类作用域

作用域可以分为**类作用域**、**类名的作用域**以及**对象的作用域**几部分内容。在类中定义的成员变量和成员函数的作用域是整个类，这些名称只有在类中（包含类的定义部分和类外函数实现部分）是可见的，在类外是不可见的，因此，可以在不同类中使用相同的成员名。另外，类作用域意味着不能从外部直接访问类的任何成员，即使该成员的访问权限是public，也要通过对象名来调用，对于static成员函数，要指定类名来调用。

如果发生"屏蔽"现象，类成员的可见域将小于作用域，但此时可借助this指针或"类名::"形式指明所访问的是类成员，这有些类似于使用::访问全局变量。例如:

```
#include <iostream>

using std::cout;
using std::endl;

int num = 1;

namespace wd
{

int num = 20;

class Example
{
public:
    void print(int num) const
    {
        cout << "形参num = " << num << endl;
        cout << "数据成员num = " << this->num << endl;
        cout << "数据成员num = " << Example::num << endl;
        cout << "命名空间中num = " << wd::num << endl;
        cout << "全局变量num = " << ::num << endl;
    }
private:
    int num;
};
}//end of namespace wd
```

和函数一样，类的定义没有生存期的概念，但类定义有作用域和可见域。使用类名创建对象时，首要的前提是类名可见，类名是否可见取决于类定义的可见域，该可见域同样包含在其作用域中，类本身可被定义在3种作用域内，这也是类定义的作用域。

1、全局作用域

在函数和其他类定义的外部定义的类称为全局类，绝大多数的`C++`类是定义在该作用域中，我们在前面定义的所有类都是在全局作用域中，全局类具有全局作用域。

2、类作用域

一个类可以定义在另一类的定义中，这是所谓**嵌套类**或者**内部类**，举例来说，如果类A定义在类B中，如果A的访问权限是public，则A的作用域可认为和B的作用域相同，不同之处在于必须使用B::A的形式访问A的类名。当然，如果A的访问权限是private，则只能在类内使用类名创建该类的对象，无法在外部创建A类的对象。

```
class Line
{
public:
    Line(int x1, int y1, int x2, int y2);
    void printLine() const;

private:
    class Point
    {
    public:
        Point(int x = 0, int y = 0)
        : _x(x), _y(y)
        {

        }

        void print() const;
    private:
        int _x;
        int _y;
    };
private:
    Point _pt1;
    Point _pt2;
};

Line::Line(int x1, int y1, int x2, int y2)
: _pt1(x1, y1)
, _pt2(x2, y2)
{

}

void Line::printLine() const
{
    _pt1.print();
    cout << " ---> ";
    _pt2.print();
    cout << endl;
}

void Line::Point::print() const
{
    cout << "(" << _x
         << "," << _y
         << ")";
}
```

2.1、设计模式之Pimpl

PIMPL（Private Implementation 或Pointer to Implementation）是通过一个私有的成员指针，将指针所指向的类的内部实现数据进行隐藏。PIMPL又称作“编译防火墙”，它的实现中就用到了嵌套类。PIMPL设计模式有如下优点：

1. 提高编译速度；
2. 实现信息隐藏；
3. 减小编译依赖，可以用最小的代价平滑的升级库文件；
4. 接口与实现进行解耦；
5. 移动语义友好。

```
//Line.h
class Line
{
public:
    Line(int,int,int,int);
    ~Line();
    void printLine() const;
private:
    class LineImpl;//类的前向声明
    LineImple *_pimpl;
};

//Line.cc
#include "Line.h"

class Line::LineImpl
{
public:
    LineImpl(int x1, int y1, int x2, int y2);
    void printLineImpl() const;
private:
    class Point
    {
    public:
        Point(int x = 0, int y = 0)
        : _x(x), _y(y)
        {

        }

        void print() const;
    private:
        int _x;
        int _y;
    };

    Point _pt1;
    Point _pt2;
};

Line::LineImpl::LineImpl(int x1, int y1, int x2, int y2)
: _pt1(x1, y1)
, _pt2(x2, y2)
{

```

```
}

void Line::LineImpl::printLineImpl() const
{
    _pt1.print();
    cout << " ---> ";
    _pt2.print();
    cout << endl;
}

Line::Line(int x1, int y1, int x2, int y2)
: _pimpl(new LineImpl(x1, y1, x2, y2))
{

}

Line::~Line()
{
    delete _pimpl;
}

void Line::printLine() const
{
    _pimpl->printLineImpl();
}
```

2.2、单例模式的自动释放

在类和对象那一章，我们讲过单例模式，其中对象是由_pInstance指针来保存的，而在使用单例设计模式的过程中，也难免会遇到内存泄漏的问题。那么是否有一个方法，可以让对象自动释放，而不需要程序员自己手动去释放呢？在学习了嵌套类之后，我们就可以完美的解决这一问题。

在涉及到自动的问题时，我们很自然的可以想到：当对象被销毁时，会自动调用其析构函数。利用这一特性，我们可以解决这一问题。

2.2.0、检测内存泄漏的工具valgrind

```
" 安装方式:
$ sudo apt install valgrind
" 使用方式:
$ valgrind --tool=memcheck --leak-check=full ./test
```

2.2.1、可以使用友元形式进行设计

```
//1、友元实现单例对象的自动释放
class AutoRelease;

class Singleton
{
    friend AutoRelease;
public:
    static Singleton *getInstance()
```

```
    {
        if(nullptr == _pInstance)
        {
            _pInstance = new Singleton();
        }

        return _pInstance;
    }

    static void destroy()
    {
        if(_pInstance)
        {
            delete _pInstance; //1、调用析构函数 2、operator delete
            _pInstance = nullptr;
        }
    }
private:
    Singleton()
    {
        cout << "Singleton()" << endl;
    }

    ~Singleton()
    {
        cout << "~Singleton()" << endl;
    }
private:
    static Singleton *_pInstance;
};

Singleton *Singleton::_pInstance = nullptr;

class AutoRelease
{
public:
    AutoRelease()
    {
        cout << "AutoRelease()" << endl;
    }

    ~AutoRelease()
    {
        cout << "~AutoRelease()" << endl;
        if(Singleton::_pInstance)
        {
            delete Singleton::_pInstance;//1、调用析构函数 2、operator delete
            Singleton::_pInstance = nullptr;
        }
    }
};
```

2.2.2、内部类加静态数据成员形式

```
class Singleton
{
public:
    static Singleton * getInstance()
    {
        if(_pInstance == nullptr)
        {
            _pInstance = new Singleton();
        }
        return _pInstance;
    }

private:
    class AutoRelease
    {
    public:
        AutoRelease()
        {
            cout << "AutoRelease()" << endl;
        }

        ~AutoReleas()
        {
            cout << "~AutoRelease()" << endl;

            if(_pInstance)
            {
                delete _pInstance;
                _pInstance = nullptr;
            }
        }
    };
private:
    Singleton()
    {
         cout << "Singleton()" << endl;
    }

    ~Singleton()
    {
         cout << "~Singleton()" << endl;
    }

private:
    static Singleton *_pInstance;
    static AutoRelease  _auto;
};
```

2.2.3、atexit方式进行

```
class Singleton
{
public:
    static Singleton *getInstance()
    {
        //对于多线程环境，不安全
        if(nullptr == _pInstance)
        {
            _pInstance = new Singleton();
            atexit(destroy);
        }

        return _pInstance;
    }

    static void destroy()
    {
        if(_pInstance)
        {
            delete _pInstance;//1、调用析构函数 2、operator delete
            _pInstance = nullptr;
        }
    }
private:
    Singleton()
    {
        cout << "Singleton()" << endl;
    }

    ~Singleton()
    {
        cout << "~Singleton()" << endl;
    }
private:
    static Singleton *_pInstance;
};

/* Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)*/
Singleton *Singleton::_pInstance = getInstance();//饿汉模式

```

2.2.4、pthread_once形式

```
//4、pthread_once,平台相关性的函数
class Singleton
{
public:
    static Singleton *getInstance()
    {
        pthread_once(&_once, init);

        return _pInstance;

```

```
    }

    static void init()
    {
        _pInstance = new Singleton();
        atexit(destroy);
    }

    static void destroy()
    {
        if(_pInstance)
        {
            delete _pInstance;//1、调用析构函数 2、operator delete
            _pInstance = nullptr;
        }
    }
private:
    Singleton()
    {
        cout << "Singleton()" << endl;
    }

    ~Singleton()
    {
        cout << "~Singleton()" << endl;
    }
private:
    static Singleton *_pInstance;
    static pthread_once_t _once;
};

Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)
/* Singleton *Singleton::_pInstance = getInstance();//饿汉模式 */
pthread_once_t Singleton::_once = PTHREAD_ONCE_INIT;

```

3、块作用域

类的定义在代码块中，这是所谓**局部类**，该类完全被块包含，其作用域仅仅限于定义所在块，不能在块外使用类名声明该类的对象。

```
void test()
{
    class Point
    {
    public:
        Point(int x, int y)
        : _x(x), _y(y)
        {

        }

        void print() const
        {
            cout << "(" << _x
```

```
                << "," << _y
                << ")" << endl;
        }
    private:
        int _x;
        int _y;
    }; //end of class Point

    Point pt(1, 2);
    pt.print();
}
```