



C++语言程序设计

第五章 数据的共享与保护

信息工程学院 王红平

Email: cugwhp@qq.com

本章主要内容

- 作用域与可见性
- 对象的生存期
- 静态成员
- 友元
- 共享数据的保护
- 多文件结构和工程
- 编译预处理命令



函数原形的作用域

作用域与可见性

- 函数原型中的参数，其作用域始于"(", 结束于")"。
- 例如，设有下列原型声明：

```
double Area(double radius);
```

radius 的作用域仅在于此，不能用于程序正文其它地方，因而可有可无。

块作用域

作用域与可见性

- 在块中声明的标识符，其作用域自声明处起，限于块中，例如：

```
void fun(int a)
{
    int b(a);
    cin>>b;
    if (b>0)
    {
        int c;
        .....
    }
}
```

b的作用域

c的作用域

类作用域

作用域与可见性

- 类的内部
 - 如果在X的成员函数中没有声明同名的局部作用域标识符，那么在该函数内可以访问成员M。
- 类的外部
 - 通过表达式x.M或者X::M访问。
 - 通过表达式ptr->M



文件作用域

作用域与可见性

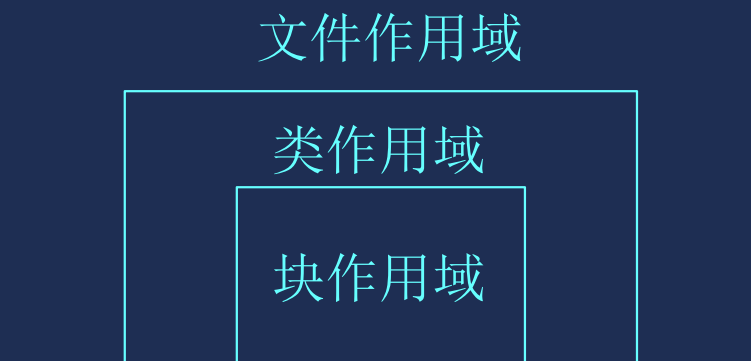
- 不在前述各个作用域中出现的声明，具有文件作用域，这样声明的标识符的作用域开始于声明点，结束于文件尾。



可见性

作用域与可见性

- 可见性表示从内层作用域向外层作用域“看”时能看见什么。
- 如果标识在某处可见，则就可以在该处引用此标识符。



可见性

作用域与可见性

- 标识符应声明在先，引用在后。
- 如果某个标识符在外层中声明，且在内层中没有同一标识符的声明，则该标识符在内层可见。
- 对于两个嵌套的作用域，如果在内层作用域内声明了与外层作用域中同名的标识符，则外层作用域的标识符在内层不可见。



例 5.1

作用域与可见性

```
#include<iostream>
using namespace std;
int i; //文件作用域
int main()
{ i=5;
  { int i; //块作用域
    i=7;
    cout<<"i="<<i<<endl; //输出7
  }
  cout<<"i="<<i; //输出5
  return 0;
}
```



对象的生存期

对象从产生到结束的这段时间就是它的生存期。在对象生存期内，对象将保持它的值，直到被更新为止。



静态生存期

对象的生存期

- 这种生存期与程序的运行期相同。
- 在文件作用域中声明的对象具有这种生存期。
- 在函数内部声明静态生存期对象，要冠以关键字 **static**。

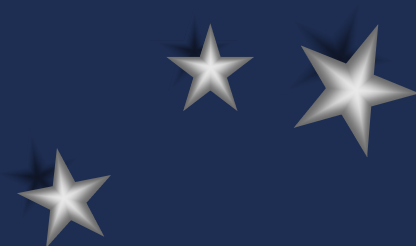


例

对象的生存期

```
#include<iostream>
using namespace std;
int i=5; //文件作用域
int main()
{
    cout<<"i="<<i<<endl;
    return 0;
}
```

i具有静态生存期



动态生存期

对象的生存期

- 块作用域中声明的，没有用**static**修饰的对象是动态生存期的对象（习惯称局部生存期对象）。
- 开始于程序执行到声明点时，结束于命名该标识符的作用域结束处。



例

对象的生存期

```
#include<iostream>
using namespace std;
void fun();
void main()
{ fun();
  fun();
}
void fun()
{ static int a=1;
  int i=5;
  a++;
  i++;
  cout<<"i="<<i<<",a="<<a<<endl;
}
```

运行结果:

i=6, a=2

i=6, a=3

i是动态生存期

a是静态生存期

例5-3 具有静态、动态生存期对象的时钟程序

对
象
的
生
存
期

```
#include<iostream>
using namespace std;
class Clock //时钟类声明
{public:    //外部接口
    Clock();
    void SetTime(int NewH, int NewM, int NewS);
    //三个形参均具有函数原型作用域
    void ShowTime();
    ~Clock(){}
private:   //私有数据成员
    int Hour,Minute,Second;
};
```



//时钟类成员函数实现

Clock::Clock() //构造函数

```
{ Hour=0;
  Minute=0;
  Second=0;
}
```

void Clock::SetTime(int NewH, int NewM, int NewS)

```
{ Hour=NewH;
  Minute=NewM;
  Second=NewS;
}
```

void Clock::ShowTime()

```
{ cout<<Hour<<":"<<Minute<<":"<<Second<<endl;
}
```



```
Clock globClock; //声明对象globClock,
                  //具有静态生存期, 文件作用域

void main() //主函数
{
    cout<<"First time output:"<<endl;
    //引用具有文件作用域的对象:
    globClock.ShowTime(); //对象的成员函数具有类作用域
    globClock.SetTime(8,30,30);
    Clock myClock(globClock);
        //声明具有块作用域的对象myClock
    cout<<"Second time output:"<<endl;
    myClock.ShowTime(); //引用具有块作用域的对象
}
```

程序的运行结果为:

First time output:

0:0:0

Second time output:

8:30:30

静态成员

静态成员

- 静态数据成员
 - 用关键字static声明
 - 该类的所有对象维护该成员的同一个拷贝
 - 必须在类外定义和初始化，用(::)来指明所属的类。
- 静态成员函数
 - 类外代码可以使用类名和作用域操作符来调用静态成员函数。
 - 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。

例5-4 具有静态数据成员的 Point类

静态成员

```
#include <iostream>
using namespace std;
class Point
{public:
    Point(int xx=0, int yy=0) {X=xx; Y=yy; countP++; }
    Point(Point &p);
    int GetX() {return X;}
    int GetY() {return Y;}
    void GetC() {cout<<" Object id="<<countP<<endl;}
private:
    int X,Y;
    static int countP;
};
```



```
Point::Point(Point &p)
{ X=p.X;
  Y=p.Y;
  countP++;
}
```

```
int Point::countP=0;
void main()
{ Point A(4,5);
  cout<<"Point A,"<<A.GetX()<<","<<A.GetY();
  A.GetC();
  Point B(A);
  cout<<"Point B,"<<B.GetX()<<","<<B.GetY();
  B.GetC();
}
```


静态成员函数举例

静态成员

```
#include<iostream>
using namespace std;
class Application
{ public:
    static void f();
    static void g();
private:
    static int global;
};
int Application::global
    =0;
```

```
void Application::f()
{ global=5;}
void Application::g()
{ cout<<global<<endl;}

int main()
{
    Application::f();
    Application::g();
    return 0;
}
```



静态成员函数举例

静态成员

```
class A
{
    public:
        static void f(A a);
    private:
        int x;
};

void A::f(A a)
{
    cout<<x; //对x的引用是错误的
    cout<<a.x; //正确
}
```



具有静态数据、函数成员的 Point类

静
态
成
员

```
#include <iostream>
using namespace std;
class Point //Point类声明
{public:    //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;countP++;}
    Point(Point &p); //拷贝构造函数
    int GetX() {return X;}
    int GetY() {return Y;}
    static void GetC()
        {cout<<" Object id="<<countP<<endl;}
private:  //私有数据成员
    int X,Y;
    static int countP;
}
```




```
Point::Point(Point &p)
{ X=p.X;
  Y=p.Y;
  countP++;
}
int Point::countP=0;
void main() //主函数实现
{ Point A(4,5);    //声明对象A
  cout<<"Point A,"<<A.GetX()<<","<<A.GetY();
  A.GetC(); //输出对象号，对象名引用
  Point B(A);      //声明对象B
  cout<<"Point B,"<<B.GetX()<<","<<B.GetY();
  Point::GetC();   //输出对象号，类名引用
}
```

友元

友元

- 友元是C++提供的一种破坏数据封装和数据隐藏的机制。
- 通过将一个模块声明为另一个模块的友元，一个模块能够引用到另一个模块中本是被隐藏的信息。
- 可以使用友元函数和友元类。
- 为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

友元函数

友元

- 友元函数是在类声明中由关键字**friend**修饰说明的非成员函数，在它的函数体中能够通过对象名访问 **private** 和 **protected**成员
- 作用：增加灵活性，使程序员可以在封装和快速性方面做合理选择。
- 访问对象中的成员必须通过对象名。



例5-6 使用友元函数计算两点距离

友
元

```
#include <iostream>
#include <cmath>
using namespace std;
class Point //Point类声明
{ public:    //外部接口
    Point(int xx=0, int yy=0) {X=xx;Y=yy;}
    int GetX() {return X;}
    int GetY() {return Y;}
    friend double Distance(Point &a, Point &b);
private:   //私有数据成员
    int X,Y;
};
```



```
double Distance( Point& a, Point& b)
{
    double dx=a.X-b.X;
    double dy=a.Y-b.Y;
    return sqrt(dx*dx+dy*dy);
}
int main()
{ Point p1(3.0, 5.0), p2(4.0, 6.0);
  double d=Distance(p1, p2);
  cout<<"The distance is "<<d<<endl;
  return 0;
}
```

友元类

友元

- 若一个类为另一个类的友元，则此类的所有成员都能访问对方类的私有成员。
- 声明语法：将友元类名在另一个类中使用 **friend** 修饰说明。



友元类举例

友
元

```
class A
{
    friend class B;
public:
    void Display()
    {cout<<x<<endl;}
private:
    int x;
}
class B
{
    public:
        void Set(int i);
        void Display();
    private:
        A a;
};
```



```
void B::Set(int i)
{
    a.x=i;
}
void B::Display()
{
    a.Display();
}
```


友元关系是单向的

如果声明**B**类是**A**类的友元，**B**类的成员函数就可以访问**A**类的私有和保护数据，但**A**类的成员函数却不能访问**B**类的私有、保护数据。



常类型

共享数据的保护

常类型的对象必须进行初始化，而且不能被更新。

- 常引用：被引用的对象不能被更新。
`const` 类型说明符 &引用名
- 常对象：必须进行初始化，不能被更新。
`const` 类名 对象名
- 常数组：数组元素不能被更新。
`const` 类型说明符 数组名[大小]...
- 常指针：指向常量的指针。

例5-7常引用做形参

共享
数据
的
保
护

```
#include<iostream>
using namespace std;
void display(const double& r);
int main()
{ double d(9.5);
  display(d);
  return 0;
}

void display(const double& r)
//常引用做形参，在函数中不能更新 r所引用的对象。
{ cout<<r<<endl; }
```

常对象举例

共享
数据
的
保护

```
class A
{
    public:
        A(int i,int j) {x=i; y=j;}
        ...
    private:
        int x,y;
};
```

const A a(3,4); //a是常对象，不能被更新★ ★



用const修饰的对象成员

共享数据的保护

- 常成员函数
 - 使用const关键字说明的函数。
 - 常成员函数不更新对象的数据成员。
 - 常成员函数说明格式：
类型说明符 函数名（参数表） const;
这里，const是函数类型的一个组成部分，因此在实现部分也要带const关键字。
 - const关键字可以被用于参与对重载函数的区分
- 通过常对象只能调用它的常成员函数。
- 常数据成员
 - 使用const说明的数据成员。



例5-8 常成员函数举例

共享数据的保护

```
#include<iostream>
using namespace std;
class R
{   public:
    R(int r1, int r2){R1=r1;R2=r2;}
    void print();
    void print() const;
    private:
        int R1,R2;
};
```



```
void R::print()
{   cout<<R1<<":"<<R2<<endl;
}
void R::print() const
{   cout<<R1<<";"<<R2<<endl;
}
void main()
{   R a(5,4);
    a.print(); //调用void print()
    const R b(20,52);
    b.print(); //调用void print() const
}
```

例5-9 常数据成员举例

共享
数据
的
保
护

```
#include<iostream>
using namespace std;
class A
{public:
    A(int i);
    void print();
    const int& r;
private:
    const int a;
    static const int b; //静态常数据成员
};
```




```
const int A::b=10;
A::A(int i):a(i),r(a) {}
void A::print()
{   cout<<a<<":"<<b<<":"<<r<<endl; }
void main()
{ /*建立对象a1和a2，并以100和0作为初值，分别调用构造函数，通过构造函数的初始化列表给对象的常数据成员赋初值*/
    A a1(100),a2(0);
    a1.print();
    a2.print();
}
```

编译预处理命令

- **#include** 包含指令
 - 将一个源文件嵌入到当前源文件中该点处。
 - **#include**<文件名>
 - 按标准方式搜索，文件位于C++系统目录的include子目录下
 - **#include**"文件名"
 - 首先在当前目录中搜索，若没有，再按标准方式搜索。
- **#define** 宏定义指令
 - 定义符号常量，很多情况下已被**const**定义语句取代。
 - 定义带参数宏，已被内联函数取代。
- **#undef**
 - 删除由**#define**定义的宏，使之不再起作用。

条件编译指令 #if 和 #endif

编译
预处理
命令

#if 常量表达式

//当“常量表达式”非零时编译

程序正文

#endif

.....



条件编译指令——#else

编译
预处理
命令

#if 常量表达式

//当“常量表达式”非零时编译

程序正文1

#else

//当“常量表达式”为零时编译

程序正文2

#endif



条件编译指令 #elif

编译
预处理
命令

#if 常量表达式1

程序正文1 //当 “ 常量表达式1”非零时编译

#elif 常量表达式2

程序正文2 //当 “ 常量表达式2”非零时编译

#else

程序正文3 //其它情况下编译

#endif



条件编译指令

编译
预处理
命令

#ifdef 标识符
程序段1

#else
程序段2

#endif

如果“标识符”经**#defined**定义过，且未经**undef**删除，则编译程序段1，否则编译程序段2。



条件编译指令

编译
译
预
处
理
命
令

#ifndef 标识符

程序段1

#else

程序段2

#endif

如果“标识符”未被定义过，则编译程序段1，否则编译程序段2。



多文件结构（例5-10）

- 一个源程序可以划分为多个源文件：
 - 类声明文件（.h文件）
 - 类实现文件（.cpp文件）
 - 类的使用文件（`main()`所在的.cpp文件）
- 利用工程来组合各个文件。



不使用条件编译的头文件

多文件结构

//main.cpp

#include "file1.h"

#include "file2.h"

void main()

{

...

}

//file1.h

#include "head.h"

...

//file2.h

#include "head.h"

...

//head.h

...

class Point

{

...

}

...



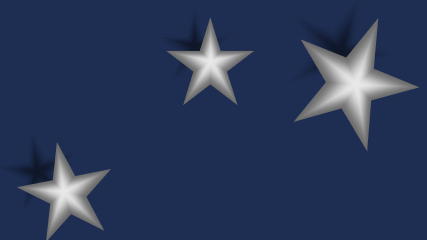
使用条件编译的头文件

多
文
件
结
构

```
//head.h
#ifndef HEAD_H
#define HEAD_H
    ...
    class Point
    {
        ...
    }
    ...
#endif
```



The End



同一作用域中的同名标识符

作用域与可见性

- 在同一作用域内的对象名、函数名、枚举常量名会隐藏同名的类名或枚举类型名。
- 重载的函数可以有相同的函数名。



例5-2 变量的生存期与可见性

对象的生存期

```
#include<iostream>
using namespace std;
int i=1; // i 为全局变量，具有静态生存期。
void main(void)
{ static int a; // 静态局部变量，有全局寿命，局部可见。
  int b=-10; // b, c为局部变量，具有动态生存期。
  int c=0;
  void other(void);
  cout<<"---MAIN---\n";
  cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
  c=c+8; other();
  cout<<"---MAIN---\n";
  cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
  i=i+10; other();
}
```

```
void other(void)
```

```
{
```

```
    static int a=2;
```

```
    static int b;
```

// a,b为静态局部变量，具有全局寿命，局部可见。

//只第一次进入函数时被初始化。

```
    int c=10; // C为局部变量，具有动态生存期，
```

//每次进入函数时都初始化。

```
    a=a+2; i=i+32; c=c+5;
```

```
    cout<<"---OTHER---\n";
```

```
    cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
```

```
    b=a;
```

```
}
```



运行结果:

---MAIN---

i: 1 a: 0 b: -10 c: 0

---OTHER---

i: 33 a: 4 b: 0 c: 15

---MAIN---

i: 33 a: 0 b: -10 c: 8

---OTHER---

i: 75 a: 6 b: 4 c: 15



数据与函数

数据与函数

- 数据存储在全局对象中，通过参数传递实现共享——函数间的参数传递。
- 数据存储在全局对象中。
- 将数据和使用数据的函数封装在类中。



使用全局对象

数
据
与
函
数

```
#include<iostream>
using namespace std;
int global;
void f()
{   global=5; }
void g()
{   cout<<global<<endl; }
int main()
{   f();
    g(); //输出“5”
    return 0;
}
```

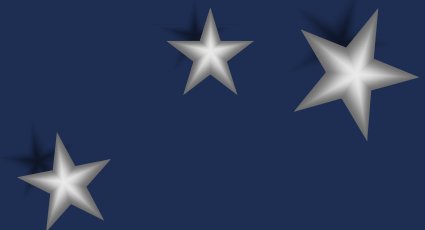


将函数与数据封装

数据与函数

```
#include<iostream>
using namespace std;
class Application
{ public:
    void f(); void g();
    private:
        int global;
};
void Application::f()
{ global=5;}
void Application::g()
{ cout<<global<<endl;}
```

```
int main()
{
    Application MyApp;
    MyApp.f();
    MyApp.g();
    return 0;
}
```



上机实习<3>

- 书本上的习题（必做）
 - 4-9/4-20
 - 扩展4-9，为Point对象增加记录Point对象数目的静态变量；增加能访问对象数目的静态成员函数。
 - 5-10/5-11
- 选做题
 - 定义个Student类，成员变量包含学号和姓名，要求每创建一个Student对象，Student类自动创建一个后续的新学号11316101，并能返回Student对象的数量。