

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：

```
1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  int num = 1;
7
8  namespace wd
9  {
10
11  int num = 20;
12
13  class Example
14  {
15  public:
16      void print(int num) const
17      {
18          cout << "形参num = " << num << endl;
19          cout << "数据成员num = " << this->num << endl;
20          cout << "数据成员num = " << Example::num << endl;
21          cout << "命名空间中num = " << wd::num << endl;
22          cout << "全局变量num = " << ::num << endl;
23      }
24  private:
25      int num;
26  };
27 }//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。

```
1  class Line
2  {
3  public:
4      Line(int x1, int y1, int x2, int y2);
5      void printLine() const;
6
7  private:
8      class Point
9      {
10     public:
11         Point(int x = 0, int y = 0)
12             : _x(x), _y(y)
13             {
14
15             }
16
17         void print() const;
18     private:
19         int _x;
20         int _y;
21     };
22 private:
23     Point _pt1;
24     Point _pt2;
25 };
26
27 Line::Line(int x1, int y1, int x2, int y2)
28 : _pt1(x1, y1)
29 , _pt2(x2, y2)
30 {
31
32 }
33
34 void Line::printLine() const
35 {
36     _pt1.print();
37     cout << " ---> ";
38     _pt2.print();
39     cout << endl;
40 }
41
42 void Line::Point::print() const
43 {
44     cout << "(" << _x
45         << "," << _y
46         << ")";
47 }
```

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。

```
1 //Line.h
2 class Line
3 {
4 public:
5     Line(int,int,int,int);
6     ~Line();
7     void printLine() const;
8 private:
9     class LineImpl; //类的前向声明
10    LineImpl *_pimpl;
11 };
12
13 //Line.cc
14 #include "Line.h"
15
16 class Line::LineImpl
17 {
18 public:
19     LineImpl(int x1, int y1, int x2, int y2);
20     void printLineImpl() const;
21 private:
22     class Point
23     {
24     public:
25         Point(int x = 0, int y = 0)
26             : _x(x), _y(y)
27         {
28
29         }
30
31         void print() const;
32     private:
33         int _x;
34         int _y;
35     };
36
37     Point _pt1;
38     Point _pt2;
39 };
40
41 Line::LineImpl::LineImpl(int x1, int y1, int x2, int y2)
42 : _pt1(x1, y1)
43 , _pt2(x2, y2)
44 {
45
```

```

46 }
47
48 void Line::LineImpl::printLineImpl() const
49 {
50     _pt1.print();
51     cout << " ---> ";
52     _pt2.print();
53     cout << endl;
54 }
55
56 Line::Line(int x1, int y1, int x2, int y2)
57 : _pimpl(new LineImpl(x1, y1, x2, y2))
58 {
59
60 }
61
62 Line::~~Line()
63 {
64     delete _pimpl;
65 }
66
67 void Line::printLine() const
68 {
69     _pimpl->printLineImpl();
70 }

```

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

```

1 " 安装方式：
2 $ sudo apt install valgrind
3 " 使用方式：
4 $ valgrind --tool=memcheck --leak-check=full ./test

```

2.2.1、可以使用友元形式进行设计

```

1 //1、友元实现单例对象的自动释放
2 class AutoRelease;
3
4 class Singleton
5 {
6     friend AutoRelease;
7 public:
8     static Singleton *getInstance()

```

```

9      {
10         if(nullptr == _pInstance)
11         {
12             _pInstance = new Singleton();
13         }
14
15         return _pInstance;
16     }
17
18     static void destroy()
19     {
20         if(_pInstance)
21         {
22             delete _pInstance; //1、调用析构函数 2、operator delete
23             _pInstance = nullptr;
24         }
25     }
26 private:
27     Singleton()
28     {
29         cout << "Singleton()" << endl;
30     }
31
32     ~Singleton()
33     {
34         cout << "~Singleton()" << endl;
35     }
36 private:
37     static Singleton *_pInstance;
38 };
39
40 Singleton *Singleton::_pInstance = nullptr;
41
42 class AutoRelease
43 {
44 public:
45     AutoRelease()
46     {
47         cout << "AutoRelease()" << endl;
48     }
49
50     ~AutoRelease()
51     {
52         cout << "~AutoRelease()" << endl;
53         if(Singleton::_pInstance)
54         {
55             delete Singleton::_pInstance; //1、调用析构函数 2、operator delete
56             Singleton::_pInstance = nullptr;
57         }
58     }
59 };

```

2.2.2、内部类加静态数据成员形式

```
1  class Singleton
2  {
3  public:
4      static Singleton * getInstance()
5      {
6          if(_pInstance == nullptr)
7          {
8              _pInstance = new Singleton();
9          }
10         return _pInstance;
11     }
12
13 private:
14     class AutoRelease
15     {
16     public:
17         AutoRelease()
18         {
19             cout << "AutoRelease()" << endl;
20         }
21
22         ~AutoRelease()
23         {
24             cout << "~AutoRelease()" << endl;
25
26             if(_pInstance)
27             {
28                 delete _pInstance;
29                 _pInstance = nullptr;
30             }
31         }
32     };
33 private:
34     Singleton()
35     {
36         cout << "Singleton()" << endl;
37     }
38
39     ~Singleton()
40     {
41         cout << "~Singleton()" << endl;
42     }
43
44 private:
45     static Singleton *_pInstance;
46     static AutoRelease _auto;
47 };
```

2.2.3、atexit方式进行

```
1  class Singleton
2  {
3  public:
4      static Singleton *getInstance()
5      {
6          //对于多线程环境，不安全
7          if(nullptr == _pInstance)
8          {
9              _pInstance = new Singleton();
10             atexit(destroy);
11         }
12
13         return _pInstance;
14     }
15
16     static void destroy()
17     {
18         if(_pInstance)
19         {
20             delete _pInstance; //1、调用析构函数 2、operator delete
21             _pInstance = nullptr;
22         }
23     }
24 private:
25     Singleton()
26     {
27         cout << "Singleton()" << endl;
28     }
29
30     ~Singleton()
31     {
32         cout << "~Singleton()" << endl;
33     }
34 private:
35     static Singleton *_pInstance;
36 };
37
38 /* Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)*/
39 Singleton *Singleton::_pInstance = getInstance(); //饿汉模式
40
```

2.2.4、pthread_once形式

```
1  //4、pthread_once, 平台相关性的函数
2  class Singleton
3  {
4  public:
5      static Singleton *getInstance()
6      {
7          pthread_once(&_once, init);
8
9          return _pInstance;
```

```

10     }
11
12     static void init()
13     {
14         _pInstance = new Singleton();
15         atexit(destroy);
16     }
17
18     static void destroy()
19     {
20         if(_pInstance)
21         {
22             delete _pInstance; //1、调用析构函数 2、operator delete
23             _pInstance = nullptr;
24         }
25     }
26 private:
27     Singleton()
28     {
29         cout << "Singleton()" << endl;
30     }
31
32     ~Singleton()
33     {
34         cout << "~Singleton()" << endl;
35     }
36 private:
37     static Singleton *_pInstance;
38     static pthread_once_t _once;
39 };
40
41 Singleton *Singleton::_pInstance = nullptr; //饱汉模式(懒汉模式)
42 /* Singleton *Singleton::_pInstance = getInstance(); //饿汉模式 */
43 pthread_once_t Singleton::_once = PTHREAD_ONCE_INIT;
44

```

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。

```

1 void test()
2 {
3     class Point
4     {
5     public:
6         Point(int x, int y)
7             : _x(x), _y(y)
8         {
9
10        }
11
12        void print() const
13        {
14            cout << "(" << _x

```



```
15         << "," << _y
16         << ")" << endl;
17     }
18     private:
19         int _x;
20         int _y;
21 }; //end of class Point
22
23 Point pt(1, 2);
24 pt.print();
25 }
```