C++ 迁移指南

Sun™ Studio 8

Sun Microsystems, Inc.
www.sun.com

部件号码 817-5806-10
2004 年 4 月，修订 A

如对本文档有任何意见，请将电子邮件发送到：http://www.sun.com/hwdocs/feedback

请回收

Adobe PostScript

目录

表

编码示例

开始之前必须了解的事项

本书介绍您从 4.0、4.0.1、4.1 或 4.2 版的 C++ 编译器中迁移代码而需要了解的事项。如果您要从更早的 3.0 或 3.0.1 版 C++ 编译器中迁移代码，这些信息也同样适用。同时，也涉及到与这些早期的编译器版本有关的其他一些主题。本手册的读者是具有 C++ 实践知识并且了解 Solaris™ 操作环境和 UNIX® 命令的程序员。

印刷惯例

表 P-1 字样惯例

字样	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑您的 `.login` 文件。 使用 `ls -a` 列出所有文件。 `% You have mail.`
`AaBbCc123`	您键入的内容，与计算机屏幕输出对比时	`% su` `Password:`
AaBbCc123	书名、新字或术语、要强调的字	阅读《*用户指南*》第 6 章。 这些称为*类*选项。 您*必须*是超级用户才能这样做。
AaBbCc123	命令行占位符文字，将用实际名称或值替代	要删除文件，请键入 **`rm`** *filename*。

表 P-2 代码惯例

代码符号	含义	表示法	代码示例
[ ]	方括号包含可选的参数。	O[*n*]	O4, O
{ }	花括号包含一个必需选项的一组选项。	d{y|n}	dy
|	“管道”或“条形”符号分隔参数，只能选择其中一个参数。	B{dynamic|static}	Bstatic
:	冒号类似逗号，有时用来分隔参数。	R*dir*[:*dir*]	R/local/libs:/U/a
…	省略号表示数列中的省略。	xinline=*fl*[,…*fn*]	xinline=alpha,dos

Shell 提示符

Shell	提示符
C shell	*machine-name*%
C shell 超级用户	*machine-name*#
Bourne shell 和 Korn shell	$
Bourne shell 和 Korn shell 的超级用户	#

访问 Sun Studio 软件和手册页

编译器和工具及其手册页未安装到标准的 /usr/bin/ 和 /usr/share/man 目录中。要访问编译器和工具，必须正确设置 PATH 环境变量（请参见第 xiii 页上的“访问编译器和工具”）。要访问手册页，必须正确设置 MANPATH 环境变量（请参见第 xiv 页上的“访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、 sh(1) 和 ksh(1) 手册页。有关 MANPATH 变量的详细信息，请参见 man(1) 手册页。有关设置 PATH 变量和 MANPATH 变量以访问此发行版的详细信息，请参见安装指南或与系统管理员联系。

注 – 本节所含信息假定您的 Sun Studio 编译器和工具安装在 /opt 目录中。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

访问编译器和工具

按照下面的步骤确定您是否需要更改您的 PATH 变量以访问编译器和工具。

▼ 确定您是否需要设置您的 PATH 环境变量

1. **通过在命令提示符后面键入以下内容，显示 PATH 变量的当前值。**

```
% echo $PATH
```

2. **检查输出以查找包含 /opt/SUNWspro/bin/ 的路径的字符串。**

   如果您找到路径，您的 PATH 变量已设置为可访问编译器和工具。如果您没有找到路径，请按照下一过程中的指令设置 PATH 环境变量。

▼ 要设置您的 PATH 环境变量以便能够访问编译器和工具

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 PATH 环境变量中。如果安装了 Sun ONE Studio 软件或 Forte Developer 软件，请在其安装路径前面增加以下路径。**

   /opt/SUNWspro/bin

访问手册页

按照下列步骤确定您是否需要更改您的 MANPATH 变量以访问手册页。

▼ 确定您是否需要设置您的 MANPATH 环境变量

1. **通过在命令提示符后面键入以下内容，请求 dbx 手册页。**

```
% man dbx
```

2. **检查输出 （如果有）。**

   如果无法找到 dbx(1) 手册页，或者显示的手册页不适用于所安装软件的当前版本，请按照下一过程中的指令设置您的 MANPATH 环境变量。

▼ 设置您的 MANPATH 环境变量以便能够访问手册页

1. **如果您正在使用 C shell，请编辑您的起始 .cshrc 文件。如果您正在使用 Bourne shell 或 Korn shell，请编辑您的起始 .profile 文件。**

2. **将以下内容增加到您的 MANPATH 环境变量中。**

   /opt/SUNWspro/man

访问集成开发环境

Sun Studio 8 集成开发环境 (IDE) 提供用于创建、编辑、生成、调试和分析 C、 C++ 或 Fortran 应用程序性能的模块。

IDE 需要 Sun Studio 8 的核心平台组件。如果核心平台组件未安装，或者安装到以下某个位置，您必须将 SPRO_NETBEANS_HOME 环境变量设置为核心平台组件所安装的位置 (*installation_directory*/netbeans/3.5R)：

- 缺省安装目录 /opt/netbeans/3.5R
- Sun Studio 8 的编译器和工具组件的同一位置 （例如，编译器和工具组件安装在 /foo/SUNWspro 中，核心平台组件安装在 /foo/netbeans/3.5R 中）

用于启动 IDE 的命令是 sunstudio。有关此命令的详细信息，参见 sunstudio(1) 手册页。

访问编译器和工具文档

您可以从以下位置访问该文档：

- 该文档可通过随软件安装在您的本地系统或网络以下位置的文档索引获得 `file:/opt/SUNWspro/docs/index.html`。

  如果软件未安装在 `/opt` 目录中，请询问系统管理员以了解系统中的等价路径。

- 大多数手册可从 `docs.sun.com`[sm] web 站点获得。以下手册只能通过安装的软件获得：
  - *标准 C++ 库类参考*
  - *标准 C++ 库用户指南*
  - *Tools.h++ 类库参考*
  - *Tools.h++ 用户指南*
- 发行说明可从 `docs.sun.com` web 站点获得。
- 在 IDE 中， IDE 的所有组件的联机帮助可通过 [帮助] 菜单获得，也可通过许多窗口和对话框上的 [帮助] 按钮获得。

`docs.sun.com` web 站点 (`http://docs.sun.com`) 使您能够通过国际互联网阅读、打印和购买 Sun Microsystems 手册。如果您无法找到某本手册，请查看随软件安装在您的本地系统或网络中的文档索引。

注 – Sun 对于本文档中提到的第三方 web 站点的可用性概不负责，并且未对此类站点或资源上或从中获得的任何内容、广告、产品或其他资料作任何保证且概不负责。对于因为使用通过任何此类站点或资源获得的任何此类内容、商品或服务而造成或宣称造成的直接或间接损害或损失， Sun 概不负责。

可访问格式的文档

该文档以可访问格式提供，残疾人用户可通过辅助技术进行阅读。您可以找到下表所述文档的可访问版本。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

文档类型	可访问版本的格式和位置
手册 （第三方手册除外）	HTML，在 http://docs.sun.com 上
第三方手册： • *标准 C++ 库类参考* • *标准 C++ 库用户指南* • *Tools.h++ 类库参考* • *Tools.h++ 用户指南*	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
自述文件和手册页	HTML，在安装的软件中，通过位于 file:/opt/SUNWspro/docs/index.html 的文档索引访问
联机帮助	HTML，可通过 IDE 中的 [帮助] 菜单访问
发行说明	HTML，在 http://docs.sun.com 上

相关的编译器和工具文档

下表描述了可从 file:/opt/SUNWspro/docs/index.html 和 http://docs.sun.com 获得的相关文档。如果软件未安装在 /opt 目录中，请询问系统管理员以了解系统中的等价路径。

文档标题	描述
数值计算指南	描述关于浮点计算的数值准确性的问题。

访问相关的 Solaris 文档

下表描述了可通过 `docs.sun.com` web 站点获得的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	参见手册页部分的标题。	提供有关 Solaris 操作环境的信息。
Solaris 软件开发者集合	*链接程序和库指南*	描述了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发者集合	*多线程编程指南*	包括 POSIX 和 Solaris 线程 API、使用同步对象编程、编译多线程程序以及查找多线程程序的工具。

市面有售的参考书

下面列出了一部分 C++ 语言方面的书籍。

《*The C++ Programming Language*》第三版， Bjarne Stroustrup 编著 （Addison-Wesley 1997 年出版）。

《*The C++ Standard Library*》， Nicolai Josuttis 编著 （Addison-Wesley 1999 年出版）。

《*Generic Programming and the STL*， Matthew Austern 编著 （Addison-Wesley 1999 年出版）。

《*Standard C++ IOStreams and Locales*》 Angelika Langer 和 Klaus Kreft 编著 （Addison-Wesley 2000 年出版）。

《*Thinking in C++, Volume 1, Second Edition*》， Bruce Eckel 编著 （Prentice Hall 2000 年出版）。

《*The Annotated C++ Reference Manual*》， Margaret A. Ellis 和 Bjarne Stroustrup 编著 （Addison-Wesley 1990 年出版）

《*Design Patterns: Elements of Reusable Object-Oriented Software*》， Erich Gamma、Richard Helm、 Ralph Johnson 和 John Vlissides 编著 （Addison-Wesley 1995 年出版）。

《*C++ Primer*》第三版， Stanley B. Lippman 和 Josee Lajoie 编著 （Addison-Wesley 1998 年出版）。

《*Effective C++—50 Ways to Improve Your Programs and Designs*》第二版， Scott Meyers 编著 （Addison-Wesley 1998 年出版）。

《*More Effective C++—35 Ways to Improve Your Programs and Designs*》， Scott Meyers 编著 （Addison-Wesley 1996 年出版）。

《*Efficient C++: Performance Programming Techniques*》， Dov Bulka 和 David Mayhew 编著 （Addison-Wesley 2000 年出版）。

开发者资源

访问 `http://developers.sun.com/prodtech/cc` 以查找以下经常更新的资源：

- 关于编程技巧和最佳实践的文章
- 包含简短编程提示的知识库
- 编译器和工具组件的文档以及随软件安装的文档的校正
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以在 `http://developers.sun.com` 上找到适用于开发者的更多资源。

联系 Sun 技术支持

如果您有关于本产品的技术问题，而本文档中未涉及，请访问：

`http://www.sun.com/service/contacting`

发送您的意见

Sun 乐于改进文档并欢迎您发表意见和建议。请通过以下地址将您的意见 Email 给 Sun：

`docfeedback@sun.com`

请在您的电子邮件主题行中包含您的文档的部件号码 (817-5806-10)。

第 1 章

简介

在本书中，C++ 4.0、4.0.1、4.1 和 4.2 编译器统称为“C++ 4”，而 C++ 5.0、5.1、5.2、5.3、5.4 和 5.5 编译器统称为“C++ 5”。绝大多数使用 C++ 4 编译器编译并运行的 C++ 源代码在 C++ 5 编译器下都可以继续使用，少数不能使用是由于 C++ 语言定义发生了更改。编译器提供兼容模式 (`-compat[=4]`)，使得几乎所有的 C++ 4 代码不作改动即可继续使用。

注 – 在标准模式（缺省模式）下通过 5.0 版、5.1 版、5.2 版、5.3 版、5.4 版或 5.5 版 C++ 编译器编译的对象代码与通过版本更旧的编译器编译的 C++ 代码不兼容。有时，您可以将旧对象代码自带的库用于 5.0、5.1、5.2、5.3、5.4 和 5.5 版的编译器。第 1-3 页上的第 1.3 节 “二进制兼容问题”中对此作了详细说明。

1.1 C++ 语言

Bjarne Stroustrup 编著的 《*The C++ Programming Language*》（1986 版）一书中第一次介绍了 C++ 语言，后来 Margaret Ellis 和 Bjarne Stroustrup 合著的 《*The Annotated C++ Reference Manual*》（简称 ARM）（1990 版）中更加正式地介绍了 C++ 语言。Sun C++ 4 编译器版本主要基于 ARM 中的定义，并且根据以后出现的 C++ 标准增加了一些定义。选中要增加到 C++ 4 中，特别是 C++ 4.2 编译器中的定义主要是那些不会引起源代码不兼容或二进制不兼容的定义。

C++ 现在是国际标准 ISO/IEC 14882:1998 *编程语言 — C++* 的主题。标准模式下的 C++ 5.4 编译器几乎实现了标准中规定的所有语言。当前发行版本中提供的自述文件介绍了与标准中的要求有哪些出入。

C++ 语言定义中的一些更改导致必须对旧的源代码做一些轻微改动才能进行编译。最明显的例子就是整个 C++ 标准库都是在名称空间 std 中定义的。传统的第一个 C++ 程序

```
#include <iostream.h>
int main() { cout << "Hello, world!" << endl; }
```

在严格符合定义的编译器下无法再进行编译，因为头的标准名称现在是 <iostream>（不带 .h），并且名称 cout 和 endl 位于名称空间 std 中，而不是位于全局名称空间中。作为一项扩展， C++ 编译器提供了头 <iostream.h>，这样甚至可以在标准模式下编译程序。除了需要更改源代码之外，这些语言更改还导致二进制不兼容，因此， 5.0 版之前的 C++ 编译器没有做这些更改。

一些更新的 C++ 语言特征还需要更改程序的二进制表示。第 1-3 页上的第 1.3 节 “二进制兼容问题”中详细介绍了该主题。

1.2 编译器的操作模式

C++ 编译器有二种操作模式：一种是*标准*模式，另一种是*兼容*模式。

1.2.1 标准模式

标准模式实现了大多数 C++ 国际标准，并且与 C++ 4 接受的语言在源代码方面有一些不兼容，这一点在前面已做了说明。

更为重要的是，在标准模式下 C++ 5 编译器使用了与 C++ 4 不相同的应用程序二进制接口 (ABI)。在标准模式下编译器生成的代码通常与各种 C++ 4 编译器生成的代码不兼容，而且两者也不能相互链接。第 1-3 页上的第 1.3 节 “二进制兼容问题”中更详细地介绍了该主题。

由于以下几种原因，您应该更新代码，以便在标准模式下编译：

- 兼容模式不适用于 64 位程序。
- 在兼容模式下您不能使用重要的标准 C++ 特征。
- 写入 C++ 标准的新代码在兼容模式下可能无法编译，这意味着您不能将后来的新代码导入应用程序。
- 由于您不能将 4.2 版的 C++ 代码与标准模式的 C++ 代码链接在一起，可能需要维护两种版本的对象库。
- 以后永远不会支持兼容模式。

1.2.2 兼容模式

为了提供一种从 C++ 4 迁移到标准模式的途径，编译器提供了兼容模式 (`-compat[=4]`)。兼容模式与 C++ 4 编译器在二进制方面完全兼容，并且大多数源代码也兼容。（*兼容*意味着*向上兼容*。旧的源代码和二进制代码能够用新的编译器编译，但是您不能指望专门为新编译器设计的代码也能够用旧的编译器编译。）兼容模式与标准模式在二进制方面不兼容。兼容模式适用于在 IA 和 SPARC 平台上运行的 Solaris 7 和 Solaris 8 操作环境，但是不适用于 SPARC V9 （64 位）处理器。

注 – 在本文档中，“IA”一词指 Intel 32 位处理器体系结构 （包括 Pentium、 Pentium Pro、 Pentium II、 Pentium II Xeon、 Celeron、 Pentium III 和 Pentium III Xeon 处理器）以及由 AMD 和 Cyrix 生产的兼容微处理器芯片。

使用兼容模式的原因：

- 您有通过 C++ 4 编译器编译的 C++ 对象库，但是您不能在标准模式下重新编译。（例如，您没有源代码。）
- 您需要立即交付产品，但是源代码不能在标准模式下进行编译。

注 – 在大多数条件下，您不能将在兼容模式下 (`-compat[=4]`) 编译的对象文件和对象库与在标准模式 （缺省模式）下编译的对象文件和库链接在一起。有关详细信息，请参见第 1-4 页上的第 1.4 节 “将旧的二进制与新的二进制混合”。

1.3 二进制兼容问题

*应用程序二进制接口*或 ABI 定义了编译器产生的对象程序所具备的机器级特点。它包括基本类型的大小和排列要求、结构化类型和聚集类型的布局、函数的调用方式、程序中定义的实体的实际名称以及其他许多特征。 Solaris 操作环境的许多 C++ ABI 与基本的 Solaris ABI （即 C 语言的 ABI）相同。

1.3.1 语言更改

C++ 语言引入了许多功能 （例如，类成员函数、重载函数和运算符、类型安全链接、异常和模板），这些功能在 C 语言的 ABI 中没有对应的功能。 C++ 的每一个重大的新版本都增加了一些语言功能，而这些功能无法通过旧版本的 ABI 实现。对 ABI 进行必要的更改涉及到类对象的排列方式、某些函数的调用方式以及类型安全链接 （“名字粉碎”）的实现方式。

C++ 4.0 编译器实现了 ARM 定义的语言。到 C++ 4.2 编译器发行之时，C++ 委员会已经引入了许多新的语言功能，有一些功能需要在 ABI 中做出更改。由于迄今未知的语言增加或更改必然需要对 ABI 做其他更改，因此 Sun 选择只实现那些不需要更改 ABI 的新功能。目的是尽可能减少因必须保持通过不同编译器版本编译的二进制文件的对应关系而造成的不便。在发布 C++ 标准时，Sun 设计了一种能够实现全部 C++ 语言的新 ABI。C++ 5 编译器在缺省情况下使用新的 ABI。

更改语言而影响 ABI 的一个实例就是 `new` 和 `delete` 空闲存储函数使用新的名称、签名和语义。另外一个实例就是签名相同的模板函数与非模板函数仍然是不同的函数这一项新的规则。该规则需要更改“名字粉碎”，造成与更旧的编译代码在二进制方面不兼容。类型 `bool` 的引入还造成 ABI 更改，尤其是标准库接口。旧的 ABI 导致无用的低效运行时代码，由于 ABI 需要更改，因此这些方面也得到了改善。

1.4 将旧的二进制与新的二进制混合

您不能将旧的二进制（通过 C++ 4 编译器或在兼容模式下通过 C++ 5 编译器编译的对象文件和库）与新的二进制（在标准模式下通过 C++ 5 编译器编译的对象文件和库）链接在一起，这种说法言过其实。但是，在 SPARC 平台上通过使用 `libExbridge` 库有可能做到这一点。

注 – 只有 SPARC 平台支持使用 `libExbridge` 的混合模式，x86 平台不支持这种模式。要使 `libExbridge` 能够用于 x86 平台，需要更改 ABI 和重新编译代码。如果您可以重新编译代码，不需要使用 `libExbridge`。我们强烈建议您不要在一个程序中混合兼容模式，而是应该在标准模式下编译所有的代码。最好在短期和长期内都能做到这一点。您尝试通过混合模式解决问题，但并不能彻底解决问题，使用 `libExbridge` 也不能长期可靠地解决该问题。

1.4.1 入门指南

在生成或运行应用程序的所有系统上安装最新的 `SUNWlibC` 修补程序。新修补程序中的 `libExbridge.so.1` 库取决于同一个修补程序中 `libC.so.5` 和 `libCrun.so.1` 的版本。因此，如果您只是将 `libExbridge.so.1` 复制到系统中，而没有安装修补程序，修补程序不能工作。

1.4.2 要求

在以下条件下，您可以将旧的二进制与新的二进制（前面已作了规定）进行链接：

1.4.2.1 使用异常

在代码使用异常时，也就是说代码包含 throw 或 catch 关键字 （包括函数的异常说明），要求如下。

- 如果标准模式代码和兼容模式代码同时使用异常，只要从抛出点一直到 （包括）捕捉点的所有活动函数都是在同一个模式下编译的，代码就能使用。换句话说，从抛出点沿着栈向上一直走到捕捉子句，具有捕捉块的所有 C++ 函数都必须在同一个模式下编译。
- 禁止静态地将 libC 库与 libCrun 库链接在一起。您必须将库的共享 (.so) 版本链接在一起，这是编译器的缺省状态。链接 libExbridge.so.1 就会自动将 libCrun.so.1 与 libC.so.5 连在一起。参见第 1-5 页上的 “ 使用 libExbridge 库 ”。
- 您不应该使用 -Bsymbolic 链接程序选项创建共享库，也不应该链接使用 -Bsymbolic 链接程序选项生成的库。

  如果您当前使用的 C++ 编译器版本通过 -xldscope 支持链接程序作用域，使用该功能控制库中符号的可视性。有关详细信息，请参见 《*C++ 语言用户指南*》中的 -xldscope 或 CC(1)。

  如果您当前使用的 C++ 编译器版本不支持链接程序作用域，使用链接程序映射文件控制库中符号的可视性。有关详细信息，参见 《*链接程序和库指南*》。

1.4.2.2 使用 libExbridge 库

使用 libExbridge 的首选方法就是将其与代码链接。为此，将 -lExbridge 选项添加到您的 CC 命令中。 -l 选项将 lib 置于库名称的前面。

如果您不能与 libExbridge 进行链接，请遵循以下说明：

- 如下所示，预装 C shell 的库：

```
example% setenv LD_PRELOAD /usr/lib/libExbrige.so.1
example% my_application arg1 arg2
```

- 如下所示，预装 Bourne 或 Korn shell 的库。请注意，设置 LD_PRELOAD 之后没有分号：

```
$ LD_PRELOAD=/usr/lib/libExbrige.so.1 my_application arg1 arg2
```

一旦设置了 LD_PRELOAD，它会影响以后启动的所有程序。预装 libExbridge 会导致以后调用 shell，或者导致产生 shell 的程序失败。 /usr/bin/sh 定义了自己的 malloc，它在预装库的 .init 段调用时没有正确初始化。

因此，在使用 C shell 时，最好在运行应用程序的 shell 脚本中设置环境变量。显示的 Bourne 和 Korn shell 语法只为同一行中的命令创建环境变量。

如果您预装 libExbridge 并且应用程序产生 shell，应用程序有可能会失败。在那种情况下，您必须使用 libExbridge 重新链接应用程序，而不是预装库。

1.4.3 确定接口的结构

文件和库表示 C 接口。

有时以 C++ 语言编码的库为了方便起见，对外界仍然只表示 C 接口。简而言之，有了 C 接口意味着客户机不能辨别程序是以 C++ 语言编写的。更确切一些，有了 C 接口意味着以下说法都正确：

- 所有外部调用的函数都有 C 链接，并且对于参数和返回值只使用 C 类型。
- 接口中的所有函数指针都有 C 链接，并且对于参数和返回值只使用 C 类型。
- 所有外部可视类型都为 C 类型。
- 所有外部可用对象都属于 C 类型。
- 不允许使用 `cin`, `cout`、 `cerr` 或 `clog`。

如果库符合 C 接口的标准，则只要可以使用 C 库，就可以使用该库。特别是，此类库可以通过一种版本的 C++ 编译器编译，并与通过其他版本的 C++ 编译器编译的对象文件链接，条件是它们没有混合异常处理。

然而，如果违反了这些条件中的其中一项，文件和库无法链接在一起。如果尝试链接成功 （对此持怀疑态度），程序也不能正确运行。

请注意，如果您使用 C 编译器 (`cc`) 将应用程序与 C 接口库链接，并且该库需要 C++ 运行时支持，必须使用以下某种方法，创建与 `libC` （兼容模式）或 `libCrun` （标准模式）的依存关系。如果 C 接口库不需要 C++ 运行时支持，则不需要链接 `libC` 或 `libCrun`。

- **归档的 C 接口库**。在提供归档的 C 接口库时，必须提供如何使用库的说明。
  - **标准模式**。如果在标准模式（缺省模式）下使用 C++ 编译器 (`CC`) 生成 C 接口库，用户在使用 C 接口库时，必须将 `-lCrun` 增加到 `cc` 命令行中。
  - **兼容模式**。如果在兼容模式 (`-compat`) 下使用 C++ 编译器 (`CC`) 生成 C 接口库，用户在使用 C 接口库时，必须将 `-lC` 增加到 `cc` 命令行中。
- **共享的 C 接口库**。在提供共享的 C 接口库时，必须在生成库时创建与 `libC` 或 `libCrun` 的依存关系。在共享的库有了正确的依存关系时，在使用库时不需要将 `-lC` 或 `-lCrun` 增加到 `cc` 命令行中。
  - **标准模式**。如果正在标准模式 （缺省模式）下生成 C 接口库，在生成库时将 `-lCrun` 增加到 `CC` 命令行中。
  - **兼容模式**。如果正在兼容模式 (`-compat`) 下生成 C 接口库，在生成库时将 `-lC` 增加到 `CC` 命令行中。

1.5 条件表达式

C++ 标准给条件表达式的规则带来了变化。只有在象下面的这种表达式中才会看出差别：

```
e ? a : b = c
```

主要的问题是在没有分组括号时冒号后面有赋值。

4.2 版编译器使用原始的 C++ 规则，并且就象您编写表达式那样处理表达式。

```
(e ? a : b) = c
```

也就是说， `c` 被赋值给 `a` 或 `b`，具体视 `e` 的值而定。

现在编译器在兼容模式和标准模式下都使用新的 C++ 规则。它就象您编写表达式那样处理表达式。

```
e ? a : (b = c)
```

也就是说，如果 `e` 为 False 并且也只有在这种情况下， `c` 被赋值给 `b`。

解决方法：始终使用括号来表示代码要表达的含义。这样，可以确保在使用任何一种编译器编译时代码的含义相同。

1.6 函数指针与 `void*`

在 C 语言中，函数指针与 `void*` 之间不存在隐式转换。“如果值相符，” ARM 增加了函数指针与 `void*` 之间的隐式转换。 C++ 4.2 实现了该规则。由于隐式转换导致意外的函数重载行为，并且降低了代码的可移植性，后来从 C++ 语言中删除了隐式转换。此外，在函数指针与 `void*` 之间不再有任何转换，甚至不会有强制类型转换。

编译器现在会针对函数指针与 void* 之间的隐式转换和显式转换发出警告。在标准模式下，编译器在解决重载的函数调用时，不再能够识别此类隐式转换。通过 4.2 编译器编译的此类代码在标准模式下现在会生成错误 （没有匹配的函数）。（编译器在兼容模式下会发出时序错误警告。）如果您的代码依靠隐式转换以便正确地解决重载问题，您需要增加强制类型转换。例如：

```
int g(int);
typedef void (*fptr)();
int f(void*);
int f(fptr);
void foo()
{
    f(g);           // 此行有不同的行为
}
```

如果是 4.2 版编译器，示例代码中的标记行调用 f(void*)。现在，在标准模式下由于没有匹配的函数，您会看到错误消息。您可以增加显式强制类型转换，例如 f((void*)g)，但是，由于代码违反了 C++ 标准，您会看到警告消息。函数指针与 void* 之间的转换适用于所有版本的 Solaris 操作环境，但是并不能在所有平台上移植。

C++ 没有与 void* 相对应的 “通用函数指针”。如果在所有支持的平台上使用 C++ 语言，所有函数指针的大小和表示法必须相同。因此，您可以使用任何一种方便的函数指针类型以容纳任何函数指针的值。这种解决方案对于大多数平台都可以移植。和往常一样，您必须将指针值转为原来的类型，然后再尝试调用指针所指向的函数。参见第 3-16 页上的第 3.11 节 “extern "C" 函数的指针”。

1.7 预见将来的粉碎变化

在一些情形下，对于指向相同条目的声明，编译器不符合 C++ 标准。在这些情形下，您的程序不能获得正确的链接行为。为了避免该问题，请遵守这些规则。在以后的版本解决了粉碎问题时，名字仍将以相同的方式被粉碎。

- 在函数声明中不要使用无关紧要的 const 关键字。

  声明值参数 const 不应该对函数签名或者函数调用方式造成影响，因此，不要将其声明为 const。

```
int f(const int); // const 没有意义，不要使用
int f(int);       // 而要这样
int f(const int i) { ... } // 不要这样
int f(int i) { ... }       // 而要这样
```

- 在任何一个函数声明中，不要同时使用 typedef 及其扩展形式。

```
typedef int int32;
int* foo(int*, int32*); // 不要这样
// 不要在同一个函数声明中同时使用 int* 和 int32*
// 而要始终写入以下一项
int* foo(int*, int*);
int32* foo(int32*, int32*);
```

- typedefs 只适用于属于函数指针的参数或返回类型。

```
void function( void (*)(), void (*)() ); // 不要这样
typedef void (*pvf)();
void function( pvf, pvf ); // 而要这样
```

- 在函数声明中不要使用 const 数组。

```
void function( const int (*)[4] ); // 不要使用这个
```

不幸的是，这种声明没有直接的解决方法。

如果您无法避免受该粉碎问题影响的代码，例如，代码出现在不属于您的头或库中，可以如下面实例所示，使用弱符号以使声明与其定义相等。

```
int cpp_function( int arg ) { return arg; }
#pragma_weak "__1c_missing_mangled_name" = cpp_function
```

在这些声明类型中，您必须使用粉碎的名字版本。

1.7.1 粉碎不当的特征

在您的代码具有第 1-8 页上的第 1.7 节 “预见将来的粉碎变化”中介绍的特点而成为问题区时，编译器不能始终以相同的方式粉碎名字。其特征就是程序链接失败，并且链接程序反映无法找到符号。链接程序错误消息中未粉碎的名字指实际上定义的函数或对象。然而，由于编译器采用不同于符号定义的方式粉碎符号引用，因此链接程序不能使名称相匹配。请看下面的例子：

```
main.cc
--------
int foo(int); // 声明中没有 "const"
int main()
{
    return foo(1);
}

file1.cc
---------
int foo(const int k) // "const" 已增加到参数声明
{
    return k;
}

example% CC main.cc file1.cc
main.cc:
file1.cc:
Undefined                       first referenced
  symbol                           in file
int foo(int)                       main.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

通过检查编译器向对象文件发出的名称，您可以看出失败的原因：

```
% nm main.o | grep foo
[2]     |        0|       0|NOTY |GLOB |0    |UNDEF |__1cDfoo6Fi_i_
% nm file1.o | grep foo
[2]     |     16|     40|FUNC |GLOB |0    |2     |__1cDfoo6Fki_i_
```

在 `main.o` 中，编译器发出对函数 `foo` 的引用，该函数的名字粉碎方式不同于 `file1.o` 中函数 `foo` 定义的名字粉碎方式。按照第 1-8 页上的第 1.7 节 “预见将来的粉碎变化”所述，解决方法是不要在 `foo` 的参数声明中使用 `const`。

使用同样的方式声明 `foo` 和 `const` 参数的程序以及使用同样的方式声明 `foo` 和非常量参数的程序，都可以成功地编译和链接。

如果我们解决了编译器的错误，现在链接的一些程序将会停止链接。例如，假设第三方二进制库包含 `file1.o`。如果我们解决了编译器错误，`foo` 声明不允许程序链接至该库中的 `foo`。如果我们没有解决该错误，您可以声明 `foo` 以及 `const` 参数并成功地链接至库。

幸运的是，与名字粉碎有关的所有已知编译器错误带来了“不可能实现的”粉碎名字。也就是说，无效的粉碎名字绝不会意外地引用其他某些函数或对象的粉碎名字。您始终可以增加其他符号，以解决由于不正确地粉碎名字而造成的问题。《*迁移指南*》对此作了说明。

第 2 章

使用兼容模式

本章介绍如何编译适用于 C++ 4 编译器的代码。

2.1 兼容模式

兼容模式的编译器选项为 （两种版本意义相同）：

```
-compat
-compat=4
```

例如：

```
example% CC -compat -O myfile.cc mylib.a -o myprog
```

在兼容模式下使用 C++ 4 编译器与使用 C++ 5 编译器稍微有一些差别，下面小节介绍了这些差别。

2.2 兼容模式下的关键字

缺省情况下，如果是兼容模式，一些新的 C++ 关键字被识别为关键字，但是您可以如下表所示，使用编译器选项关闭这些大多数关键字。更改源代码以避免关键字比使用编译器选项更有优越性。

表 2-1 兼容模式下的关键字

关键字	禁止关键字的编译器选项
`explicit`	`-features=no%explicit`
`export`	`-features=no%export`
`mutable`	`-features=no%mutable`
`typename`	*无法禁止*

无法禁止关键字 `typename`。在缺省情况下，如果是兼容模式，表 3-1 中介绍的其它新 C++ 关键字将被禁止。

2.3 语言语义

C++ 5 在强制执行一些 C++ 语言规则方面表现更出色。它们对时序错误的控制也更加严格。

如果您使用 C++ 4 编译并且启用时序错误警告，可能会发现一直是无效的代码，但是更早的 C++ 编译器却接受这些代码。将来的编译器版本将停止支持时序错误，这一项策略一直很明确 （在手册中已经有规定）。时序错误主要由以下错误组成：违反访问 （专用、受保护内容）规则、违反类型匹配规则以及将编译器生成的临时变量用作引用参数的目标。

本节接下来讨论以前未强制执行，但现在 C++ 编译器强制执行的规则。

注 – C++ 编译器在兼容模式和标准模式下强制执行这些规则。

2.3.1 拷贝构造函数

在初始化对象或者传递或返回类类型的值时，必须可以访问拷贝构造函数。

```
class T {
  T(const T&); // 专用
public:
  T();
};
T    f1(T t) { return t; }  // 错误，无法返回 T
void f2()    { f1( T() ); } // 错误，无法传递 T
```

解决方法：使拷贝构造函数可以访问。通常，可以任意对其访问。

2.3.2 Static 存储类

static 存储类适用于对象和函数，而不适用于类型。

```
static class C {...};   // 错误，此处不能使用 static 关键字
static class D {...} d; // 正确， d 是 static 关键字
```

解决方法：在该实例中， static 关键字对于类 C 没有含义，应该将其删除。

2.3.3 运算符 new 和 delete

在使用 new 分配对象时，必须可以访问匹配的 operator delete。

```
class T {
    void operator delete(void*); // 专用
  public:
    void* operator new(size_t);
};
T* t = new T;               // 错误，运算符 delete 不能使用
```

解决方法：使 delete 运算符可以访问。通常，可以任意对其访问。

在 delete 表达式中不允许使用计数。

```
delete [5] p; // 错误：应该是 delete [] p;
```

2.3.4 new const

如果您使用 new 分配 const 对象，对象必须经过初始化。

```
const int* ip1 = new const int;     // 错误
const int* ip2 = new const int(3); // 正确
```

2.3.5 条件表达式

C++ 标准给条件表达式的规则带来了变化。 C++ 编译器在标准模式和兼容模式下都使用新的规则。有关详细信息，请参见第 1-7 页上的第 1.5 节 “条件表达式”。

2.3.6 缺省参数值

重载运算符或函数指针不允许有缺省的参数值。

```
T operator+(T t1, T t2 = T(0) );    // 错误
void (*fptr)(int = 3);              // 错误
```

解决方法：您必须以其他某种方式编写代码，有可能需要提供其他函数或函数指针声明。

2.3.7 结尾逗号

函数参数列表中不允许使用结尾逗号。

```
f(int i, int j, ){ ... } // 错误
```

解决方法：删除多余的逗号。

2.3.8 传递 const 和字面值

不允许将 const 或字面值传递给非恒定的引用参数。

```
void f(T&);
extern const T t;
void g() {
    f(t);  // 错误
}
```

解决方法：如果函数不修改自身参数，请更改声明以采用 const 引用 （在本例中为 const T&）。如果函数修改了参数，不能向参数传递 const 或字面值。另外一种方法就是创建一个显式非恒定临时 const 或字面值，然后再传递。有关信息，请参见第 3-9 页上的第 3.7 节 “字符串文字和 char*”。

2.3.9 在函数指针与 void* 之间转换

C++ 编译器在兼容模式和标准模式下现在都会针对函数指针与 void* 之间的隐式或显式转换发出警告。有关详细信息，请参见第 1-7 页上的第 1.6 节 “函数指针与 void*”。

2.3.10 类型 enum

如果对 enum 类型的对象赋值，其值必须具有相同的 enum 类型。

```
enum E { zero=0, one=1 };
E foo(E e)
{
   e = 0;     // 错误
   e = E(0); // 正确
   return e;
}
```

解决方法：使用强制类型转换。

2.3.11 成员初始化程序列表

在成员初始化程序列表中隐含的基类名称不允许使用旧的 C++ 语法。

```
struct B { B(int); };
struct D : B {
    D(int i) : (i) { }     // 错误，应该是 B(i)
};
```

2.3.12 const 和 volatile 限定符

在向函数传递参数或初始化变量时，函数指针的 const 和 volatile 限定符必须正确匹配。

```
void f(char *);
const char* p = "hello";
f(p);               // 错误：正在将 const char* 传递到 non-const char*
```

解决方法：如果函数不修改它所指向的字符，将参数声明为 const char*。否则，创建字符串的非常量副本，然后再传递。

2.3.13 嵌套类型

在没有类限定符的封装类的外面不能访问嵌套类型。

```
struct Outer {
    struct Inner { int i; };
    int j;
};
Inner x;             // 错误：应该是 Outer::Inner
```

2.3.14 类模板定义和声明

在类模板定义和声明中，将带括号 < > 的类型参数附加到类的名称中的做法始终无效，但是 4.0 和 5.0 版的 C++ 编译器不会报错。例如，在以下代码中 <T> 附加到 MyClass 中，这对于定义和声明都无效。

```
template<class T> class MyClass<T> { ... }; // 定义
template<class T> class MyClass<T>;          // 声明
```

解决方法：如下面的代码所示，从类名称中删除带括号的类型变量。

```
template<class T> class MyClass { ... }; // 定义
template<class T> class MyClass;         // 声明
```

2.4 模板编译模式

在处于兼容模式时模板编译模式不同于 4.2 版编译模式。有关新模式的更多信息，请参见第 3-5 页上的第 3.3.5 节 “模板系统信息库”。

第 3 章

使用标准模式

本章介绍标准模式的使用，标准模式是 C++ 编译器的缺省编译模式。

3.1 标准模式

由于标准模式是主要的缺省模式，因此不需要任何选项。您也可以选择编译器选项：

```
-compat=5
```

例如：

```
example% CC -O myfile.cc mylib.a -o myprog
```

3.2 标准模式下的关键字

C++ 增加了一些新的关键字。如果您将其中任何一个关键字用作标识符，则会看到许多并且有时很奇怪的错误消息。（确定程序员在何时将关键字用作标识符非常困难，并且编译器的错误消息这时可能也不会有什么帮助。）

大多数新关键字可以通过编译器选项禁止，如下表所示。一些关键字在逻辑上相互关联，并且可以作为一个组启用或禁止。

表 3-1 标准模式下的关键字

关键字	禁止关键字的编译器选项
`bool, true, false`	`-features=no%bool`
`explicit`	`-features=no%explicit`
`export`	`-features=no%export`
`mutable`	`-features=no%mutable`
`namespace, using`	*无法禁止*
`typename`	*无法禁止*
`and, and_eq, bitand, compl, not, not_eq, or, bitor, xor, xor_eq`	`-features=no%altspell` （参见下面）

国际标准化组织 (ISO) C 标准的补充标准引入了 C 标准头 `<iso646.h>`，它定义了新的宏以生成特殊的标记。 C++ 标准直接将这些拼写作为保留字引入。（在启用替代的拼写时，在程序中包括 `<iso646.h>` 无丝毫影响。）下表中显示了这些标记的含义。

表 3-2 替代的标记拼写

标记	拼写
`&&`	`and`
`&&=`	`and_eq`
`&`	`bitand`
`~`	`compl`
`!`	`not`
`!=`	`not_eq`
`||`	`or`
`|`	`bitor`
`~`	`xor`
`~=`	`xor_eq`

3.3 模板

C++ 标准有一些新的模板规则，使旧的代码不再适应要求，尤其是涉及到使用新关键字 typename 的代码。C++ 编译器不强制执行这些规则，但是确实会识别该关键字。尽管 4.2 版编译器接受一些无效的模板代码，在大多数情况下，在 4.2 版编译器下工作的模板代码仍可以继续工作。由于将来的编译器会强制执行新的规则，因此在开发时间表允许的情况下，您应该将代码迁移至新的 C++ 规则。

3.3.1 解析类型名称

C++ 标准有新的规则，用于确定标识符是否为类型的名称。下面举例说明新的规则：

```
typedef int S;
template< class T > class B { typedef int U; };
template< class T > class C : public B<T> {
  S s; // 正确
  T t; // 正确
  U x; // 1 不再有效
  T::V z; // 2 不再有效
};
```

新的语言规则表明不会自动搜索依靠模板参数的基类以解析模板中的类型名称，并且来自基类或模板参数类的名称不是类型名称，除非是使用关键字 typename 将名称声明为类型名称。

示例代码中的第一个无效行 (1) 尝试从 B 中继承 U 作为一项类型，但是没有使用合格的类名称，也没有使用关键字 typename。第二个无效行 (2) 使用来自模板参数的类型 V，但是省略了关键字 typename。由于 s 类型不依赖于模板参数的基类或成员，因此 s 的定义有效。同样，由于 t 的定义直接使用类型 T，表示一定是某种类型的模板参数，因此该定义也有效。

下面是修改之后正确的示例。

```
typedef int S;
template< class T > class B { typedef int U; };
template< class T > class C : public B<T> {
  S s; // 正确
  T t; // 正确
  typename B::U x; // 正确
  typename T::V z; // 正确
};
```

3.3.2 转换为新的规则

迁移代码的问题在于 typename 以前不是关键字。如果现有的代码将 typename 用作标识符，必须首先将该名称更改为其他名称。

对于必须用于新旧编译器的代码，您可以在整个项目的头文件中增加与下面的示例类似的语句。

```
#ifdef TYPENAME_NOT_RECOGNIZED
#define typename
#endif
```

其效果就是有条件地将 typename 替换为空白。在使用不能识别 typename 的更早的编译器 （例如 C++ 4.1）时，在 makefile 的编译器选项设置中增加 -DTYPENAME_NOT_RECOGNIZED。

3.3.3 显式实例化和专门化

在 ARM 和 4.2 版编译器中，没有一种标准的方式来使用模板定义来请求模板的显式实例化。C++ 标准和标准模式下的 C++ 编译器提供了一种使用模板定义进行显式实例化的语法；即关键字 template 后面加上类型的声明。例如，以下代码的最后一行使用缺省的模板定义，强制将类 MyClass 实例化为类型 int。

```
template<class T> class MyClass {
    ...
};
template class MyClass<int>; // 显式实例化
```

显式专门化的语法已被更改。要声明显式专门化或提供完整的定义，现在给声明加上前缀 template<>。（注意空白的尖括号。）例如：

```
// MyClass 专门化
class MyClass<char>;          // 旧式声明
class MyClass<char> { ... }; // 旧式定义
template<> class MyClass<char>;          // 标准声明
template<> class MyClass<char> { ... }; // 标准定义
```

这样的声明形式表示程序员在其他某处为提供的参数提供了不相同的模板定义 （专门化），编译器不会使用那些参数的缺省模板定义。

在标准模式下，编译器接受旧的语法，作为时序错误。 4.2 版编译器接受新的专门化语法，但不是每一种情形都会使用新的语法正确处理代码。（在将该功能结合到 4.2 版编译器中之后，草案标准有了改变。）为了尽可能提高模板专门化代码的可移植性，可以在整个项目的头中增加与下面类似的语法：

```
#ifdef OLD_SPECIALIZATION_SYNTAX
#define Specialize
#else
#define Specialize template<>
#endif
```

然后您编写代码，例如：

```
Specialize class MyClass<char>; // 定义
```

3.3.4 类模板定义和声明

在类模板定义和声明中，给类型参数加上括号 < > 并附加到类的名称中，这种做法从未有效，但是 4.0 和 5.0 版的 C++ 编译器不会报告错误。例如，在以下代码中 <T> 附加到 MyClass 中，这对于定义和声明都无效。

```
template<class T> class MyClass<T> { ... }; // 定义
template<class T> class MyClass<T>;          // 声明
```

要解决该问题，如下面的代码所示，从类名称中删除带括号的类型变量。

```
template<class T> class MyClass { ... }; // 定义
template<class T> class MyClass;         // 声明
```

3.3.5 模板系统信息库

Sun 实现 C++ 模板使用了模板实例的系统信息库。 C++ 4.2 版编译器将系统信息库存储在一个称为 Templates.DB 的目录中。 C++ 5 编译器在缺省情况下使用称为 SunWS_cache 和 SunWS_config 的目录。 SunWS_cache 包含工作文件，而 SunWS_config 包含配置文件，尤其是模板选项文件 (SunWs_config/CC_tmpl_opt)。（参见 《*C++ 用户指南*》。）

如果您的 makefile 因某些原因提及到按名称列出的系统信息库目录，需要修改 makefile。此外，系统信息库的内部结构已经变化，因此，任何访问 Templates.DB 内容的 makefile 都不再起作用。

除此之外，标准的 C++ 程序可能大量地使用模板。注意要考虑到多个程序或项目共享目录，这一点非常重要。如果有可能，使用最简单的方式：只编译任何一个目录中属于相同程序或库的文件。模板系统信息库就会只适用于一个程序。如果您在相同的目录中编译不同的程序，使用 `CCadmin -clean` 清除系统信息库。有关详细信息，请参见《*C++ 用户指南*》。

多个程序共享同一个系统信息库的危险在于同一个名称可能需要不同的定义。在共享系统信息库时，不能正确处理这种情形。

3.3.6 模板和标准库

C++ 标准库包含许多模板以及访问这些模板的许多新的标准头名称。 Sun C++ 标准库在模板头中进行声明，并且在单独的文件中实现模板。如果某个项目文件名与新模板头的名称相匹配，编译器可能会选择错误的实行文件，并且导致出现许多奇怪的错误。假设您自己有一个称为 `vector` 的模板，并且在称为 `vector.cc` 的文件中实现模板。在编译器需要标准库中的这个模板时，可能会根据文件位置和命令行选项选择您的 `vector.cc`，反之亦然。在将来的编译器版本中实现导出关键字和导出的模板时，情形会更糟糕。

为了防止当前和将来出现问题，下面提供两点建议：

- 不要将任何标准头名用作模板文件名。所有的标准库位于名称空间 `std` 中，因此，名称不会直接与自己的模板或类冲突。`using` 声明或指令仍然会引起间接冲突，因此不要使用与标准库中的头文件重名的模板名称。涉及模板的标准头如下所示：

```
algorithm     bitset      complex      deque      exception
fstream       functional  iomanip      ios        iosfwd
iostream      istream     iterator     limits     list
locale        map         memory       numeric    ostream
queue         set         sstream      stack      stdexcept
streambuf     string      typeinfo     utility    valarray
vector
```

- 在头 (`.h`) 文件而不是单独的文件中实现模板，以防出现实现文件名冲突。有关详细信息，请参见 《*C++ 用户指南*》。

3.4 类名注入

C++ 标准表明类名被 “注入”类本身中。这是根据早期的 C++ 规则作出的更改。以前，类名不会出现在类中。

在大多数情况下，这种微小的更改对现有的程序没有影响。在某些情况下，这种更改使以前有效的程序无效，有时导致含义变化。例如：

编码示例 3-1 类名注入问题 1

```
const int X = 5;

class X {
    int i;
public:
    X(int j = X) : // 缺省值 X 是什么?
    i(j) { }
};
```

要确定 X 作为缺省参数值的含义，编译器在当前的范围中查找名称 X，然后在连续的外部范围中查找，直到找到 X：

- 根据旧的 C++ 规则，类名 X 不会出现在类范围中，并且文件范围中的整型名称 X 隐藏了类名 X。因此，缺省值为 5。
- 根据新的 C++ 规则，类名 X 可出现在类本身中。编译器在类中找到 X 并生成错误，这是因为它找到的 X 是类型名称，而不是整数值。

由于在同一个范围中类型和对象的名称相同被视为编程经验不足，因此，这种错误应该很少出现。如果您看到这样的错误，可以通过正确的范围使变量符合要求，从而修复代码，例如：

```
X(int j = ::X)
```

下面举例 （取自标准库）说明另一个范围问题。

编码示例 3-2　类名注入问题 2

```
template <class T> class iterator { ... };

template <class T> class list {
     public:
       class iterator { ... };
       class const_iterator : public ::iterator<T> {
         public:
            const_iterator(const iterator&); // 哪个迭代器?
       };
};
```

const_iterator 构造函数的参数类型是什么？根据旧的 C++ 规则，编译器在类的范围 const_iterator 中找不到名称 iterator，因此，它搜索下一个外部范围，即类 list<T>。该范围具有成员类型 iterator。因此，参数类型为 list<T>::iterator。

根据新的 C++ 规则，类名将插入自己的范围。尤其是，基类名称将插入基类。当编译器开始在派生类范围中搜索名称时，现在它可以找到基类的名称。由于 const_iterator 构造函数的参数类型没有范围限定符，因此找到的名称是 const_iterator 基类的名称。因此，参数类型为 global::iterator<T>，而不是 list<T>::iterator。

要取得理想的结果，可以更改一些名称，或者使用范围限定符，例如：

```
const_iterator(const list<T>::iterator&);
```

3.5　for- 语句变量

ARM 规则表明 for- 语句的头中声明的变量插入到包含 for 语句的范围中。 C++ 委员会认为这项规则不正确，并且变量的范围应该在 for- 语句结尾处结束。（此外，该规则没有包括一些常见情形，因此，一些代码在使用不同的编译器时工作方式各不相同。）C++ 委员会相应地更改了该规则。许多编译器 （包括 C++ 4.2）继续使用旧的规则。

在下面的示例中， if- 语句根据旧的规则有效，但是根据新的规则无效，因为 k 已超出范围。

```
for( int k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...        // 此代码是否可以?
```

在兼容模式下， C++ 编译器缺省情况下使用旧的规则。您可以通过 -features=localfor 编译器选项，要求编译器使用新的规则。

在标准模式下， C++ 编译器缺省情况下使用新的规则。您可以通过 -features=no%localfor 编译器选项，要求编译器使用旧的规则。

您可以从 for- 语句头中拉出声明，编写在任何模式下都可以正常用于所有编译器的代码，如下面的例子所示。

```
int k;
for( k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...       // 始终可以
```

3.6 在函数指针与 void* 之间转换

C++ 编译器在兼容模式和标准模式下现在都会针对函数指针与 void* 之间的隐式或显式转换发出警告。在标准模式下，编译器在解决重载的函数调用时，不再能够识别此类隐式转换。有关详细信息，请参见第 1-7 页上的第 1.6 节 “函数指针与 void*”。

3.7 字符串文字和 char*

一些历史记录可能有助于解释清楚这个小问题。标准的 C 语言引入了 const 关键字和常数对象概念，这两者在最初的 C 语言中都没有出现 ("K&R" C)。为了防止出现毫无意义的结果，“Hello world”等字符串文字逻辑上应该是 const，如下面的例子所示。

```
#define GREETING "Hello world"
char* greet = GREETING; // 没有编译器错误消息
greet[0] = 'G';
printf("%s", GREETING); // 在某些系统上打印 “Gello world”
```

C 和 C++ 语言都没有规定尝试修改字符串文字的结果。如果实现选择相同的字符串文字使用相同的可写入存储器，上面的例子则产生所示的单一结果。

由于从过去到现在多数代码看上去象前面示例的第二行，因此在 1989 年 C 标准委员会不想使字符串文字成为 const。 C++ 语言最初遵守 C 语言规则。 C++ 标准委员会后来决定要保证 C++ 类型安全性的目标更重要，因此更改了规则。

在标准的 C++ 语言中，字符串文字是常数，并且类型为 const char[]。前面例子中的第二行代码在标准的 C++ 语言中无效。同样，不再向声明为 char* 的函数参数传递字符串文字。然而， C++ 标准还提供了一种从 const char[] 到 char* 的过时字符串文字转换。一些例子包括：

```
char *p1 = "Hello";          // 以前可以，现在已过时
const char* p2 = "Hello";  // 正确
void f(char*);
f(p1);         // 始终可以，因为 p1 不是已声明的 const
f(p2);         // 始终错误，正在将 const char* 传递到 char*
f("Hello");    // 以前可以，现在已过时
void g(const char*);
g(p1);         // 始终可以
g(p2);         // 始终可以
g("Hello");    // 始终可以
```

如果函数不直接或间接修改作为参数传递的字符数组，应该声明参数 const char*（或者 const char[]）。您可能会发现整个程序都需要增加 const 限定符，在您增加限定符时，就仍有必要增加更多的限定符。（这种现象有时称为 “常数中毒”。）

在标准模式下，编译器就从字符串文字到 char* 的过时转换发出警告。如果您在现有程序的恰当之处小心地使用 const，根据新的规则，它们可能会编译，并且不会生成这些警告。

为了达到函数重载的目的，在标准模式下字符串文字始终作为 const。例如：

```
void f(char*);
void f(const char*);
f("Hello"); // 调用哪个 f?
```

如果在兼容模式下 （或使用 4.2 版编译器）编译上面的示例，将调用函数 f(char*)。如果在标准模式下编译，将调用函数 f(const char*)。

在标准模式下，编译器在缺省情况下将文字串放在只读存储器中。如果您之后尝试修改字符串 （由于自动转换为 char*，可能会出现这种情况），程序由于存储器违规而异常终止。

对于下面的例子， C++ 编译器在兼容模式下将字符串文字放入可写入的存储器中，就象 4.2 版编译器那样。尽管程序在技术上有一些行为未定义，但程序会运行。在标准模式下，编译器在缺省情况下将字符串文字放入只读存储器中，并且程序由于存储器故障而异常终止。因此，您应该留意有关字符串文字转换的所有警告，并且试着修复程序，从而不再出现转换。这样的更改确保您的程序对于实现每一个 C++ 都正确。

```
void f(char* p) { p[0] = 'J'; }

int main()
{
    f("Hello"); // 从 const char[] 转换为 char*
}
```

您可以使用编译器选项更改编译器的行为：

- `-features=conststrings` 编译器选项要求编译器将字符串文字放入只读存储器中，即使处于兼容模式也是如此。
- `-features=no%conststrings` 编译器选项导致编译器将字符串文字放入可写入的存储器中，即使处于标准模式也是如此。

您可能会发现使用标准的 C++ `string` 类 （而不是 C 式样的字符串）很方便。 C++ 字符串类没有与字符串文字有关的问题，这是因为标准的 `string` 对象可以分别声明为 `const` 或不声明，并且可以通过引用、指针和函数值进行传递。

3.8 条件表达式

C++ 标准给条件表达式的规则带来了变化。 C++ 编译器在标准模式和兼容模式下都使用新的规则。有关详细信息，请参见第 1-7 页上的第 1.5 节 “条件表达式”。

3.9 new 和 delete 的新形式

有四个问题与 `new` 和 `delete` 的新形式有关：

- 数组形式
- 异常规范
- 替换函数
- 头文件

在兼容模式下，编译器的缺省设置是使用旧的规则，而在标准模式下编译器的缺省设置是使用新的规则。建议不要更改缺省设置，因为兼容模式运行时库 (libC) 依靠旧的定义和行为，而标准模式运行时库 (libCstd) 依靠新的定义和行为。

在实行新的规则时，编译器预定义宏 _ARRAYNEW 的值为 1。在使用旧的规则时未定义宏。下面一段举例详细说明这一点：

```
// 替换函数
#ifdef _ARRAYNEW
    void* operator new(size_t) throw(std::bad_alloc);
    void* operator new[](size_t) throw(std::bad_alloc);
#else
    void* operator new(size_t);
#endif
```

3.9.1 new 和 delete 的数组形式

C++ 标准为在分配数组或解除数组分配时调用的 operator new 和 operator delete 增加了新的形式。以前，这些运算符函数只有一种形式。此外，在您分配数组时，只使用 operator new 和 operator delete 的全局形式，而从不使用特定于类的形式。C++ 4.2 版编译器不支持新的形式，这是因为使用新的形式需要更改 ABI。

除了这些函数之外：

```
void* operator new(size_t);
void operator delete(void*);
```

现在还有以下函数：

```
void* operator new[](size_t);
void operator delete[](void*);
```

在所有情形 （以前和当前）下，您可以编写替换函数以替换运行时库中的版本。编译器提供了两种形式，这样您可以使用不同的存储区来存储数组，而不是存储单个对象，并且类可以为数组提供自己的 operator new 版本。

根据这两套规则，在您编写 new T 时，其中 T 指一些类型，将调用函数 operator new(size_t)。然而，在您根据新的规则编写 new T[n] 时，将调用函数 operator new[](size_t)。

同样，根据这两套规则，在您编写 delete p 时，将调用函数 operator delete(void*)。根据新的规则，在您编写 delete [] p 时，将调用函数 operator delete[](void*)。

您也可以为这些函数的数组形式编写特定于类的版本。

3.9.2 异常规范

根据旧的规则，如果分配失败，所有的 operator new 形式返回空指针。根据新的规则，如果分配失败，普通的 operator new 形式将抛弃异常，并且不返回任何值。同时也有 operator new 的特殊形式，它返回零，而不抛弃异常。所有的 operator new 和 operator delete 版本都有*例外规范*。标准头 <new> 中的声明为：

编码示例 3-3 标准头 <new>

```
namespace std {
      class bad_alloc;
      struct nothrow_t {};
      extern const nothrow_t nothrow;
}
// 单个对象形式
void* operator new(size_t size) throw(std::bad_alloc);
void* operator new(size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
// 数组形式
void* operator new[](size_t size) throw(std::bad_alloc);
void* operator new[](size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

例如，下面示例中的防范代码不再能够按照以前的计划工作。如果分配失败， new 表达式自动调用的 operator new 抛弃异常，并且再也不出现零测试。

```
T* p = new T;
if( p == 0 ) {            // 不再可以
    ...                   // 句柄分配失败
}
...                       // 使用 p
```

有两种解决方法：

- 重写代码以捕捉例外。例如：

```
T* p = 0;
try {
    p = new T;
}
catch( std::bad_alloc& ) {
    ...        // 句柄分配失败
}
...            // 使用 p
```

- 使用 operator new 的 nothrow 版本。例如：

```
T* p = new (std::nothrow) T;
... remainder of code unchanged from original
```

如果您不喜欢在代码中使用任何异常，可以使用第二种形式。如果您要在代码中使用异常，请考虑使用第一种形式。

如果您以前没有检验 operator new 是否成功，可以保留现有的代码不动。然后，在分配失败时它会立即终止，而不会再继续运行到出现无效内存引用的某个点。

3.9.3 替换函数

如果您有 operator new 和 delete 的替换版本，它们必须与编码示例 3-3 中显示的签名匹配，包括函数的异常规范。此外，它们*必须*实行相同的语义。 operator new 的正常形式必须在失败时抛弃 bad_alloc 异常； nothrow 版本禁止抛出任何异常，但是必须在失败时返回零。 operator delete 的形式禁止抛弃任何异常。标准库中的代码使用全局 operator new 和 delete，并且依靠该行为才能正确操作。第三方库可能有类似的依存性。

根据 C++ 标准的要求，C++ 运行时库中的 operator new[]() 全局版本只是调用单个对象版本 operator new()。如果您替换 C++ 标准库中 operator new() 的全局版本，不需要替换 operator new[] () 的全局版本。

C++ 标准禁止替换 operator new 的预定义 “站点”形式。

```
void* operator new(std::size_t, void*) throw();
void* operator new[](std::size_t, void*) throw();
```

它们不能在标准模式下替换，尽管 4.2 版编译器允许这样做。当然，您可以使用不同的参数列表编写自己的站点版本。

3.9.4 头包括的内容

在兼容模式下始终包括 <new.h>。而在标准模式下包括 <new> （无 .h）。为了便于过渡，在标准模式下提供了头 <new.h>，它使得名称空间 std 中的名称适用于全局名称空间。该头还提供了类型定义，使旧的异常名称对应于新的异常名称。参见第 3-20 页上的第 3.13 节 “标准异常”。

3.10 布尔类型

布尔关键字 — bool、 true 和 false — 由编译器中是否出现布尔关键字识别功能控制：

- 在兼容模式下，缺省设置是关闭布尔关键字识别功能。您可以使用编译器选项 -features=bool，打开布尔关键字识别功能。
- 在标准模式下，缺省设置是打开布尔关键字识别功能。您可以使用 -features=no%bool 编译器选项，关闭识别这些关键字的功能。

在兼容模式下最好打开关键字，因为这会显示代码中当前使用的任何关键字。

注 – 即使您的旧代码使用了兼容的布尔类型定义，实际类型并不相同，它会影响名字粉碎。如果您这样做，必须使用函数参数中的布尔类型重新编译所有旧的代码。

在标准模式下最好不要关闭布尔关键字，这是因为 C++ 标准库依靠内建的 bool 类型，而该类型不可用。您以后打开 bool 时，更多的问题接踵而至，尤其是名字粉碎。

在启用布尔关键字时，编译器预定义的宏 _BOOL 为 1。在禁用布尔关键字时，不定义该宏。例如：

```
// 定义兼容性很好的布尔类型
#if !defined(_BOOL) && !defined(BOOL_TYPE)
    #define BOOL_TYPE           // 本地包含保护
    typedef unsigned char bool; // 标准模式布尔使用 1 个字节
    const bool true = 1;
    const bool false = 0;
#endif
```

在兼容模式下，您不能定义一个象新的内建 bool 型那样工作的布尔型。这是将内建的布尔类型增加至 C++ 语言的一项原因。

3.11 extern "C" 函数的指针

可以声明具有语言链接的函数，例如

```
extern "C" int f1(int);
```

如果您不指定链接，则假设使用 C++ 链接。您可以显式指定 C++ 链接：

```
extern "C++" int f2(int);
```

您还可以组合声明：

```
extern "C" {
    int g1(); // C 链接
    int g2(); // C 链接
    int g3(); // C 链接
} // 不带分号
```

该技术大量用于标准头中。

3.11.1 语言链接

*语言链接*表示调用函数的方式：在参数放置的位置就可以找到返回值，依此类推。对语言链接的声明并不意味着函数就是用该语言编写的。这意味着函数就*好象*以该语言编写的那样会被调用。因此，声明 C++ 函数具有 C 链接意味着可以通过以 C 语言编写的函数调用 C++ 函数。

函数声明所使用的语言链接应用于返回类型及其具有函数或函数指针类型的所有参数。

在兼容模式下，编译器实现语言链接不属于函数类型的 ARM 规则。尤其是，您可以声明函数指针，而不必考虑指针链接或赋值的函数链接。该行为与 C++ 4.2 编译器相同。

在标准模式下，编译器执行新的规则：语言链接是其类型的一部分，同时也是函数指针类型的一部分。因此，链接必须相匹配。

下面的示例为带有 C 和 C++ 链接的函数和函数指针显示了全部四种可能的组合。在兼容模式下，编译器接受所有的组合，就象 4.2 版编译器。在标准模式下，编译器则仅将不匹配的组合作为时序错误接受。

```
extern "C" int fc(int) { return 1; }  // fc 包含 C 链接
int fcpp(int) { return 1; }           // fcpp 包含 C++ 链接
// fp1 and fp2 have C++ linkage
int (*fp1)(int) = fc;                 // 不匹配
int (*fp2)(int) = fcpp;               // 正确
// fp3 and fp4 have C linkage
extern "C" int (*fp3)(int) = fc;      // 正确
extern "C" int (*fp4)(int) = fcpp;    // 不匹配
```

如果您遇到问题，确保要用于 C 链接函数的指针使用 C 链接声明，并且要用于 C++ 链接函数的指针不使用链接说明符声明，或者使用 C++ 链接声明。例如：

```
extern "C" {
    int fc(int);
    int (*fp1)(int) = fc; // 两者均包含 C 链接
}
int fcpp(int);
int (*fp2)(int) = fcpp;   // 两者均包含 C++ 链接
```

最糟糕的情况是，您确实有不匹配的指针和函数，这时您可以编写一个函数 “包装器”，以免编译器报错。在下面的示例中， composer 是 C 函数，它采用了具有 C 链接的函数指针。

```
extern "C" void composer( int(*)(int) );
extern "C++" int foo(int);
composer( foo ); // 不匹配
```

要将函数 foo （具有 C++ 链接）传递至函数 composer，创建 C 链接函数 foo_wrapper，表示 foo 的 C 接口：

```
extern "C" void composer( int(*)(int) );
extern "C++" int foo(int);
extern "C" int foo_wrapper(int i) { return foo(i); }
composer( foo_wrapper ); // 正确
```

除了避免编译器报错之外，即使 C 和 C++ 函数确实具有不同的链接，这种解决方法仍然起作用。

3.11.2 可移植性更小的解决方法

Sun 实现 C 和 C++ 函数链接时与二进制兼容。这并不表示实现每一项 C++ 都是如此，尽管它相当常见。如果您对可能出现的不兼容并不关心，可以执行强制类型转换，将 C++ 链接函数当作 C 链接函数使用。

这里是有关静态成员函数的一个很好的例子。在实行将链接视为函数类型的一部分这种新的 C++ 语言规则之前，通常建议将类的静态成员函数作为具有 C 链接的函数处理。这样可以摆脱无法为类成员函数声明任何链接的限制。您的代码可能如下所示：

```
// 现有的代码
typedef int (*cfuncptr)(int);
extern "C" void set_callback(cfuncptr);
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // 不再有效
```

根据前面一段的建议，您可以创建调用 `T::memfunc` 的函数包装器，然后将所有的 `set_callback` 调用更改为使用包装器，而不使用 `T::memfunc`。这样的代码正确无误，并且完全可以移植。

另外一种方法就是创建 `set_callback` 的重载版本，该版本采用具有 C++ 链接的函数，并调用原始函数，如下面的例子所示：

```
// 修改后的代码
extern "C" {
    typedef int (*cfuncptr)(int); // ptr 至 C 函数
    void set_callback(cfuncptr);
}
typedef int (*cppfuncptr)(int); // ptr 至 C++ 函数
inline void set_callback(cppfuncptr f) // 重载的版本
    { set_callback((cfuncptr)f); }
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // 未对原始代码进行更改
```

该例子只需对现有的代码稍作修改。这样增加了另一个设置回调的函数版本。调用原始 `set_callback` 的现有代码现在调用重载版本，而重载版本调用原始版本。由于重载版本是内联函数，因此根本不存在运行时开销。

虽然该项技术可在 Sun C++ 语言中使用，但并不保证能够用于每一个 C++ 实现，因为在其他系统上对 C 和 C++ 函数的调用顺序可能不同。

3.11.3 函数指针作为函数参数

新的语言链接规则带来的细微结果涉及到将函数指针用作参数的函数，例如 :

```
extern "C" void composer( int(*)(int) );
```

有关语言链接未改变的规则就是：如果您声明具有语言链接的函数，并且在后面加上*相同函数*的定义，但不指定语言链接，这将应用上一个语言链接。例如：

```
extern "C" int f(int);
int f(int i) { ... } // 包含 "C" 链接
```

在本例子中，函数 f 具有 C 链接。声明 （声明可能在被包括的头文件中）之后的定义继承声明的链接规范。但是，假设函数采用函数指针类型的参数，如下面的例子所示：

```
extern "C" int g( int(*)(int) );
int g( int(*pf)(int) ) { ... } // 这是 "C" 还是 "C++" 链接?
```

根据旧的规则以及在用于 4.2 版编译器时，只有一个函数 g。根据新的规则，第一行声明具有 C 链接的函数 g，它采用了具有 C 链接的函数指针。第二行定义的函数采用具有 C++ 链接的函数指针。这两个函数不相同，第二个函数具有 C++ 链接。由于链接是函数指针类型的一部分，这两行指的是两个都调用 g 的不同的重载函数。它们具有相同的功能，基于它们的代码会有问题。很有可能这些代码在编译或链接时会失败。

好的编程做法是在函数定义以及声明中加上链接规范：

```
extern "C" int g( int(*)(int) );
extern "C" int g( int(*pf)(int) ) { ... }
```

为进一步减少类型方面的混淆，您可以对函数参数使用 typedef。

```
extern "C" {typedef int (*pfc)(int);} // ptr 至 C 链接函数
extern "C" int g(pfc);
extern "C" int g(pfc pf) { ... }
```

3.12 运行时环境类型标识 (RTTI)

在兼容模式下，缺省设置是关闭 RTTI， 4.2 版编译器也是关闭 RTTI。在标准模式下， RTTI 已打开并且不能关闭。在使用旧的 ABI 时， RTTI 在数据大小和效率方面的成本相当高。（在使用旧的 ABI 时不能直接实现 RTTI，并且需要使用低效的间接方法。）在使用新的 ABI 的标准模式下， RTTI 的成本可以忽略不计。（这是 ABI 众多改进中的其中一项改进。）

3.13 标准异常

C++ 4.2 版编译器使用与标准异常相关的名称，在准备编译器时，该名称出现在 C++ 草案标准中。标准 C++ 中的名称从此以后发生变化。在标准模式下， C++ 5 编译器使用标准名称，如下表所示。

表 3-3 与异常相关的类型名称

旧名称	标准名称	说明
`xmsg`	`exception`	标准异常的基类
`xalloc`	`bad_alloc`	由失败的分配请求抛出
`terminate_function`	`terminate_handler`	终止处理程序函数的类型
`unexpected_function`	`unexpected_handler`	未预料的异常处理程序函数的类型

类的公共成员 （`xmsg` 与 `exception` 以及 `xalloc` 与 `bad_alloc`）不相同，这正是您使用这些类的方式。

3.14 析构静态对象的顺序

静态对象 指具有静态的存储持续时间的对象。静态对象可以是全局对象，或者位于名称空间中。它可以是函数的静态局部变量，或者它可以是类的静态数据成员。

C++ 标准要求销毁静态对象的顺序与构造对象的顺序相反。此外，析构这些对象可能需要与向 `atexit()` 函数登记的函数混合。

早期版本的 C++ 编译器会销毁在任一模块中创建的全局静态对象，销毁这些对象的顺序与构造时相反。然而，并不保证在整个程序中析构对象的顺序正确。

从 5.1 版的 C++ 编译器开始，严格按照构造静态对象的相反顺序析构静态对象。例如，假定有三个类型为 T 的静态对象：

- 一个对象在 file1 的全局范围中。
- 第二个对象在 file2 的全局范围中。
- 第三个对象在函数的局部范围中。

我们不能判定会首先创建两个全局对象中的哪一个对象，是 file1 中的对象，还是 file2 中的对象。然而，先创建的全局对象将在销毁另一个全局对象后再销毁。

局部静态对象在调用其函数时创建。如果在创建两个全局静态对象之后调用函数，局部对象将在全局对象之前销毁。

C++ 标准对析构与向 atexit() 函数登记的函数有关的静态对象提出其他一些要求。如果在构造静态对象 X 之后函数 F 向 atexit() 登记，在销毁 X 之前并且在退出程序时，必须调用 F。反过来，如果在构造 X 之前函数 F 向 atexit() 登记，在销毁 X 之后并且在退出程序时，必须调用 F。

下面举例说明该规则。

```
// T 是包含析构函数的类型
void bar();
void foo()
{
  static T t2;
  atexit(bar);
  static T t3;
}
T t1;
int main()
{
  foo();
}
```

在程序启动时创建 t1，然后运行主程序。主程序调用 foo()。 foo() 函数按照该顺序执行以下任务。

1. 创建 t2
2. 向 atexit() 登记 bar()。
3. 创建 t3

在快要结束主程序时，自动调用 exit。退出过程必须按照以下顺序。

1. 销毁 t3； t3 是在 bar() 向 atexit() 登记之后构造的。

2. 运行 bar()

3. 销毁 t2； t2 是在 bar() 向 atexit() 登记之前构造的。

4. 销毁 t1； t1 是第一个构造的对象，因此最后一个销毁。

支持这种静态析构函数的交替和 atexit() 处理需要 Solaris 运行时库 libc.so 的帮助。从 Solaris 8 软件开始提供这种支持。使用 5.1 版、 5.2 版、 5.3 版或 5.4 版的 C++ 编译器编译的 C++ 程序在运行时查找库中的特殊符号，确定程序当前是否在提供这种支持的 Solaris 软件版本上运行。如果提供了这种支持，静态的析构函数正确地与 atexit 登记的函数交替。如果程序在不提供该支持的 Solaris 软件版本上运行，析构函数仍然按照正确的顺序执行，但是不会与 atexit 登记的函数交替。

注意程序每次运行时会作出决定。它与您生成程序所使用的 Solaris 软件版本没有关系。只要 Solaris 运行时库 libc.so 动态地链接 （缺省设置是这样），如果运行程序的 Solaris 软件版本支持交替，程序退出时将会出现交替。

不同的编译器提供不同的支持级别，以便按照正确的顺序析构对象。为了提高代码的可移植性，程序的正确性不应该依靠销毁静态对象的正确顺序。

如果您的程序依靠特定的析构顺序并且在使用比较旧的编译器，在标准模式下标准所要求的顺序可能会破坏程序。-features=no%strictdestrorder 命令选项禁止按照严格的析构顺序。

第 4 章

使用 Iostream 和库头

本章介绍在 C++ 5.0 编译器中库和头文件做了哪些更改。在迁移原本用于 C++ 4 编译器的代码以便用于 C++ 5 编译器时，必须考虑这些更改。

4.1 Iostream

C++ 4.2 版编译器实现*传统的* iostream，它从来没有正式的定义。此实现与 1990 年随 `Cfront` 发行的版本兼容，并且修复了一些错误。

标准的 C++ 定义了新的和扩展的 iostream （*标准的* iostream）。它现在的定义更完善、功能更丰富，并且支持编写国际化代码。

在兼容模式下，您会获得传统的 iostream，该版本与随 C++ 4.2 版编译器提供的版本相同。在兼容模式 (`-compat[=4]`) 下编译时，用于 4.2 版编译器的任何现有 iostream 代码应该同样能够使用。

注 – 编译器提供了两种传统的 iostream 运行时库版本。一种版本是在兼容模式下使用编译器编译的库，并且与 C++ 4.2 使用的库相同。另一个版本是根据相同的源代码在标准模式下使用编译器编译的库。源代码的接口相同，但是库中的二进制代码有标准模式的 ABI。参见第 1-3 页上的第 1.3 节 “二进制兼容问题”。

在标准模式下，缺省设置是您使用标准的 iostream。如果您使用标准形式的头名称 （不带 “`.h`”），您就会得到标准的头文件，所有声明都在名称空间 `std` 中。

还提供了 4 个以 “`.h`” 结尾的标准头文件，这样通过使用声明，就可以在全局名称空间中使用这些头名称。

- `<fstream.h>`
- `<iomanip.h>`
- `<iostream.h>`
- `<strstream.h>`

这些头是 Sun 公司提供的扩展头，依靠这些头的代码可能无法移植。这些头使您可以编译现有（简单的）iostream 代码，而且不必更改代码，即使使用标准的 iostream（不使用传统的 iostream）也是如此。例如，编码示例 4-2 使用传统的 iostream 或 Sun 实现的标准 iostream 进行编译。

编码示例 4-1 使用标准的 iostream 名称形式

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
}
```

编码示例 4-2 使用传统的 iostream 名称形式

```
#include <iostream.h>
int main()
{
    cout << "Hello, world!" << endl;
}
```

并非所有的传统 iostream 代码都与标准的 iostream 代码兼容。如果传统的 iostream 代码不编译，必须修改代码或者全部使用传统的 iostream。

要在标准模式下使用传统的 iostream，在 CC 命令行中使用编译器选项 `-library=iostream`。在使用该选项时，它会搜索包含传统 iostream 头文件的特殊目录，并且传统的 iostream 运行时库与您的程序链接。您必须在构造程序的所有编译工作以及最终的链接阶段中使用该选项，否则，程序结果会不一致。

注 – 在同一个程序中混合新旧形式的 iostream（包括标准的输入和输出流 cin、cout 和 cerr）可能会导致严重的问题，建议不要这样做。

在使用传统的 iostream 时，您可以为 iostream 类编写自己的前向声明，而不要包括某个 iostream 头。例如：

编码示例 4-3 传统 iostream 的前向声明

```
// 仅对传统 iostream 有效
class istream;
class ostream;
class Myclass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

这种方法不适用于标准的 iostream，因为传统的名称（istream、ofstream、streambuf 等等）不是标准的 iostream 中的类名称。它们是指类模板专门化的类型定义。

在使用标准的 iostream 时，您不能提供自己的 iostream 类前向声明。而是应该提供正确的 iostream 类前向声明，包括标准头 <iosfwd>。

编码示例 4-4 标准 iostream 的前向声明

```
// 仅对标准 iostream 有效
#include <iosfwd>
using std::istream;
using std::ostream;
class MyClass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

要编写可以同时用于标准和传统 iostream 的代码，您可以包括完整的头，而不使用前向声明。例如：

编码示例 4-5 传统和标准 iostream 的代码

```
// 对使用 Sun C++ 的传统和标准 iostream 有效
#include <iostream.h>
class MyClass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

4.2 任务（协同程序）库

现在不再支持协同程序库，该库可以通过 <task.h> 头访问。与协同程序库相比，Solaris 线程已更好地集成到语言开发工具（尤其是调试器）和操作系统中。

4.3 Rogue Wave Tools.h++

C++ 编译器包含两种版本的 Tools.h++ 库：

- **一种版本用于传统的 iostream**。该版本的 Tools.h++ 库与随早期的编译器版本提供的 Tools.h++ 库兼容。
  - **标准模式**。要在标准模式 （缺省模式）下使用 Tools.h++ 的传统 iostream 版本，使用 `-library=rwtools7,iostream` 选项。
  - **兼容模式**。要在兼容模式 (`-compat[=4]`) 下使用 Tools.h++ 传统的 iostream 版本，请使用 `-library=rwtools7` 选项。
- **一种版本用于标准的 iostream**。该版本的 Tools.h++ 库与 Tools.h++ 的传统 iostream 版本不兼容。该版本只适用于标准模式。它不适用于兼容模式 (`-compat[=4]`)。

  要使用库的标准 iostream 版本，请使用 `-library=rwtools7_std` 选项。

有关访问 `Tools.h++` 的更多信息，请参阅 《*C++ 用户指南*》或 `CC`(1) 手册页。

4.4 C 库头

在兼容模式下，您象以前那样使用 C 语言的标准头。头位于 `/usr/include` 目录中，它随您在使用的 Solaris 软件版本一起提供。

C++ 标准更改了标准的 C 头的定义。

为了表达清楚，此处讨论的头为 ISO C 标准 (ISO 9899:1990) 及以后的增补标准 （1994 年）中规定的 17 个头：

```
<assert.h>  <ctype.h>   <errno.h>   <float.h>   <iso646.h>
<limits.h>  <locale.h>  <math.h>    <setjmp.h>  <signal.h>
<stdarg.h>  <stdio.h>   <stdlib.h>  <string.h>  <time.h>
<wchar.h>   <wctype.h>
```

`/usr/include` 目录及子目录中数百个其他的头不会受到该语言更改的影响，因为它们不属于 C 语言标准。

您可以象使用以前的 Sun C++ 版本那样，在 C++ 程序中包括和使用这些头，但是有一些限制。

C++ 标准需要这些头中的类型、对象和函数名称出现在名称空间 std 以及全局名称空间中。这也就意味着不能直接使用随 Solaris 7 操作环境提供的这些头的版本。如果您在标准模式下编译，必须使用随 C++ 编译器提供的这些头的版本。如果您使用了错误的头，程序编译或链接可能会失败。

在使用 Solaris 7 操作环境时，您必须使用标准的头名称拼写，而不是路径名称。例如，编写以下代码：

```
#include <stdio.h> // 正确
```

而不是编写这些代码：

```
#include "/usr/include/stdio.h"    // 错误
#include </usr/include/stdio.h>    // 错误
```

在使用 Solaris 8 操作环境时， /usr/include 中的标准 C 头对于 C++ 语言正确无误，并且 C++ 编译器会自动使用这些头。也就是说，如果您编写

```
#include <stdio.h>
```

您在 Solaris 7 操作环境中编译时，将会得到 stdio.h 的 C++ 编译器版本，而在 Solaris 8 操作环境中编译时，将会得到 stdio.h 的 Solaris 版本。在使用 Solaris 8 操作环境时，在包含语句中使用显式路径名没有任何限制。然而，使用路径名 （例如 </usr/include/stdio.h>）确实使代码不可移植。

对于 17 个标准的 C 头，C++ 标准还为每一个头引入了另外一个版本。对于 <NAME.h> 形式的每一个头文件，都有另外一个 <cNAME> 形式的头文件。也就是说，删除结尾的 “.h”，并在前面加入 “c”。一些例子包括：<cstdio>, <cstring>, <cctype>。

这些头文件包含头文件原始形式的名称，但是只出现在名称空间 std 中。以下是符合 C++ 标准的一个实例：

```
#include <cstdio>
int main() {
    printf("Hello, ");        // 错误，printf 未知
    std::printf("world!\n"); // 正确
}
```

由于代码使用了 <cstdio>，而不是使用 <stdio.h>，因此名称 printf 只出现在名称空间 std 中，而不出现在全局名称空间中。您必须限定名称 printf，或者增加一个 *using- 声明*：

```
#include <cstdio>
using std::printf;
int main() {
    printf("Hello, ");        // 正确
    std::printf("world!\n"); // 正确
}
```

/usr/include 中的标准 C 头包含了许多 C 标准不允许使用的声明。这些声明的存在是由于历史的原因，主要是因为 UNIX 系统通常在那些头中有额外的声明，或因为其他标准 （如 POSIX 或 XOPEN）需要它们。为了继续兼容，这些额外的名称出现在 <NAME.h> 头的 Sun C++ 版本中，但是只出现在全局名称空间中。这些额外的名称不出现在头的 <cNAME> 版本中。

由于以前的任何程序从未使用这些新头，因此不存在兼容问题或历史问题。结果，您可能不会发觉 <cNAME> 头对一般编程非常有用。然而，如果您要编写尽可能可以移植的标准 C++ 代码，一定要确保 <cNAME> 头不包含任何不可移植的声明。下面的例子使用了 <stdio.h>：

```
#include <stdio.h>
extern FILE* f; // std::FILE 应该也可以
int func1() { return fileno(f); }      // 正确
int func2() { return std::fileno(f); } // 错误
```

下面的例子使用了 <cstdio>：

```
#include <cstdio>
extern std::FILE* f; // FILE 仅在名称空间 std 中
int func1() { return fileno(f); }      // 错误
int func2() { return std::fileno(f); } // 错误
```

函数 fileno 是一个额外的函数，为了兼容，它继续留在 <stdio.h> 中，但是它只出现在全局名称空间中，而不出现在名称空间 std 中。由于它是额外的函数，它根本不出现在 <cstdio> 中。

C++ 标准允许在同一个编译单元中同时使用标准 C 头的 <NAME.h> 版本和 <cNAME> 版本。尽管您不会故意这样做，但这种情况还是可能会发生，例如，您可能会在自己的代码中包括 <cstdlib>，或者您使用的某些项目头可能包括 <stdlib.h>。

4.5 实现标准头

《*C++ 用户指南*》详细介绍了如何实现标准头以及为什么要使用这种实现方法。在您包括任何标准的 C 或 C++ 头时，编译器实际上搜索指定名称中以 “.SUNWCCh”作为后缀的文件。例如， <string> 导致搜索 <string.SUNWCCh>，而 <string.h> 导致搜索 <string.h.SUNWCCh>。编译器的 include 目录包含名称的两种拼写，并且每一对拼写指同一个文件。例如，在目录 include/CC/Cstd 中，您会发现 string 和 string.SUNWCCh 同时存在。它们指同一个文件，即您在包括 <string> 时获得的文件。

在错误消息和调试器信息中，后缀不会显示。如果您包括 <string>，错误消息和该文件的调试引用将提到 string。文件依存性信息使用名称 string.SUNWCCh，从而避免由于名称不带后缀使缺省的 makefile 规则有问题。如果您只是想搜索头文件 （例如使用 find 命令），可以搜索 .SUNWCCh 后缀。

第 5 章

从 C 语言移至 C++ 语言

本章介绍如何将程序从 C 语言移至 C++ 语言。

C 程序一般需要作少量的修改，才能作为 C++ 程序编译。 C 和 C++ 在链接方面相互兼容。您不必修改已编译的 C 代码以便与 C++ 代码链接。有关 C++ 语言的参考书列表，请参阅在第 xvii 页上的 “ 市面有售的参考书 ” 。

5.1 保留字和预定义字

表 5-1 显示了 C++ 语言和 C 语言中的保留的关键字，以及 C++ 语言预定义的关键字。C++ 语言保留但 C 语言未保留的关键字以**粗体**表示。

表 5-1 保留的关键字

asm	do	if	return	typedef
auto	double	**inline**	short	**typeid**
bool	**dynamic_cast**	int	signed	typename
break	else	long	sizeof	union
case	enum	**mutable**	static	unsigned
catch	**explicit**	**namespace**	**static_cast**	**using**
char	**export**	**new**	struct	**virtual**
class	extern	**operator**	switch	void
const	**false**	**private**	**template**	volatile
const_cast	float	**protected**	**this**	**wchar_t**

表 5-1　保留的关键字　*（续）*

continue	for	**public**	**throw**	while
default	**friend**	register	**true**	
delete	goto	**reinterpret_cast**	**try**	

__STDC__ 预定义的值为 0。例如：

```
#include <stdio.h>
main()
{
    #ifdef __STDC__
        printf("yes\n");
    #else
        printf("no\n");
    #endif

    #if    __STDC__ ==0
        printf("yes\n");
    #else
        printf("no\n");
    #endif
}
```

产生：

```
yes
yes
```

下表列出的保留字是 C++ 标准中某些运算符和标点符的另一种表示法。

表 5-2　运算符和标点符的 C++ 保留字

and	bitor	not	or	xor
and_eq	compl	not_eq	or_eq	xor_eq
bitand				

5.2 创建通用头文件

K&R C、ANSI C 和 C++ 语言需要不同的头文件。要使 C++ 头文件符合 K&R C 和 ANSI C 标准以便能够通用，请使用宏 `__cplusplus`，将 C++ 代码与 C 代码区别开来。宏 `__STDC__` 在 ANSI C 和 C++ 中同时定义。使用该宏，将 C++ 或 ANSI C 代码与 K&R C 代码区别开来。有关详细信息，请参见 《*C++ 编程指南*》。

注 – 早期的 C++ 编译器预定义了宏 `c_plusplus`，现在的编译器不再支持该宏，而是使用 `__cplusplus`。

5.3 链接至 C 函数

编译器给 C++ 函数名编码以便允许重载。要调用 C 函数或调用 “伪装”成 C 函数的 C++ 函数，您必须阻止这种编码。为此，可以使用 `extern "C"` 声明。例如：

```
extern "C" {
double sqrt(double); //sqrt(double) 包含 C 链接
    }
```

该链接规范不影响程序使用 `sqrt()` 的语义，而只是导致编译器对 `sqrt()` 使用 C 语言命名惯例。

重载 C++ 函数集合中只能有一个函数具有 C 链接。您可以将 C 链接用于要通过 C 程序调用的 C++ 函数，但是您只能使用该函数的一个实例。

您不能在函数定义内部指定 C 链接。此类声明只能在全局范围中完成。

5.4 C 语言和 C++ 语言中的内联函数

如果内联函数定义位于可以同时使用 C 编译器和 C++ 编译器编译的源代码中，函数必须符合以下限制条件：

- 内联函数声明和定义必须加上有条件的 `extern "C"` 语句，如下面的例子所示。

```
#ifdef __cplusplus
extern "C" {
#endif
inline int twice( int arg ) { return arg + arg; }
#ifdef __cplusplus
}
#endif
```

- 内联函数的声明和定义必须符合两种语言强加的限制条件。
- 函数的语义在使用两种编译器时必须相同。请参见 C++ 标准的附录 D，了解在两种语言中可能产生不同语义的程序构造。

索引

数字

英文字母