OpenFOAM 中的 tmp 类

2016-10-13 · OpenFOAM · 1516 words · 4 mins read · 12865 times read

tmp 类是 OpenFOAM 中用来封装对象的一个类，这里将介绍 tmp 的机制及用法。

基础知识

在介绍 tmp 类之前我们有必要了解一些 C++ 的机制。在 C++ 中，当一个函数的返回值为对象时，一般情况下将执行以下过程：

- 调用该对象的拷贝构造函数（copy constructor）构造一个临时对象；
- 将新构造的临时对象返回；
- 销毁原对象。

为了说明以上过程，我们来看一段简单的代码：

C++

```
#include <iostream>

using namespace std;

struct C
{
    C() { cout << "constructor" << endl; }
    C(const C& other) { cout << "copy constructor" << endl; }
    C& operator=(const C& other) { cout << "assignment operator" << endl; }
};

C foo()
{
    C retObj;
    return retObj;
}

int main(int argc, char *argv[])
{
    C obj = foo();
    return 0;
}
```

用 g++ 编译，并关闭返回值优化，输出如下：

```
$ g++ -fno-elide-constructors main.cpp
$ ./a.out
constructor
copy constructor
copy constructor
```

上述代码调用了两次拷贝构造函数：第一次在函数 `foo` 返回时调用，用 return 返回的对象 `retObj` 拷贝构造了一个匿名的临时对象，并销毁 `retObj` ，函数返回的其实是这个匿名临时对象；第二次在函数返回后的 `main` 函数中调用，用临时对象拷贝构造 `obj` 。

这样做的目的是出于对内存管理安全性的考虑。然而当一个对象非常大时，在函数中返回这个对象将消耗大量的内存。如何才能避免大量内存开销呢?

tmp 类的机制

tmp 类实际上是智能指针，其功能类似 C++11 中的 `shared_ptr` （最新的 OpenFOAM-dev 版本中，tmp 类已经限定引用计数不能大于2）。tmp 类将大型对象封装起来，tmp 类型的对象本身只保存指向大型对象的指针或引用，因此占用内存非常小。在对 tmp 类型的对象执行拷贝构造函数时基本可以忽略其消耗的内存。

下面以 OpenFOAM-3.0.0 为例分析其源码实现，相关代码：

```
src/OpenFOAM/memory/tmp/tmp.H
src/OpenFOAM/memory/tmp/tmpI.H
```

成员变量

C++

```
template<class T>
class tmp
{
    // Private data

        //- Flag for whether object is a temporary or a constant object
        bool isTmp_;

        //- Pointer to temporary object
        mutable T* ptr_;

        //- Const reference to constant object
        const T& ref_;
    ...
```

和 autoPtr 类相比，tmp 类的成员变量多出了 `isTmp_` 和 `ref_` 。

构造函数

tmp 类可以从指针构造，也可以从常引用构造，对应以下两个构造函数：

C++

```
template<class T>
inline Foam::tmp<T>::tmp(T* tPtr)
:
    isTmp_(true),
    ptr_(tPtr),
    ref_(*tPtr)
{}


template<class T>
inline Foam::tmp<T>::tmp(const T& tRef)
:
    isTmp_(false),
    ptr_(0),
    ref_(tRef)
{}
```

若从指针构造，则说明 tmp 类管理的对象是通过 `new` 申请得到的，存储在堆内存（heap）上，指针 `ptr_` 和引用 `ref_` 同时指向被管理对象；若从常引用构造，则说明 tmp 类管理的对象不是通过 `new` 申请得到

的，存储在栈内存（stack）上，指针 `ptr_` 为空，引用 `ref_` 指向被管理对象。OpenFOAM 中用的比较多的是前一种。

拷贝构造函数

C++

```
template<class T>
inline Foam::tmp<T>::tmp(const tmp<T>& t)
:
    isTmp_(t.isTmp_),
    ptr_(t.ptr_),
    ref_(t.ref_)
{
    if (isTmp_)
    {
        if (ptr_)
        {
            ptr_->operator++();
        }
        else
        {
            FatalErrorIn("Foam::tmp<T>::tmp(const tmp<T>&)")
                << "attempted copy of a deallocated temporary"
            << " of type " << typeid(T).name()
            << abort(FatalError);
        }
    }
}
```

这里需要注意的是只有派生自 `refCount` 的类才能被 `tmp<>` 封装。拷贝构造函数中调用了 `ptr_->operator++()` ，实际上调用的是 `refCount::operator++()` ， 这个函数只是在引用计数 `count_` 上加一，其代码如下：

C++

```
//- Increment the reference count
void operator++()
{
    count_++;
}
```

`refCount` 还有个函数用来判断对象是否可被销毁。当引用计数为零时，对象才能被销毁：

C++

```
//- Return true if the reference count is zero
bool okToDelete() const
{
    return !count_;
}
```

这个函数在 `tmp` 的析构函数中被调用，如果引用计数为零才销毁被封装的对象，否则只是执行 `ptr_->operator--()` ，将引用计数减一：

C++

```
template<class T>
inline Foam::tmp<T>::~tmp()
{
    if (isTmp_ && ptr_)
    {
        if (ptr_->okToDelete())
        {
            delete ptr_;
            ptr_ = 0;
```

```
        }
        else
        {
            ptr_->operator--();
        }
    }
}
```

操作符重载

括号操作符 operator()

括号操作符返回被管理对象本身，有 const 和非 const 两个版本，代码如下：

C++

```
template<class T>
inline T& Foam::tmp<T>::operator()()
{
    if (isTmp_)
    {
        if (!ptr_)
        {
            FatalErrorIn("T& Foam::tmp<T>::operator()()")
                << "temporary of type " << typeid(T).name() << " deallocated"
                << abort(FatalError);
        }

        return *ptr_;
    }
    else
    {
        // Note: const is cast away!
        // Perhaps there should be two refs, one for const and one for non const
        // and if the ref is actually const then you cannot return it here.
        //
        // Another possibility would be to store a const ref and a flag to say
        // whether the tmp was constructed with a const or a non-const argument.
        //
        // eg, enum refType { POINTER = 0, REF = 1, CONSTREF = 2 };
        return const_cast<T&>(ref_);
    }
}


template<class T>
inline const T& Foam::tmp<T>::operator()() const
{
    if (isTmp_)
    {
        if (!ptr_)
        {
            FatalErrorIn("const T& Foam::tmp<T>::operator()() const")
                << "temporary of type " << typeid(T).name() << " deallocated"
                << abort(FatalError);
        }

        return *ptr_;
    }
    else
    {
        return ref_;
    }
}
```

tmp 类的使用

示例代码

C++

```
// 用 tmp 封装大型类
tmp<volScalarField> tvsf = someFunction();

// 用 operator() 获得实际引用，并对其进行操作；
volScalarField& vsf = tvsf();

// 对 vsf 进行数据操作
......

// 销毁对象
tvsf.clear();
```

Author : wwzhao

LastMod : 2016-10-13

#OpenFOAM #tmp #smart pointers

‹ C++11 中的智能指针