C++语言程序设计

第五章 数据的共享与保护

本章主要内容

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

标识符的作用域

作用域与可见性

- 标识符在程序正文中有效的区域
  - 函数原型作用域
  - 局部作用域
  - 类作用域
  - 命名空间作用域

函数原型的作用域

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他地方，因而可有可无。

局部作用域

作用域与可见性

- 函数的形参，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;
        ......
    }
}
```

a的作用域

b的作用域

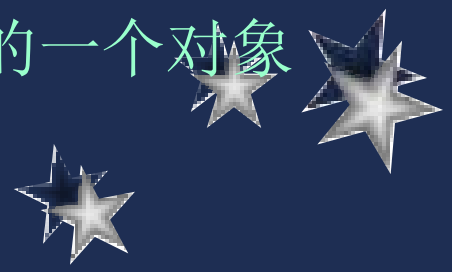
c的作用域

块：一对大括号括起来的一段程序

类作用域

作用域与可见性

- 类作用域作用于特定的成员名。
- 类X的成员m具有类作用域，对m的访问方式如下：
  - 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。
  - 通过表达式x.m或者X::m访问。 X::m的形式访问类的静态成员。
  - 通过表达式ptr->m，ptr为指向类X的一个对象的指针



命名空间

作用域与可见性

- 命名空间可以解决类名、函数名等的命名冲突
- 命名空间的声明

```
namespace 命名空间名 {
    各种声明（函数声明、类声明、……）
}
```

- 例

```
namespace SomeNs {
    class SomeClass { ... };
}
```

命名空间

作用域与可见性

- 命名空间允许嵌套

```
namespace OuterNs {
  namespace InnerNs {
    class SomeClass { ... };
  }
}
```

- 特殊的命名空间
  - 全局命名空间：默认的命名空间
    - 在显式声明的命名空间外声明的标识符都在全局命名空间中
  - 匿名命名空间：对每个源文件是唯一的

作用域与可见性

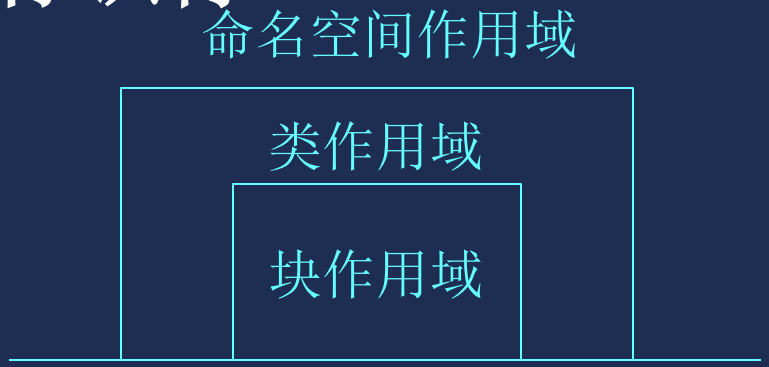
命名空间作用域

- 一个命名空间确定了一个命名空间作用域
- 引用其它命名空间作用域中的标识符
  - 命名空间名::标识符名
  - 例：声明一个SomeClass型的对象
    - SomeNs::SomeClass obj1;
    - OuterNs::InnerNs::SomeClass obj2;
- 将其它命名空间作用域的标识符暴露于当前作用域
  - 对指定标识符
    - using 命名空间名::标识符名;
  - 对所有标识符
    - using namespace 命名空间名;

作用域与可见性

可见性

- 可见性是从对标识符的引用的角度来谈的概念
- 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。

作用域与可见性

同一作用域中的同名标识符

- 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 5.1

作用域与可见性

```
#include <iostream>
using namespace std;
int i;                  //在全局命名空间中的全局变量
namespace Ns {
   int j;               //在Ns命名空间中的全局变量
}
int main() {
   i = 5;                           //为全局变量i赋值
   Ns::j = 6;                       //为全局变量j赋值
   {                                //子块1
     using namespace Ns;  //当前块中可以直接引用Ns中的标识符
     int i;             //局部变量，局部作用域
     i = 7;
     cout << "i = " << i << endl;//输出7
     cout << "j = " << j << endl;//输出6
   }
   cout << "i = " << i << endl;  //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。
- 在命名空间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** 。
- 定义时未指定初值的基本类型静态生存期变量，会被赋0值初始化

例

对象的生存期

```
#include<iostream>
using namespace std;
int i = 5;    //命名空间作用域
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修是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();
int 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5-2 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 other() {
  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 += 2; i += 32; c += 5;
  cout<<"---OTHER---\n";
  cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c: "<<c<<endl;
  b = a;
}
```

```
int main() {
  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;
   cout << "---MAIN---\n";
  cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c:
   "<<c<<endl;
  c += 8; other();
  cout<<"---MAIN---\n";
  cout<<" i: "<<i<<" a: "<<a<<" b: "<<b<<" c:
   "<<c<<endl;
  i += 10; other();
   return 0;
}
```

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
```

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 {      //时钟类定义
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();
private:     //私有数据成员
   int hour, minute, second;
};
```

```
Clock::Clock() : hour(0), minute(0), second(0) { }
      //构造函数

void Clock::setTime(int newH, int newM, int newS) {
//三个形参均具有局部作用域
  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，命名空间作用域
int main() { //主函数
   cout << "First time output:" << endl;
   //引用具有命名空间作用域的对象：
   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
   return 0;
}
```

程序的运行结果为：

```
First time output:
0:0:0
Second time output:
8:30:30
```

数据与函数

- 数据存储在局部对象中，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


```
#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;
}
```

静态成员

- 实例属性

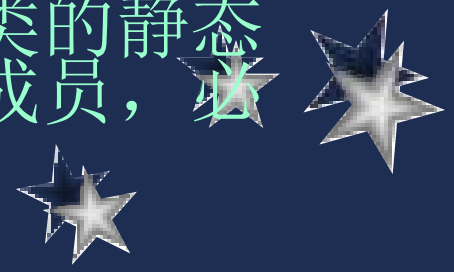
静态成员

- 类属性：用static声明为静态成员
  - 实现同一个类的不同对象之间的数据和函数共享

静态成员

静态成员

- 静态数据成员
  - 该类的所有对象维护该成员的同一个拷贝
  - 必须在类内声明并在类外定义和初始化，用(::)来指明所属的类。
  - 具有静态生存期
- 静态成员函数
  - 类外代码可以使用类名(或对象名)和作用域操作符来调用静态成员函数。
  - 静态成员函数可以直接访问该类的静态数据和函数成员。访问非静态成员，必须通过对象名



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)
   { count++; }
   Point(Point &p);
   int getX() { return x; }
   int getY() { return y; }
   void showCount() {
     cout << " Object count= " << count << endl;
   }
private:
   int x,y;
   static int count;
};
```

```
Point::Point(Point &p) {
   x = p.x;
   x = p.y;
   count++;
}

int Point::count=0;
int main() {
   Point a(4,5);
   cout<<"Point A:"<<a.getX()<<","<<a.getY();
   a.showCount();
   Point b(a);
   cout<<"Point B:"<<b.getX()<<","<<b.getY();
   b.showCount();
   return 0;
}
```

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

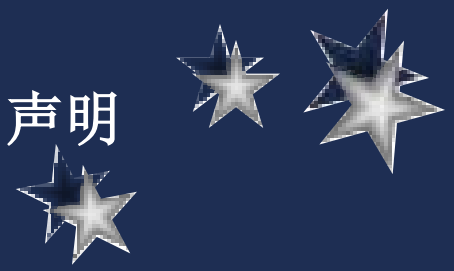
静态成员

```
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类定义
public:       //外部接口
   Point(int x = 0, int y = 0) : x(x), y(y) { count++; }
   Point(Point &p);
   ~Point() {  count--; }
   int getX() { return x; }
   int getY() { return y; }
   static void showCount() {      //静态函数成员
     cout << "  Object count = " << count << endl;
   }
private:      //私有数据成员
   int x, y;
   static int count;          //静态数据成员声明
};
```

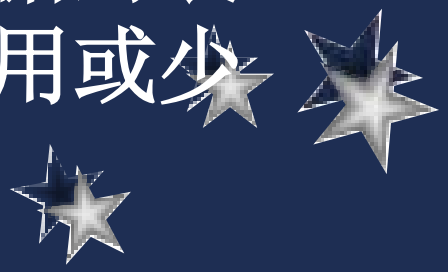


```
Point::Point(Point &p) {
   x = p.x;
   y = p.y;
   count++;
}
int Point::count=0;
int main() { //主函数实现
   Point a(4,5);   //声明对象A
   cout<<"Point A,"<<a.getX()<<","<<a.getY();
   Point::showCount();   //输出对象个数
   Point b(a);      //声明对象B
   cout<<"Point B,"<<b.GetX()<<","<<b.GetY();
   Point:: showCount(); //输出对象个数
   return 0;
}
```

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>
class Point {      //Point类声明
public:      //外部接口
   Point(int x=0, int y=0) : x(x), y(y) { }
   int getX() { return x; }
   int getY() { return y; }
   friend float dist(Point &a, Point &b);
private:     //私有数据成员
   int x, y;
};
```

```
float dist( Point& a, Point& b) {
  double x = a.x - b.x;
  double y = a.y - b.y;
  return static_cast<float>(sqrt(x * x + y
  * y));
}
int main() {
  Point p1(1, 1), p2(4, 5);
  cout <<"The distance is: ";
  cout << dist(p1, p2) << 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和保护成员。
- 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

友元

- 友元关系是不能传递的
  - B类是A类的友元，C类是B类友元，如无声明，C类和A类之间没有任何友元关系
- 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。
- 友元关系是不被继承的
  - 如果类B是类A的友元，类B的派生类BB并不会自动成为类A的友元

常类型

共享数据的保护

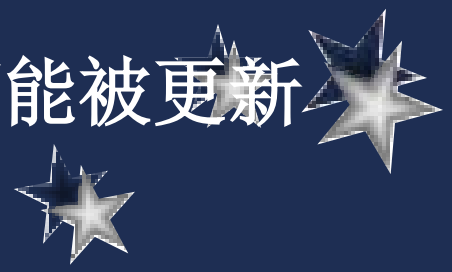
常类型的对象必须进行初始化，而且不能被更新。

- 常对象：必须进行初始化,不能被更新。
  const 类名 对象名
- 常引用：被引用的对象不能被更新。
  const 类型说明符 &引用名
- *常数组：数组元素不能被更新(下一章介绍)。*
  类型说明符 const 数组名[大小]...
- *常指针：指向常量的指针(下一章介绍)。*

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;
};
A const 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。
  - 构造函数初始化时只能通过初始化列表

例5-7 常成员函数举例

共享数据的保护

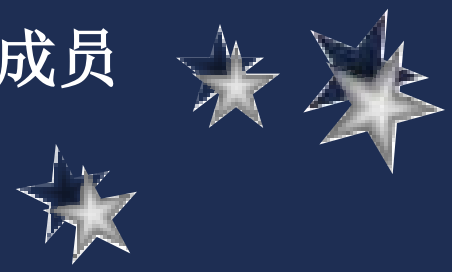
```
#include<iostream>
using namespace std;
class R {
public:
   R(int r1, int r2) : r1(r1), r2(r2) { }
   void print();
   void print() const;
private:
   int r1, r2;
};
```

```
void R::print() {
  cout << r1 << ":" << r2 << endl;
}
void R::print() const {
  cout << r1 << ";" << r2 << endl;
}
int main() {
  R a(5,4);
  a.print(); //调用void print()
  const R b(20,52);
  b.print(); //调用void print() const
  return 0;
}
```

例5-8 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();
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) { }
void A::print() {
  cout << a << ":" << b <<endl;
}
int main() {
/*建立对象a和b，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();
  return 0;
}
```

常引用

共享数据的保护

- 常引用：被引用的对象不能被更新
- 非const引用只能绑定到普通的对象，不能绑定要常对象
- 常引用可以绑定到普通对象和常对象

- 在函数中无须改变值的参数，不宜使用普通引用方式传递
  - 传值
  - 传递常引用：常用在大对象和复制构造函数

例5-9常引用作形参

共享数据的保护

```
#include <iostream>
#include <cmath>
using namespace std;
class Point {    //Point类定义
public:    //外部接口
  Point(int x = 0, int y = 0)
    : x(x), y(y) { }
  int getX() { return x; }
  int getY() { return y; }
  friend float dist(const Point &p1, const
  Point &p2);
private:   //私有数据成员
  int x, y;
};
```

例5-9常引用作形参

共享数据的保护

```
float dist(const Point &p1, const Point &p2) {
   double x = p1.x - p2.x;
   double y = p1.y - p2.y;
   return static_cast<float>(sqrt(x * x + y * y));
}

int main() { //主函数
   const Point myp1(1, 1), myp2(4, 5);
   cout << "The distance is: ";
   cout << dist(myp1, myp2) << endl;
   return 0;
}
```

多文件结构（例5-10）

多文件结构

- 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。

外部变量

多文件结构

- 定义性声明
  - 命名空间作用域中，不用extern关键字声明的变量
  - 用extern关键字声明的变量，同时指定初值
- 引用性声明
  - 命名空间作用域中，用extern关键字声明的变量，但未指定初值

外部函数

多文件结构

- 在所有类之外声明的函数，都是具有命名空间作用域的
- 在不同源文件中调用前，要进行引用性声明（即声明函数原型）

外部变量与外部函数

多文件结构

```
// 源文件1
int i=3;
void other()
void next();

int main(){
  i++;
  next();
  return 0;
}

void next(){
  i++;
  other();
}
```

```
// 源文件2
extern int i;

void other(){
  i++;
}
```

把变量和函数限制在编译单元内

多文件结构

- 命名空间作用域中声明的变量和函数，在默认情况下都可以被其他编译单元访问
- 把函数和变量的定义限制在源文件内
  - 安全性
  - 防止重名

把变量和函数限制在编译单元内

多文件结构

- 方法一：定义全局变量和函数时使用static关键字
- 方法二：使用匿名的命名空间

```
namespace {
   int n;
   void f(){
     n++;
   }
n = 4;
f();
```

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  //其他情况下编译
#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。

不使用条件编译的头文件

多文件结构

```
//main.cpp
#include "file1.h"
#include "file2.h"
int 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
```

常成员函数的声明原则

深度探索

- 适当地将成员函数声明为常成员函数，能够提高代码质量。
- 凡是不会改变对象状态的函数，都应当声明为常成员函数。
- 什么是改变对象状态？
  - 改变对象状态，不简单地等同于改变成员数据的值。
  - 只要一个成员函数执行与否，不会影响以后接口函数的调用结果，都可以认为它不会改变对象状态。

常成员函数的声明原则

深度探索

```
class Line {    //Line类的定义
public: //外部接口
   Line(const Point &p1, const Point &p2)
      : p1(p1), p2(p2), len(-1) { }
   double getLen();
private:        //私有数据成员
   Point p1, p2;          //Point类的对象p1,p2
   double len;
};
double Line::getLen() {
   if (len < 0) {
        double x = p1.getX() - p2.getX();
        double y = p1.getY() - p2.getY();
        len = sqrt(x * x + y * y);
   }
   return len;
}
```

改变数据成员，但不改变对象状态

常成员函数的声明原则

深度探索

- 在原则上，应当将getLen声明为常成员函数，但由于修改了数据成员的值，语言规则不允许
- 怎么办？使用mutable关键字
  - mutable关键字使得被修饰的成员对象无视“常对象的成员对象被视为常对象”这一语言原则
- mutable须慎用

修改后的程序代码

深度探索

```
class Line {    //Line类的定义
public: //外部接口
   Line(const Point &p1, const Point &p2)
      : p1(p1), p2(p2), len(-1) { }
   double getLen() const;
private:        //私有数据成员
   Point p1, p2;        //Point类的对象p1,p2
   mutable double len;
};
double Line::getLen() const {
   if (len < 0) {
        double x = p1.getX() - p2.getX();
        double y = p1.getY() - p2.getY();
        len = sqrt(x * x + y * y);
   }
   return len;
}
```

代码的编译

深度探索

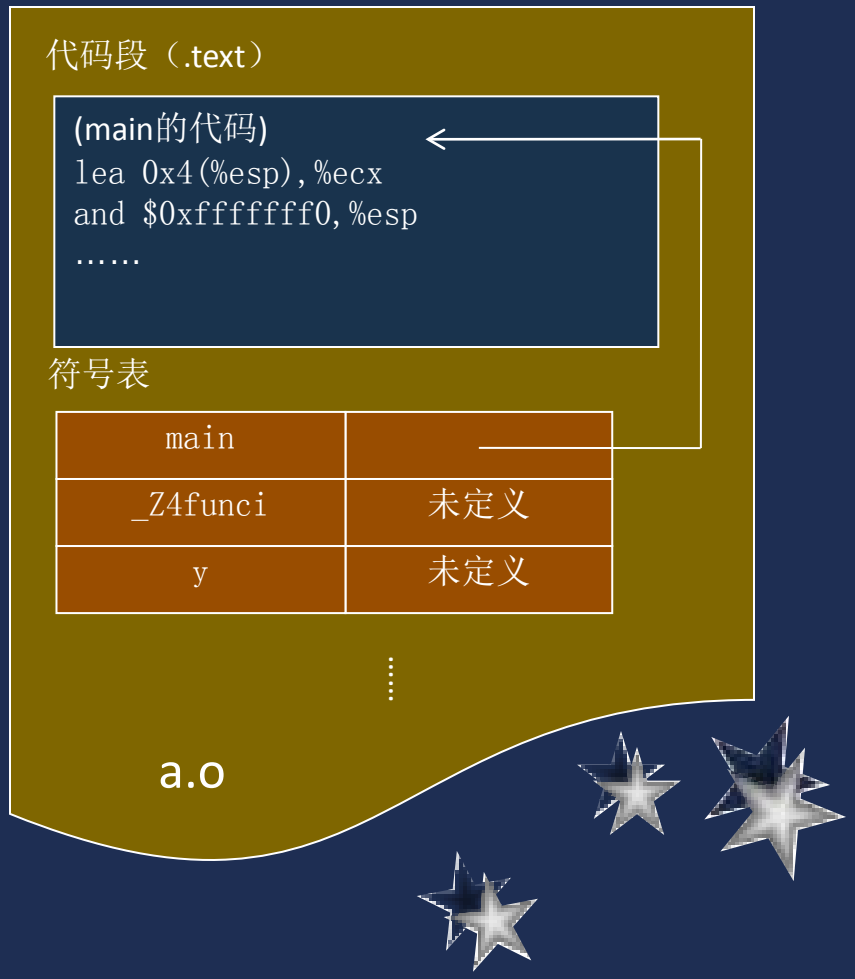
- 编译：源文件➔目标文件
  - 源文件的函数代码➔目标文件的代码段
  - 源文件的静态对象➔目标文件的数据段
    - 分为初始化的数据段和未初始化的数据段
  - 符号表：将静态对象和函数的名字与地址关联
  - 重定位信息、其它信息

示例代码(1)

深度探索

- a.cpp

```
extern int y;
int func(int v);
int main() {
    int z = 1;
    y = func(z);
    return 0;
}
```

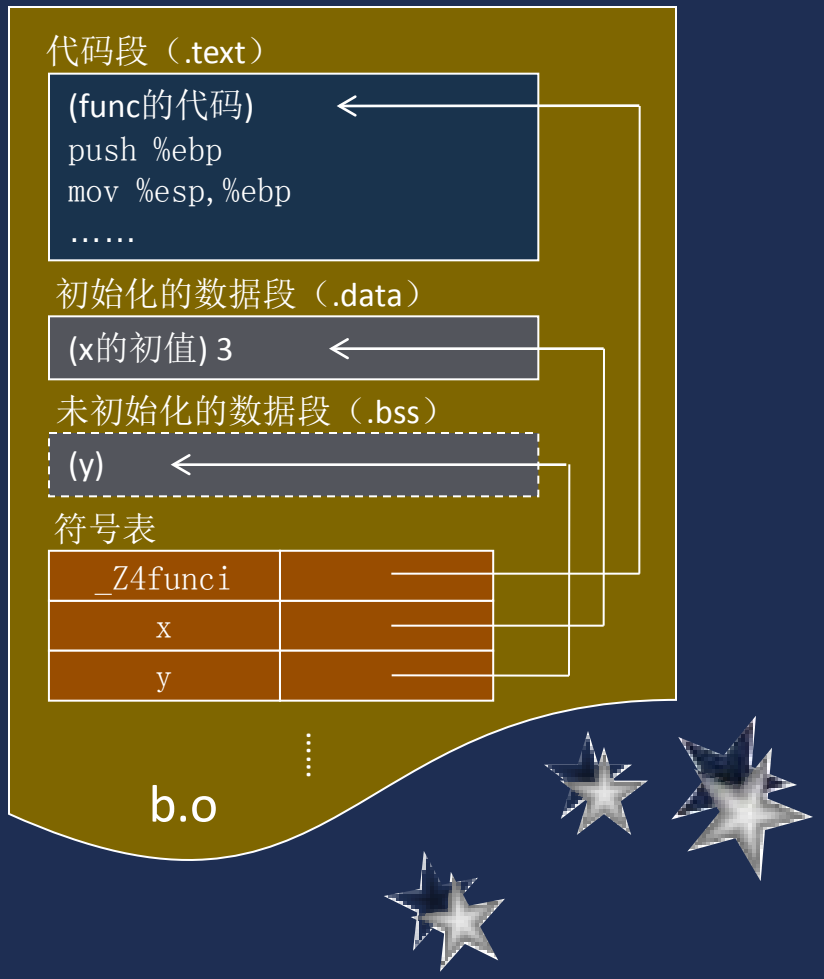


示例代码(2)

深度探索

- b.cpp

```
int x = 3;
int y;
int func(int v) {
  return v + x;
}
```



代码的连接与执行

深度探索

- 连接
  - 将各段合并
  - 将符号表综合
  - 根据重定位信息，确定代码中用到的全局地址
- 代码的执行
  - 操作系统首先将文件从磁盘读入，初始化各段——一些静态数据就在此时被初始化
  - 从引导代码开始执行，引导代码启动main，main返回后，引导代码通知操作系统程序结束

小结与复习建议

- 主要内容
  - 作用域与可见性、对象的生存期、数据的共享与保护、友元、编译预处理命令、多文件结构和工程
- 达到的目标
  - 深入理解程序的结构、模块间的关系、数据共享。