计算机程序设计语言系列丛书

C++编程技巧与实例

甘 特
杨任润 等编

学苑出版社

计算机程序设计语言系列丛书

C++ 编程技巧与实例

甘　特　杨仕润
江丽艳　梁　革　等编写
刘　军　晓　鸥
夏　薇　审　校

学苑出版社
1993

(京) 新登字 151 号

内 容 提 要

本书着重讨论了 C++程序设计技巧及其应用方法，主要包括以下内容：怎样从 C 过渡到 C++、工具库、串类、范围和下标、为数组建立块、可排序数组、整数数组、浮点数数组、基本统计、永久对象、散列表、用散列表索引文件、树结构、用 B 树的索引文件等内容，附录 A 给出各章程序的完整代码。

要想透彻地理解本书，读者首先应当了解 C++的基本机制，并写过一些 C++程序，还要有兴趣探索怎样用 C++来开发应用程序，读者还应有 Microsoft C / C++ 7.0 或 Borland C++ 3.0 才能使用本书中的源代码。

本书适用于大专院校有关专业的师生、从事计算机工作的人员及科研单位有关专业技术人员参考使用。

欲购本书的用户，请直接与北京 8721 信箱联系，邮政编码：100080，电话 2562329。

计算机程序设计语言系列丛书

C++编程技巧与实例

编　　写：甘　特　杨仕润　江丽艳
　　梁　革　刘　军　晓　鸥
审　　校：夏　薇
责任编辑：徐建军
出版发行：学苑出版社　　邮政编码：100032
社　　址：北京市西城区成方街 33 号
印　　刷：北京市朝阳区小红门印刷厂
开　　本：787×1092　1 / 16
印　　张：38.0　　字数：895 千字
印　　数：1—5000
版　　次：1993 年 12 月北京第 1 版第 1 次
ISBN 7-5077-0807-1 / TP・18
本册定价：39.00 元

学苑版图书印、装错误可随时退换

目 录

第一章　C++发展史

本书将首先从 C++和面向对象的程序设计谈起，并提出一些新观点，这些观点可能与那些被普遍接受的概念相异．了解了作者意图，将有助于读者了解本书的其余章节．

我们在这里所谈的不是一些绝对的规定，它只是帮助大家思考怎样开发 C++程序，但也不要只将它当成琐碎的重复而弃之不顾．

1.1　C++的起源

众所周知，C++是 C 的一个扩展本．对 C 的 C++扩展首先是由 Bjarne Stroustrup 于 1980 年在贝尔实验室发明的．起初，他称这一新的语言为"带类的 C 语言"．然而，在 1988 年，名字改为 C++．

尽管 C++的前身 C，目前在世界上是受欢迎且被广泛使用的专业编程语言之一，但 C++的发明是编程中最主要的一个因素—复杂性所需要的．在 C 中，一旦程序从 25000 行扩展到 100000 行，它就变得相当复杂，很难合为一个整体．C++的目的就是要打破这一障碍．C++的本质就是让程序员充分理解并管理更大型，且更复杂的程序．

Stroustrup 添加到 C 中的绝大多数内容支持面向对象编程（有时简写为 OOP）．据 Stroustrup 所言，某些 C++的面向对象特性来自于另一个称为 Simula67 的面向对家语言．因此，C++代表着两种强功能编程方法的折衷．

当发明 C++时，Stroustrup 深知，最为重要的是保留 C 的原始精华，其中包括有效性、灵活性和基础原理．而与此同时添加了对面向对象编程的支持．令人高兴的是，他的目标都达到了．C++仍给程序员提供了 C 语言的灵活和控制，以及对家的强功能．用 Stroustrup 的话说，C++中的面向对象特性"使程序构造得清晰，可扩充、易维护，且不失有效性"．

尽管 C++在开始时是为管理非常大型的程序而设计的，但无法限制这一使用．实际上，C++的面向对家属性可有效地适用于任意编程任务．人们常常会看到 C++用于像编辑程序、数据库、个人文件系统和通讯程序之类的项目．此外，由于 C++共享 C 的有效性，所以更高性能的软件是用 C++构造的．

1.2　从函数到对象

当程序设计员们讨论起哪种语言最完美时，发现各种面向功能的程序设计语言之间是没有本质性差别的，用户从 BASIC 语言转到 C 或 FORTRAN、PASCAL 语言并无很大困难，if 语句就是 if 语句，函数就是函数，它与所用语言无关：在设计面向功能的程序时，不需要用特定语言的术语进行思考，因为各种语言间的语法和功能基本上是等价的．

用面向对象的语言进行程序设计则不同于面向功能的设计，面向对家的程序应当在结

构上不同于面向功能的程序。面向功能的程序围绕着要执行的动作进行组织，而设计良好的面向对象式程序按照被操作的对象进行安排，对于习惯了面向功能式程序设计的人员来说，这个转变就比较困难。

有些人说，C++是扩充进对象的 C 语言，那只是一种简化性描述，因为从 C 到 C++的过渡只是加进了一些面向对象的特征，它使得面向对象的技术应用起来快速、小巧、易于程序维护。但是，大家千万不要被这种说法引入歧途，C++不只是比 C 新增加了一些关键字，虽然我们能够用 C 语言写出面向对象式程序，也能用 C++并发出面向功能的程序，但面向对象式程序设计和面向功能的程序设计的基本区别在于程序员必须从围绕数据类型及其交互作用的编程框中解脱出来，如果不能深刻地理解这一点就会陷入困境。

另外，人们在学习使用 C++的时候可能会出现下面一些问题：

学习者畏于 C++的复杂性而放弃学习了，这时他会问：C++能够做到的 C 又怎么做不到呢？这是由于他没有更好地理解"对象"这一概念所引起的。

学习者使用 C++的新特性编出了一些有趣的但颇难理解的程序，例如嵌入(inline)函数、操作符重载和局部声明的使用，但并没有定义有实际用途的类。

另外一些人在读过了一些有关 C++的著作，经过了艰苦的实验之后，终于理解了面向对象式程序设计的精髓所在。

每一个 C++学习者由于各自的态度不同，必然成为上面三种人中的一种。如果希望 C++能够自动地处理用户定义的对象，就会产生混淆；如果只是将 C++看成是 C 语言的增强性集合，也会导致程序中只是应用了 C++的一些特性而没有真正地理解其含义。但只要学习者有耐心，能够积极地开动大脑，并改变编写程序所使用的老路子，就一定能发现通往 C++的成功之路。

1.3 基本概念

重新定义术语能够建立新的规则，也使它充满神秘感并难于理解，定义良好的术语将有助于描述某些事情，但新的思想也需要用新的术语将它同旧的术语区分开来，只是这些术语给初学者带来了一些困难。

让我们先从一个简单的代码段开始讨论：

```
double  a, b, c, s;
  b = 1.0;
  c = 2.0;
  a = b + c;
  s = sqrt(a);
```

其中，double 类型是对浮点数的抽象表示，程序员在使用它的时候不必关心其格式，编译程序能够处理与 double 型有关的细节，这就简化了程序员的工作。

这种抽象并没有停止在数据类型一层上，函数抽象(如上例中的 sqrt 函数)为确定浮点类型使的平方根提供了"黑箱"机制，多数程序员不需要了解怎样求浮点类型使的平方根，只需要引用 sqrt 来完成这一工作。

C++允许编程者创建新的抽象，类定义了与一系列有关的程序成份的接口，通过这一接口(这种情况下使用的是 public 函数)，用户就能同所定义的抽象进行交互，而不必关心这种抽象内部是怎样工作的．就像使用 double 和 sqrt 能够更为方便地处理浮点型数一样，用户在设计类时也应考虑到它将能够简化抽象的使用．

关键字 double 能够识别一个抽象，当程序员在程序中声明一个 double 类型时，就创建了一个对象，该对象具有 double 类型所具有的特性．

对象通常作为计算机模拟思维，表示真实世界的抽象．一个对象像一个软件构造块，它包含了数据结构和提供相关的行为（操作）．对象本身可为用户提供一系列服务——可以改变对象状态、测试、传递消息等等，用户无需知道服务的任何实现细节．操作完全是封闭的．

抽象数据类型是面向对象程序设计的中心概念之一．一个抽象数据类型是一个模型，此模型包含一个类型和与之相关的操作集．定义这些操作集的是基本类型的行为．

内部数据类型如 double 和 int 是 C++本身所带的，它们在任何时候、任何地方都能使用，其特性是预定义的并且不能改变．

对象一般来说与结构一样，是相当复杂的．像处理结构一样，编译程序并未提前知晓对象含有什么内容或如何处理．即，编译程序对对象类型一无所知．

在 C++中，给出对象的类型完全是程序员的事．程序员必须确定对象中含有什么内容，以及如何处理数据．在 C++中，这样的用户定义类型称为类．类是 C++用于实现抽象数据类型的方法．

C++提供数据抽象（相对复杂数据的封装）以及抽象数据类型（用户定义类型或类)．

C++中的新型结构是类．其说明语法类似 C 语言中的 struct．

不要将类和对象仅仅作为一种数据类型．例如：PASCAL 允许用户在外层外壳函数中定义几个有关的函数，外壳为一进程提供一个入口点，内部函数能够执行这一进程，并能引用其中的数据类型，实际上是在一个程序内又创建了一个程序．

在 C++中，程序员能够使用类来完成上述内容，即创建一个类，类中私有函数供内部使用，公其(public)函数提供了入口点，各种函数所分享的数据可以是类定义的一部分．同 PASCAL 一样，C++可用来产生自包含(self-contained)进程．

封装定义为:

1. 所有对象内部软件范围具有清晰的边界;

2. 描述该对象与其他对象如何相互作用的一个接口．

3. 受保护的内部实现．该实现给出了软件对象提供的功能的细节，实现细节不能在定义该对象的类的范围外进行访问．

封装概念不仅涉及到类的描述，而且如何将问题解的各个组件组装在一起的求精过程．封装的单位是对象，该对象的特性由它自己的类说明来描述．这些特性为相同类的其他对象所共享，对象的封装比起一个类表示的封装更具体化．有了封装这个定义，一个类的每个实例（instance）在一个问题求解中是一个独立的封装，或称作组件（问题解的分量)．

例如：类 complex 定义了复数类型的数据结构，同时定义了能够对 complex 值进行

操作的函数，并使这些函数成为 complex 定义的一部分。

在日常生活中，我们经常用到封装：由多个部件构成的汽车通常视为一个实体。实际上，汽车也包含几个封装，"发动机"和"车闸"是由几个部分构成的系统，这些部分共同完成同一任务。封装与抽象结合在一起，允许程序员将某一较大范围的各成份组合在一起，形成单一的、简化的模型。

继承允许我们根据现有类来创建新类，例如：Sortable Array 类从 Array 类继承来特性，Array 类定义了数组的核心特征，Sortable Array 类将它本身的特性加进去以支持对数组的排序。依次地，IntArray 类由 Sortable Array 类派生(即继承后者)，它创建了 int 型的可排序数组。Array 类定义了所有数组的公共特性；Sortable Array 加进本身的功能对数组排序；Int Array 类定义了一特殊类型的数组，这样，使用继承性就建立了相关类的层次结构。

继承还允许多态性(polymorphism)，Poly 是许多的意思，morphus 意即采用一格式，这二者合起来的含义是：可采用多种形式的能力。在面向对象的程序设计中，多态数据类型就是能有几种变体的类型，并可用同样的方式看待它们。例如：Array 类就具有多态性，它所派生的任何类都可看成是类属 Array，允许程序以类属的方式处理 Array 的各种类型。

在后面创建类的过程中，作者将指出在何地、为什么要使用封装、继承性和多态性。

1.4 仔细认真地使用 C++

一直有这样一种说法：C 语言本身的灵活性给程序员套上了能够致命的绳索，幸运的是，好的程序员知道怎样避免使绳索绕到自己的脖子上，经过实践，他们知道了在 C 中哪些可以做而哪些不可以做。

C++所允许的不可预知的表达式灵活性要求我们小心谨慎并彻底了解它。C++不只是提供了一根绳索，它还将用户置于陷井的大门口，但避免的方法也就在眼前。这种说法有点夸张但也不言过其实。

当程序员熟悉了 C++之后，情况是不会这么糟的。我们早已知道，独特的编程技巧并不是一种好的程序设计习惯，多数 C++程序员只用它编写了几个月、最多是几年的程序，一般人还不能够区分哪些是好的、哪些是坏的。

1.4.1 C++的前景

每隔几年，就会有一种新的程序设计技术问世，并有希望使软件开发更为容易，而且，提倡者总是向我们许诺它将有很好的发展前景，问题是，确实是这样吗？

在推广面向对象的程序设计语言时，我们也听到这样的许诺：程序的构造将更为容易，原因是，可以将现存成份快速、方便地链接在一起。使用对象能够简化程序的调试过程，因为问题可以在给定的数据类型中单独解决，而且，从现有的构件派生出新的程序只需在以后进行一些变化。

上述观点在理论上是成立的，但经过多年实践后发现：面向对象式程序设计所提供的额外益处只能建立在开发者所设计和开发的软件是在经过精心构思的基础上的，这就要花

费更多的时间来分析应用程序、识别其成份、找出各成份之间的关系并创建类网。

许多 C 程序员只是将程序堆积在一起，主要原因是缺乏充足的设计时间和设计机制，C 程序员习惯于在终端上编写程序，没有对设计进行充分的考虑。程度员需要清楚地知道：要使用的是什么样的数据类型、怎样使用它们，它们之间的关系如何。如果在我们的工作环境中很难得到程序的规格说明，或者开发是在匆忙之中，就最好不要使用 C++ 语言。

但是，C++所提供的环境还是很诱人的，它有许多不易被察觉的特性，以致用户即使花费几年时间来开发 C++程序也没有完全弄懂它。一方面，C++功能强大，也很复杂；另一方面，复杂性也增大了出错的概率。

以上所述并不是说 C++有什么错，它只提供我们要小心谨慎地使用它，但也不要因为某些权威人士告诫大家哪些应当做，哪些不该做而缩手缩脚，不断尝试，权衡利弊，才能真正掌握 C++。

1.4.2 继承中会出现的蠢事

在进行面向对象的程序设计时，首先要考虑的问题是：怎样组织对象类，这也是最重要的一点，如果组织错误在后期才发现，那将是很致命的，例如：发现某处需要的只是一个分枝而不是一个类。就像对一棵真正的树一样，我们不能砍掉并移动分枝，或改变它在树上的附着点。由于这个原因，在补进新类时，类端承常常会蜕化成未来的聚集(thicket)，要想解决这个问题，可以产生多条祖先路径，实现多重端承。

当我们用多重继承的方法创建一个新类的时候，一定要搞清楚那些被结合在一起的类祖先是什么。

1.4.3 单树形结构

一个很大的层次结构就像一截弹簧，一小段弹簧掉在地上能够很容易地识别其首尾所在，若弹簧很长，要找到它的两端恐怕就不容易了。如果弹簧很长，沿着它的一头搜索下去直到找到它的另一头就很烦人，这就是复杂类结构不易理解的原因，尤其是在读别人写的源代码或者是自己很久以前所写程序的时候。

什么时候需要一数据项来识别它自己？是否每一对象都要支持 I／O 特性？类属数据结构对应用程序来说足够用了吗？使用层次化的虚函数需要多大的开销呢？要解决这些问题，面向对象的方法并不是唯一的途径。C++的最佳特性是允许使用以往所用的一般 C 语言，必要的时候使用 C++的对象，只有必要的时候才加进 C++成份，而不是说要尽可能地加入。

1.4.4 认真构思、反复修改

不知读者注意到没有，在编写软件的时候，过一段时间后我们会发现：现在比以前做得更好，尤其在进行一个大项目的时候，有时还会发现原来的设想是错误的，这时将会面临一种选择，或者进行大幅度修改，或者勉强进行下去，不管走哪条路，结果都不会十分令人满意，程序往往在第二次编写的时候才会工作得更好。

继承和多态性都不能掩盖错误的设计，权威人士总是说从已有类派生出新的类会简化

程序设计，这往往言过其实．试设想要建造一座房屋，总体结构设计已经做好，突然发现地基有问题，这时不管怎样对地基进行修补，整个建筑还是有潜在缺陷的．同样的道理也适用于类结构，如果基类有缺陷，派生类尽管能够隐藏它，却不能修补它．

程序常常是在第二次、第三次构思的时候才会更全面、具体，在实际生活中，我们往往没有足够的时间来计划预先设计软件中遇到的所有可能情况．尽管我们由于种种原因(时间啦、经费啦、或者是程序本身就没有这个价值)一般不能重写现存的代码，但只要挪出一部分时间来重新建造(rebuild)，所得程序就会更好些．

C++的程序设计风格不是随意性的，如果在用户的工作环境中很难得到程序的规格说明，或者时间很紧迫，就不要采用C++语言．

1.5 软件成份

计算机是由几大模块构成的，一旦硬件被制造出来，软件就能操纵它并指示如何完成任务．有些热衷于面向对象程序设计的人提出：软件可以由一些软件成份构造出来，就像计算机能够由标准芯片和电路板构成一样．他们还说，这样做的结果将开辟软件生产的新纪元，编程者只需将各成份连接起来就能形成新的应用程序．

这种假定当然是说软件如同硬件．对已给定的硬件(如计算机)一定有CPU、I/O端口、指定数目的扩展槽，一旦一硬件元件被选好，就几乎不能对它进行大的改动，例如：我们想在当前使用的PC机上加进一新的扩展槽，就会发现改变主板是很难的，加进扩展槽所需的工作量与其价值相比是不相应的，我们会面临两种选择：更换现有扩展卡，或者将就使用老的卡．

要想改变硬件的工作方式是很困难的，对于PC用户来说，ROM BIOS是固定的，更换成其它的BIOS也必须以类似的方式进行工作，一旦硬件做成或者软件固化进芯片中，再进行改变就很困难了．

如果离散的软件功能能被封装进成份(component)中，我们就能像制造汽车或计算机一样来制造软件了，编写程序的工作就变成：开发固定的网络和不可改变的软件芯片(IC)．软件和硬件的唯一区别是：软件比硬件更容易重新组合．这样做的结果是减少了创造性，用软件芯片进行程序设计就像用积木进行建筑一样做动人心．坏的情况是，软件芯片的出现使程序员的第一重要位置从软件业中消失了，而没有人类的参与，程序就会失去生命力、缺少创造性和天才的火花，组装线式软件对有些人来说可能有用，但不提倡这样做的．

一种更好的设计方法是创建灵活的、可扩充的对象，灵活性允许对对象进行修改，可扩充性即是将对象作为其它对象的基础．在本书的后面章节中将发现：Array类试图使用灵活的结构来提供灵活的扩充．

1.6 有关类的更多讨论

两个类的关系可以为以下形式之一或多种形式的结合：

• is a 关系：类B定义为类A的变体，类B的主要特性从类A继承来，而且常常是：通

过指向类 A 对象的指针来引用类 B 对象，实现多态性．例如，定义 int 型数组的类可以从另外一类派生出来，该类指定了所有数组类型的公共特性．

- modifies关系：类B扩充或定义了类A的功能．例如：SortableArray类(第七章)将排序功能加进了 Array 类(第六章)中．
- made of 关系：类A对象为类B的成份，这时，类A定义了传统的数据类型，例如：整数或复数，类 B 定义的对象类型包含类 A 的对象．
- uses关系：类A对象是类B对象使用的工具，例如：一个对象使用另一文件对象来存贮数据．

程序员必须知道程序中类的关系以确定类的类型和交互，这就需要常常查询完整的规格说明书，如果对类的描述不清楚，就会使程序难于理解和维护．

本书随后的几章中将给出一些类集合，当涉及到类的设计问题时，还会引用到上面所列出的关系．

第二章　怎样从 C 过渡到 C++

本章主要是为原来不懂 C++，但熟悉 C 语言的程序员编写的，目的就是使他们能很快地了解 C++，进而掌握 C++。对于有一定 C++基础的读者可以跳过这章而直接阅读第三章。

2.1　从 C 快速进入 C++

虽然对面向对象程序设计是否支持是 C 和 C++之间最重要的差别，但在较小范围内也可用 C++增强 C，因此我们先要了解 C++的非面向对象的特点。

1. 注释行

C++的注释行是用行限定符／／结束，而 C 中注释行括号／＊和＊／仍然适用。

2. 枚举名

一个枚举名是类型名，在 C++中不必在枚举类型名前使用限定词 enum。

3. 结构体或类名

一个结构体或类名是类型名，在 C++中结构体或类名前使用限定词 struct 或 class 是不必要的。在 C 中不存在类的构造。

4. 在块（分程序）内说明

C++允许在块（分程序）内和代码语句之后出现说明，这是允许程序员说明一个实体，此实体紧挨着它的第一次使用处被说明。也允许在循环内说明一个下标，如：

```
for (int i=0; i<12; i++)
  ...
```

5. 作用域限定运算符

新的运算符::用于解决名字冲突。例如，如果自动变量 point__count 在函数内说明并存在一个全程变量 point__count，则运算符::point__count 允许自动变量 point__sum 的作用域内存取全局变量。在这种意义上来说，::运算符用于存取一个被隐藏在当前作用域内的一个项。请看下述例子：

```
#include "stream.hpp"
int point__count;
int main( )
  {
  float point__count;
  point__count=1.5;
  ::point__count=2;
  cout<<"local point__count="<<point__count<<"\n";
  cout<<"global point__count="<<::point__count<<"\n";
```

```
}
```

这样，::运算符要求"不要用于局部，而要用于此范围外部说明!"。上面程序将显示:

```
local point__count=1.5
global point__count=2
```

这个作用域限定运算符，总是在类方法函数定义中用来说明给定方法的自己的类；这个作用域限定符主要可用于区分具有同名的基本类的成员。

逆过程是不成立的。即在自动变量作用域外存取自动变量 point__count 是不可能的，作用域限定符::总是与类相联系而使用。

6. const 说明符

const 说明符在实体作用域范围内可冻结一个实体值。它能冻结用一个指针变量指向的数据、指针地址的值、或者指针地址和指向的数据两者的值。

一个函数的参数能用 const 说明，即可冻结函数内的该参数值。

7. 无名共用体

没有名字的共用体（union）可在结构内定义，他们允许为节省内存区共享两个或更多的结构体域内存空间。

8. 显式类型转换

预先定义类型或程序员定义类型名字，都能够作为一个函数使用，以便将数据从一种类型强制转换到另一种类型。

9. 函数原型

C++在函数接口的风格上接近于 PASCAL, Modula__2 和 Ada，它允许程序员指定每个函数参数（在函数名之后，括号内）的类型和名字，一个函数原型的例子如下:

```
float point__count (float point[ ],int size)
  {
   ...
  }
```

相当于 C 的接口应是

```
float point__count (point, size)
 float point[ ],int size;
  {
   ...
  }
```

C++编译程序执行类型检测，以确保在这个函数被调用时，传送到函数内的参数个数和值的类型与函数定义的参数的个数与类型匹配，这种检测使返回类型与引用的函数表达式的变量相匹配的。这种合乎需要的参数检测机制在大多数 C 语言中是没有的。

10. 函数名重载

C++函数可使用同名（在同样作用域内），如果程序员指定了重载，即每个重载函数能识别它们的参数类型和个数。

下面举例说明函数名重载问题。

在 C 语言中，函数名必须唯一，即

```
int abs (int i);
long labs (long l);
double fabs (double d);
```

在 C++中，允许用相同的函数名（称其为重载）代替上面说明，即

```
int abs (int i);
long abs (long l);
double abs (double d);
```

通过名字分割机制（name mangling），C++编译程序能精确地确定 abs 函数对给定调用的实现。例如，

```
abs (-100); // calls int abs (int i)
abs (-1000000); // calls long abs (long l)
abs (-14.356); // calls double abs (double d)
```

C++用函数名重载处理上述过程不会产生二义性。此外，C++编译器调用的实现提供方便的一系列转换：

```
abs ('b'); //calls int abs (int i)
abs (3.1415F); //calls double abs (double d);
```

11. 函数参数的缺省值

在 C++函数中一串参数能指定缺省值。基于这种情况，函数能用少于全部参数个数的参数来引用。任何一串遗漏的参数均可假设为缺省值。

12. 具有不确定参数个数的函数

使用省略号…，C++函数能指定不确定参数个数和参数类型的函数，这个特征能消除参数类型的检测并允许对函数接口的灵活性。

13. 函数中引用参数

C++允许函数的形参用&操作符作为引用参数说明，如：

```
void addition (int & count)
  {
   count++;
  }
int j;
addition (j);
```

因为 count 作为引用参数定义。当 addition 引用时，参数 count 的地址是由 j 的地址指定。在函数 addition 内，送入 j 的值是递增的而返回到变量 j 的函数 addition 之外。像 C 语言中，显式传送到函数 addition 的 j 的地址是不必要的。

引用变量在 C++中可使用更一般的模式，即为程序员建立类型使用特殊的操作。

14. inline 说明符