第9章　面向对象语言的编译

本章内容

- 概述面向对象语言的重要概念和实现技术
- 以C++语言为例，介绍如何将C++程序翻译成C程序
- 实际的编译器大都把C++程序直接翻译成低级语言程序

9.1 面向对象语言的概念

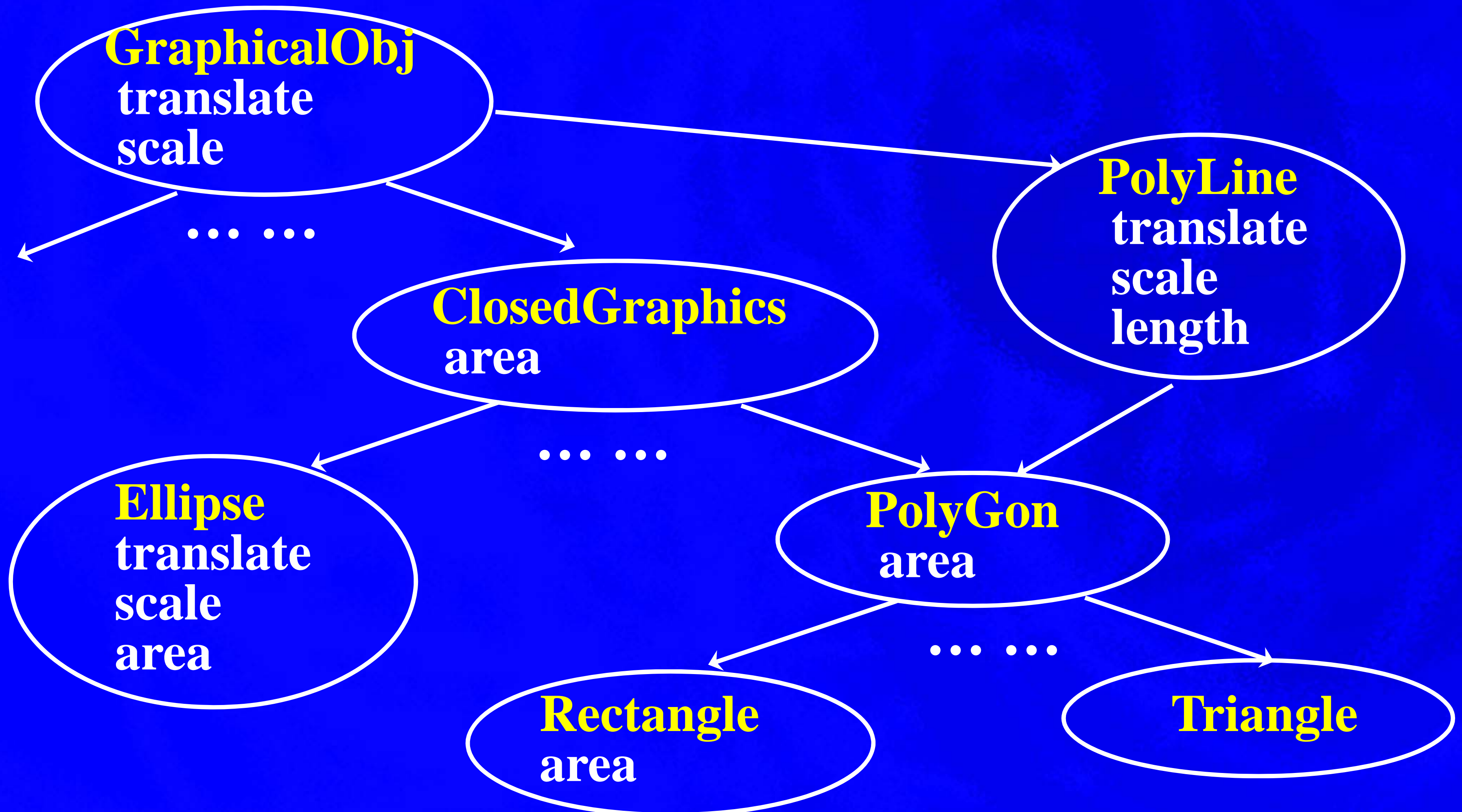
9.1.1 对象和对象类

- 对象
  - 由一组属性和操作于这组属性的过程组成
  - 属性到值的映射称为对象的状态，过程称为方法
- 对象类
  - 一类对象的总称，规范了该类中对象的属性和方法，包括它们的类型和原型
  - 对象有自己存放属性的存储单元；同一个类的对象可以共享方法的代码
  - 对象类形成了面向对象语言的模块单元
  - 下面将把术语“类”和“类型”看成是同义的

9.1 面向对象语言的概念

9.1.2 继承

图形对象的继承层次结构



9.1 面向对象语言的概念

继承

- 基类、派生类、子类、抽象类
- 子类型规则

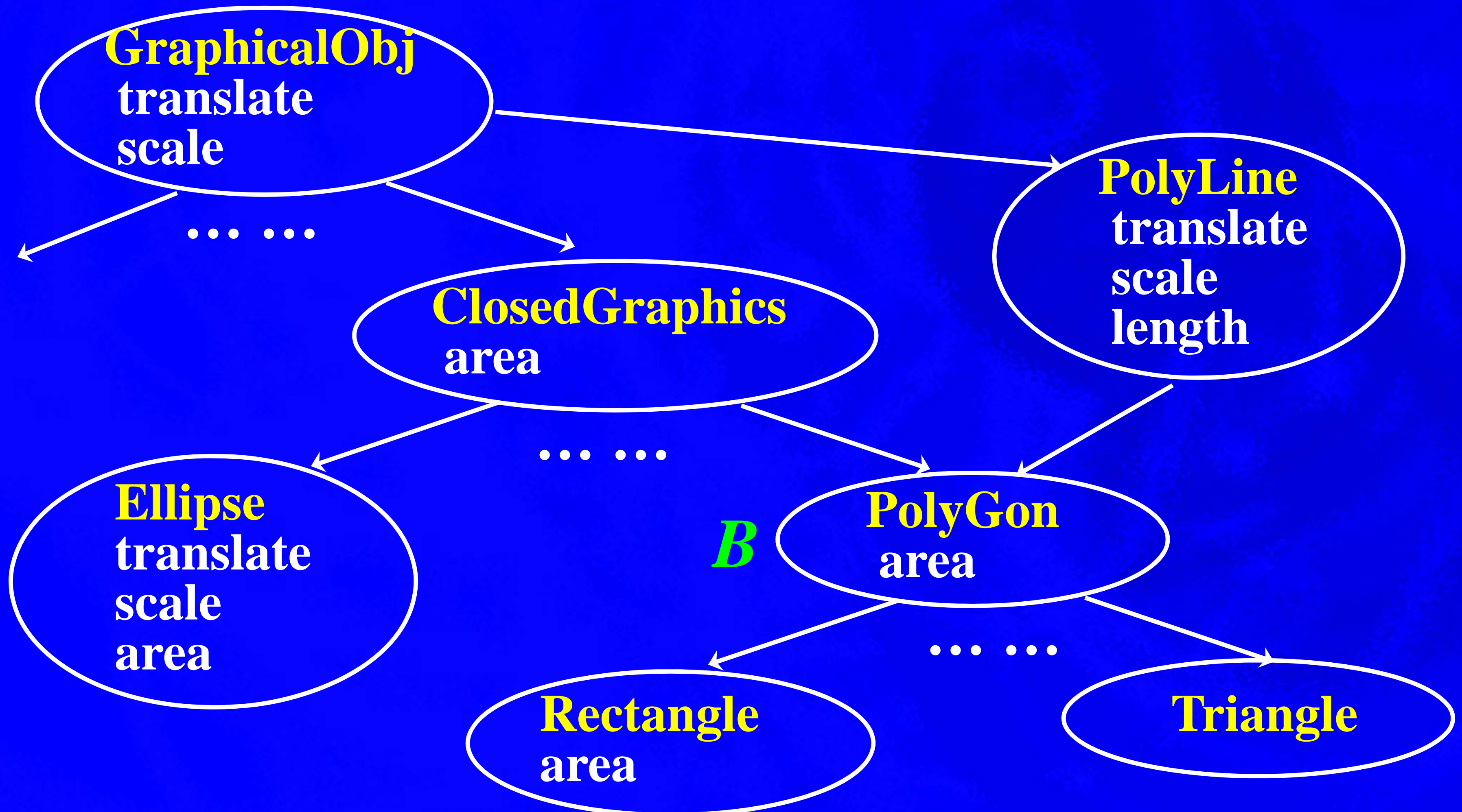
  当某个类型的一个对象在某个输入位置被需要或作为函数的返回值时，其任何子类型的对象允许出现在这些地方

- 类*B*的一个对象，若它不同时是*B*的某个真子类的对象，那么称该对象是*B*的真对象，称*B*是该对象的运行时类型

9.1 面向对象语言的概念

9.1.2 继承

图形对象的继承层次结构



9.1 面向对象语言的概念

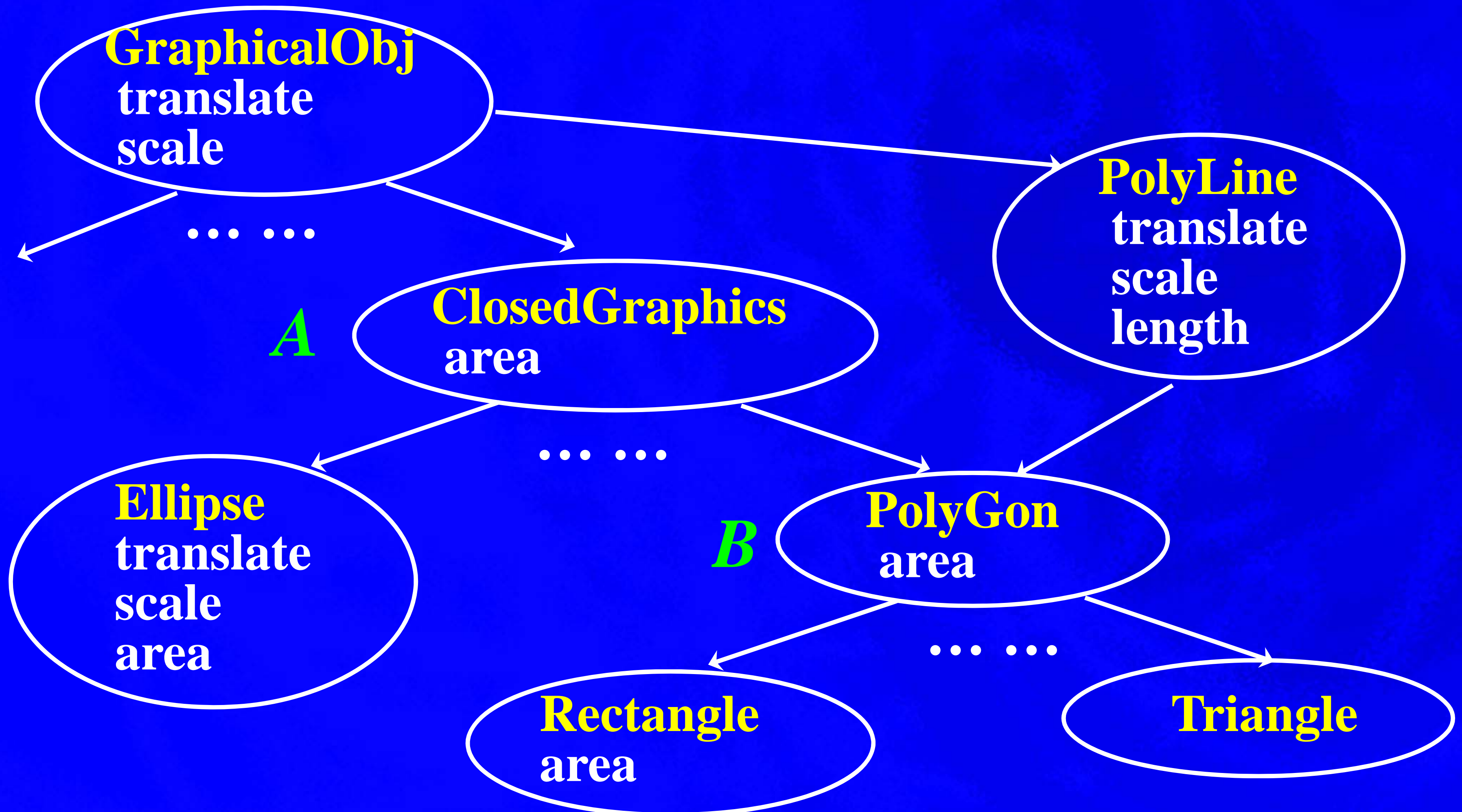
- 方法选择规则

  如果类*B*继承类*A*并且重写了方法m，那么对类*B*的对象b来说, 即使它作为类*A*的对象使用, 也必须使用在类*B*中定义的方法m

9.1 面向对象语言的概念

9.1.2 继承

图形对象的继承层次结构



9.1 面向对象语言的概念

- 动态绑定规则

  当对象o的一个方法可能被子类重新定义时，如果编译器不能确定o的运行时类型，那么必须对该方法进行动态绑定

```
void zoom (GraphicalObj &obj, double
                              zoom_factor, Point &center) {
     obj.translate (–center.x, –center.y);
                              // 将“中心”移至“点(0, 0)”
     obj.scale (zoom_factor);    // 缩放
}
```

9.1 面向对象语言的概念

9.1.3 信息封装

- 大多数面向对象语言提供了一种机制，它可用来将类的特征分成私有的和公共的
- 某些面向对象语言用不同的上下文区分作用域，如“在一个类中”、“在派生类中”、“在友元类中”等等
- 由编译器来实现这些作用域规则是简单而又明显的

9.2 方法的编译

先定义一般的图形对象类GraphicalObj如下:

```
class GraphicalObj {
virtual void translate (double x_offset, double
                                        y_offset);
virtual void scale (double factor);
. . .              // 可能还有一些其它方法
};
```

9.2 方法的编译

```
class Point : public GraphicalObj {
    double xc, yc;
  public :
    void translate (double x_offset, double y_offset) {
            xc += x_offset;  yc += y_offset;
    }
    void scale (double factor) {
            xc *= factor;  yc *= factor;
    }
    Point(double x0 = 0, double y0 = 0) {xc = x0; yc = y0; }
    void set(double x0, double y0) {xc = x0; yc = y0;}
    double x(void) {return xc;}
    double y(void) {return yc;}
    double dist (Point &);
};
```

9.2　方法的编译

将一个C++语言的类翻译成C语言的程序段，主要工作有如下几点(由继承引出的问题暂不考虑)

- 将C++语言中一个类的所有非静态属性构成一个C语言的结构体类型，取类的名字作为结构体类型的名字
- 类的静态属性是该类的所有对象所共有的, 应当翻译成C中的全局变量，但是需要改一个名字
- C++语言中类的对象声明不加翻译就成了C语言中相应结构体类型的变量声明

9.2　方法的编译

- 将C++语言中类的非静态方法翻译成C语言的函数，对应的方法和函数的区别有下面几点：
  - 函数的名字必须在原来方法名的基础上修改
  - 函数声明增加一个形参this
  - 在函数体中出现的函数调用也要增加一个实参
  - 在方法中对本对象的非静态属性的访问，改成对this相应域的访问。在方法中对其它对象的非静态属性的访问不必修改
- 类的静态方法在定义和调用的地方都需要改名

9.2　方法的编译

类C的方法m被翻译成函数fm

	方　法	函　数
原型	返回类型 m(形参表)	返回类型 fm(C &this, 形参表)
调用	m（实参表） o.n（实参表）	fm（this，实参表） fn（o，实参表）
属性访问	k o.k	this.k o.k

9.2 方法的编译

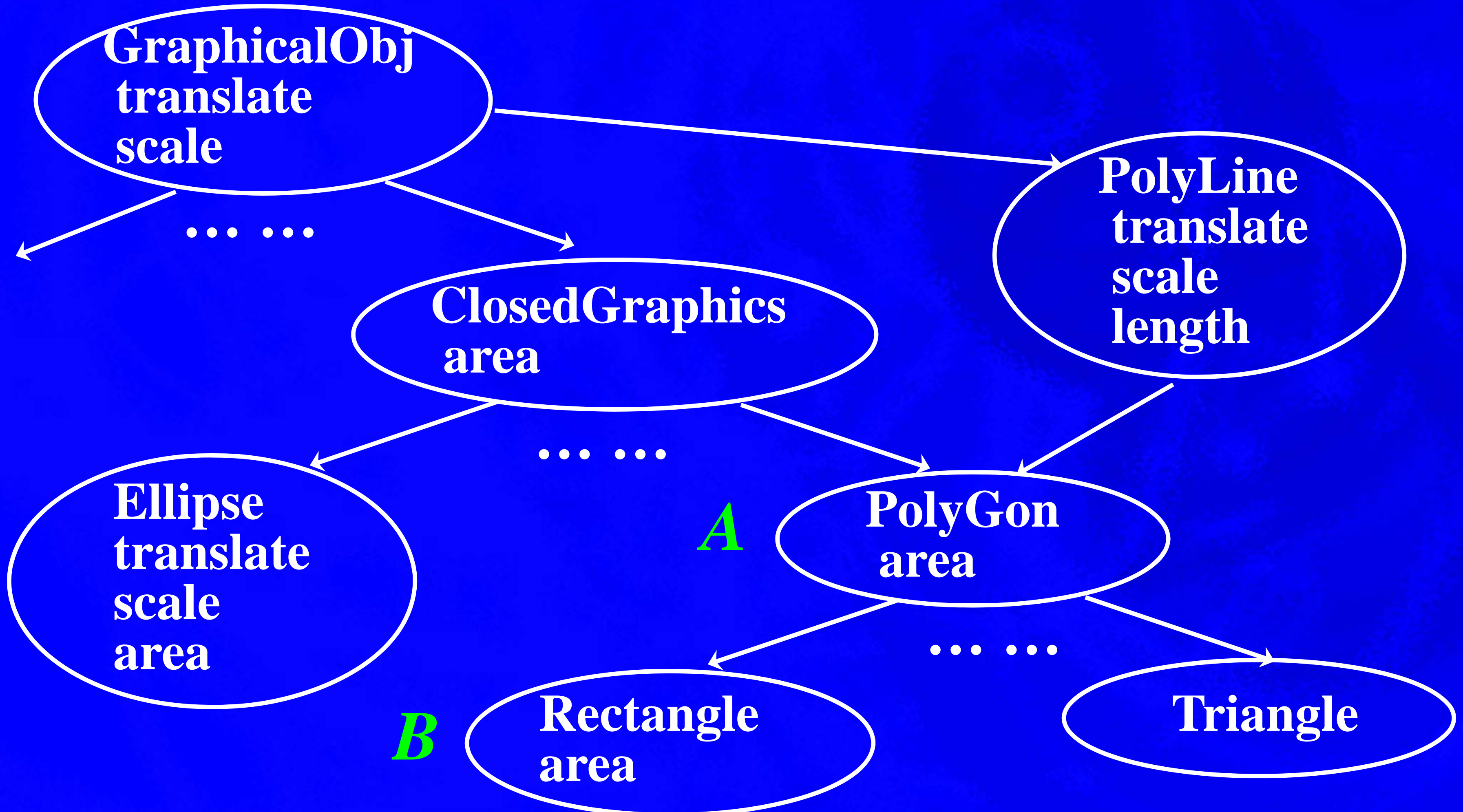
类Point的方法translate翻译成函数

translate_ _5Pointdd

```
void translate_ _5Pointdd(Point this,
                          double x_offset , double
   y_offset) {
   this.xc += x_offset;  this.yc += y _offset;
}
```

9.3 继承的编译方案

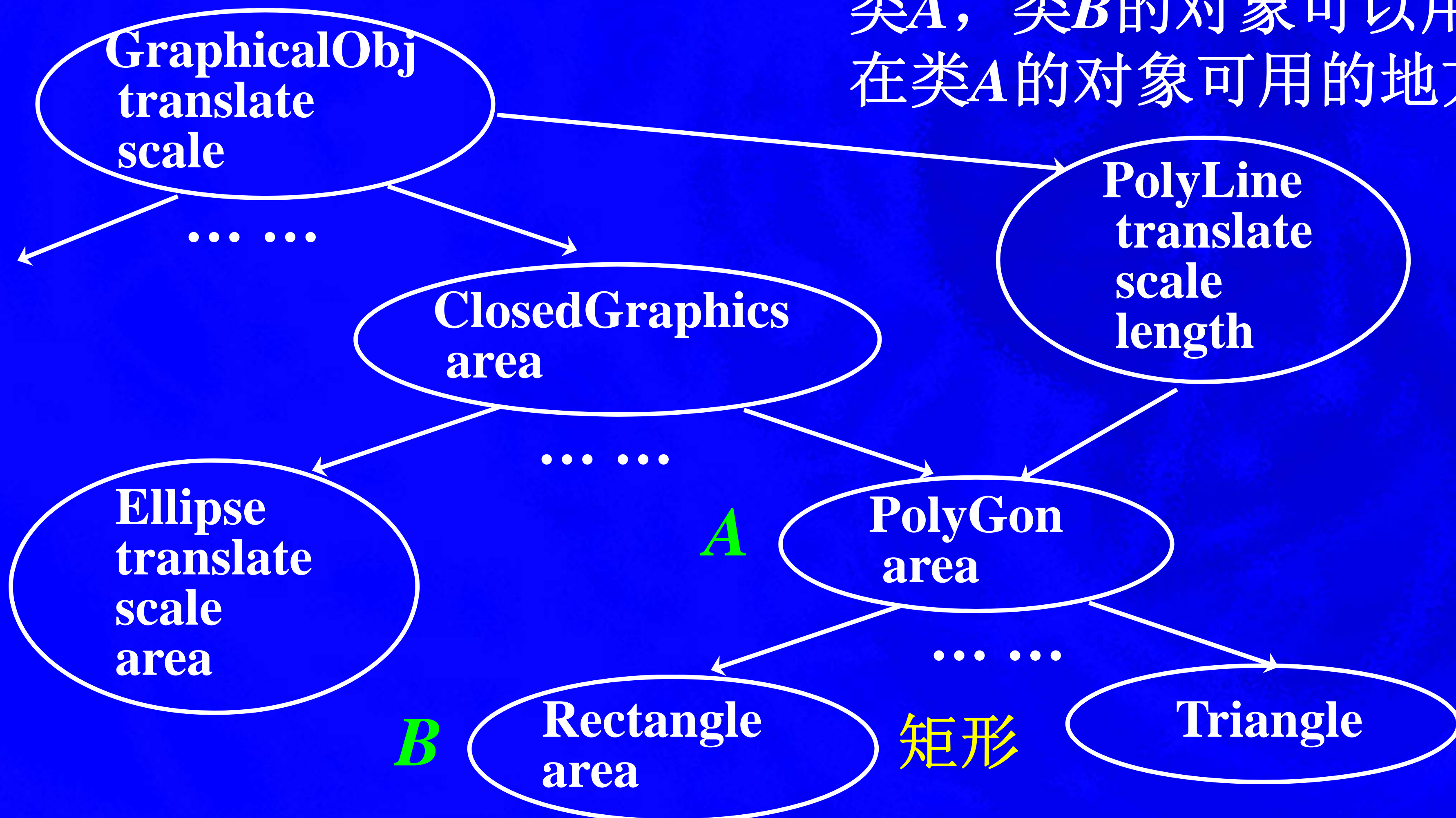
图形对象的继承层次结构



9.3 继承的编译方案

图形对象的继承层次结构

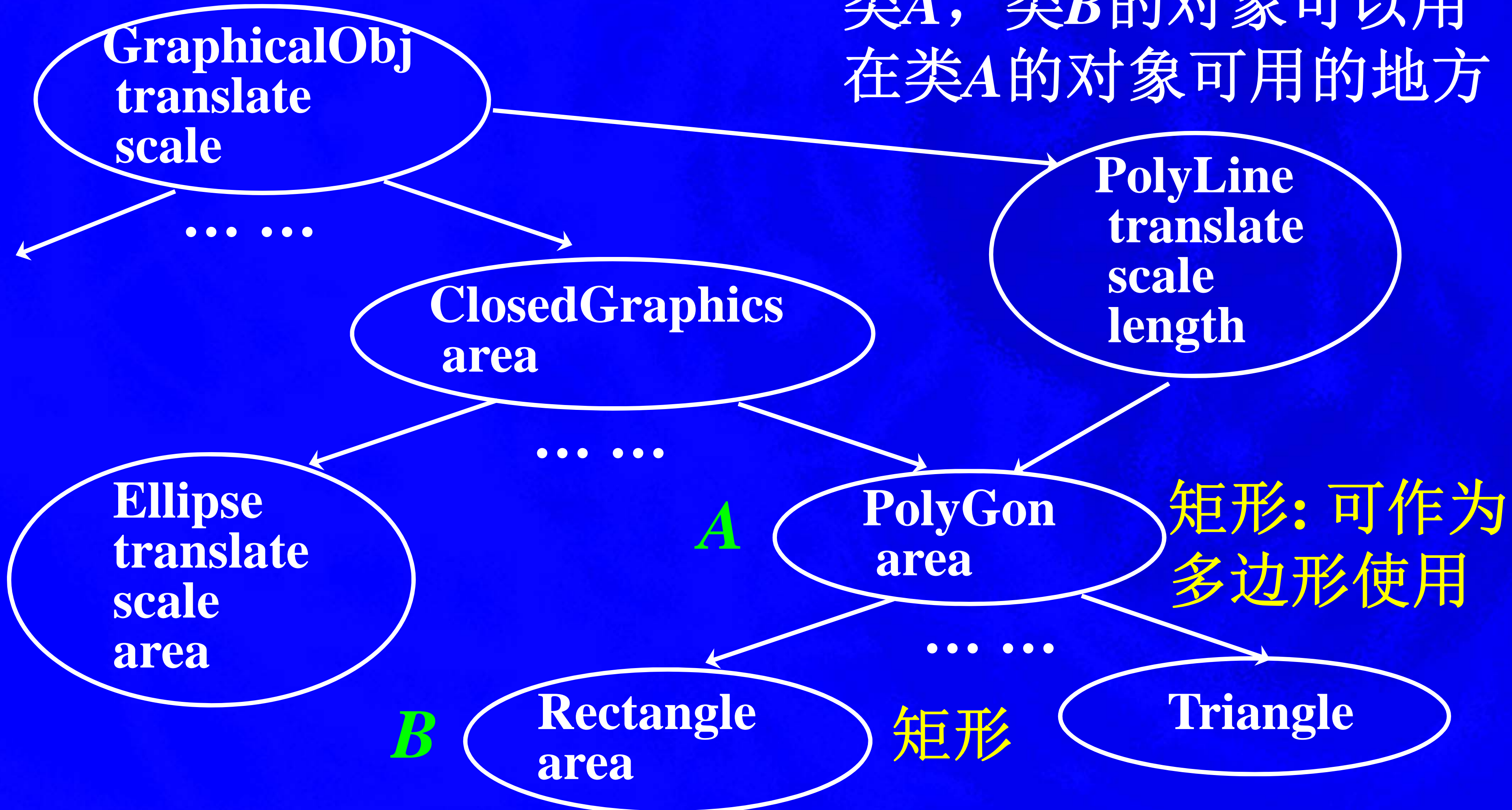
若类*B*直接或间接继承类*A*，类*B*的对象可以用在类*A*的对象可用的地方



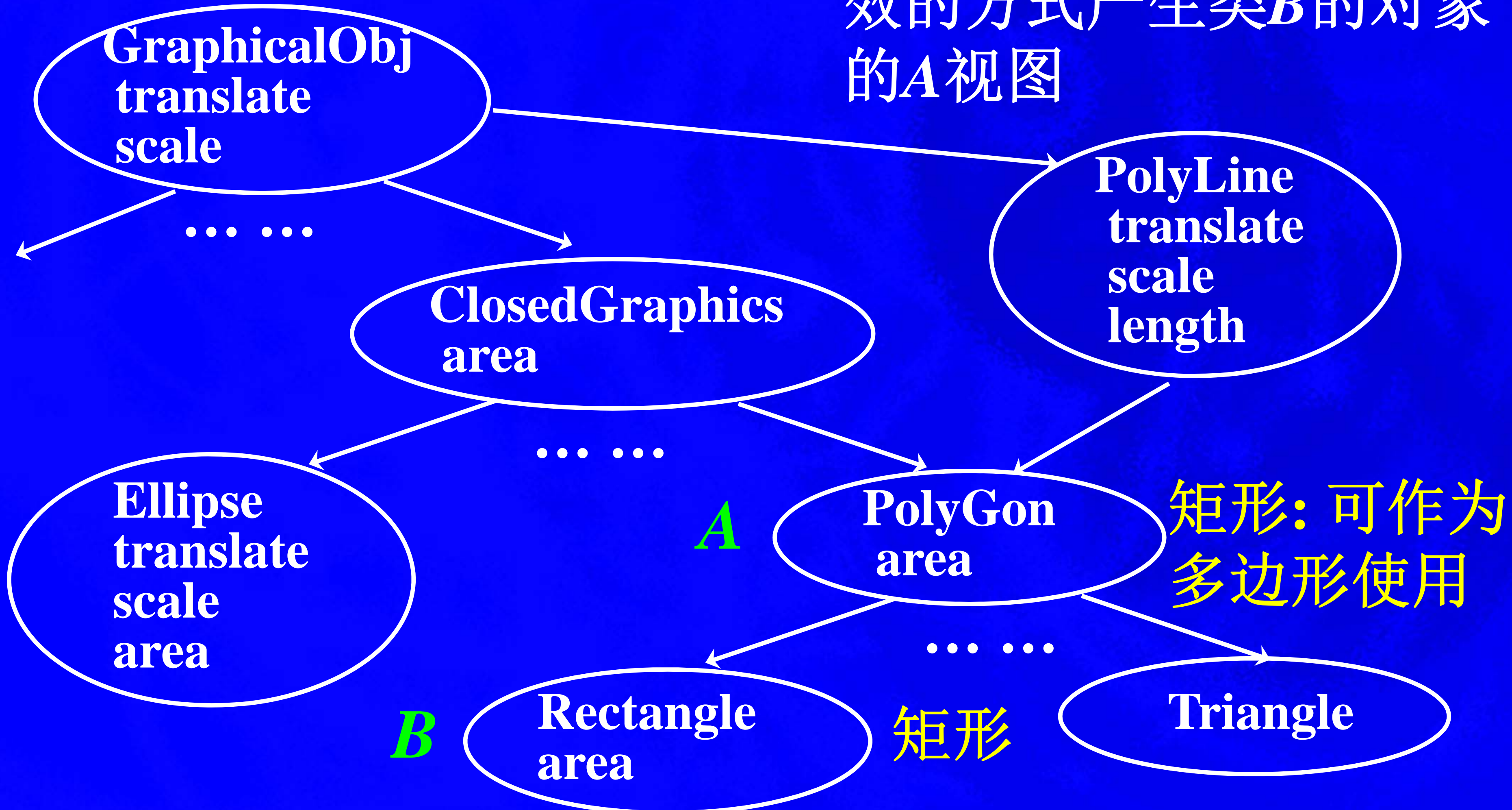
9.3 继承的编译方案

图形对象的继承层次结构

若类*B*直接或间接继承类*A*，类*B*的对象可以用在类*A*的对象可用的地方



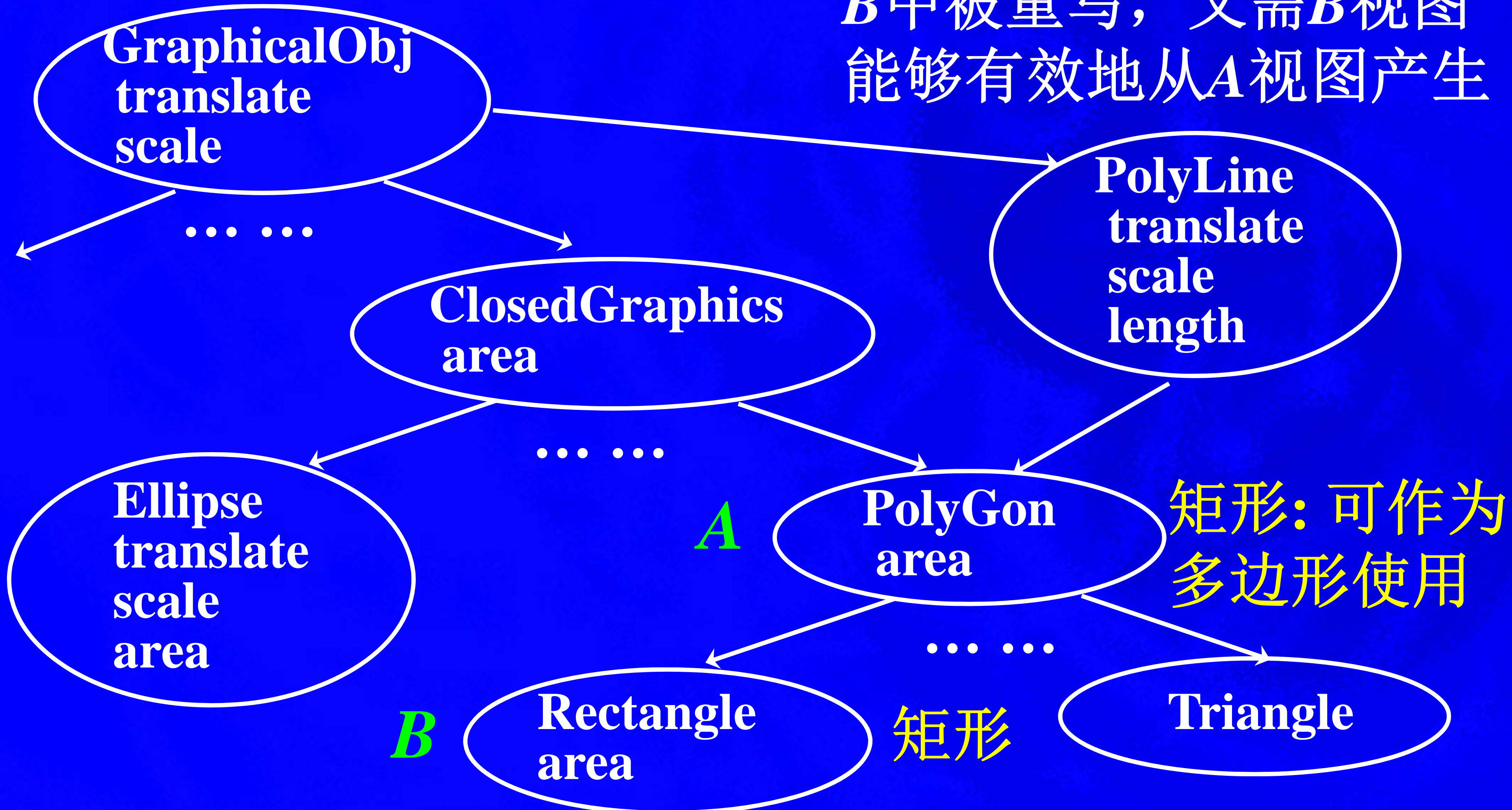
9.3 继承的编译方案



9.3 继承的编译方案

图形对象的继承层次结构

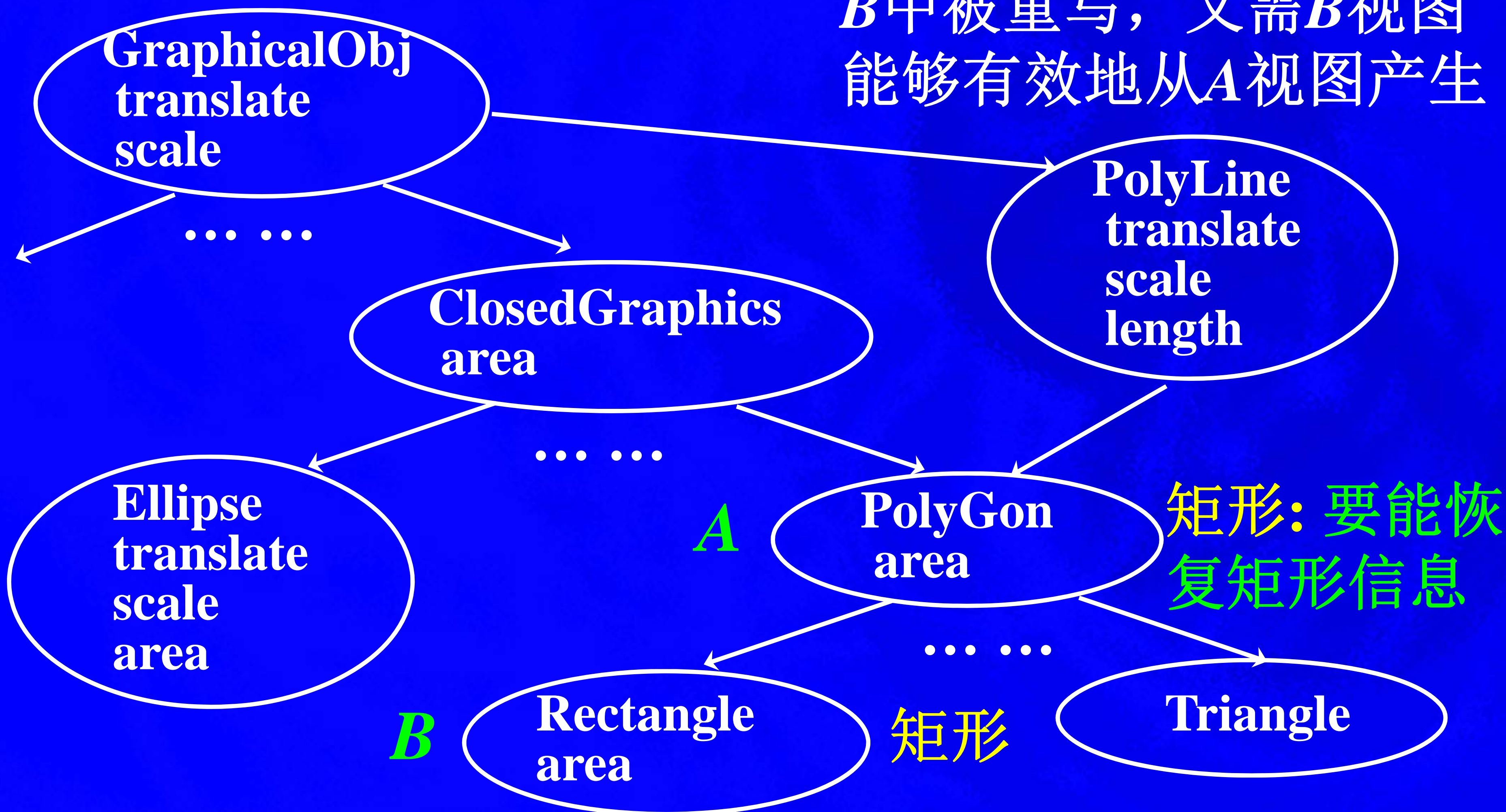
因类*A*的虚方法可在类*B*中被重写，又需*B*视图能够有效地从*A*视图产生



9.3 继承的编译方案

图形对象的继承层次结构

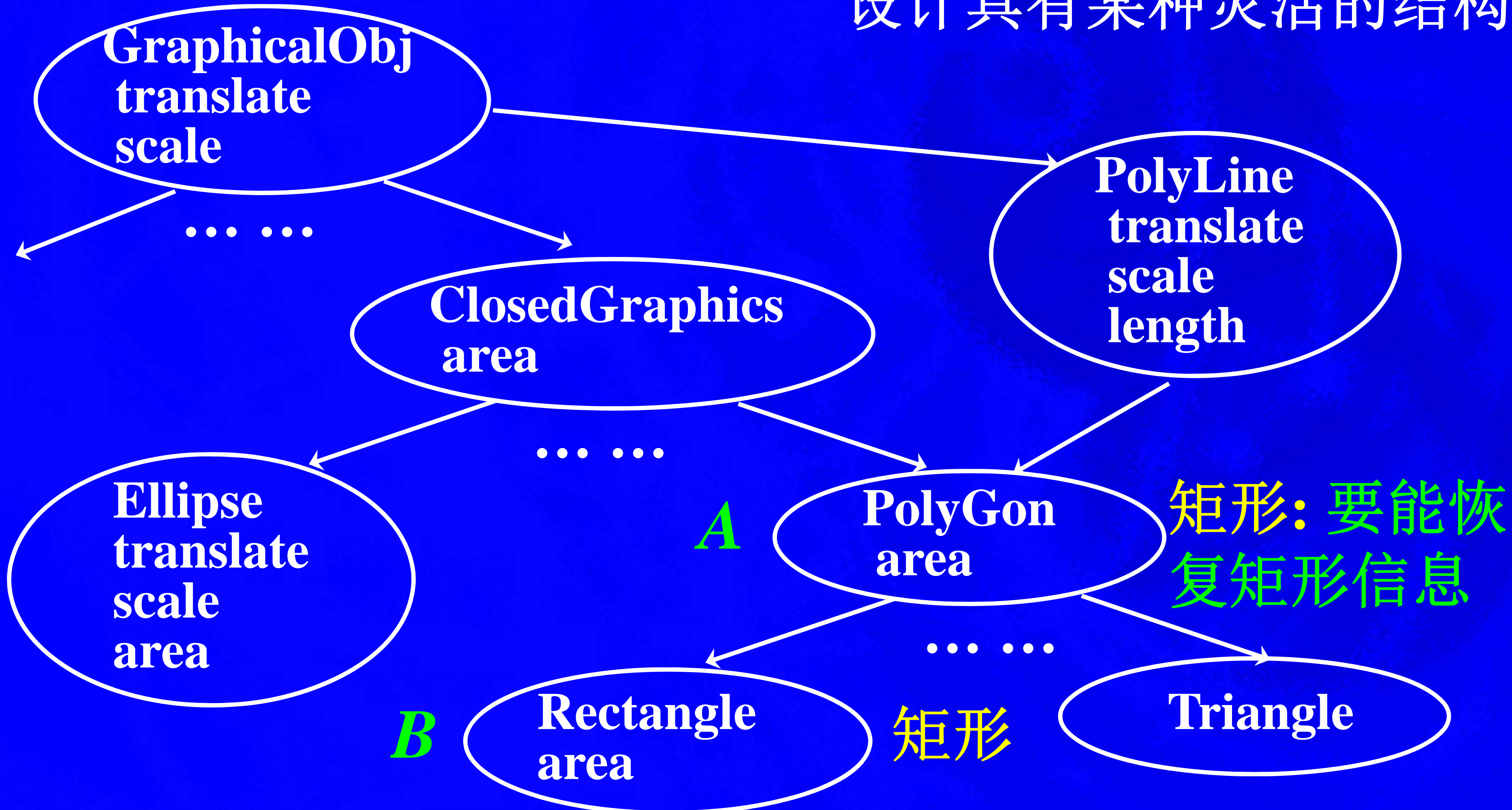
因类*A*的虚方法可在类*B*中被重写，又需*B*视图能够有效地从*A*视图产生



9.3　继承的编译方案

图形对象的继承层次结构

需要编译器为类的对象设计具有某种灵活的结构



9.3 继承的编译方案

9.3.1 单一继承的编译方案

```
#include "graphicalobj.h"
#include "list.h"
#include "point.h"
class PolyLine : public
                    GraphicalObj {
   list <Point> points;
 public:
   void translate (double
        x_offset, double y_offset);
   virtual void scale (double
                          factor);
   virtual double length (void);
};
```

```
#include "polyline.h"
class Rectangle : public
                         PolyLine {
    double side1_length,
    double  side2_length;
  public :
   Rectangle (double s1_len,
                 double s2_len,
              double x_angle = 0);
   void scale (double factor);
   double length (void);
};
```

9.3　继承的编译方案

```
void zoom (GraphicalObj &obj, double
                    zoom_factor, Point &center) {
    obj.translate (–center.x, –center.y);
                    // 将“中心”移至“点(0, 0)”
    obj.scale (zoom_factor);    // 缩放
}
```

如果函数zoom作用于矩形，那么zoom的体必须调用Rectangle的缩放函数，而不是PolyLine甚至GraphicalObj的缩放函数

9.3 继承的编译方案

编译器怎样有效地实现动态绑定？

- 编译器为每个类建立一个方法表，它们包含该类或它的超类中所有定义为**virtual**的方法的入口
- 每个对象在**C**程序中有对应的结构体，再为这种结构体增加一个域，该域是方法表的指针
- 继承类方法表的产生
  - 首先拷贝基类的方法表，被重新定义的方法由新的定义覆盖
  - 然后把新引入的方法追加到这张表上

9.3 继承的编译方案

图形对象的不同子类的方法表

translate_GO
scale_GO

GraphicalObj

translate_PL
scale_PL
length_PL

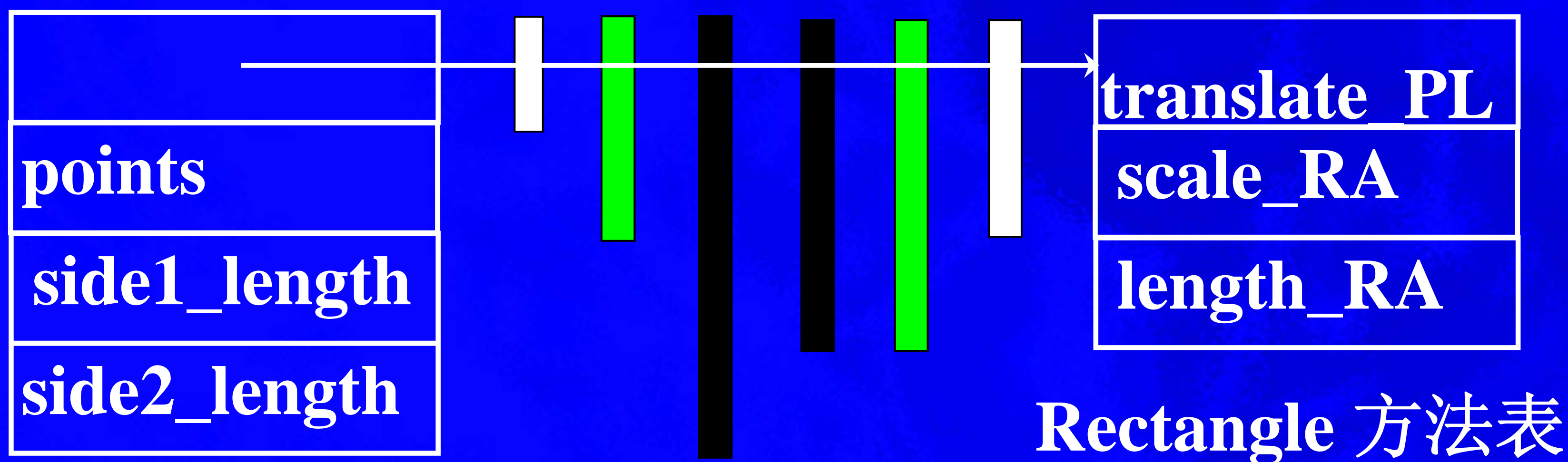
PolyLine

translate_PL
scale_RA
length_RA

Rectangle

9.3　继承的编译方案

Rectangle的对象表示



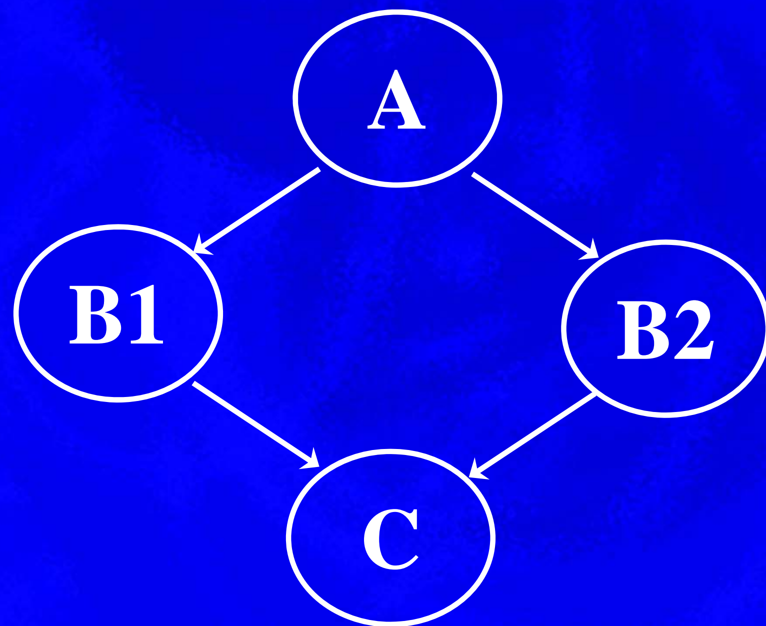
视图：GraphicalObj PolyLine Rectangle

9.3　继承的编译方案

9.3.2 多重继承的编译方案

多重继承对语言定义和编译器设计来说，都具有很大的挑战性

- B1和B2之间的冲突与矛盾
- 多重继承
  - 可以有多个实例
  - 只能有一个实例

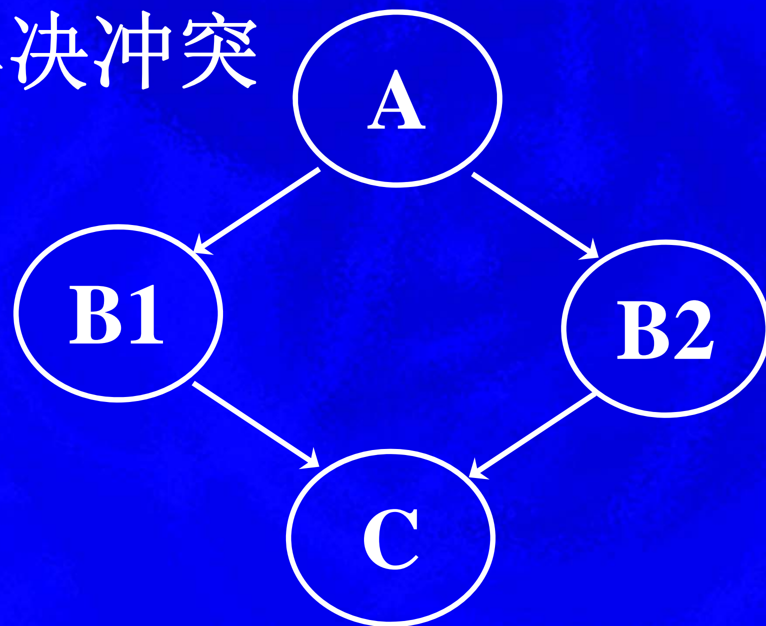


9.3 继承的编译方案

B1和B2之间的冲突与矛盾

这是语言定义问题，解决办法：

- 将B1定义为主要后代，冲突解决优先于B1
- 语言允许重新命名被继承的特征
- 语言提供显式地手段来解决冲突
  - B1::n或B2::n



- 实现起来并无什么困难，只涉及到编译器符号表的组织和管理问题

9.3 继承的编译方案

下面两种方式都有应用，仅讨论前者

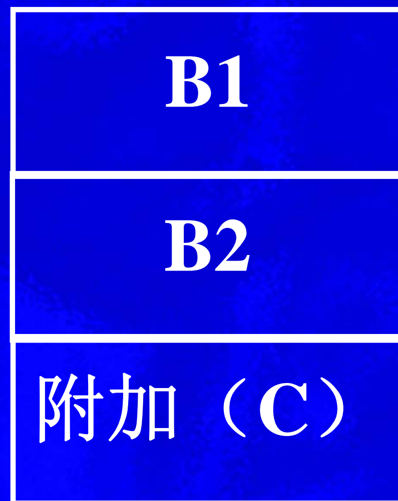
多重继承的多个实例

多重继承的单个实例

9.3　继承的编译方案

独立的多重继承的编译方案

- 继承类C的对象包含基类B1和B2的完整拷贝
- 来自基类的继承是相互独立的



独立的多重继承时的对象结构（程序视图）

9.3 继承的编译方案

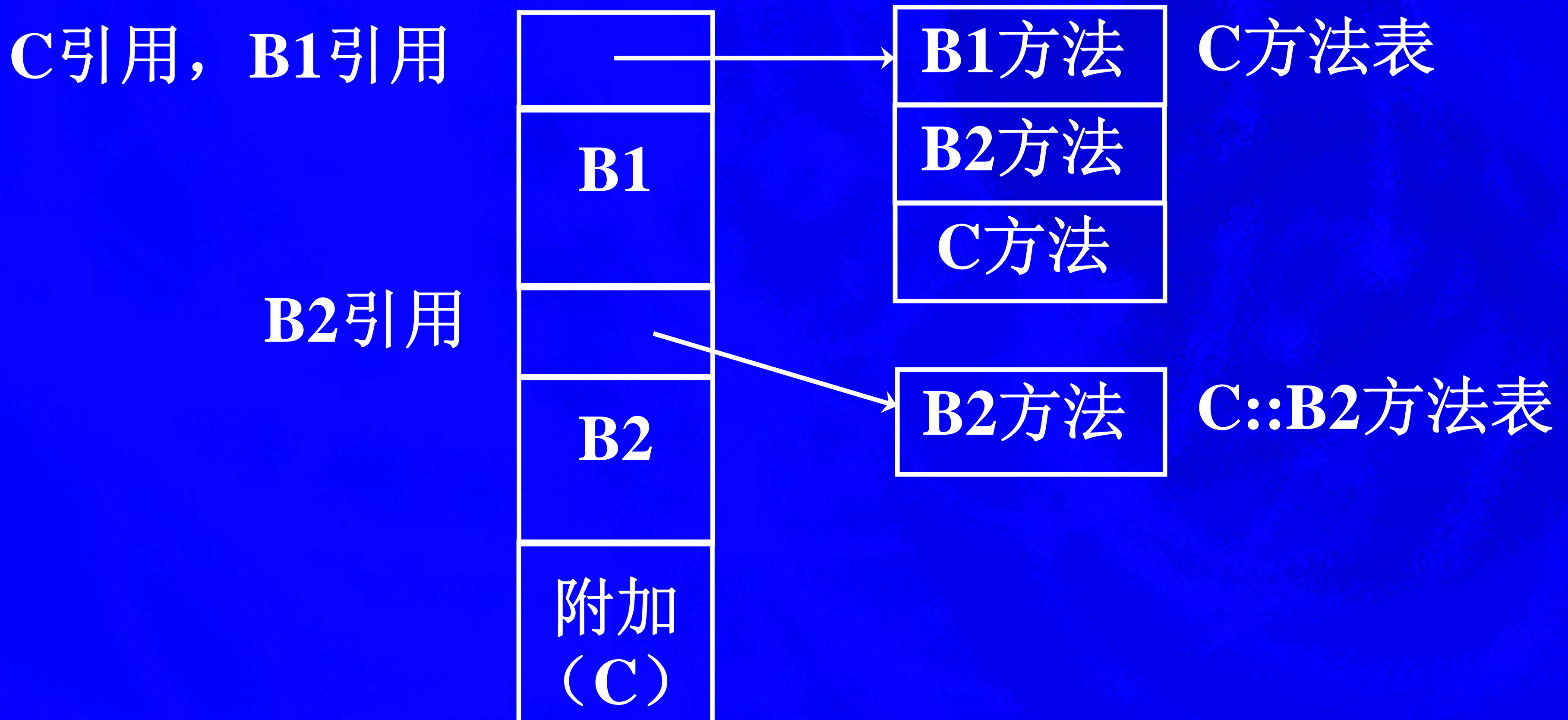
多重继承在下述情况导致冲突和二义

- 当多实例的特征被用于访问、调用和覆盖的时候
- 当类C的对象的A视图被建立时，因为类C的对象包含多个类A子对象
- 可见性规则可以在某些情况下帮助避免这些困难

独立的多重继承时的对象结构（程序视图）

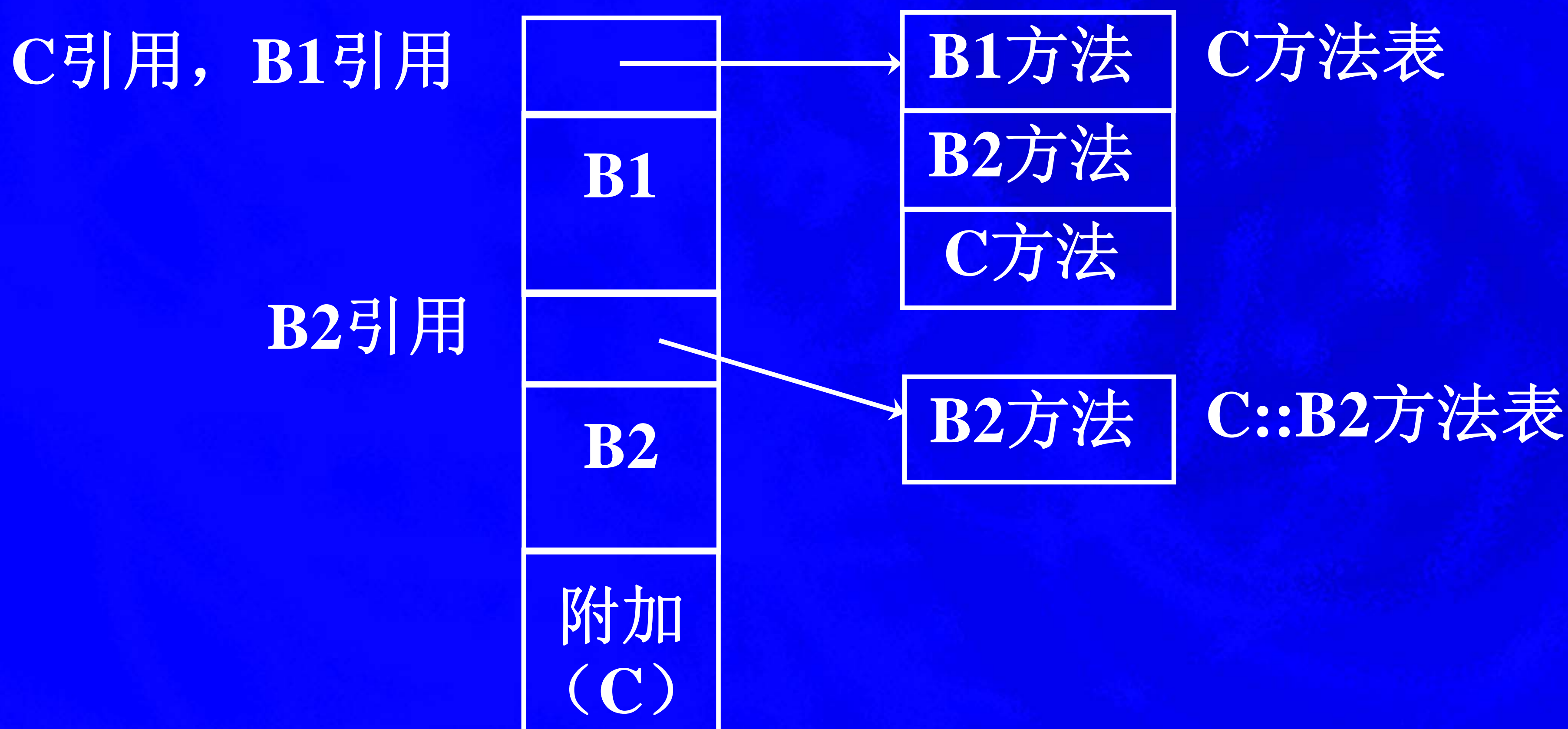
9.3 继承的编译方案

独立的多重继承的对象结构（实现视图）
（把单一继承的编译方案加以扩充 ）



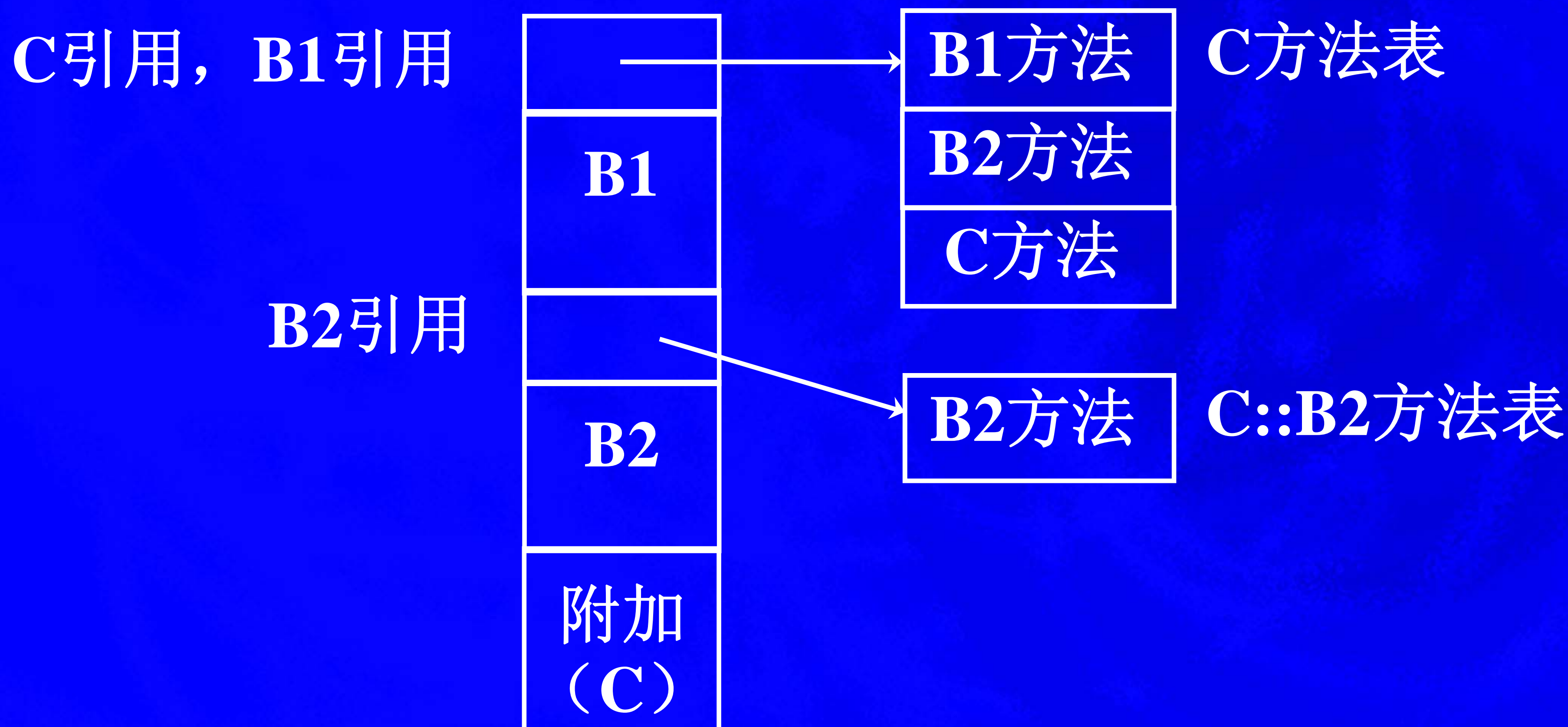
9.3 继承的编译方案

C对象的B1视图是C视图的开头部分
C视图的开头部分不能作为B2视图



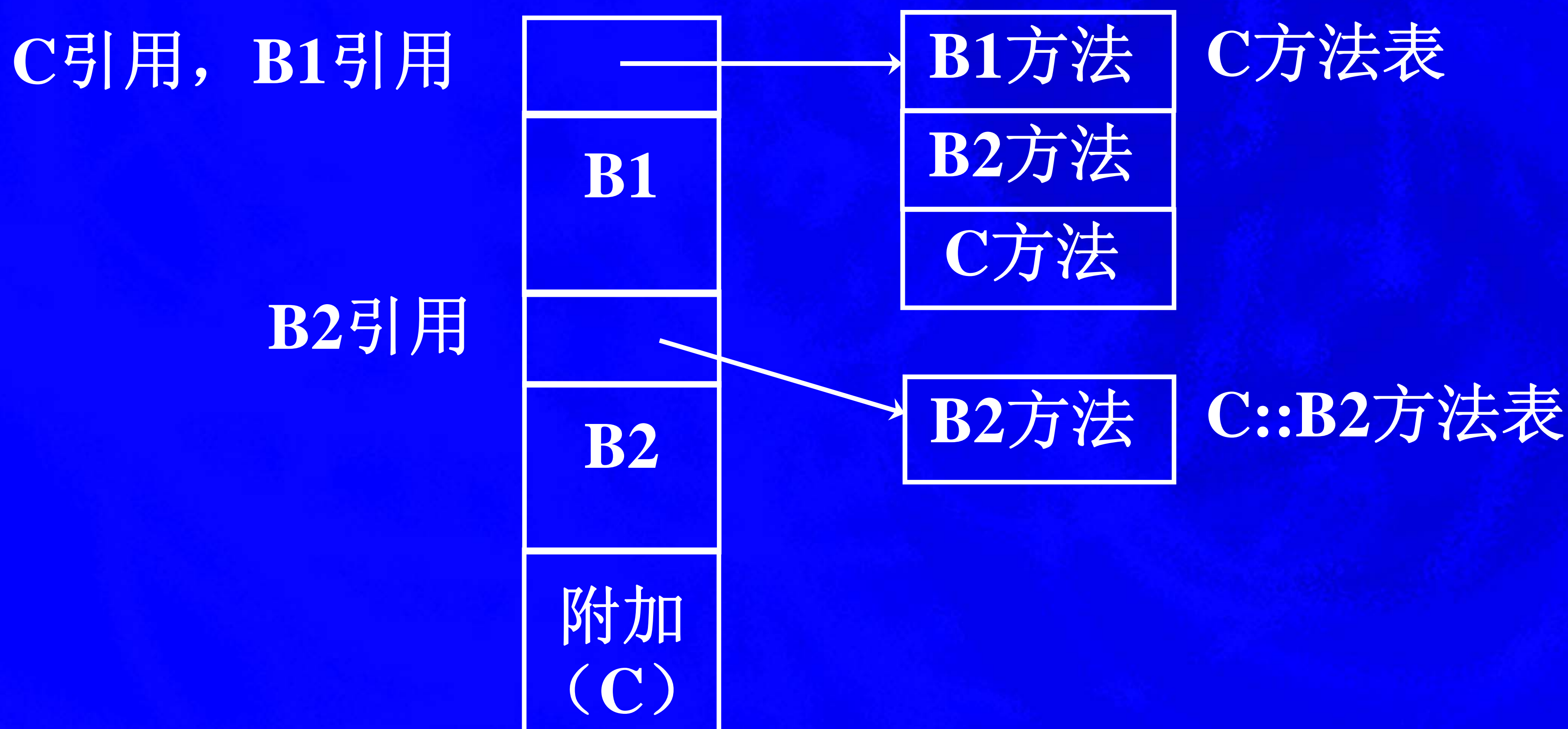
9.3 继承的编译方案

困难的事情是，从B2的视图来恢复C的视图



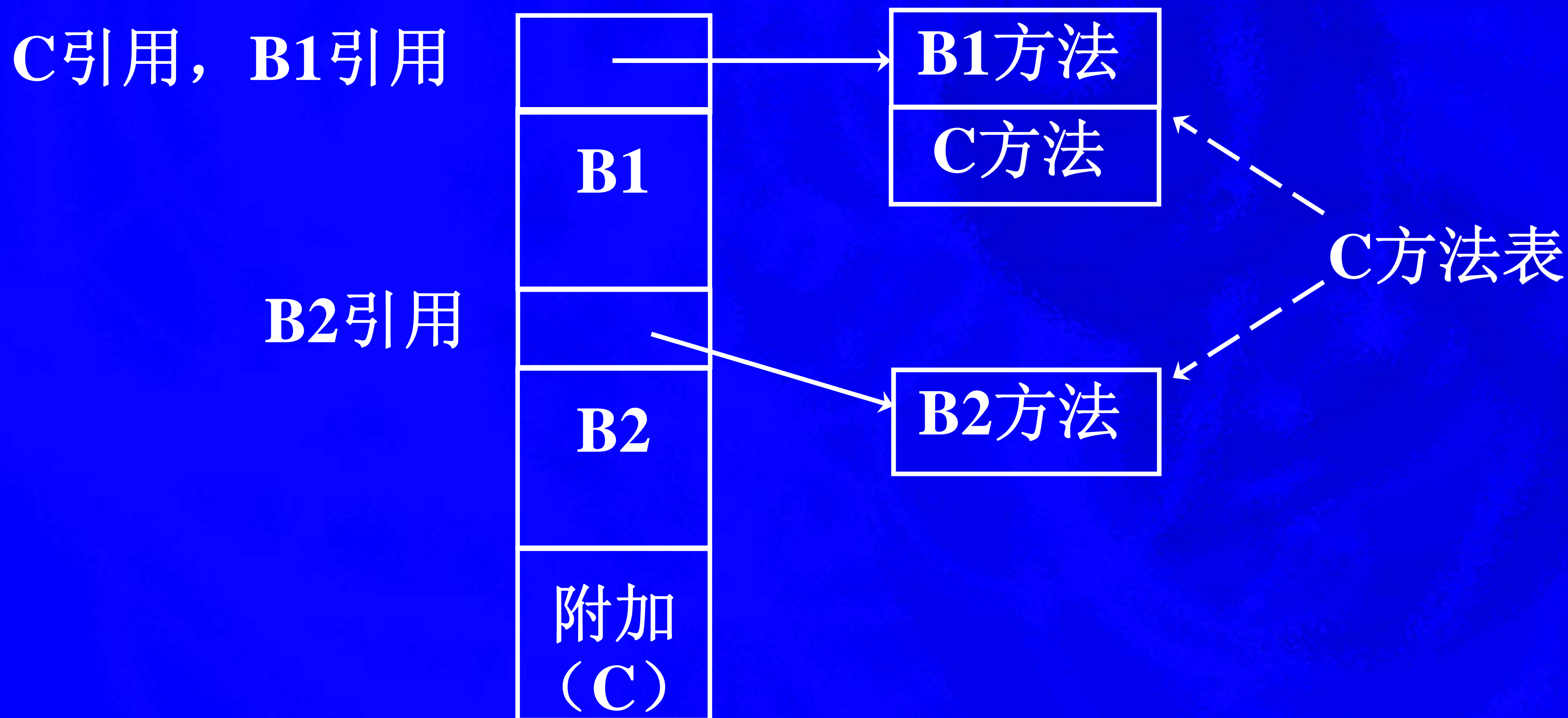
9.3 继承的编译方案

编译器把用于确定所需视图的偏移存放在方法表中下邻该方法指针的地方



9.3　继承的编译方案

独立的多重继承的对象结构（实际视图）



习　　题

第1次：9.1, 9.2, 9.3