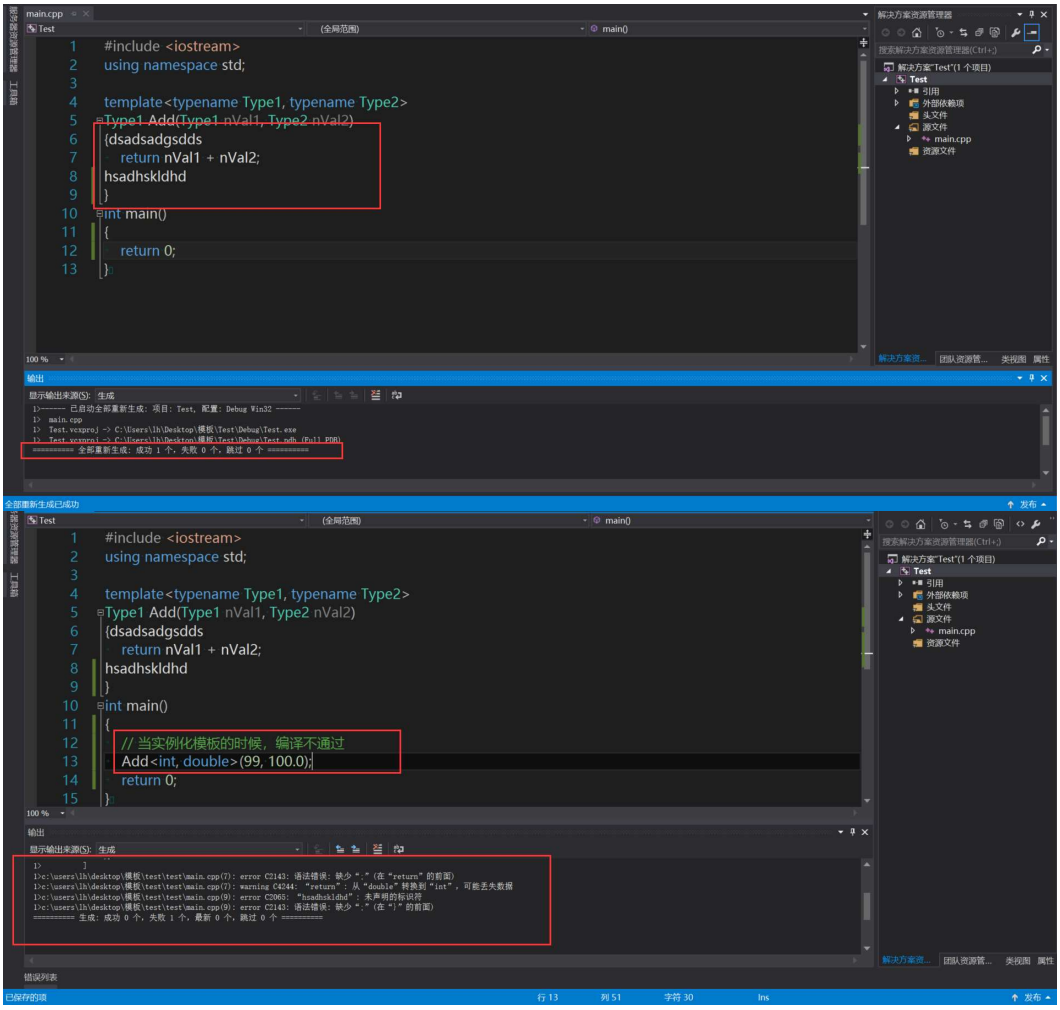
```
}

int main()
{
    // 显示实例化（推荐使用），明确参数类型
    /*
    替换后的函数为：
    CInteger Add(CInteger val1, int val2)
    {
        return val1 + val2;
    }
    */
    Add<CInteger, int>(10, 99.9);

    // 编译不通过，调用Add函数的时候，将类型带入后为：对象 + int，运算符没有重载
    // error C2676: 二进制“+”:“T1”不定义该运算符或到预定义运算符可接收的类型的转换
    return 0;
}
```

模板也可以重载

代码示例:

```
template<typename T1, typename T2>
T1 Add(T1 val1, T2 val2)
{
    return val1 + val2;
}
template<typename T1, typename T2, typename T3>
T1 Add(T1 val1, T2 val2, T3 val3)
{
    return val1 + val2 + val3;
}
int main()
{
    Add<int, double>(999, 1.00);
    Add<int, double, int>(999, 1.00, 666);
    return 0;
}
```

VS2015 模板的小Bug



模板是在使用的时候才会进行实例化

注意： 调试模板一定要进行实例化

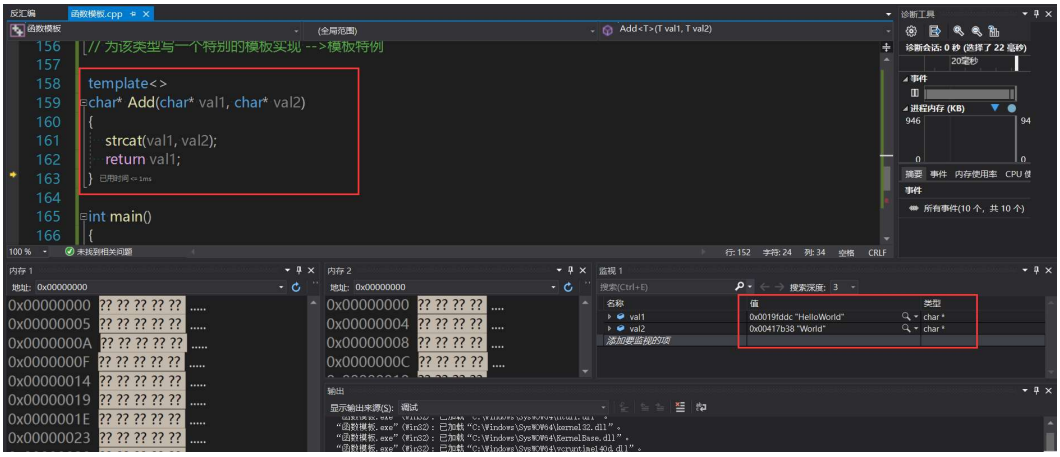
没有实例化，编译出来的文件没有模板的任何信息（代码，声明等）

模板特例

示例：

```
// 模板特例
template<typename T>
T Add(T val1, T val2)
{
    // 此时 编译不通过
    // error C2110: “+”：不能添加两个指针
    return val1 + val2;
}

// 一般的模板内有些类型和其它的类型不匹配
// 为该类型写一个特别的模板实现 -->模板特例

template<>
char* Add(char* val1, char* val2)
```

```
{
    strcat(val1, val2);
    return val1;
}

int main()
{
    Add<int>(999, 66.99);

    char szBuff[255] = { "Hello" };
    Add<char*>(szBuff, (char*)"World");     // 字符串拼接
    return 0;
}
```



模板的实现和声明

函数模板的 声明和实现是否可以分开写?

分开写，没有实例化，链接时不通过，在main函数中通过包含头文件是没有用的，只是将模板的声明拷贝过来，编译器检查到只有声明，没有实现，编译可以通过，但是链接的时候就不通过。

特性：用的时候才进行实例化

注意：模板的声明和实现都应放在头文件内

类模板

成员函数模板

C++允许一个类的成员函数是一个模板-->成员函数模板

```
class CFoo
{
```

```
public:
    // 成员函数模板
    template<typename T>
    void Print(T Val)
    {
        cout << Val << endl;
        cout << m_nVal << endl;
    }
private:
    int m_nVal = 999;
};
int main()
{
    CFoo foo;
    foo.Print<int>(666);
    foo.Print<float>(99.99f);
    foo.Print<double>(6.18);
    foo.Print<char>('A');

    return 0;
}
```

类模板

```
// 类的成员函数模板
template<typename T>
class CFoo
{
public:

    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val)
    {
        cout << Val << endl;
        cout << m_Val << endl;
    }
private:
    T m_Val = 999;
};
int main()
{
    // 类模板只能显示实例化，不能隐式实例化
    // 这行代码的意思是，当定义一个类对象foo的时候，定义的时候，需要知
```

```
其类型
    // 当知道类型的时候，模板就需要进行实例化
    //CFoo foo(666);

    CFoo<int> foo(777);
    foo.Print(678);

    CFoo<int> foo1(666);
    foo1.Print(678);
    return 0;
}
```

成员函数的模板可以写在类外

使用操作:

```
template<typename T>
class CFoo
{
public:

    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val);
private:
    T m_Val = 999;
};

// 该成员函数属于CFoo类模板中的成员，需要在类名后加<T>
template<typename T>
void CFoo<T>::Print(T Val)
{
    cout << Val << endl;
    cout << m_Val << endl;
}

int main()
{
    // 类模板只能显示实例化，不能隐式实例化
    // 这行代码的意思是，当定义一个类对象foo的时候，定义的时候，需要知
其类型
    // 当知道类型的时候，模板就需要进行实例化
    //CFoo foo(666);
```

```
    CFoo<int> foo(777);
    foo.Print(678);

    CFoo<float> foo1(666.66f);
    foo1.Print(678);
    return 0;
}
```

类成员函数的特例

使用操作:

```
template<typename T>
class CFoo
{
public:
    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val);
private:
    T m_Val = 999;
};

template<typename T>
void CFoo<T>::Print(T Val)
{
    cout << Val << endl;
    cout << m_Val << endl;
}

// 类型参数列表需要清除
template<>
void CFoo<char>::Print(char Val)
{
    cout << Val << endl;
    cout << m_Val << endl;
}

int main()
{
    CFoo<int> foo(777);
    foo.Print(678);
```

```
    CFoo<float> foo1(666.66f);
    foo1.Print(678);

    CFoo<double> foo2(666.66f);
    foo2.Print(678);

    CFoo<char> foo3('A');
    foo3.Print('B');
    return 0;
}
```

类模板特例

```
template<typename T>
class CFoo
{
public:
    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val)
    {
        cout << Val << endl;
        cout << m_Val << endl;
    }
private:
    T m_Val = 999;
};

// char 类型的特例
template<>
class CFoo<char*>
{
public:
    CFoo(char* Val) : m_Val(Val)
    {
    }
    void Print(char* Val)
    {
        cout << Val << endl;
        cout << m_Val << endl;
    }
private:
    char* m_Val;
};
```

```
/*类型参数列表需要清除*/
template<>
void CFoo<char>::Print(char Val)
{
    cout << Val << endl;
    cout << m_Val << endl;
}

int main()
{
    CFoo<int> foo(777);
    foo.Print(678);

    CFoo<float> foo1(666.66f);
    foo1.Print(678);

    CFoo<char*> foo3((char*)"Hello");
    foo3.Print((char*)"World");
    return 0;
}
```

继承类模板

```
template<typename T>
class CFoo
{
public:
    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val)
    {
        cout << Val << endl;
        cout << m_Val << endl;
    }
private:
    T m_Val;
};

// 此时该子类也是一个模板

// 子类继承一个模板，其必须是一个模板
template<typename T>
class CChild :public CFoo<T>
{
```

```
public:
    CChild(T Val): CFoo<T>(Val)
    {

    }
};

int main()
{
    CChild<int> child(999);
    child.Print(666);
    return 0;
}
```

继承指定类型的类模板

```
template<typename T>
class CFoo
{
public:
    CFoo(T Val) : m_Val(Val)
    {
    }
    void Print(T Val)
    {
        cout << Val << endl;
        cout << m_Val << endl;
    }
private:
    T m_Val;
};

class CChild :public CFoo<float>
{
public:
    CChild(float Val) : CFoo<float>(Val)
    {
    }
};

int main()
{
    CChild child(999.99f);
    child.Print(666.66f);
```

```
    return 0;
}
```