C++ 用户指南

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

文件号码 819-4815-10
2005 年 1 月，修订版 A

请将有关本文档的意见和建议提交至： http://www.sun.com/hwdocs/feedback

目录

表

代码样例

阅读本书之前

本手册指导您使用 Sun™ Studio 11 的 C++ 编译器，并提供了有关命令行编译器选项的详细信息。本手册适用于具有 C++ 使用经验并对 Solaris™ 操作系统和 UNIX® 命令有一定了解的编程人员。

本书的结构

本手册包含以下主题：

C++ 编译器。第 1 章提供了有关编译的介绍性信息，诸如标准一致性和新特性。第 2 章说明了如何使用编译器。第 3 章介绍了如何使用编译器的命令行选项。

编写 C++ 程序。第 4 章介绍了如何编译通常可被其他 C++ 编译器接受的非标准代码。第 5 章建议设置和组织头文件和模板定义。第 6 章介绍了如何创建和使用模板。第 7 章说明了用于编译模板的各种选项。第 8 章介绍了异常处理，类型转换操作的信息则位于第 9 章。第 10 章介绍了显著影响 C++ 编译器的性能技术。第 11 章则提供了生成多线程程序的信息。

库。第 12 章说明了如何使用编译器提供的库。第 13 章介绍了 C++ 标准库。第 14 章介绍了用于兼容模式的传统 `iostream` 库。第 15 章介绍了用于兼容模式的复数运算库。第 16 章则提供了关于生成库的信息。

附录。附录 A 按字母顺序列出了一整套 C++ 编译器选项；附录 B 列出了 C++ 编译器 pragma。

印刷约定

表 P-1 字体约定

字体*	含义	示例
AaBbCc123	命令、文件和目录的名称；计算机屏幕输出。	编辑 .login 文件。 使用 ls -a 列出所有文件。 % You have mail.
AaBbCc123	用户键入的内容，与计算机屏幕输出的显示不同。	% **su** Password:
AaBbCc123	保留未译的新词或术语以及要强调的词。要使用实名或值替换的命令行变量。	这些称为 *class* 选项。 要删除文件，请键入 **rm** *filename*。
新词术语强调	新词或术语以及要强调的词。	您**必须**成为超级用户才能执行此操作。
《书名》	书名	阅读 《用户指南》的第 6 章。

* 浏览器的设置可能会与这些设置不同。

表 P-2 代码约定

代码符号	含义	表示法	代码示例
[ ]	方括号中包含可选参数。	O[*n*]	-O4, -O
{ }	花括号中包含所需选项的选项集合。	d{y|n}	-dy
|	分隔变量的 “|” 或 “-” 符号，只能选择其一。	B{dynamic|static}	-Bstatic
:	与逗号一样，分号有时可用于分隔参数。	R*dir*[:*dir*]	-R/local/libs:/U/a
…	省略号表示一系列的省略。	-xinline=*f1*[,…*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、 Korn shell 和 GNU Bourne-Again shell	$
Bourne shell、 Korn shell 和 GNU Bourne-Again shell 的超级用户	#

支持的平台

此 Sun Studio 发行版本支持使用 SPARC® 和 x86 系列处理器体系结构（UltraSPARC®、SPARC64、 AMD64、 Pentium 和 Xeon EM64T）的系统。通过访问 `http://www.sun.com/bigadmin/hcl` 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本的支持系统。这些文档列出了实现各个平台类型的所有差别。

在本文档中，这些与 x86 有关的术语具有以下含义：

- “x86” 是指较大的 64 位和 32 位 x86 兼容产品系列。
- “x64” 表示有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。

有关所支持的系统，请参见硬件兼容性列表。

访问 Sun Studio 软件和手册页

Sun Studio 软件及其手册页未安装到 `/usr/bin/` 和 `/usr/share/man` 标准目录中。要访问软件，必须正确设置 `PATH` 环境变量 （请参见第 xxx 页的 “访问软件”）。要访问手册页，必须正确设置 `MANPATH` 环境变量 （请参见第 xxx 页的 “访问手册页”）。

有关 PATH 变量的详细信息，请参见 csh(1)、 sh(1)、 ksh(1) 和 bash(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本的详细信息，请参见安装指南或询问系统管理员。

注 – 本节中的信息假设 Sun Studio 软件安装在 Solaris 平台上的 /opt 目录中和 Linux 平台上的 /opt/sun 目录中。如果未将软件安装在默认目录中，请询问系统管理员以获取系统中的相应路径。

访问软件

使用以下步骤来决定是否需要更改 PATH 变量以访问该软件。

决定是否需要设置 PATH 环境变量

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

```
% echo $PATH
```

2. **在 Solaris 平台上，查看输出中是否包含有 /opt/SUNWspro/bin 的路径字符串。在 Linux 平台上，查看输出中是否包含有 /opt/sun/sunstudio11/bin 的路径字符串。**

   如果找到该路径，则说明已设置了访问该软件的 PATH 变量。如果没有找到该路径，则按照下一步的说明来设置 PATH 环境变量。

设置 PATH 环境变量以实现对软件的访问

- **在 Solaris 平台上，将以下路径添加到 PATH 环境变量中。如果以前安装了 Forte Developer 软件、 Sun ONE Studio 软件，或其他发行版本的 Sun Studio 软件，则将以下路径添加到这些安装路径之前。**

  /opt/SUNWspro/bin

- **在 Linux 平台上，将以下路径添加到 PATH 环境变量中。**

  /opt/sun/sunstudio11/bin

访问手册页

使用以下步骤决定是否需要更改 MANPATH 变量以访问手册页。

决定是否需要设置 MANPATH 环境变量

1. **通过在命令提示符后键入以下内容以请求 dbx 手册页。**

```
% man dbx
```

2. **请查看输出 （如果有）。**

   如果找不到 dbx(1) 手册页或者显示的手册页不是软件当前版本的手册页，请按照下一步的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以实现对手册页的访问

- **在 Solaris 平台上，将以下路径添加到 MANPATH 环境变量中。**

  /opt/SUNWspro/man

- **在 Linux 平台上，将以下路径添加到 MANPATH 环境变量中。**

  /opt/sun/sunstudio11/man

访问集成开发环境

Sun Studio 集成开发环境 (integrated development environment, IDE) 提供了创建、编辑、生成、调试 C、 C++ 或 Fortran 应用程序并分析其性能的模块。

启动 IDE 的命令是 sunstudio。有关该命令的详细信息，请参见 sunstudio(1) 手册页。

IDE 是否可以正确操作取决于 IDE 能否找到核心平台。 sunstudio 命令查找两个位置的核心平台：

- 该命令首先查找 Solaris 平台上的默认安装目录 /opt/netbeans/3.5V11 和 Linux 平台上的默认安装目录 /opt/sun/netbeans/3.5V11。
- 如果该命令在默认目录找不到核心平台，则它会假设包含 IDE 的目录和包含核心平台的目录均安装在同一位置上。例如，在 Solaris 平台上，如果包含 IDE 的目录的路径是 /foo/SUNWspro，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。在 Linux 平台上，如果包含 IDE 的目录的路径是 /foo/sunstudio11，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。

如果核心平台未安装在 sunstudio 命令查找它的任一位置上，则客户端系统上的每个用户必须将环境变量 SPRO_NETBEANS_HOME 设置为安装核心平台的位置 (/*installation_directory*/netbeans/3.5V11)。

在 Solaris 平台上，IDE 的每个用户还必须将 */installation_directory*/SUNWspro/bin 添加到其他任何 Forte Developer 软件、 Sun ONE Studio 软件或 Sun Studio 软件发行版本路径前面的 $PATH 中。在 Linux 平台上， IDE 的每个用户还必须将 */installation_directory*/sunstudio11/bin 添加到其他任何发行版本的 Sun Studio 软件路径前面的 $PATH 中。

路径 */installation_directory*/netbeans/3.5V11/bin 不能添加到用户的 $PATH 中。

访问 Sun Studio 文档

您可以访问下列位置的文档：

- 可以通过随软件一起安装在本地系统或网络上的文档索引获取文档，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。

  如果未将软件安装在 Solaris 平台上的 /opt 目录中或 Linux 平台上的 /opt/sun 目录中，请询问系统管理员以获取系统中的相应路径。

- 大多数的手册都可以从 docs.sun.com℠ Web 站点获取。以下书目只能从 Solaris 平台上安装的软件中找到：
  - 《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》
- 适用于 Solaris 平台和 Linux 平台的发行说明可以通过 docs.sun.com Web 站点获取。
- 在 IDE 中通过“帮助”菜单以及许多窗口和对话框上的“帮助”按钮，可以访问 IDE 所有组件的联机帮助。

您可以通过 Internet 访问 docs.sun.com Web 站点 (http://docs.sun.com) 以阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到手册，请参见和软件一起安装在本地系统或网络中的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的 （或通过它们获得的）任何内容、产品或服务而造成的或连带产生的实际或名义损坏或损失，Sun 概不负责，也不承担任何责任。

使用易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。您还可以按照下表所述，找到文档的易读版本。如果未将软件安装在 /opt 目录中，请询问系统管理员以获取系统中的相应路径。

文档类型	易读版本的格式和位置
手册 （第三方手册除外）	HTML，位于 http://docs.sun.com
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，位于 Solaris 平台上所安装软件中的文档索引 file:/opt/SUNWspro/docs/zh/index.html
自述文件	HTML，位于 Sun Developer Network 门户网站 http://developers.sun.com/prodtech/cc/documentation/
手册页	HTML，位于安装的软件上的文档索引，位置为 Solaris 平台上的 file:/opt/SUNWspro/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。
联机帮助	HTML，可通过 IDE 中的 “帮助”菜单和 “帮助”按钮访问
发行说明	HTML，位于 http://docs.sun.com

相关文档

对于 Solaris 平台，下表描述的相关文档可以在 /opt/SUNWspro/docs/zh/index.html 和 http://docs.sun.com 上获取。如果未将软件安装在 /opt 目录中，请询问系统管理员以获取系统中的相应路径。

文档标题	描述

对于 Linux 平台，下表描述的相关文档可以在 file:/opt/sun/sunstudio11/docs/zh/index.html 和 http://docs.sun.com 上获取。如果未将软件安装在 /opt/sun 目录中，请询问系统管理员以获取系统中的相应路径。

文档标题	描述

访问相关的 Solaris 文档

下表描述了可从 `docs.sun.com` Web 站点上获取的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参见手册页部分的标题。	提供有关 Solaris 操作系统的信息。
Solaris 软件开发者集合	《链接程序和库指南》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。

访问相关的 C++ 手册页

本手册提供了可用于 C++ 库的手册页列表。下表列出了与 C++ 相关的其他手册页。

标题	描述
`c++filt`	按顺序复制每个文件名，并在解码类似 C++ 还原名称的符号之后将文件名写入标准输出
`dem`	还原指定的一个或多个 C++ 名称
`fbe`	从汇编语言源文件创建对象文件
`fpversion`	输出系统 CPU 和 FPU 的相关信息
`gprof`	生成程序的可执行性能分析数据
inline	扩展汇编程序的内联过程调用
`lex`	生成词法分析程序
`rpcgen`	生成 C/C++ 代码以实现 RPC 协议
`sigfpe`	允许对指定 SIGFPE 代码进行信号处理
`stdarg`	处理变量参数列表
`varargs`	处理变量参数列表
版本	显示对象文件或二进制文件的版本标识
`yacc`	将上下文无关的语法转换成一组表，用于执行 LALR(1) 分析算法的简单自动化

其他公司出版的书籍

以下是部分 C++ 语言书籍的列表：

The C++ Programming Language 3rd edition， Bjarne Stroustrup 所著 (Addison-Wesley, 1997)。

The C++ Standard Library， Nicolai Josuttis 所著 (Addison-Wesley, 1999)。

Generic Programming and the STL， Matthew Austern 所著 (Addison-Wesley, 1999)。

Standard C++ IOStreams and Locales，Angelika Langer 和 Klaus Kreft 所著 (Addison-Wesley, 2000)。

Thinking in C++, Volume 1, Second Edition， Bruce Eckel 所著 (Prentice Hall, 2000)。

The Annotated C++ Reference Manual， Margaret A. Ellis 和 Bjarne Stroustrup 所著 (Addison-Wesley, 1990)。

Design Patterns: Elements of Reusable Object-Oriented Software， Erich Gamma、Richard Helm、 Ralph Johnson 和 John Vlissides 所著 (Addison-Wesley, 1995)。

C++ Primer, Third Edition，Stanley B. Lippman 和 Josee Lajoie 所著 (Addison-Wesley, 1998)。

Effective C++－50 Ways to Improve Your Programs and Designs Second Edition，Scott Meyers 所著 (Addison-Wesley, 1998)。

More Effective C++ － 35 Ways to Improve Your Programs and Designs，Scott Meyers 所著 (Addison-Wesley, 1996)。

开发者资源

访问 Sun Developer Network Sun Studio 门户网站
`http://developers.sun.com/prodtech/cc` 以查找以下经常更新的资源：

- 关于编程技术和最佳实例的文章
- 有关编程小技巧的知识库
- 软件的文档，以及随软件一同安装的文档的更正信息
- 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

Sun Studio 门户网站是 Sun Developer Network 网站 `http://developers.sun.com` 上的很多额外开发者资源之一。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下网址：

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

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下网址提交您的意见和建议：

`http://www.sun.com/hwdocs/feedback`

请在电子邮件的主题行中注明文档的文件号码。例如，本文档的文件号码是 819-4815-10。

第 I 部分　C++ 编译器

第 1 章

C++ 编译器

本章提供有关以下内容的信息：

1.1 Sun Studio 10 C++ 5.8 编译器的新特性和新功能

本节简要介绍在 Sun Studio 10 C++ 5.8 编译器发行版本中引入的 C 编译器的新特性和新功能。有关详细的说明，请参见每项的交叉引用。

- 用于 x86 开发的 `-xarch` 新标志

  `-xarch` 选项目前支持将以下新标志用于 x86 平台开发：`amd64a`、`pentium_proa`、`ssea`、`sse2a`。请参见第 A-69 页的第 A.2.105 节 "`-xarch=isa`"。

- 支持 x86 `-xpagesize` 选项

  目前为 x86 平台和 SPARC 启用了 `-xpagesize`、`-xpagesize_heap`、`-xpagesize_stack` 选项。请参见第 A-114 页的第 A.2.147 节 "`-xpagesize=n`"。

- 用于指定 x86 内存模型的 `-xmodel` 新选项

  通过使用 `-xmodel` 新选项，可以在 64 位 AMD 体系结构上指定内核、小型或中型内存模型。如果全局变量和静态变量的大小超过了 2 千兆字节，请指定 `-xmodel=medium`。否则，请使用 `-xmodel=small` 默认设置。有关详细信息，请参见第 A-106 页的第 A.2.140 节 "`-xmodel=[a]`" 。

- 支持 SSE/SSE2 整型介质内部函数

  本发行版本支持 SSE2 128 位 XMM 寄存器整型介质指令的内部函数。在源代码中包含 `sunmedia_intrin.h` 头文件，并指定 `-xbuiltin` 选项以利用这些函数。再者，这些内部函数需要 SSE2 支持，因此，请指定

  `-xarch=sse2`、`-xarch=amd64` 或 `-xtarget=opteron` 等选项。

  实质上，编译器为这些内部函数生成内联代码。这比通过汇编语言处理这些指令简单一些，并且编译器可以对其进行优化。

  有关内部函数的详细信息、头文件中包含的函数原型以及这些函数使用的数据类型的说明，请参见《用于 Linux 系统的 Intel(R) C++ 编译器》手册中的“Intel C++ 内部函数参考”一节。

- 用于 x86 SSE2 平台的 `-xvector` 新标志

  利用 `-xvector` 选项，可以自动生成向量库函数调用和 / 或生成 SIMD （Single Instruction Multiple Data, 单指令多数据）指令。有关详细信息，请参见第 A-147 页的第 A.2.173 节 "`-xvector[=a]`" 。

- 用于 SPARC 的二进制文件优化器

  通过 `-xbinopt` 新选项，编译器可以准备由 `binopt(1)` 二进制文件优化器进一步优化的二进制文件。有关详细信息，请参见第 A-75 页的第 A.2.107 节 "`-xbinopt={prepare|off}`" 。

- 从函数模板调用相关静态函数

  C++ 标准规定，取决于模板参数的函数调用只能引用具有外部链接的可见函数声明。如果应用程序代码取决于忽略此规则并从函数模板中调用相关静态函数的编译器，则指定 `-features=[no%]tmplrefstatic`。有关详细信息和示例，请参见第 A-17 页的第 A.2.19 节 "`-features=a[,a...]`"。

- `SPARC -xtarget` 和 `-xchip` 的新值

  `-xtarget` 的新标志 `ultra3iplus`、`ultra4plus` 和 `ultraT1` 以及 `-xchip` 的新标志 `ultra3iplus`、`ultra4plus` 和 `ultraT1` 为 UltraSPARC IIIiplus、UltraSPARC T1 和 UltraSPARC IVplus 处理器提供代码生成。有关详细信息，请参见第 A-136 页的第 A.2.167 节 "`-xtarget=t`" 和第 A-80 页的第 A.2.113 节 "`-xchip=c`"。

- 新的调试器信息格式

  C++ 编译器目前可以使用 dwarf 格式来生成调试器信息。默认格式仍然为 stabs 格式，但可以通过将新选项 `-xdebugformat` 设置为 `-xdebugformat=dwarf` 来生成 dwarf 数据。请参见第 A-85 页的第 A.2.116 节 "`-xdebugformat=[stabs|dwarf]`"。

- STACKSIZE 环境变量增强功能

  已增强 STACKSIZE 环境变量的语法，可接受用于表示从属线程栈大小的单位关键字：B 表示字节； K 表示千字节； M 表示兆字节； G 表示千兆字节。

  例如， setenv STACKSIZE 8192 将从属线程栈大小设置为 8 MB。 1235B 将从属线程栈大小设置为 1235 字节； 1235G 将其设置为 1235 千兆字节。默认情况下，没有后缀字母的整数值仍然为千字节。

- OpenMP 自动确定作用域

自动确定作用域目前适用于 C++ 程序。 Sun Studio 《OpenMP API 用户指南》的第 3 章中介绍了该功能。

1.2 Sun Studio 10 C++ 5.7 编译器的新特性和新功能

本节简要介绍在 Sun Studio 10 C++ 5.7 编译器发行版本中引入的 C++ 编译器的新特性和新功能。有关详细的说明，请参见每项的交叉引用。

- `-xarch` 新选项 `-xarch=amd64` 指定了 64 位 AMD 指令集编译。有关 `-xarch=amd64` 的详细信息，请参见第 A-69 页的第 A.2.105 节 "`-xarch=isa`"。
- `-xtarget` 新选项 `-xtarget=opteron` 为 32 位 AMD 编译指定了 `-xarch`、 `-xchip` 和 `-xcache` 设置。有关 `-xtarget=opteron` 的详细信息，请参见第 A-136 页的第 A.2.167 节 "`-xtarget=t`"。

注 – 要生成 64 位代码，您必须在命令行中 `-fast` 和 `-xtarget` 的右侧指定 `-xarch=amd64`。例如，指定 `CC -fast -xarch=amd64` 或 `CC -xtarget=opteron -xarch=amd64`。 `-xtarget=opteron` 新选项并不自动生成 64 位代码。它扩展为 `-xarch=sse2`、`-xchip=opteron` 和 `-xcache=64/64/2:1024/64/16`，而产生 32 位代码。`-fast` 选项也会产生 32 位代码，因为它也是一个定义 `-xtarget=native` 的宏。

- 除传统 SPARC 平台外，现有 `-xarch=generic64` 选项现在还支持 x86 平台。
- 如果指定了 `-xarch=amd64`， C++ 编译器现在预定义 `__amd64` 和 `__x86_64`。
- 利用 `-xregs` 选项 `-xregs=[no%]frameptr` 仅限 x86 的新标志，您可以将帧指针寄存器用作未分配的被调用方保存寄存器以提高应用程序的运行时性能。

  有关 `-xregs=[no%]frameptr` 的详细信息，请参见第 A-131 页的第 A.2.160 节 "`-xregs=r[,r...]`"。
- C++ 编译器现在支持模板 - 模板参数。这意味着，可以使用本身就是模板的参数来指定模板定义，而不是使用类型或值来指定。请回想一下，在类型上实例化的模板本身就是类型。考虑以下代码示例：

```
template<typename T> class MyClass { ... };
std::list< MyClass<int> > x;
```

因为 MyClass<int> 是一种类型，所以代码示例并不使用模板 — 模板参数。然而，在以下代码示例中，类模板 C 的参数是类模板，而对象 x 则是 C 的实例，它使用类模板 A 作为其参数。C 的成员 y 具有**类型** A<int>。

```
// 普通类模板
template<typename T> class A {
    T x;
};

// 具有模板参数的类模板
template< template<typename U> class V > class C {
    V<int> y;
};
// 在模板上实例化 C
C<A> x;
```

- 在默认标准模式下， C++ 编译器现在允许嵌套类访问封装类的专有成员。

  C++ 标准规定嵌套类没有封装类成员的特殊访问权限。然而，大多数人认为这种限制不合理，因为成员函数拥有专有成员的访问权限，因此，成员类也应该有此权限。在以下示例中，函数 foo 试图访问类 outer 的专有成员。按照 C++ 标准，函数没有访问权限，除非将其声明为友元函数：

```
class outer {
    int i; // 在 outer 中是专有的
    class inner {
        int foo(outer* p) {
            return p->i; // 无效
        }
    };
};
```

  C++ 委员会正在采纳对访问规则所做的更改，以便给成员类授予与成员函数相同的访问权限。由于预料到将会更改语言规则，很多编译器已经实现了这种规则。

  要恢复旧的编译器行为，而禁用这种访问权限，请使用编译器选项 -features=no%nestedaccess。默认为 -features=nestedaccess。有关 -features 的详细信息，请参见第 A-17 页的第 A.2.19 节 "-features=a[,a...]"。

- 此发行版在基于 x86 系统的 Solaris 操作系统和基于 SPARC 系统的 Solaris 操作系统上提供了 OpenMP API 以实现共享内存并行性。这两种平台现在都启用相同的功能。

1.3 标准一致性

C++ 编译器 (CC) 支持 C++ ISO 国际标准 ISO IS 14882:1998，**编程语言** C++。当前发行版本附带的自述文件描述了与标准需求的所有差异。

在 SPARC™ 平台上，编译器提供了对 SPARC V8 和 SPARC V9（包括 UltraSPARC™ 实现）优化开发功能的支持。在 Prentice-Hall for SPARC International 发行的第 8 版 (ISBN 0-13-825001-4) 和第 9 版 (ISBN 0-13-099227-5) SPARC Architecture Manual 中定义了这些功能。

在本文档中，“标准”是指与上面列出的标准版本相一致。“非标准”或“扩展”是指超出这些标准版本的功能。

负责标准的一方可能会不时地修订这些标准。C++ 编译器兼容的适用标准版本可能被修订或替换，这将会导致以后的 Sun C++ 编译器发行版本在功能上与旧的发行版本产生不兼容。

1.4 C++ 自述文件

C++ 编译器的自述文件强调了关于编译器的重要信息，其中包括：

- 在手册印刷之后发现的信息
- 新特性和更改的特性
- 软件更正
- 问题和解决办法
- 限制和不兼容
- 可发送库
- 未实现的标准

要查看 C++ 自述文件的文本格式文件，请在命令提示符后键入以下命令：

```
example% CC -xhelp=readme
```

要访问自述文件的 HTML 格式文件，请在您的 Netscape Communicator 4.0 或兼容版本的浏览器中打开以下文件：

```
/opt/SUNWspro/docs/zh/index.html
```

（如果您的 C++ 编译器软件没有安装在 /opt 目录中，请通过系统管理员获取系统中的相应路径。）浏览器可以显示 HTML 文档的索引。要打开自述文件，请在索引中查找它的对应条目，然后单击主题。

1.5 手册页

联机手册 (man) 页提供了关于命令、函数、例行程序以及收集这些信息的文档。

可以通过运行以下命令来显示手册页：

```
example% man topic
```

在整个 C++ 文档中，手册页参考以主题名称和手册节编号显示：cc(1) 通过 man cc 进行访问。其他部分 （例如用 ieee_flags(3M) 表示的节）要使用 man 命令和该命令的 -s 选项来访问：

```
example% man -s 3M ieee_flags
```

1.6 C++ 实用程序

以下 C++ 实用程序现已并入传统的 UNIX® 工具并且与 UNIX 操作系统捆绑在了一起：

- lex 生成文本简单词法分析的程序
- yacc 根据语法生成 C 函数分析输入流
- prof 生成程序模块的执行性能分析
- gprof 按过程来配置程序运行时性能
- tcov 按语句来配置程序运行时性能

关于这些 UNIX 工具的详细信息，请参见 《程序性能分析工具》和相关的手册页。

1.7 本地语言支持

此发行版本的 C++ 支持使用英语以外的其他语言进行应用程序的开发，包括了大多数欧洲语言和日语。因此，您可以十分便捷地将应用程序从一种语言切换到另一种语言。此功能被称为**国际化**。

通常， C++ 编译器按如下方式实现国际化：

- C++ 从国际化的键盘识别 ASCII 字符（也就是说，它具有键盘独立性和 8 位清除）。
- C++ 允许使用本地语言打印某些消息。

- C++ 允许在注释、字符串和数据中使用本地语言。
- C++ 只支持符合扩展 UNIX 字符 (EUC) 的字符集，在该字符集中字符串内的每个空字节是一个空字符，而每个 "/" 的 ascii 值是 "/" 字符。

变量名称不能国际化，必须使用英文字符集。

您可以设置语言环境将应用程序从一种本地语言更改为另一种语言。关于这一点和其他本地语言支持功能的信息，请参见操作系统文档。

第2章

使用 C++ 编译器

本章描述了如何使用 C++ 编译器。

任何编译器的主要用途是将高级语言 （如 C++）编写的程序转换成目标计算机硬件可执行的数据文件。您可以使用 C++ 编译器完成以下任务：

- 将源文件转换成可重定位的二进制 (`.o`) 文件、静态 （归档）库 (`.a`) 文件 （使用 `-xar`）或动态 （共享）库 (`.so`) 文件。其中二进制文件可以在以后链接成可执行文件。
- 将对象文件或库文件 （或两者）链接或重链接成可执行文件
- 启用运行时调试 (`-g`) 来编译可执行文件
- 启用运行时语句或过程级别的文件配置 (`-pg`) 来编译可执行文件

2.1 入门

本节简要概述了如何使用 C++ 编译器编译和运行 C++ 程序。关于命令行选项的完整参考，请参见附录 A。

注 – 本节中的命令行示例显示了 `CC` 的用法。打印输出可能会稍有不同。

生成和运行 C++ 程序的基本步骤包括：

1. 使用编辑器创建具有表 2-1 中列出的有效后缀之一的 C++ 源文件
2. 调用编译器来生成可执行文件
3. 通过输入可执行文件的名称来启动程序

以下程序在屏幕上显示消息：

```
example% cat greetings.cc
       #include <iostream>
    int main() {
      std::cout << "Real programmers write C++!" << std::endl;
      return 0;
    }
example% CC greetings.cc
example% a.out
 Real programmers write C++!
example%
```

在此示例中，CC 编译源文件 greetings.cc，并且在默认情况下将编译可执行程序生成 a.out 文件。要启动程序，请在命令行提示符后键入可执行文件 a.out 的名称。

按传统方法，UNIX 编译器为可执行文件 a.out 命名。每次编译都写入到同一个文件是比较笨拙的方法。另外，如果已经有这样一个文件存在，下次运行编译器时该文件将被覆盖。如以下示例所示，改为使用 -o 编译器选项来指定可执行输出文件的名称：

```
example% CC -o greetings greetings.C
```

在此示例中，-o 选项告知编译器将可执行代码写入文件 greetings。（由单独源文件组成的程序通常提供无后缀的源文件名称。）

或者，可以在每次编译后使用 mv 命令来重命名默认的 a.out 文件。无论是哪种方式，都可以键入可执行文件的名称来运行程序：

```
example% greetings
Real programmers write C++!
example%
```

2.2 调用编译器

本节的其他部分讨论了使用 CC 命令的约定、编译器源代码行指令和编译器使用的其他有关问题。

2.2.1 命令语法

编译器命令行的通用语法如下所示：

```
CC [options] [source-files] [object-files] [libraries]
```

option 是前缀为短线 (-) 或加法符号 (+) 的选项关键字。某些选项带有参数。

通常，编译器选项的处理是由左到右，允许有选择地覆盖宏选项（即包括其他选项的选项）。在大多数的情况下，如果您多次指定同一个选项，那么最右边的赋值会覆盖前面的赋值，而不会累积。注意以下特殊情况：

- 所有的链接程序选项和 `-features`、`-I`、`-l`、`-L`、`-library`、`-pti`、`-R`/`-staticlib`、`-U`、 `-verbose`、 `-xdumpmacros` 和 `-xprefetch` 选项将会累积而不会覆盖。
- 所有的 `-U` 选项都在所有的 `-D` 选项后处理。

源文件、对象文件和库按它们在命令行上出现的顺序编译并链接。

在以下示例中，`CC` 用于编译两个源文件（`growth.C` 和 `fft.C`）以在启用运行时调试的情况下生成名为 `growth` 的可执行文件。

```
example% CC -g -o growth growth.C fft.C
```

2.2.2 文件名称约定

出现在命令行上文件名称的附加后缀决定了编译器处理文件的方式。如果文件名称的后缀没有在下表中列出，或文件名称没有后缀，那么都要传递到链接程序。

表 2-1 C++ 编译器识别的文件名称后缀

后缀	语言	操作
`.c`	C++	作为 C++ 源文件编译，将对象文件放入当前目录；对象文件的默认名称是带有 `.o` 后缀的源名称。
`.C`	C++	与 `.c` 后缀相同的操作。
`.cc`	C++	与 `.c` 后缀相同的操作。
`.cpp`	C++	与 `.c` 后缀相同的操作。
`.cxx`	C++	与 `.c` 后缀相同的操作。
`.c++`	C++	与 `.c` 后缀相同的操作。
`.i`	C++	将预处理程序输出文件作为 C++ 源文件处理。与 `.c` 后缀相同的操作。

表 2-1　C++ 编译器识别的文件名称后缀 (续)

后缀	语言	操作
.s	汇编程序	使用汇编程序的汇编源文件。
.S	汇编程序	使用 C 语言预处理程序和汇编程序的汇编源文件。
.il	内联扩展	处理内联扩展的汇编内联模板文件。编译器将使用模板来扩展选定例程的内联调用。（内联模板文件是特殊的汇编文件。请参见 inline(1) 手册页。）
.o	对象文件	将对象文件传递到链接程序。
.a	静态（归档）库	将对象库名传递到链接程序。
.so .so.*n*	动态（共享）库	将共享对象的名称传递到链接程序。

2.2.3　使用多个源文件

C++ 编译器在命令行上接受多个源文件。编译器编译的单独源文件与编译器直接或间接支持的任何文件一起称为**编译单元**。 C++ 将每个源作为一个单独的编译单元处理。

2.3　使用不同编译器版本进行编译

从 C++ 5.1 编译器开始，编译器就使用标识模板缓存版本的字符串来标记模板缓存目录。

此编译器在默认情况下不使用缓存。只有指定 -instances=extern 后，该编译器才使用缓存。如果编译器使用缓存，就要检查缓存目录的版本，并在遇到缓存版本问题时发出错误消息。以后的 C++ 编译器也会检查缓存的版本。例如，具有不同模板缓存版本标识的未来版本编译器在处理此发行版本的编译器生成的缓存目录时，会发出与以下消息类似的错误：

```
./SunWS_cache 的模板数据库与此编译器不兼容
```

编译器遇到新版本的编译器生成的缓存目录时，也会发出类似的错误。

尽管 C++ 5.0 编译器生成的模板缓存目录没有标记版本标识符，但是当前编译器在处理 5.0 缓存目录时不会发出错误或警告。编译器将 5.0 缓存目录转换为编译器使用的目录格式。

C++ 5.0 编译器不能使用新版本编译器生成的缓存目录。C++ 5.0 编译器不能识别格式差异，并在遇到由 C++ 5.1 或新版本编译器生成的缓存目录时将会发出断言。

升级编译器时，最好清除缓存。在包含模板缓存目录（在大多数的情况下，模板缓存目录命名为 SunWS_cache）的每个目录中运行 CCadmin -clean。或者，可以使用 rm -rf SunWS_cache。关于如何清除模板的最新说明，请参见 http://forte.sun.com/s1scc/articles/index.html 上的技术文章“升级 C++ 编译器”。

2.4 编译和链接

本节描述了编译和链接程序的某些方面。在以下示例中，CC 用来编译三个源文件，并链接对象文件以生成名为 prgrm 的可执行文件。

```
example% CC file1.cc file2.cc file3.cc -o prgrm
```

2.4.1 编译和链接序列

在上述示例中，编译器自动生成加载器对象文件（file1.o、file2.o 和 file3.o），然后调用系统链接程序为文件 prgrm 创建可执行程序。

编译以后，仍然保留对象文件（file1.o、file2.o 和 file3.o）。此约定让您易于重新链接和重新编译文件。

注 – 如果在同一个操作中仅编译一个源文件和链接一个程序，则相应的 .o 文件被自动删除。要保留所有的 .o 文件，除非要编译多个源文件，否则请不要在同一操作中进行编译和链接。

如果编译失败，您将收到每个错误的对应消息。不会为那些有错的源文件生成 .o 文件，也不会为它们生成可执行程序。

2.4.2 分别编译和链接

您可以分别进行编译和链接。 -c 选项编译源文件并生成 .o 对象文件，但不创建可执行文件。没有 -c 选项，编译器将调用链接程序。通过将编译和链接步骤分开，仅修复一个文件而不需要完全重新编译。以下示例显示了如何以独立的步骤编译一个文件并与其他文件链接：

```
example% CC -c file1.cc                                  生成新的对象文件
example% CC -o prgrm file1.o file2.o file3.o    生成可执行文件
```

确保链接步骤列出了生成完整程序所需的**全部**对象文件。如果这一步骤中丢失了所有的对象文件，则链接将失败并出现 “未定义的外部引用”错误 （丢失例程）。

2.4.3 一致编译和链接

如果以独立的步骤编译和链接，那么使用以下编译器选项来进行一致的编译和链接是十分重要的：

- -B
- -compat
- -fast
- -g
- -g0
- -library
- -misalign
- -mt
- -p
- -xa
- -xarch
- -xcg92 and -xcg89
- -xipo
- -xpagesize
- -xpg
- -xprofile
- -xtarget

如果您使用这些选项之一来**编译**子程序，请确保使用相同的选项进行**链接**：

- 在使用 -library、 -fast、 -xtarget 和 -xarch 选项时，必须确保包括了链接程序选项。如果编译和链接同时进行的话，就可以忽略这些链接程序选项。
- 对于 -p、-xpg 和 -xprofile，将选项包括在一个阶段而从其他阶段排除并不影响程序的正确性，但是您将不能进行文件配置。

- 对于 -g 和 -g0，将选项包括在一个阶段而从其他状态排除不影响程序的正确性，但影响调试程序的能力。任何使用 -g 或 -g0 来链接程序，但没有使用这两个选择中的一个进行编译的模块将无法正确调试。注意，调试通常需要使用 -g 选项或 -g0 选项编译具有函数 main 的模块。

以下示例中，使用 -xcg92 编译器选项来编译程序。此选项是用于 -xtarget=ss1000 的宏，可以扩展为：-xarch=v8 -xchip=super -xcache=16/64/4:1024/64/1。

```
example% CC -c -xcg92 sbr.cc
example% CC -c -xcg92 smain.cc
example% CC -xcg92 sbr.o smain.o
```

如果程序使用模板，则某些模板可能会在链接期间被实例化。在这种情况下，来自最后一行 （链接行）的命令行选项将用于编译实例化的模板。

2.4.4 为 SPARC V9 编译

只有在运行 64 位内核的 V9 SPARC Solaris 8 操作系统中才支持 64 位对象的编译、链接和执行。 64 位编译由 -xarch=v9、-xarch==v9a 和 -xarch=v9b 选项来指示。

2.4.5 诊断编译器

可以使用 -verbose 选项在编译程序的同时显示帮助信息，例如编译器调用的程序名称和版本号以及每个编译阶段的命令行。

编译器无法识别的命令行上的任何参数被解释为链接程序选项、对象程序文件名或库名。

基本区别是：

- 无法识别的**选项**会生成警告。这些选项一般带有前缀短线 (-) 或加法符号 (+)。
- 无法识别的**非选项**不会生成警告。这些非选项一般不带有前缀短线或加法符号。（然而，这些选项会传递到链接程序。如果链接程序无法识别它们，将会生成链接程序错误消息。）

以下示例中，注意 CC 无法识别 -bit，该选项被传递到尝试解释它的链接程序 (ld)。因为单字母 ld 选项可以被连在一起，所以链接程序将 -bit 视为 -b -i -t，所有这些都是合法的 ld 选项。这可能并不是您所希望看到的结果：

```
example% CC -bit move.cc              <- -bit 不是一个可识别的 CC 选项

CC: 警告：如果调用了 ld，则将选项 -bit 传递至 ld，否则会将其忽略
```

以下示例中，用户打算键入 CC 选项 -fast，但是省略前导短线。编译器又一次将参数传递到链接程序，而链接程序将参数解释为文件名称：

```
example% CC fast move.cc              <- 用户要键入 -fast。
move.CC:
ld: 致命的：文件 fast：打开失败；errno=2
ld: 致命的：文件处理失败。No output written to a.out
```

2.4.6 了解编译器的组织

C++ 编译器软件包由前端、优化器、代码生成器、汇编程序、模板预链接程序和链接编辑器组成。除非您使用命令行选项进行指定，否则 CC 命令将逐个自动调用这些组件。

因为这些组件中的任何一个都可能生成错误，并且各个组件执行不同的任务，所以标识生成错误的组件是有意义的。使用 -v 和 -dryrun 选项来帮助实现这一目的。

正如下表所示，不同编译器组件的输入文件拥有不同的文件名后缀。后缀建立了要进行的编译类型。关于文件后缀的含义请参见表 2-1。

表 2-2 C++ 编译系统的组件

组件	描述	使用说明
ccfe	前端 （编译器预处理程序和编译器）	
iropt	SPARC：代码优化器	-xO[2-5], -fast
ir2hf	x86：中间语言转换器	-xO[2-5], -fast
inline	SPARC：汇编语言模板的内联扩展	指定的 .il 文件
ube_ipa	x86：程间的分析器	-xcrossfile=1 和 -xO4、-xO5 或 -fast
fbe	汇编程序	
cg	SPARC：代码生成器、内联函数、汇编程序	
ube	x86：代码生成器	-xO[2-5], -fast
CClink	模板预链接程序	
ld	非递增式链接编辑器	
ild	递增式链接编辑器	-g, -xildon

2.5 预处理指令和名称

本节讨论了关于预处理 C++ 编译器所特有的指令信息。

2.5.1 Pragma

预处理程序关键字 `pragma` 是 C++ 标准的一部分，但 pragma 的形式、内容和含义对每个编译器是不同的。关于 C++ 编译器识别的 pragmas 列表，请参见附录 B。

2.5.2 具有可变数量参数的宏

C++ 编译器接受以下形式的 `#define` 预处理程序指令。

```
#define identifier (...) replacement_list
#define identifier (identifier_list, ...) replacement_list
```

如果列出的宏参数以省略号结尾，那么该宏的调用允许使用除了宏参数以外的其他更多参数。附加参数被收集在一个单独的字符串中，该字符串可以包括逗号。可以使用宏替换列表中的名称 `__VA_ARGS__` 来引用这些附加参数。以下示例说明了如何使用可变参数列表的宏。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...)((test)?puts(#test):\
                          printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n",x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

其结果如下：

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):printf("x is %d but y is %d", x, y));
```

2.5.3 预定义的名称

附录中的表 A-3 显示了预定义的宏。您可以在 `#ifdef` 这样的预处理程序条件中使用这些值。 `+p` 选项防止了 `sun`、 `unix`、 `sparc` 和 `i386` 预定义宏的自动定义。

2.5.4 `#error`

在发出警告以后，`#error` 指令不会继续编译。指令原来的行为是发出警告并继续编译。其新行为 （和其他编译器保持一致）是发出错误消息并立即停止编译。编译器退出并报告失败。

2.6 内存要求

编译需要的内存量取决于多个参数，包括：

- 每个过程的大小
- 优化级别
- 为虚拟内存设置的限制
- 磁盘交换文件的大小

在 SPARC 平台上，如果优化器用完了所有内存，那么它将通过在较低优化级别上重试当前过程来尝试恢复。然后优化器将在命令行用 -x*O**level* 选项指定的原优化级别上，继续随后的例程。

如果编译包括大量例程的单独源文件，编译器可能会用完所有内存或交换空间。如果编译器用完了内存，可以尝试降低优化级别。或者，可以将多例程的源文件分割为单例程的文件。

2.6.1 交换空间大小

`swap -s` 命令显示了可用的交换空间。更多信息请参见 `swap`(1M) 手册页。

以下示例显示了 `swap` 命令的使用：

```
example% swap -s
total:40236k bytes allocated + 7280k reserved = 47516k used,
1058708k available
```

2.6.2 增加交换空间

使用 mkfile(1M) 和 swap (1M) 来增加工作站上交换空间的大小。（您必须成为超级用户才能执行该操作。）mkfile 命令创建指定大小的文件，而 swap -a 将文件增加到系统交换空间：

```
example# mkfile -v 90m /home/swapfile
/home/swapfile 94317840 bytes
example# /usr/sbin/swap -a  /home/swapfile
```

2.6.3 虚拟内存的控制

在 -xO3 或更高级别上编译大型例程 （单个过程中包含了几千行代码）会需要大量的内存。在这种情况下，系统性能可能降低。您可以通过限制单个进程的可用虚拟内存量来控制这种情况。

要限制 sh shell 的虚拟内存，请使用 ulimit 命令。更多信息请参见 sh(1) 手册页。

以下示例显示了如何将虚拟内存限制为 16M。

```
example$ ulimit -d 16000
```

在 csh shell 中，使用 limit 命令来限制虚拟内存。更多信息请参见 csh(1) 手册页。

下一个示例也显示了如何将虚拟内存限制为 16M。

```
example% limit datasize 16M
```

这些示例都使优化器在数据空间达到 16M 时尝试恢复。

虚拟空间的限制不能大于系统总的可用交换空间。在实际使用时，虚拟空间的限制要足够的小，以允许在大型编译过程中正常使用系统。

请确保编译不会消耗一半以上的交换空间。

对于 32M 的交换空间，请使用以下命令：

在 sh shell 中：

```
example$ ulimit -d 16000
```

在 csh shell 中：

```
example% limit datasize 16M
```

最佳设置取决于要求的优化程度、实际内存量和可用的虚拟内存量。

2.6.4 内存要求

工作站至少需要 64M 的内存，推荐使用 128M。

要决定实际内存，请使用以下命令：

```
example% /usr/sbin/dmesg | grep mem
mem = 655360K (0x28000000)
avail mem = 602476544
```

2.7 简化命令

您可以通过使用 CCFLAGS 环境变量或通过使用 make 来定义特殊的 shell 别名，简化复杂的编译器命令。

2.7.1 在 C Shell 中使用别名

以下示例为带有常用选项的命令定义了别名。

```
example% alias CCfx "CC -fast -xnolibmil"
```

下面的示例使用了别名 CCfx。

```
example% CCfx any.C
```

命令 CCfx 现在等价于：

```
example% CC -fast -xnolibmil any.C
```

2.7.2 使用 CCFLAGS 来指定编译选项

您可以设置 CCFLAGS 变量来指定选项。

CCFLAGS 变量可以在命令行中显式使用。下列示例说明了如何设置 CCFLAGS (C Shell)：

```
example% setenv CCFLAGS '-xO2 -xsb'
```

下面的示例显式使用 CCFLAGS。

```
example% CC $CCFLAGS any.cc
```

当您使用 make 时，如果 CCFLAGS 变量像上述示例那样设置，并且 makefile 的编译规则是隐式的，那么调用 make 会导致编译等价于：

CC -xO2 -xsb *files*...

2.7.3 使用 make

make 实用程序是功能十分强大的程序开发工具，可以方便地和所有 Sun 编译器一起使用。更多信息请参见 make(1S) 手册页。

2.7.3.1 和 make 一起使用 CCFLAGS

使用 makefile 的**隐式**编译规则时（即没有 C++ 编译行），make 程序自动使用 CCFLAGS。

2.7.3.2 为 Makefile 增加后缀

您可以将不同的文件后缀增加到 makefile 以使它们收入 C++ 中。以下示例将 .cpp 增加为 C++ 文件的有效后缀。将 SUFFIXES 宏增加到 makefile：

```
SUFFIXES:.cpp .cpp~
```

（此行可以放置在 makefile 的任何位置。）

将以下各行增加到 makefile。缩进的行必须以制表符开头。

```
.cpp:
    $(LINK.cc) -o $@ $< $(LDLIBS)
.cpp~:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(LINK.cc) -o $@ $*.cpp $(LDLIBS)
.cpp.o:
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp~.o:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) $(OUTPUT_OPTION) $<
.cpp.a:
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
    $(RM) $%
.cpp~.a:
    $(GET) $(GFLAGS) -p $< > $*.cpp
    $(COMPILE.cc) -o $% $<
    $(COMPILE.cc) -xar $@ $%
```

2.7.3.3 和标准库头文件一起使用 make

标准库文件的名称不包括 .h 后缀。相反，它们命名为 istream、fstream 等等。此外，模板源文件命名为 istream.cc、 fstream.cc 等等。

第 3 章

使用 C++ 参阅编译器选项

本章说明了如何使用命令行 C++ 编译器选项，并按功能汇总它们的使用。关于选项的详细解释，请参见附录 A。

3.1 语法

下表显示了本书中使用的典型选项语法格式的示例。

表 3-1 选项语法格式示例

语法格式	示例
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

圆括号、大括号、括号、管道字符和省略号是选项说明中使用的**元字符**，而不是选项自身的一部分。关于使用语法的详细解释，请参见本手册前面的“阅读本书之前”中的印刷约定。

3.2 通用指南

C++ 编译器选项的某些通用指南：

- `-l`*lib* 选项和 `lib`*lib*`.a` 库 （或 `lib`*lib*`.so`）一起链接。将 `-l`*lib* 放置在源文件和对象文件后面可以确保搜索库时的顺序更加安全。

- 一般，编译器选项的处理是由左到右 （除了 -U 选项要在所有 -D 选项处理以后），允许有选择性地覆盖宏选项 （即包括其他选项的选项）。此规则不适用于链接程序选项。
- features、-I、-l、-L、-library、-pti、-R、-staticlib、-U、-verbose 和 -xprefetch 选项将会累积而不会覆盖。
- D 选项会积累，不过同一个名称的多个 -D 选项会互相覆盖。

源文件、对象文件和库按它们在命令行上出现的顺序编译和链接。

3.3 按功能汇总的选项

在本节中，编译器选项按功能分组以便提供快速参考。关于每个选项的详细描述，请参见附录 A。

选项适用于除了说明以外的所有平台；基于 SPARC 系统的 Solaris 操作系统专有的特性标识为 *SPARC*，而基于 x86 系统的 Solaris 操作系统专有的特性标识为 *x86*。

3.3.1 代码生成选项

以下代码生成选项按字母顺序列出。

表 3-2 代码生成选项

选项	操作
-compat	设置编译器的主发行版本兼容模式。
+e{0|1}	控制虚拟表的生成。
-g	用于与调试一起使用的编译。
-KPIC	生成位置独立的代码。
-KPIC	生成位置独立的代码。
-mt	编译和链接多线程代码。
-xcode=*a*	*(SPARC)* 指定代码地址空间。
-xMerge	*(SPARC)* 将数据段和文本段合并。
+w	标识会产生不可预料结果的代码。

表 3-2　代码生成选项 (续)

选项	操作
+w2	发出由 +w 提供的所有警告，以及关于技术违规的附加警告。这类技术违规可能是无害的，但可能会降低系统的最大可移植性。
-xregs	如果编译器可以使用更多的寄存器用于临时存储 （临时寄存器），那么编译器将能生成速度更快的代码。该选项使得附加临时寄存器可用，而这些附加寄存器通常是不适用的。
-z *arg*	链接程序选项。

3.3.2 编译时性能选项

以下编译时性能选项按字母顺序列出

表 3-3　编译时性能选项

选项	操作
-instlib	禁止生成已出现在指定库中的模板实例。
-xjobs	设置编译器可以为完成工作而创建的进程数量。
-xpch	可以减少应用程序的编译时间，该应用程序的源文件共享共同的包括文件集合。
-xpchstop	指定最后一个包含文件，用于考虑用 -xpch 创建预编译头文件。
-xprofile_ircache	*(SPARC)* 重用 -xprofile=collect 期间保存的编译数据。
-xprofile_pathmap	*(SPARC)* 支持单个性能分析目录中的多个程序或共享库。

3.3.3 调试选项

以下调试选项按字母顺序列出。

表 3-4　调试选项

选项	操作
+d	不扩展 C++ 内联函数。
-dryrun	显示但不编译由驱动程序传递到编译器的选项。
-E	仅在 C++ 源文件上运行预处理程序，并将结果发送至 stdout。
-g	用于与调试一起使用的编译。
-g0	为了调试而编译，但不禁止内联。

表 3-4　调试选项 (续)

选项	操作
-H	打印包含文件的路径名称。
-keeptmp	保留编译时创建的临时文件。
-migration	解释可以从早期编译器获得有关移植信息的位置。
-P	仅预处理源文件，输出到 .i 文件。
-Qoption	直接将选项传递给编译阶段。
-readme	显示联机 README 文件的内容。
-s	从可执行文件中去掉符号表，这样可以保护调试代码的能力。
-temp=*dir*	为临时文件定义目录。
-verbose=*vlst*	控制编译器冗长。
-xcheck	增加堆栈溢出的运行时检查。
-xdumpmacros	打印诸如定义、定义及未定义的位置和已使用的位置的宏信息。
-xe	仅检查语法和语义错误。
-xhelp=flags	显示编译器选项汇总列表。
-xildoff	关闭增量式链接程序。
-xildon	打开增量式链接程序。
-xport64	对 32 位体系结构到 64 位体系结构的移植过程中的常见问题发出警告。
-xs	允许在用 dbx 调试时不包括对象 (.o) 文件。
-xsb	为源码浏览器生成表信息。
-xsbfast	仅生成源码浏览器信息，而不进行编译。

3.3.4　浮点选项

以下浮点选项按字母顺序列出。

表 3-5　浮点选项

选项	操作
-fns[={no|yes}]	*(SPARC)* 禁用或启用 SPARC 的非标准浮点模式。
-fprecision=*p*	*x86*：设置浮点精度模式。
-fround=*r*	设置启动时生效的 IEEE 舍入模式。
-fsimple=*n*	设置浮点优化首选项。
-fstore	*x86*：强制浮点表达式的精度。

表 3-5　浮点选项 (续)

选项	操作
-ftrap=*tlst*	设置启动时生效的 IEEE 捕获模式。
-nofstore	*x86*：禁用表达式的强制精度。
-xlibmieee	在异常情况下，使 libm 返回数学例程的 IEEE 754 值。

3.3.5 语言选项

以下语言选项按字母顺序列出。

表 3-6　语言选项

选项	操作
-compat	设置编译器的主发行版本兼容模式。
-features=*alst*	启用或禁用各种 C++ 语言特性。
-xchar	在 char 类型定义为无符号的系统上，简化代码的移植。
-xldscope	控制变量和函数定义的默认链接程序范围，以创建更快更安全的共享库。
-xthreadvar	*(SPARC)* 更改默认的线程局部存储访问模式。
-xtrigraphs	启用三字母序列的识别。
-xustr	启用识别由 16 位字符构成的字符串文字。

3.3.6 库选项

以下库链接选项按字母顺序列出。

表 3-7　库选项

选项	操作
-B*binding*	请求符号、动态或静态库链接。
-d{y|n}	允许或不允许整个可执行文件的动态库。
-G	生成动态共享库来取代可执行文件。
-D*name*	为生成的动态共享库指定名称。
-i	告知 ld(1) 忽略任何 LD_LIBRARY_PATH 设置。
-L*dir*	将 *dir* 增加到搜索库的目录列表。

表 3-7 库选项 (续)

选项	操作
-l*lib*	将 lib*lib*.a 或 lib*lib*.so 增加到链接程序的库搜索列表。
-library=*llst*	强制将特定库和相关文件包含到编译和链接中。
-mt	编译和链接多线程代码。
-norunpath	不将库的路径生成到可执行文件中。
-R*plst*	将动态库搜索路径生成到可执行文件中。
-staticlib=*llst*	说明哪些 C++ 库是静态链接的。
-xar	创建归档库。
-xbuiltin[=*opt*]	启用或禁用标准库调用的更多优化
-xia	*(SPARC)* 链接合适的区间运算库并设置适当的浮点环境。
-xlang=*l*[,*l*]	包含适当的运行库，并确保指定语言的正确运行时环境。
-xlibmieee	在异常情况下，使 libm 返回数学例程的 IEEE 754 值。
-xlibmil	内联选定的 libm 库例程用于进行优化。
-xlibmopt	使用优化的数学例程库。
-xlic_lib=sunperf	*(SPARC)* 在 Sun Performance Library™ 中的链接。请注意对于 C++，-library=sunperf 是链接该库的最佳方法。
-xnativeconnect	包含对象文件中的接口信息和随后的共享库，使得这些共享库可以与使用 Java™ 编程语言编写的代码接口。
-xnolib	禁用与默认系统库进行链接。
-xnolibmil	在命令行上取消 -xlibmil。
-xnolibmopt	不使用数学例程库。

3.3.7 许可证选项

以下许可证选项按字母顺序列出。

表 3-8 许可证选项

选项	操作
-xlic_lib=sunperf	*(SPARC)* 在 Sun Performance Library™ 中的链接。请注意对于 C++，-library=sunperf 是链接该库的最佳方法。
-xlicinfo	显示许可证服务器信息。

3.3.8 废弃的选项

以下选项已废弃或将要废弃。

表 3-9 废弃的选项

选项	操作
`-library=%all`	废弃的选项，在以后的发行版本中将被删除。
`-noqueue`	禁用许可证队列。
`-ptr`	编译器忽略。以后的编译器发行版本可以使用其他行为来重用该选项。
`-vdelx`	废弃的选项，在以后的发行版本中将被删除。
`-xprefetch=yes`	而是使用 `-xprefetch=auto,explicit`。
`-xprefetch=no`	而是使用 `-xprefetch=no%auto,no%explicit`。

3.3.9 输出选项

以下输出选项按字母顺序列出。

表 3-10 输出选项

选项	操作
`-c`	仅编译；生成对象 (`.o`) 文件，但禁止链接。
`-dryrun`	显示但不编译由驱动程序传递到编译器的选项。
`-E`	仅在 C++ 源文件上运行预处理程序，并将结果发送至 `stdout`。
`-erroff`	禁止编译器警告消息。
`-errtags`	显示每条警告消息的消息标记。
`-errwarn`	如果指示的警告消息已发出，则 cc 以失败状态退出。
`-filt`	禁止编译器应用到链接程序错误消息的过滤。
`-G`	生成动态共享库来取代可执行文件。
`-H`	打印包含文件的路径名称。
`-migration`	解释可以从早期编译器获得有关移植信息的位置。
`-o` *filename*	将输出文件或可执行文件的名称设置为 *filename*。
`-P`	仅预处理源文件，输出到 `.i` 文件。
`-Qproduce` *sourcetype*	使 `CC` 驱动程序生成类型 *sourcetype* 的输出。
`-s`	从可执行文件去掉符号表。

表 3-10 输出选项 (续)

选项	操作
-verbose=*vlst*	控制编译器冗长。
+w	必要时打印附加警告。
-w	禁止警告消息。
-xdumpmacros	打印诸如定义、定义及未定义的位置和已使用的位置的宏信息。
-xe	仅对源文件进行语法和语义检查，但不生成任何对象或可执行代码。
-xhelp=flags	显示编译器选项汇总列表
-xhelp=readme	显示联机 README 文件的内容。
-xM	输出 makefile 依存信息。
-xM1	生成依存信息，但排除 /usr/include。
-xsb	为源码浏览器生成表信息。
-xsbfast	仅生成源码浏览器信息，而不进行编译。
-xtime	报告每个编译阶段的执行时间。
-xwe	通过返回非零的退出状态，将所有警告转换成错误。
-z *arg*	链接程序选项。

3.3.10 运行时性能选项

以下运行时性能选项按字母顺序列出。

表 3-11 运行时性能选项

选项	操作
-fast	选择编译选项的组合以优化某些程序的执行速度。
-g	指示编译器和链接程序准备程序以进行性能分析 （以及调试）。
-s	从可执行文件去掉符号表。
-xalias_level	启用编译器执行基于类型的别名分析和优化。
-xarch=*isa*	指定目标架构指令集。
-xbuiltin[=*opt*]	启用或禁用标准库调用的更多优化
-xcache=*c*	*(SPARC)* 定义优化器的目标缓存属性。
-xcg89	为通用 SPARC 架构编译。

表 3-11　运行时性能选项 (续)

选项	操作
-xcg92	为 SPARC V8 架构编译。
-xchip=*c*	指定目标处理器芯片。
-xF	启用函数和变量的链接程序重新排序。
-xinline=*flst*	指定用户编写的哪些例程可以被优化器内联
-xipo	执行过程间优化。
-xlibmil	内联选定的 libm 库例程用于进行优化。
-xlibmopt	使用优化数学例程的库。
-xlinkopt	*(SPARC)* 在对象文件中的任何优化上生成的可执行文件或动态库上，执行链接时优化。
-xmemalign=*ab*	*(SPARC)* 指定最大的假定内存对齐和未对齐的数据访问行为。
-xnolibmil	在命令行上取消 -xlibmil。
-xnolibmopt	不使用数学例程库。
-xO*level*	将优化等级指定为 *level*。
-xpagesize	*(SPARC)* 设置栈和堆所需的页面大小。
-xpagesize_heap	*(SPARC)* 设置堆所需的页面大小。
-xpagesize_stack	*(SPARC)* 设置栈所需的页面大小。
-xprefetch[=*lst*]	*(SPARC)* 在支持预取的架构上启用预取指令。
-xprefetch_level	控制由于设置 -xprefetch=auto 而造成自动插入预取指令的攻击性。
-xprofile	*(SPARC)* 使用运行时文件配置数据来收集或优化。
-xregs=*rlst*	*(SPARC)* 控制临时寄存器的使用。
-xsafe=mem	*(SPARC)* 允许不基于内存的自陷。
-xspace	*(SPARC)* 不允许增加代码大小的优化。
-xtarget=*t*	指定目标指令集和优化系统。
-xthreadvar	*(SPARC)* 更改默认的线程局部存储访问模式。
-xunroll=*n*	启用在可能的场合下解开循环。
-xvis	*(SPARC)* 启用在 VIS™ 指令集中定义的汇编语言模板的编译器识别

3.3.11 预处理程序选项

以下预处理程序选项按字母顺序列出。

表 3-12 预处理程序选项

选项	操作
-D*name*[=*def*]	将符号 *name* 定义为预处理程序。
-E	仅在 C++ 源文件上运行预处理程序，并将结果发送至 stdout。
-H	打印包含文件的路径名称。
-P	仅预处理源文件，输出到 .i 文件。
-U*name*	删除预处理程序符号 *name* 的最初定义。
-xM	输出 makefile 依存信息。
-xM1	生成依存信息，但排除 /usr/include。

3.3.12 文件配置选项

以下文件配置选项按字母顺序列出。

表 3-13 文件配置选项

选项	操作
-p	准备对象代码以使用 prof 收集文件配置的数据。
-xa	为文件配置生成代码。
-xpg	使用 gprof 配置程序为文件配置编译。
-xprofile	*(SPARC)* 使用运行时文件配置数据来收集或优化。

3.3.13 参考选项

以下选项提供了编译器信息的快速参考。

表 3-14 参考选项

选项	操作
`-migration`	解释可以从早期编译器获得有关移植信息的位置。
`-xhelp=flags`	显示编译器选项汇总列表。
`-xhelp=readme`	显示联机 `README` 文件的内容。

3.3.14 源文件选项

以下源文件选项按字母顺序列出。

表 3-15 源文件选项

选项	操作
`-H`	打印包含文件的路径名称。
`-I`*pathname*	将 *pathname* 增加到 `include` 文件搜索路径。
`-I-`	更改包含文件搜索规则
`-xM`	输出 makefile 依存信息。
`-xM1`	生成依存信息，但排除 `/usr/include`。

3.3.15 模板选项

以下模板选项按字母顺序列出。

表 3-16 模板选项

选项	操作
`-instances=`*a*	控制模板实例的放置和链接。
`-pti`*path*	为模板源文件指定附加搜索目录。
`-template=`*wlst*	启用或禁用各种模板选项。

3.3.16 线程选项

以下线程选项按字母顺序列出。

表 3-17 线程选项

选项	操作
`-mt`	编译和链接多线程代码。
`-xsafe=mem`	*(SPARC)* 允许不基于内存的自陷。
`-xthreadvar`	*(SPARC)* 更改默认的线程局部存储访问模式。

第 II 部分 编写 C++ 程序

第4章

语言扩展参阅

本章记录了特定于本编译器的语言扩展。附录 B 也提供了实现的特定信息。在命令行上指定某些编译器选项之后，编译器才能识别本章中描述的某些特性。相关编译器选项在相应章节中列出。

`-features=extensions` 选项使您能编译非标准代码，该代码通常可以被其他 C++ 编译器接受。必须编译无效代码且不允许修改代码而使之有效时，您可以使用该选项。

本章描述了使用 `-features=extensions` 选项时编译器支持的语言扩展。

注 – 可以很容易的将每个支持无效代码的实例转变为所有编译器接受的有效代码。如果允许使代码有效，那么您应该使代码有效而不是使用该选项。使用 `-features=extensions` 选项可以使某些编译器拒绝的无效代码永远存在。

4.1 链接程序作用域

使用以下声明说明符帮助约束外部符号的声明和定义。文件链接到共享库或可执行文件之前，静态归档或对象文件指定的作用域限制不会生效。尽管如此，编译器仍然可以执行显示链接程序作用域说明符的某些优化。

通过使用这些说明符，您不必再使用链接程序作用域的 mapfile。此外也可以通过在命令行指定 `-xldscope` 来控制变量作用域的默认设置。

有关更多信息请参见第 A-99 页的第 A.2.128 节 "-xldscope={v}"。

表 4-1 链接程序作用域声明说明符

值	含义
__global	符号定义具有全局链接程序作用域，并且是限制最少的链接程序作用域。对符号的所有引用都绑定到定义符号的第一个动态装入模块中的定义。该链接程序作用域是外部符号的当前链接程序作用域。
__symbolic	符号定义具有符号链接程序作用域，并且具有比全局链接程序作用域更多的限制。将对链接的动态装入模块内符号的所有引用绑定到模块内定义的符号。在模块外部，符号也显示为全局符号。该链接程序作用域对应于链接程序选项 -Bsymbolic。尽管不能与 C++ 库一起使用 -Bsymbolic，但是可以使用 __symbolic 说明符而不会引起问题。关于链接程序的更多信息，请参见 ld(1)。
__hidden	符号定义具有隐藏的链接程序作用域。隐藏链接程序作用域具有比符号和全局链接程序作用域更高的限制。将动态装入模块内的所有引用绑定到该模块内的定义。符号在模块外部是不可视的。

符号定义可以用更多限制的说明符来重新声明，但是不可以用较少限制的说明符重新声明。符号定义后，不可以用不同的说明符声明符号。

__global 是限制最少的作用域，__symbolic 是限制较多的，而 __hidden 是限制最多的作用域。

因为虚函数的声明影响虚拟表的结构和解释，所以所有虚函数对包括类定义的所有编译单元必须是可视的。

因为 C++ 类可能需要生成诸如虚拟表和运行时类型信息等的隐式信息，所以您可以将链接程序说明符应用到结构、类和联合的声明及定义。在这种情况下，说明符后跟结构、类或联合关键字。这种应用程序为其所有隐式成员隐含了相同的链接程序作用域。

4.2 线程局部存储

通过声明线程局部变量可以利用线程局部存储的优点。线程局部变量声明由具有附加声明说明符 __thread 的普通变量声明组成。有关更多信息请参见第 A-143 页的第 A.2.168 节 "-xthreadvar[=o]"。

必须将 __thread 说明符包括在线程变量的第一个声明中。用 __thread 说明符声明的变量是没有 __thread 说明符时的边界。

只可以用 __thread 说明符声明静态持续时间的变量。具有静态持续时间的变量包括了文件全局、文件静态、函数局部静态和类静态成员。不能用 __thread 说明符声明具有动态或自动持续时间的变量。线程变量可以具有静态初始化函数，但是不可以具有动态

初始化函数或析构函数。例如， `__thread int x = 4;` 是允许的，但是 `__thread int x = f();` 是不允许的。线程变量不能包含具有重要构造函数和析构函数的类型。具体来讲，线程变量不可以具有类型 `std::string`。

线程变量的地址运算符 (&) 在运行时求值并返回当前线程变量的地址。因此，线程变量的地址不是常量。

线程变量的地址在相应线程的生命周期中是稳定的。进程中的任何线程可以在变量的生命周期中自由使用线程变量的地址。不能在线程终止后使用线程变量地址。线程变量的所有地址在线程终止后都是无效的。

4.3 用限制较少的虚函数覆盖

C++ 标准声称覆盖的虚函数在允许的异常中包含的限制并不比覆盖的任何函数少。该虚函数可能与覆盖的任何函数具有相同或更多的限制。请注意，不存在异常规范也允许任何异常。

例如，假定通过指向基类的指针调用函数。如果函数具有异常规范，则可以计算出没有其他正在抛出的异常。如果覆盖函数具有限制较少的规范，则意外的异常可能会被抛出，这会导致奇怪的程序行为并且终止程序。这就是规则的原因。

使用 `-features=extensions` 时，编译器允许使用限制较少的异常规范来覆盖函数。

4.4 生成 enum 类型和变量的向前声明

使用 `-features=extensions` 时，编译器允许 `enum` 类型和变量的向前声明。此外，编译器允许声明不完整 enum 类型的变量。编译器总是假定不完整 `enum` 类型与当前平台上的 `int` 类型具有相同的大小和范围。

以下两行代码是无效代码的示例，使用 `-features=extensions` 选项时将编译该无效代码。

```
enum E; // 无效：不允许 enum 的向前声明
E e;    // 无效：类型 E 不完整
```

因为 `enum` 定义不能互相引用，并且 `enum` 定义不能交叉引用另一种类型，所以 enum 类型的向前声明从来都是不需要的。要使代码有效，可以在使用 `enum` 之前始终提供它的完全定义。

注 – 在 64 位架构上，enum 有可能需要比 int 类型更大的大小。如果是这种情况，并且如果向前声明和定义在同一编译中是可视的，那么编译器将发出错误。如果实际大小不是假定的大小并且编译器没有发现这个差异，那么代码将编译并链接，但有可能不能正常运行。可能出现奇怪的程序行为，尤其是 8 字节值存储在 4 字节变量中时。

4.5 使用不完整 enum 类型

使用 -features=extensions 时，不完整的 enum 类型用作向前声明。例如，使用 -features=extensions 选项时，以下无效代码将编译。

```
typedef enum E F; // 无效，E 不完整
```

如前所述，在使用之前可以总是包括 enum 类型的定义。

4.6 将 enum 名称作为作用域限定符

因为 enum 声明不引入作用域，所以 enum 名称不能用为作用域限定符。例如，以下代码是无效的。

```
enum E {e1, e2, e3};
int i = E::e1; // 无效: E 不是作用域名称
```

要编译该无效代码，请使用 -features=extensions 选项。 -features=extensions 选项说明如果编译器的作用域限定符是 enum 类型的名称，那么忽略该限定符。

要使代码有效，请删除无效的限定符 E::。

注 – 使用该选项提高了排字错误的可能性，产生了编译没有错误消息的错误程序。

4.7 使用匿名 struct 声明

匿名结构声明既不是声明结构标记的声明，也不是声明对象或 typedef 名称的声明。C++ 中不允许匿名结构。

-features=extensions 选项允许使用匿名 struct 声明，但仅作为联合的成员。

以下代码是无效匿名 struct 声明的示例，该声明在使用 -features=extensions 选项时编译。

```
union U {
  struct {
    int a;
    double b;
  };  // 无效：匿名结构
  struct {
    char* c;
    unsigned d;
  };  // 无效：匿名结构
};
```

struct 成员的名称是可视的，没有 struct 成员名称的限定。在该编码示例中给出了 U 的定义，您可以这样编写：

```
U u;
u.a = 1;
```

匿名结构与匿名联合服从相同的限制。

注意，通过赋予每个 struct 名称而使代码有效，例如：

```
union U {
  struct {
    int a;
    double b;
  } A;
  struct {
    char* c;
    unsigned d;
  } B;
};
U u;
U.A.a = 1;
```

4.8 传递匿名类实例的地址

不允许获取临时变量的地址。例如，因为以下代码获取了构造函数调用创建的变量地址，所以这些代码是无效的。不过，使用 -features=extensions 选项时，编译器接受该无效代码。

```
class C {
  public:
    C(int);
    ...
};
void f1(C*);
int main()
{
  f1( &C(2) ); // 无效
}
```

注意，可以通过使用显式变量来使该代码有效。

```
C c(2);
f1(&c);
```

函数返回时，临时对象被销毁。编程人员的责任就是确保不保留临时变量的地址。此外，临时变量销毁时存储在临时变量 （例如 f1）中的数据会丢失。

4.9 将静态名称空间作用域函数声明为类友元

下面的代码是无效的：

```
class A {
  friend static void foo(<args>);
  ...
};
```

因为类名具有外部链接并且所有定义必须是相等的，所以友元函数也必须具有外部链接。不过，使用 -features=extensions 选项时，编译器接受了该代码。

推测起来，编程人员准备为该无效代码在类 A 的实现文件中提供非成员“帮助程序”函数。通过生成 foo 静态成员函数可以得到相同的效果。如果不要客户端调用函数，则可以使该函数私有化。

注 – 如果使用该扩展，则类可以被任何客户端“截获”。任何客户端都可以包括类的头文件，然后定义客户端自身的静态函数 foo，该函数自动成为类的友元。结果就好像是您使类的所有成员成为了公共的。

4.10 使用函数名称的预定义 __func__ 符号

使用 -features=extensions 时，编译器将每个函数中的标识符 __func__ 隐式声明为 const char 的静态数组。如果程序使用标识符，那么编译器也在 *function-name* 是函数原始名称的位置提供以下定义。类成员关系、名称空间和重载不反映在名称中。

```
static const char __func__[] = "function-name";
```

例如，请考虑以下代码段。

```
#include <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);
}
```

每次调用函数时，函数将把以下内容打印到标准输出流。

```
myfunc
```

第5章

程序组织

C++ 程序的文件组织需要比典参阅型的 C 程序更加小心。本章说明了如何建立头文件和模板定义。

5.1 头文件

创建有效的头文件是很困难的。头文件通常必须适应 C 和 C++ 的不同版本。要容纳模板，请确保头文件可以采取多个包含 （幂等）。

5.1.1 可适应语言的头文件

可能需要开发能够包含在 C 和 C++ 程序中的头文件。不过，称为“传统 C”的 Kernighan 和 Ritchie C (K&R C)、ANSI C、*Annotated Reference Manual* C++ (ARM C++) 以及 ISO C++ 有时需要在单个头文件中对同一个程序元素给出不同的声明和定义。（关于各个语言和版本的附加信息，请参见 《C++ 迁移指南》。）要使头文件能够被所有这些标准接受，可能需要使用基于预处理程序宏 `__STDC__` 和 `__cplusplus` 或其值的条件编译。

宏 `__STDC__` 在 K&R C 中没有定义，但在 ANSI C 和 C++ 中都有定义。使用这个宏可以从 ANSI C 或 C++ 代码中区分出 K&R C 代码。该宏最适用于从非原型函数定义中区分原型函数定义。

```
#ifdef __STDC__
int function(char*,...);       // C++ & ANSI C 声明
#else
int function();                // K&R C
#endif
```

宏 `__cplusplus` 不在 C 中定义，但在 C++ 中定义。

注 – C++ 的早期版本定义的是宏 `c_plusplus`，而非 `__cplusplus`。宏 `c_plusplus` 现在不再定义。

使用 `__cplusplus` 宏的定义来区分 C 和 C++。这个宏在保护函数声明的 `extern "C"` 接口规范时非常有用，如以下示例所示。为了防止不一致的 `extern "C"` 规范，请勿在 `extern "C"` 链接规范的作用域中放置 `#include` 指令。

```
#include "header.h"
...                          // ... 其他包括文件 ...
#if defined(__cplusplus)
extern "C" {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

在 ARM C++ 中，`__cplusplus` 宏值为 1。在 ISO C++ 中，宏的值为 199711L（标准年月用 `long` 常量来表示）。使用这个宏的值区分 ARM C++ 和 ISO C++。这个宏值在保护模板语法的更改时极为有用。

```
// 模板函数规范
#if __cplusplus < 199711L
int power(int,int);                          // ARM C++
#else
template <> int power(int,int);              // ISO C++
#endif
```

5.1.2 幂等头文件

头文件应当是幂等的。也就是说，多次包括头文件的效果和仅包括一次的效果完全相同。该特性对于模板尤其重要。通过设置预处理程序条件以防止头文件体多次出现，可以很好的实现幂等。

```
#ifndef HEADER_H
#define HEADER_H
/* 头文件内容 */
#endif
```

5.2 模板定义

可以用两种方法组织模板定义：使用包括的定义和使用独立的定义。包括的定义组织允许对模板编译进行更多的控制。

5.2.1 包括的模板定义

将模板的声明和定义放置在使用模板的文件中时，这就是**包括定义的结构**。例如：

main.cc	template <class Number> Number twice(Number original); template <class Number> Number twice(Number original) { return original + original; } int main() { return twice<int>(-3); }

使用模板的文件包括了包含模板声明和定义的文件时，使用模板的该文件也具有包括的定义组织。例如：

```
twice.h     #ifndef TWICE_H
            #define TWICE_H
            template <class Number>
            Number twice(Number original);
            template <class Number> Number
            twice( Number original )
                { return original + original; }
            #endif

main.cc     #include "twice.h"
            int main()
                { return twice(-3); }
```

注 – 使模板头文件幂等是非常重要的。（请参见第 5-2 页的第 5.1.2 节“幂等头文件”。）

5.2.2 独立的模板定义

组织模板定义的另一种方法是将定义保留在模板定义文件中，如以下示例所示。

```
twice.h     #ifndef TWICE_H
            #define TWICE_H
            template <class Number>
            Number twice(Number original);
            #endif TWICE_H

twice.cc    template <class Number>
            Number twice( Number original )
                { return original + original; }

main.cc     #include "twice.h"
            int main( )
                { return twice<int>( -3 ); }
```

模板定义文件**不得**包括任何非幂等的头文件，而且通常根本不包括任何头文件。（请参见第 5-2 页的第 5.1.2 节 “幂等头文件”。）注意，并非所有编译器都支持模板的独立定义模型。

一个单独的定义文件作为头文件时，该文件可能会被隐式包括在许多文件中。因此，它不应该包含任何函数或变量定义 （除非这些定义是模板定义的一部分）。一个单独的定义文件可以包含类型定义，包括 typedef。

注 – 尽管模板定义文件的源文件通常会使用扩展名（即，`.c`、`.C`、`.cc`、`.cpp`、`.cxx` 或 `.c++`），但模板定义文件就是头文件。如果需要，编译器会自动包括这些它们。模板定义文件**不**可以单独编译。

如果将模板声明放置在一个文件中，而将模板定义放置在另一个文件中，则必须仔细考虑如何构造定义文件，如何命名定义文件和如何放置定义文件。此外也需要向编译器显式指定定义的位置。关于模板定义搜索规则的信息，请参见第 7-7 页的第 7.5 节 “模板定义搜索”。

第6章

创建和使用模板

有了模板，就可以采用类型安全方法参阅来编写适用于多种类型的单一代码。本章介绍了模板的概念和函数模板上下文中的术语，讨论了更复杂的（更强大的）类模板，描述了模板的组成，此外还讨论了模板实例化、默认模板参数和模板专门化。本章的结尾部分讨论了模板的潜在问题。

6.1 函数模板

函数模板描述了仅用参数或返回值的类型来区分的一组相关函数。

6.1.1 函数模板声明

使用模板之前，请先声明。以下示例中所示的**声明**提供了使用模板的足够信息，但没有提供实现模板的足够信息。

```
template <class Number> Number twice( Number original );
```

在本示例中，*Number* 是**模板参数**，指定了模板描述的函数范围。尤其特别的是 *Number* 是**模板类型参数**，该参数在模板定义中的使用代表了在使用模板的位置所决定的类型。

6.1.2 函数模板定义

如果要声明模板，请先定义该模板。**定义**提供了实现模板的足够信息。以下示例定义了在前一个示例中声明的模板。

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

因为模板定义通常出现在头文件中，所以模板定义必须在多个编译单元中重复。不过所有的定义都必须是相同的。这种约束称为**单次定义规则**。

编译器不支持函数参数列表中非类型模板参数的表达式，如以下示例所示。

```
// 具有非类型模板参数的表达式
// 在函数参数列表中是不支持的
template<int I> void foo( mytype<2*I> ) { ... }
template<int I, int J> void foo( int a[I+J] ) { ... }
```

6.1.3 函数模板用法

声明后，模板可以像其他函数一样使用。模板的**使用**由命名模板和提供函数参数组成。编译器可以从函数参数类型推断出模板类型参数。例如，您可以使用上面声明的模板，具体步骤如下所示。

```
double twicedouble( double item )
    { return twice( item ); }
```

如果模板参数不能从函数参数类型推断出，则调用函数时必须提供模板参数。例如：

```
template<class T> T func(); // 无函数参数
int k = func<int>(); // 显式提供模板参数
```

6.2 类模板

类模板描述了一组相关的类或数据类型，它们只能通过类型来区分：整数值、指向（或引用）具有全局链接的变量的指针、其他的组合。类模板尤其适用于描述通用但类型安全的数据结构。

6.2.1 类模板声明

类模板声明仅提供了类的名称和类的模板参数。这种声明是**不完整的类模板**。

以下示例是命名为 Array 类的模板声明，该类可以将任何类型作为参数。

```
template <class Elem> class Array;
```

该模板是命名为 String 类的模板，该类将 unsigned int 作为参数。

```
template <unsigned Size> class String;
```

6.2.2 类模板定义

类模板定义必须声明类数据和函数成员，如以下示例所示。

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize( );
        Elem& operator[]( int idx );
};
```

```
template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

与函数模板不同，类模板可以同时具有类型参数（例如 class Elem）和表达式参数（例如 unsigned Size）。表达式参数可以是：

- 具有整数类型或枚举的值
- 指向对象的指针或到对象的引用
- 指向函数的指针或到函数的引用
- 指向类成员函数的指针

6.2.3 类模板成员定义

类模板的完整定义需要类模板函数成员和静态数据成员的定义。动态（非静态）数据成员由类模板声明完全定义。

6.2.3.1 函数成员定义

模板函数成员的定义由模板参数专门化后跟函数定义组成。函数标识符通过类模板的类名称和模板参数来限定。以下示例说明了 `Array` 类模板的两个函数成员的定义，该模板具有 `template <class Elem>` 的模板参数专门化。每个函数标识符都通过模板类名称和模板参数 `Array<Elem>` 来限定。

```
template <class Elem> Array<Elem>::Array( int sz )
    {size = sz; data = new Elem[size];}

template <class Elem> int Array<Elem>::GetSize( )
    { return size; }
```

该示例说明了 `String` 类模板的函数成员定义。

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    {int len = 0;
      while (len < Size && data[len]!= '\0') len++;
      return len;}

template <unsigned Size> String<Size>::String( char *initial )
    {strncpy(data, initial, Size);
      if (length( ) == Size) overflows++;}
```

6.2.3.2 静态数据成员定义

模板静态数据成员的定义由后跟变量定义的模板参数专门化组成，在此处变量标识符通过类模板名称和类模板实际参数来限定。

```
template <unsigned Size> int String<Size>::overflows = 0;
```

6.2.4 类模板的用法

模板类可以在使用类型的任何地方使用。指定模板类包括了提供模板名称和参数的值。以下示例中的声明创建了基于 Array 模板的 int_array 变量。变量的类声明和方法集与 Array 模板中的类声明和方法集相同，不同之处是 Elem 替换为 int （请参见第 6-5 页的第 6.3 节 “模板实例化”）。

```
Array<int> int_array( 100 );
```

本示例中的声明使用 String 模板创建 short_string 变量。

```
String<8> short_string( "hello" );
```

需要任何其他成员函数时，您可以使用模板类成员函数。

```
int x = int_array.GetSize( );
```

```
int x = short_string.length( );
```

6.3 模板实例化

模板**实例化**包含了为模板参数的特定组合生成固定类或函数 （**实例**）。例如，编译器生成了 Array<int> 的类和 Array<double> 的不同类。通过替换模板类定义中模板参数的模板参数，可以定义这些新的类。在上一节 “类模板”所述的 Array<int> 示例中，编译器在 Elem 出现的位置替换 int。

6.3.1 隐式模板实例化

使用模板函数或模板类时需要实例。如果这种实例还不存在，则编译器隐式实例化模板参数组合的模板。

6.3.2 显式模板实例化

编译器仅为实际使用的那些模板参数组合而隐式实例化模板。该方法不适用于构造提供模板的库。 C++ 提供了显式实例化模板的功能，如以下示例所示。

6.3.2.1 模板函数的显式实例化

要显式实例化一个模板函数， `template` 关键字的后面应跟有该函数的声明 （而不是定义）；其中，函数识别符后面跟有模板参数。

```
template float twice<float>( float original );
```

在编译器可以推断出模板参数时，模板参数可以省略。

```
template int twice( int original );
```

6.3.2.2 模板类的显式实例化

要显式实例化模板类，请在 `template` 关键字后跟类的声明 （无定义）；而要实例化类标识符，请后跟模板参数。

```
template class Array<char>;
```

```
template class String<19>;
```

显式实例化类时，所有的类成员也必须实例化。

6.3.2.3 模板类函数成员的显式实例化

要显式实例化模板类函数成员，请在 `template` 关键字后跟函数的声明 （无定义）；而要实例化由模板类限定的函数标识符，请后跟模板参数。

```
template int Array<char>::GetSize( );
```

```
template int String<19>::length( );
```

6.3.2.4 模板类静态数据成员的显式实例

要显式实例化模板类静态数据成员，请在 template 关键字后跟成员的声明（无定义）；而要实例化由模板类限定的成员标识符，请后跟模板参数。

```
template int String<19>::overflows;
```

6.4 模板组合

可以用嵌套方式使用模板。这种方式尤其适用于在通用数据结构上定义通用函数，与在标准 C++ 库中相同。例如，模板排序函数可以通过一个模板数组类进行声明：

```
template <class Elem> void sort( Array<Elem> );
```

并定义为：

```
template <class Elem> void sort( Array<Elem> store )
    {int num_elems = store.GetSize();
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( store[j-1] > store[j] )
                  {Elem temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp;}}
```

上述示例定义了预先声明的 Array 类模板对象上的排序函数。下一个示例说明了排序函数的实际用法。

```
Array<int> int_array( 100 );    // 构造整型数组
sort( int_array );              // 排序该数组
```

6.5 默认模板参数

您可以将默认值赋予类模板 （但不是函数模板）的模板参数。

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

如果模板参数具有默认值，则该参数后的所有参数也必须具有默认值。模板参数仅能具有一个默认值。

6.6 模板专门化

将模板参数的某些组合视为特殊的参数可以优化性能，如以下 `twice` 的示例所示。或者，模板描述将无法处理一组可能的参数，如以下 `sort` 的示例所示。模板专门化允许您定义实际模板参数给定组合的可选实现。模板专门化覆盖了默认实例化。

6.6.1 模板专门化声明

使用模板参数的组合之前，您必须声明专门化。以下示例声明了 *twice* 和 *sort* 的专用实现。

```
template <> unsigned twice<unsigned>( unsigned original );
```

```
template <> sort<char*>( Array<char*> store );
```

如果编译器可以明确决定模板参数，则您可以省略模板参数。例如：

```
template <> unsigned twice( unsigned original );
```

```
template <> sort( Array<char*> store );
```

6.6.2 模板专门化定义

必须定义声明的所有模板专门化。下例定义了上一节中声明的函数。

```
template <> unsigned twice<unsigned>( unsigned original )
    {return original << 1;}
```

```
#include <string.h>
template <> void sort<char*>( Array<char*> store )
    {int num_elems = store.GetSize();
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( strcmp( store[j-1], store[j] ) > 0 )
                  {char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp;}}
```

6.6.3 模板专门化使用和实例化

专门化与其他任何模板一样使用并实例化，除此以外，完全专用模板的定义也是实例化。

6.6.4 部分专门化

在前一个示例中，模板是完全专用的。也就是说，模板定义了特定模板参数的实现。模板也可以部分专用，这意味着只有某些模板参数被指定，或者一个或多个参数被限定到某种类型。生成的部分专门化仍然是模板。例如，以下代码样例说明了主模板和该模板的完全专门化。

```
template<class T, class U> class A {...}; // 主模板
template<> class A<int, double> {...};    // 专门化
```

以下代码说明了主模板部分专门化的示例。

```
template<class U> class A<int> {...};           // 示例 1
template<class T, class U> class A<T*> {...}; // 示例 2
template<class T> class A<T**, char> {...};   // 示例 3
```

- 示例 1 提供了第一个模板参数是 `int` 类型时的特殊模板定义。

- 示例 2 提供了第一个模板参数是任何指针类型时的特殊模板定义。
- 示例 3 为第一个模板参数是指针到指针的任何类型而第二个模板参数是 `char` 类型时的情况提供了特殊模板定义。

6.7 模板问题部分

本节描述了使用模板时会遇到的问题。

6.7.1 非本地名称解析和实例化

有时模板定义使用模板参数或模板本身未定义的名称。如此，编译器解决了封闭模板作用域的名称，该模板可以在定义或实例化点的上下文中。名称可以在不同的位置具有不同的含义，产生不同的解析。

名称解析比较复杂。因此，您不应该依赖除一般全局环境中提供的名称外的非本地名称。也就是说，仅使用在任何地方都用相同方法声明和定义的非本地名称。在以下示例中，模板函数 `converter` 使用了非本地名称 `intermediary` 和 `temporary`。这些名称在 `use1.cc` 和 `use2.cc` 中具有不同的定义，并且可能在不同编译器下产生不同的结果。为了模板能可靠地工作，所有非本地名称（该示例中的 `intermediary` 和 `temporary`）都必须在任何地方具有相同的定义。

```
use_common.h     // 通用模板定义
                 template <class Source, class Target>
                 Target converter( Source source )
                        {temporary = (intermediary)source;
                        return (Target)temporary;}

use1.cc          typedef int intermediary;
                 int temporary;

                 #include "use_common.h"

use2.cc          typedef double intermediary;
                 unsigned int temporary;

                 #include "use_common.h"
```

非本地名称的一个常见的用法是模板内的 `cin` 和 `cout` 流。有时编程人员要将流作为模板参数传递，这时就要引用到全局变量。不过，`cin` 和 `cout` 在任何地方都必须具有相同的定义。

6.7.2 作为模板参数的本地类型

模板实例化系统取决于类型名称，等价于决定哪些模板需要实例化或重新实例化。因此本地类型用作模板参数时，会导致严重的问题。小心在代码中也出现类似的问题。例如：

代码样例 6-1 本地类型用作模板参数问题的示例

```
array.h      template <class Type> class Array {
                     Type* data;
                     int size;
                 public:
                     Array( int sz );
                     int GetSize( );
             };

array.cc     template <class Type> Array<Type>::Array( int sz )
                 {size = sz; data = new Type[size];}
             template <class Type> int Array<Type>::GetSize( )
                 {return size;}

file1.cc     #include "array.h"
             struct Foo {int data;};
             Array<Foo> File1Data(10);

file2.cc     #include "array.h"
             struct Foo {double data;};
             Array<Foo> File2Data(20);
```

在 file1.cc 中注册的 Foo 类型与在 file2.cc 中注册的 Foo 类型不同。以这种方法使用本地类型会出现错误和不可预料的结果。

6.7.3 模板函数的友元声明

模板在使用之前必须先声明。模板的使用由友元声明构成，不是由模板的声明构成。实际的模板声明必须在友元声明之前。例如，编译系统尝试链接以下示例中生成的对象文件时，会生成未实例化 operator<< 函数的未定义错误。

代码样例 6-2 友元声明问题的示例

```
array.h     // 生成 operator<< 函数的未定义错误
            #ifndef ARRAY_H
            #define ARRAY_H
            #include <iosfwd>

            template<class T> class array {
                int size;
            public:
                array();
                friend std::ostream&
                    operator<<(std::ostream&, const array<T>&);
            };
            #endif

array.cc    #include <stdlib.h>
            #include <iostream>

            template<class T> array<T>::array() {size = 1024;}

            template<class T>
            std::ostream&
            operator<<(std::ostream& out, const array<T>& rhs)
            {return out <<'[' << rhs.size <<']';}

main.cc     #include <iostream>
            #include "array.h"

            int main()
            {
                std::cout
                  << "creating an array of int..." << std::flush;
                array<int> foo;
                std::cout << "done\n";
                std::cout << foo << std::endl;
                return 0;
            }
```

注意，因为编译器作为普通函数 （array 类的 friend）的声明读取以下代码，所以编译期间不会出现错误消息。

```
friend ostream& operator<<(ostream&, const array<T>&);
```

因为 operator<< 实际上是模板函数，所以您需要在 template class array 的声明之前提供模板声明。不过，因为 operator<< 具有 type array<T> 的参数，所以 array<T> 的声明必须先于函数声明。文件 array.h 必须显示如下：

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// 下面两行将 operator<< 声明为模板函数。
template<class T> class array;
template<class T>
    std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<< <T> (std::ostream&, const array<T>&);
};
#endif
```

6.7.4 在模板定义内使用限定名称

C++ 标准需要具有限定名称的类型，该名称取决于用 typename 关键字显式标注为类型名称的模板参数。即使编译器“知道”应该是类型，也需要具有限定名称的类型。以下示例中的注释说明了需要用 typename 关键字来限定名称的类型。

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // 不是类型
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;    // 不是类型
template <class T> struct example {
  static typename T::a_type variable1;               // 依存
  static typename parametric<T>::a_type variable2; // 依存
  static simple::a_type variable3;                   // 不依存
};
template <class T> typename T::a_type              // 依存
  example<T>::variable1 = 0;                       // 不是类型
template <class T> typename parametric<T>::a_type // 依存
  example<T>::variable2 = 0;                       // 不是类型
template <class T> simple::a_type   // 不依存
example<T>::variable3 = 0;          // 不是类型
```

6.7.5 嵌套模板名称

由于 ">>" 字符序列解释为右移运算符，因此在一个模板名称中使用另一个模板名称时必须要非常小心。确保将邻近的 ">" 字符用至少一个空格分隔开。

例如，以下是形式错误的语句：

```
Array<String<10>> short_string_array(100); // >> = 右移
```

被解释为：

```
Array<String<10  >> short_string_array(100);
```

正确的语法为：

```
Array<String<10> > short_string_array(100);
```

6.7.6 引用静态变量和静态函数

在模板定义中，编译器不支持引用在全局作用域或名称空间中声明为静态的对象或函数。如果生成了多个实例，则每个示例引用到了不同的对象，因此违背了单次定义规则（C++ 标准的 3.2 节）。通常的失败指示是链接时丢失符号。

如果想要所有模板实例化共享单一对象，那么请使对象成为已命名空间的非静态成员。如果想要模板类的每个实例化不同对象，那么请使对象成为模板类的静态成员。如果想要模板函数每个实例化的不同对象，那么请使对象具有函数局部化属性。

6.7.7 在同一目录中使用模板生成多个程序

如果通过指定 `-instances=extern` 生成多个程序或库，那么建议在不同的目录中生成这些程序或库。如果要在同一目录中生成多个程序，那么您需要清除不同生成程序之间的系统信息库。这样可以避免出现任何不可预料的错误。更多信息请参见第 7-6 页的第 7.4.4 节 “共享模板系统信息库”。

考虑具有 `make` 文件 `a.cc`、`b.cc`、`x.h` 和 `x.cc` 的以下示例。注意该示例仅适用于指定 `-instances=extern` 时：

```
........
Makefile
........
CCC = CC

all:a b

a:
    $(CCC) -I. -instances=extern -c  a.cc
    $(CCC) -instances=extern -o a a.o

b:
    $(CCC) -I. -instances=extern -c  b.cc
    $(CCC) -instances=extern -o b b.o

clean:
    /bin/rm -rf SunWS_cache *.o a b
```

```
...
x.h
...
template <class T> class X {
public:
  int open();
  int create();
  static int variable;
};
```

```
...
x.cc
...
template <class T> int X<T>::create() {
  return variable ;
}

template <class T> int X<T>::open() {
  return variable ;
}

template <class T> int X<T>::variable = 1;
```

```
...
a.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.open();
  temp1.create();
}
```

```
...
b.cc
...
#include "x.h"

int main()
{
  X<int> temp1;

  temp1.create();
}
```

如果同时生成了 a 和 b，那么在两个生成之间增加 make clean。以下命令会引起错误：

```
example% make a
example% make b
```

以下命令不会产生任何错误：

```
example% make a
example% make clean
example% make b
```

第7章

编译模板

C++ 编译器在模板编译方面处理的工作要比传统 UNIX 编译器处理的工作多。C++ 编译器必须按需为模板实例生成对象代码。该编译器会使用模板系统信息库在多个独立的编译间共享模板实例，此外还接受某些模板编译选项。编译器必须在各个源文件中定位模板定义，并维护模板实例和主线代码之间的一致性。

7.1 冗余编译

给定标志 `-verbose=template` 后， C++ 编译器通知您模板编译期间的重要事件。相反，给定默认 `-verbose=no%template` 时，编译器不会通知您。 `+w` 选项可以指示模板实例化发生时其他潜在的问题。

7.2 系统信息库管理

`CCadmin`(1) 命令管理模板系统信息库。例如，程序中的更改会引起某些实例化过量，这样会浪费存储空间。 `CCadmin -clean` 命令（以前的 `ptclean`）清除所有实例化和有关的数据。实例化仅在需要时才重新创建。

7.2.1 生成的实例

为了生成模板实例，编译器将内联模板函数看作内联函数。编译器像管理其他内联函数一样管理这些内联模板函数，另外本章中的说明不适用于模板内联函数。

7.2.2 整个类实例化

编译器通常实例化独立于其他成员的模板类成员，因此编译器仅实例化程序中使用的成员。仅用于调试器的方法会因此而不正常地实例化。

有两种方法确保调试成员可用于调试器。

- 首先，编写使用模板类实例成员 （否则无用）的非模板函数，不需要调用该函数。
- 其次，使用 `-template=wholeclass` 编译器选项，通知编译器实例化模板类的所有非模板非内联成员，前提是任何这些相同成员都是可实例化的。

ISO C++ 标准允许开发者编写模板类，因为并不是所有成员都可以使用模板参数。只要非法成员未被实例化，程序就仍然完好。 ISO C++ 标准库使用了这种技术。不过，`-template=wholeclass` 选项实例化所有成员，因此在使用有问题的模板参数来进行实例化时就不能使用这种模板类。

7.2.3 编译时实例化

实例化是 C++ 编译器从模板创建可用的函数或对象的过程。 C++ 编译器使用了编译时实例化，在编译对模板的引用时强制进行实例化。

编译时实例化的优点是：

- 调试更为容易 - 错误消息出现在上下文中，同时允许编译器完整回溯到引用点。
- 模板实例化始终保持最新。
- 包括链接阶段在内的总编译时间减少了。

如果源文件位于不同的目录或您使用了具有模板符号的库，则模板可以多次实例化。

7.2.4 模板实例的放置和链接

从 Sun C++ 编译器的 5.5 版本开始，实例将被放入特殊的地址区，并且链接程序可以识别并丢弃重复实例。您可以指示编译器使用五个实例放置和链接方法之一：外部、静态、全局、显式和半显式。

- 外部实例适用于大多数程序的开发，并且满足以下条件时可以达到最好的执行效率：
  - 程序中的实例集比较小，但是每个编译单元引用了实例较大的子集。
  - 很少有在多于一个或两个编译单元中引用的实例。
- 静态，过时 - 见下文。
- 默认的全局实例适用于所有程序的开发，并且当对象引用不同的实例时执行效率可以达到最好。
- 显式实例适用于某些需精确控制的应用程序编译环境。

- 半显式实例需要较少控制的编译环境，但是生成了较大的对象文件，并且有严格的使用规则。

本节讨论了五种实例放置和链接方法。关于生成实例的详细信息，请参见第 6-5 页的第 6.3 节 “模板实例化”。

7.3 外部实例

对于外部实例方法，所有实例都放置在模板系统信息库中。编译器确保只有一个一致的模板实例存在；这些实例既不是未定义的也不是多重定义的。模板仅在需要时才重新实例化。对于非调试代码，使用 `-instances=extern` 时的所有对象文件的总计大小小于使用 `-instances=global` 时的所有对象文件的总计大小。

模板实例接收系统信息库中的全局链接。实例使用外部链接从当前编译单元引用。

注 – 如果以不同的步骤编译和链接并为编译步骤指定了 `-instance=extern`，则也必须为链接步骤指定 `-instance=extern`。

这种方法的缺点是更改程序或程序发生重大更改时必须清除缓存。缓存是并行编译的瓶颈，因为使用 `dmake` 时每次只能有一个编译访问缓存。另外，每个目录内仅能生成一个程序。

决定缓存中是否存在有效的模板实例比直接在主对象文件中创建实例（如果需要，用完后可以丢弃）要花费更长的时间。

使用 `instances=extern` 选项指定外部链接。

因为实例存储在模板系统信息库中，所以您必须使用 `CC` 命令将使用外部实例的 C++ 对象链接到程序中。

如果要创建包含所有模板实例的库，则使用具有 `-xar` 选项的 `CC` 命令。**不要**使用 `ar` 命令。例如：

```
example% CC -xar -instances=extern -o libmain.a a.o b.o c.o
```

有关详细信息，请参见第 16 章 。

7.3.0.1 可能的缓存冲突

指定 `-instance=extern` 时，请勿允许在同一目录中运行不同的编译器版本，以避免可能的缓存冲突。使用 `-instances=extern` 模板模型时，请考虑以下问题：

- 请勿在同一目录中创建不相关的二进制文件。在目录中创建任何二进制文件（.o、.a、.so，可执行程序）都应该是相关的，在所有对象、函数的名称中，将公共区键入到两个或多个具有相同定义的对象文件。
- 在同一目录中同时运行多个编译是安全的，例如使用 `dmake` 时。与另外一个链接步骤同时运行任何编译或链接步骤是不安全的。“链接步骤”指创建库或可执行程序的任何操作。确保 makefile 中的依存关系不允许任何内容与链接步骤以并行方式运行。

7.3.1 静态实例

注 – `-instances=static` 选项已过时。没有任何理由再使用 `-instances=static`，因为 `-instances=global` 现在向您提供了所有**静态**的优点而没有它的缺点。以前编译器中提供的该选项用于克服 C++ 5.5 中不存在的问题。

对于静态实例方法，所有实例都被放置在当前编译单元内。因此，模板在每个重新编译期间重新实例化；这些实例不保存到模板系统信息库。

这种方法的缺点是不遵循语言语义，并且会生成很大的对象和可执行文件。

实例接收静态链接。这些实例在当前编译单元外部是不可视的或不可用的。因此，模板可以在多个对象文件中具有相同的实例化。因为多重实例产生了不必要的大程序，所以对于不可能多重实例化模板的小程序可以使用静态链接。

静态实例的编译速度很快，因此这种方法也适用于修复并继续方式的调试。（请参见《使用 dbx 调试程序》。）

注 – 如果您的程序取决于多个编译单元间的共享模板实例 （例如模板类或模板函数的静态数据成员），请勿使用静态实例方法。否则程序会工作不正常。

使用 `-instances=static` 编译器选项指定静态实例链接。

7.3.2 全局实例

与以前的编译器发行版本不同，本版本的编译器无需保护全局实例的多个副本。

这种方法的优点是通常由其他编译器接受的不正确源代码也能在这种模式中接受。特别的是，从模板实例内对静态变量的引用是不合法的，但通常是可以接受的。

这种方法的缺点是单个对象文件会很大，原因是多个文件中模板实例有多个副本。如果您使用 `-g` 选项 （或没有该选项）编译调试的某些对象文件，那么很难预测是否可以获得链接到程序中模板实例的调试或非调试版本。

模板实例接收全局链接。这些实例在当前编译单元外部是可视的和可用的。

使用 -instances=global 选项 （默认选项）指定全局实例。

7.3.3 显式实例

在显式实例方法中，仅为显式实例化的模板生成实例。隐式实例化不能满足该要求。实例被放置在当前编译单元内。因此，模板在每个重新编译期间重新实例化，这些模板不保存到模板系统信息库。

这种方法的优点是拥有最少的模板编译和最小的对象大小。

缺点是您必须手动执行所有的实例化。

模板实例接收全局链接。这些实例在当前编译单元外部是可视的和可用的。链接程序识别并丢弃重复项目。

使用 -instances=explicit 选项指定显式实例。

7.3.4 半显式实例

使用半显式实例方法时，仅为显式实例化或模板体内隐式实例化的模板生成实例。那些被显式创建实例所需要的实例将会自动生成。主线代码中隐式实例化不满足该要求。实例被放置在当前编译单元内。因此，模板在每个重新编译期间重新实例化；生成的实例接收全局链接，且不会被保存到模板系统信息库中。

使用 -instances=semiexplicit 选项指定半显式实例。

7.4 模板系统信息库

模板系统信息库存储了多个独立编译之间的模板实例，因此模板实例仅当需要时才编译。模板系统信息库包含了使用外部实例方法时模板实例化所需的所有非源文件。系统信息库不用于其他种类的实例。

7.4.1 系统信息库结构

默认情况下，模板系统信息库包含在名为 SunWS_cache 的缓存目录中。

缓存目录包含在放置对象文件的目录中。您可以通过设置 SUNWS_CACHE_NAME 环境变量更改缓存目录的名称。注意， SUNWS_CACHE_NAME 变量的值必须是目录名称，不能是路径的名称。这是因为编译器自动将模板缓存目录放置到了对象文件目录中，因此编译器已经具有了路径。

7.4.2 写入模板系统信息库

编译器必须存储模板实例时，编译器将模板实例存储在对应于输出文件的模板系统信息库中。例如，以下命令行将对象文件写入 ./sub/a.o，并将模板实例写入 ./sub/SunWS_cache 内包含的系统信息库中。如果缓存目录不存在，且编译器需要实例化模板，则编译器将创建目录。

```
example% CC -o sub/a.o a.cc
```

7.4.3 从多模板系统信息库读取

编译器从对应于编译器读取的对象文件的模板系统信息库读取。也就是说，以下命令行从 ./sub1/SunWS_cache 和 ./sub2/SunWS_cache 读取，并且如果需要，则写入 ./SunWS_cache。

```
example% CC sub1/a.o sub2/b.o
```

7.4.4 共享模板系统信息库

系统信息库内的模板不能违反 ISO C++ 标准的单次定义规则。也就是说，使用所有的模板时模板必须具有相同的源。违反该规则会产生不可预料的行为。

确保不违反该规则的最简单和最保守的方法是在任何一个目录内仅生成一个程序或库。两个不相关的程序可以使用相同类型的名称或外部名称来表示不同的内容。如果程序共享模板系统信息库，则模板定义会出现冲突，会产生不可预料的结果。

7.4.5 模板实例自动与 -instances=extern 一致

模板系统信息库管理器确保了指定 -instances=extern 时，系统信息库中的实例状态与源文件保持一致并为最新。

例如，如果源文件用 -g 选项（即打开调试）编译，则从数据库所需的文件也用 -g 编译。

此外，模板系统信息库会跟踪编译中的更改。例如，如果设置 -DDEBUG 标志以定义名称 DEBUG，则数据库进行跟踪。如果在以后的编译中省略该标志，则编译器重新实例化设置依存关系的这些模板。

7.5 模板定义搜索

使用独立定义模板组织时，模板定义在当前编译单元不可用，并且编译器必须搜索该定义。本节描述了编译器如何找到定义。

定义搜索有点复杂且易于出错。因此如果可能，您应该使用定义包括模板文件组织。这样有助于避免一起定义搜索。请参见第 5-3 页的第 5.2.1 节 “包括的模板定义”。

注 – 如果使用 -template=no%extdef 选项，则编译器不会搜索独立源文件。

7.5.1 源文件位置约定

无需某个选项文件提供特定的指示说明，编译器可以采用 Cfront 样式的方法来查找模板定义文件。这种方法需要模板定义文件包含了与模板声明文件相同的基名。该方法也需要模板定义文件位于当前 include 路径中。例如，如果模板函数 foo() 位于 foo.h 中，则相匹配的模板定义文件应该命名为 foo.cc 或某些其他可识别的源文件扩展（.C、 .c、 .cc、 .cpp、 .cxx 或 .c++）。模板定义文件必须位于其中一个普通 include 目录或与定义文件相匹配的头文件的相同目录中。

7.5.2 定义搜索路径

另外一种可以替代用 -I 选项设定标准搜索路径的方法是：您可以用选项 -pti *directory* 指定模板定义文件的搜索目录。多个 -pti 标志定义了多个搜索目录，即搜索路径。如果您使用 -pti *目录*，则编译器在该路径查找模板定义文件并忽略 -I 标志。因为 -pti *目录*标志将源文件的搜索规则复杂化，所以使用 -I 选项代替 -pti *目录* 选项。

7.5.3 诊断有问题的搜索

有时，编译器会生成一些莫名其妙的警告或错误消息，原因是它正在查找您并不打算进行编译的文件。此问题通常是由于某个文件（如 foo.h）包含模板声明，而另一个文件（如 foo.cc）又隐式包含了该文件。

如果头文件 foo.h 具有模板声明，则在默认情况下，编译器将搜索名为 foo 且带有 C++ 文件扩展名（C、c、cc、cpp、.cxx 或 c++）的文件。如果找到这样的文件，编译器将自动把它包含进来。有关这类搜索的详细信息，请参见第 7-7 页的第 7.5 节 “模板定义搜索”。

如果您拥有文件 foo.cc，但是不想采用这种方式进行处理，则您有两个选择：

- 更改 .h 或 .cc 文件的名称，以消除名称匹配。
- 通过指定 -template=no%extdef 选项禁止自动搜索模板定义文件。然后必须在代码中显式包含所有模板定义，并且不能使用 “独立定义”模型。

第8章

异常处理

本章讨论了 C++ 编译器如何实现异常处理。更多信息请参见第 11-2 页的第 11.2 节“在多线程程序中使用异常”。更多关于异常处理的信息，请参见 Bjarne Stroustrup 编著的《*The C++ Programming Language*》第三版（Addison-Wesley， 1997）。

8.1 同步和异步异常

异常处理设计用于仅支持同步异常，例如数组范围检查。术语**同步异常**意味着异常仅可以来源于 `throw` 表达式。

C++ 标准支持具有终止模型的同步异常处理。**终止**意味着异常抛出后，控制不会返回到抛出点。

异常处理没有设计用于直接处理诸如键盘中断等异步异常。不过，如果小心处理，在出现异步事件时也可以进行异常处理。例如，要用信号进行异常处理工作，您可以编写设置全局变量的信号处理程序，并创建另外一个例程来定期轮询该变量的值，当该变量值发生更改时抛出异常。不能从信号处理程序抛出异常。

8.2 指定运行时错误

有五个与异常有关的运行时错误消息：

- 没有异常处理程序
- 未预料到的异常抛出
- 异常只能在处理程序中重新抛出
- 在解开堆栈时，析构函数必须处理自身的异常
- 内存不足

在运行时检测到错误时，错误消息显示了当前异常的类型和这五个错误消息之一。默认情况下，预定义的函数 `terminate()` 被调用，然后调用 `abort()`。

编译器使用异常规范中提供的信息来优化代码生成。例如，禁止不抛出异常的函数表条目，而函数异常规范的运行时检查在任何可能的地方被消除。

8.3 禁用异常

如果知道程序中不使用异常，则可以使用编译器选项 `-features=no%except` 来禁止生成支持异常处理的代码。该选项的使用可以稍微减小代码的大小，并能加快代码的执行速度。不过，用禁用的异常编译的文件链接到使用异常的文件时，在用禁用的异常编译的文件中的某些局部对象在发生异常时不会销毁。默认情况下，编译器生成支持异常处理的代码。通常都要启用异常，只有时间和空间的开销是考虑的重要因素时才禁止异常。

注 – 因为 C++ 标准库 `dynamic_cast` 和默认运算符 `new` 需要异常，所以在标准模式（默认模式）中编译时不能关闭异常。

8.4 使用运行时函数和预定义的异常

标准头文件 <exception> 提供了 C++ 标准中指定的类和与异常相关的函数。仅在标准模式 （编译器默认模式，或使用选项 -compat=5）中编译时才可访问该头文件。以下摘录了 <exception> 头文件声明。

```
// 标准头文件 <exception>
namespace std {
    class exception {
                exception() throw();
                exception(const exception&) throw();
                exception& operator=(const exception&) throw();
                virtual ~exception() throw();
                virtual const char* what() const throw();
    };
    class bad_exception:public exception {...};
    // 意外的异常处理
       typedef void (*unexpected_handler)();
       unexpected_handler
          set_unexpected(unexpected_handler) throw();
       void unexpected();
    // 终止处理
       typedef void (*terminate_handler)();
       terminate_handler set_terminate(terminate_handler)
throw();
       void terminate();
       bool uncaught_exception() throw();
}
```

标准类 exception 是由所选语言结构或 C++ 标准库抛出的所有异常的基类。类型 exception 的对象可以被构造、复制，在不生成异常销毁。虚拟成员函数 what() 返回了描述异常的字符串。

为了与 C++ 发行版本 4.2 中所用的异常兼容，头文件 <exception.h> 也被提供用于标准模式。该头文件允许转换到标准 C++ 代码，并包含了不是标准 C++ 部分的声明。您可以按照开发计划的许可来更新代码以遵循 C++ 标准 （使用 <exception> 而不使用 <exception.h>）。

```
// 头文件 <exception.h>，用于转换
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

在兼容模式 (-compat[=4]) 中，头文件 <exception> 不可用，并且头文件 <exception.h> 引用到 C++ 发行版本 4.2 提供的相同头文件。头文件在这里不会重新生成。

8.5 将异常与信号和 Setjmp/Longjmp 混合

只要 setjmp/longjmp 函数不交互，您就可以在发生异常的程序中使用这些函数。

使用异常和 setjmp/longjmp 的所有规则分别应用。此外，只要在 A 处抛出和在 B 处捕获的异常具有相同的结果，从点 A 到点 B 的 longjmp 就是有效的。具体来讲，您不必 longjmp 进或出 try 块或 catch 块 （直接或间接），或 longjmp 超过自动变量或临时变量的初始化或不常用销毁。

不能从信号处理程序抛出异常。

8.6 生成具有异常的共享库

永远不要使用具有包含 C++ 代码的程序的 -Bsymbolic，相反要使用链接程序映射文件。使用 -Bsymbolic 时，在不同模块中的引用可以绑定到被假设为全局对象内容的不同副本中。

异常机制依赖对地址的比较。如果您具有某项内容的两个副本，它们的地址就不等同且异常机制可能失败，这是由于异常机制依赖对假设为唯一地址内容的比较。

使用 dlopen 打开共享库时，必须将 RTLD_GLOBAL 用于异常。

第9章

类型转换操作

本章讨论 C++ 标准中较新的类型转换操作符：const_cast、reinterpret_cast、static_cast 和 dynamic_cast。类型转换可以将对象或值从一种类型转换为另一种类型。

这些类型转换操作比以前的类型转换操作更好控制。dynamic_cast<> 操作符提供了一种方法来检查到多态类的指针的实际类型。您可以用文本编辑器来搜索所有新样式的类型转换（搜索 _cast），而查找旧样式的类型转换需要语法分析。

否则，新的类型转换全部执行传统类型转换符号允许的类型转换子集。例如，const_cast<int*>(v) 可以写为 (int*)v。新的类型转换仅将各种可用的操作分类以更清楚地表示您的意图，并允许编译器提供更完善的检查。

类型转换操作符是始终启用的。类型转换符不能被禁用。

9.1 const_cast

表达式 const_cast<*T*>(*v*) 可用于更改指针或引用的 const 或 volatile 限定符。（在新样式的类型转换中，只有 const_cast<> 可以去掉 const 限定符。） T 必须是指针、引用或指向成员的指针的类型。

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_cast<A&>(a1).f( );                    // 去掉 const
ip = const_cast<int*> (cvip);     // 去掉 const 和 volatile
}
```

9.2 reinterpret_cast

表达式 reinterpret_cast<*T*>(*v*) 更改了表达式 *v* 值的解释。该表达式可用于在指针和整数类型之间，在不相关的指针类型之间，在指向成员的指针类型之间，和在指向函数的指针类型之间转换类型。

reinterpret_cast 操作符的用法可以具有未定义的或依赖于实现的结果。以下几点描述了唯一确定的行为：

- 指向数据对象或函数的指针（但不是指向成员的指针）可以转换为足够包含该指针的任何整数类型。（long 类型总是足以包含 C++ 编译器支持的体系结构上的指针值。）转换回原始类型时，指针值将与原始指针值相比较。
- 指向 （非成员）函数的指针可以转换为指向不同 （非成员）函数类型的指针。如果转换回原始类型，指针值将与原始指针相比较。
- 假设新类型的对齐要求没有原始类型严格，则指向对象的指针可以转换为指向不同对象类型的指针。转换回原始类型时，指针值将与原始指针值相比较。
- 如果使用重新解释的类型转换将 “指向 *T1* 的指针”类型的表达式转换为 “指向 *T2* 的指针”类型的表达式，则 *T1* 类型左值可以转换为 “对 *T2* 的引用”类型。

- 如果 *T1* 和 *T2* 都是函数类型或都是对象类型，则可以将 “指向 *T1* 类型 X 成员的指针”类型右值显式转换为 “指向 *T2* 类型 *Y* 成员的指针”类型右值。
- 在所有允许的情况下，空指针类型转换为不同的空指针类型后仍然是空指针。
- reinterpret_cast 操作符不能用于转换 const，转换 const 要使用 const_cast。
- reinterpret_cast 操作符不能用于转换指向位于同一类分层结构中不同类的指针，需使用静态或动态类型转换来实现这一目的。(reinterpret_cast 不执行所需的调整。）这一点在以下示例中描述：

```
class A {int a; public:A();};
class B:public A {int b, c;};
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);       // 安全
     B* bp = reinterpret_cast<B*>(&a1);     // 不安全
     const A a2;
     ap = reinterpret_cast<A*>(&a2);  // 错误，没有 const
}
```

9.3 static_cast

表达式 static_cast<*T*>(*v*) 将表达式 *v* 的值转换为 *T* 类型。该表达式可用于任何隐式允许的转换类型。此外，任何值都可以转换为 void，并且如果类型转换与旧样式一样合法，则任何隐式转换都可以反向转换。

```
class B            {...};
class C:public B {...};
enum E {first=1, second=2, third=3};
void use_of_static_cast(C* c1)
{
  B* bp = c1;                    // 隐式转换
  C* c2 = static_cast<C*>(bp); // 反向隐式转换
  int i = second;                // 隐式转换
  E e = static_cast<E>(i);     // 反向隐式转换
}
```

static_cast 操作符不能用于转换 const。您可以使用 static_cast 来 “向下”转换分层结构 （从基到派生的指针或引用），但是不检查转换，因此结果有可能无法使用。static_cast 不能用于从虚拟基类向下转换。

9.4 动态类型转换

指向类的指针（或引用）可以实际指向（引用）从该类派生的任何类。有时希望指向完全派生类的指针，或指向完整对象的某些其他子对象。动态类型转换可以实现这些功能。

注 – 用兼容模式 (`-compat[=4]`) 编译时，如果程序使用动态类型转换，您必须用 `-features=rtti` 编译。

动态类型转换将指向一个类 *T1* 的指针（或引用）转换到指向另一个类 *T2* 的指针（引用）。*T1* 和 *T2* 必须位于同一分层结构，类必须是可访问的（经公共派生），并且转换必须要是明确的。此外，除非转换是从派生的类到该派生类的一个基类，包括 *T1* 和 *T2* 的分层结构的最小部分必须是多态的（至少具有一个虚函数）。

在表达式 `dynamic_cast<`*T*`>(`*v*`)` 中，*v* 是要被类型转换的表达式，而 *T* 是要转换到的类型。*T* 必须是完整类的类型（其中一个的定义是可视的）的指针或引用，或指向 *cv* `void`、`const`、`volatile` 或 `const volatile` 的指针，其中 *cv* 是空字符串。

9.4.1 将分层结构向上类型转换

向上类型转换分层结构时，如果 *T* 指向（或引用）由 *v* 指向（引用）类型的基类，则该转换等价于 `static_cast<`*T*`>(`*v*`)`。

9.4.2 类型转换到 `void*`

如果 *T* 是 `void*`，则该结果是指向完整对象的指针。也就是说，*v* 可能指向某些完整对象的其中一个基类。在这种情况下，`dynamic_cast<void*>(`*v*`)` 的结果如同将分层结构向下转换 *v* 到完整对象的类型，然后转换到 `void*`。

类型转换到 `void*` 时，分层结构必须是多态的（具有虚函数）。

9.4.3 将分层结构向下或交叉类型转换

向下或交叉类型转换分层结构时，分层结构必须是多态的（具有虚函数）。结果在运行时检查。

向下或交叉类型转换分层结构时，从 *v* 到 *T* 的转换并不总是可行的。例如，尝试的转换可以是不明确的， *T* 可能不可访问，或 *v* 不能指向 （或引用）必要类型的对象。如果运行时检查失败且 *T* 是指针类型，则类型转换表达式的值是 *T* 类型的空指针。如果 *T* 是引用类型，什么都不会返回（C++ 中没有空引用），且标准异常 `std::bad_cast` 被抛出。

例如，此公共派生的示例成功：

```
#include <assert.h>
#include <stddef.h> // 为空

class A {public:virtual void f();};
class B {public:virtual void g();};
class AB:public virtual A, public B {};

void simple_dynamic_casts( )
{
  AB ab;
  B*  bp  = &ab;         // 无需类型转换
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // 成功
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

然而该示例失败，原因是基类 B 是不可访问的。

```
#include <assert.h>
#include <stddef.h> // 为空
#include <typeinfo>

class A {public:virtual void f() {}};
class B {public:virtual void g() {}};
class AB:public virtual A, private B {};

void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab; // 中断保护所必需的 C 样式类型转换
  A* ap  = dynamic_cast<A*>(bp); // 失败，B 是不可访问的
  assert(ap == NULL);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // 失败，B 是不可访问的
  }
  catch(const std::bad_cast&) {
    return; // 此处捕获的失败引用类型转换
  }
  assert(0); // 不应到此处
}
```

如果在一个单独的基类中存在虚拟继承和多重继承，则实际动态类型转换必须能够识别出唯一的匹配。如果匹配不唯一，则类型转换失败。例如，假定有如下附加类定义：

```
class AB_B:public AB,         public B {};
class AB_B__AB:public AB_B,       public AB {};
```

示例：

```
void complex_dynamic_casts( )
{
  AB_B__AB ab_b__ab;
  A*ap = &ab_b__ab;
                    // 正确：找到唯一的静态 A
  AB*abp = dynamic_cast<AB*>(ap);
                    // 失败：不明确
  assert( abp == NULL );
                    // 静态错误：AB_B* ab_bp = (AB_B*)ap;
                    // 不是动态类型转换
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // 动态转换正确
  assert( ab_bp != NULL );
}
```

dynamic_cast 返回的空指针错误可用作两个代码体之间的条件，一个用于键入正确时处理类型转换，而另一个用于键入错误时停止类型转换。

```
void using_dynamic_cast( A* ap )
{
  if ( AB *abp = dynamic_cast<AB*>(ap) )
    {             // abp 非空，
                  // 因此 ap 是指向 AB 对象的指针
                  // 继续并使用 abp
      process_AB(abp);}
  else
    {           // abp 为空，
                // 因此 ap 不是指向 AB 对象的指针
                // 不使用 abp
      process_not_AB( ap );
    }
}
```

在兼容模式 (-compat[=4]) 中，如果运行时信息还未启用 -features=rtti 编译器选项，则编译器将 dynamic_cast 转换到 static_cast 并发出警告。

如果禁用异常，则编译器将 dynamic_cast<T&> 转换到 static_cast<T&> 并发出警告。（如果发现转换在运行时无效，则需要抛出对引用类型的 dynamic_cast 异常。关于异常的信息，请参见第 8 章。

动态转换需要比对应的设计模式慢，例如虚函数的转换。请参见《Design Patterns:Elements of Reusable Object-Oriented Software》， Erich Gamma 著 (Addison-Wesley，1994)。

第 10 章

改善程序性能

采用编译器易于编译优化的方式编写函数，可以改善 C++ 函数的性能。有许多关于软件性能的书籍，尤其是关于 C++。例如，请参见《C++ Programming Style》， Tom Cargill 所著 (Addison-Wesley, 1992)、《Writing Efficient Programs》，Jon Louis Bentley 所著 (Prentice-Hall, 1982)、《Efficient C++: Performance Programming Techniques》 ， Dov Bulka 和 David Mayhew 所著 (Addison-Wesley, 2000) 以及《Effective C++ — 50 Ways to Improve Your Programs and Designs》，第二版，Scott Meyers 所著 (Addison-Wesley, 1998)。本章不重复这些有价值的信息，而是讨论了主要影响 C++ 编译器的那些性能技术。

10.1 避免临时对象

C++ 函数经常会产生必须创建并销毁的隐式临时对象。对于重要的类，临时对象的创建和销毁会占用很多处理时间和内存。C++ 编译器消除了某些临时类，但是并不能消除所有的临时类。

您编写的函数要将临时对象的数目减少到理解程序所需的最小数目。这些技术包括：使用显式变量而不使用隐式临时对象，以及使用引用变量而不使用值参数。另外一种技术是实现和使用诸如 `+=` 这样的操作，而不实现和使用只包含 `+` 和 `=` 的操作。例如，下面的第一行引入了 `a + b` 结果的临时对象，而第二行则不是。

```
T x = a + b;
T x( a ); x += b;
```

10.2 使用内联函数

使用扩展内联而不使用正常调用时，对小而快速的函数的调用可以更小更快速。反过来，如果使用扩展内联而不建立分支，则对又长又慢的函数的调用会更大更慢。另外，只要函数定义更改，就必须重新编译对内联函数的所有调用。因此，使用内联函数时要格外小心。

在期望更改函数定义**而且**重新编译所有调用方很费时间时，请不要使用内联函数。否则，如果扩展函数内联的代码比调用函数的代码少，**或**使用函数内联时**应用程序**执行速度显著提高，那么可以使用内联函数。

编译器不能内联所有函数调用，因此使用最耗时的函数内联会需要某些源码的更改。使用 `+w` 选项以了解何时不发生函数内联。在以下情况中，编译器将**不会**内联函数：

- 函数包含了复杂控制构造，例如循环、 switch 语句和 try/catch 语句。这些函数很少多次执行复杂控制构造。要内联这种函数，请将函数分割为两部分，里边的部分包含了复杂控制构造，而外边的部分决定了是否调用里边的部分。即使编译器可以内联完整函数，从函数常用部分中分隔出不常用部分的这种技术也可以改善性能。
- 内联函数体又大又复杂。因为对函数体内其他内联函数的调用，或因为隐式构造函数和析构函数调用 （通常发生在派生类的构造函数和析构函数中），所以简单函数体可以非常复杂。对于这种函数，内联扩展很少提供显著的性能改善，所以函数一般不内联。
- 内联函数调用的参数既大又复杂。对于内联成员函数调用的对象是内联函数调用的自身这种情况，编译器特别敏感。要内联具有复杂参数的函数，只需将函数参数计算到局部变量并将变量传递到函数。

10.3 使用默认运算符

如果类定义不声明无参数的构造函数、复制构造函数、复制赋值运算符或析构函数，那么编译器将隐式声明它们。它们都是调用的默认运算符。类似 C 的结构具有这些默认运算符。编译器生成默认运算符时，可以了解大量关于需要处理的工作和可以产生优良代码的工作。这种代码通常比用户编写的代码的执行速度快，原因是编译器可以利用汇编级功能的优点，而编程人员则不能利用该功能的优点。因此默认运算符执行所需的工作时，程序不能声明这些运算符的用户定义版本。

默认运算符是内联函数，因此内联函数不合适时不使用默认运算符 （请参见上一节）。否则，默认运算符是合适的：

- 用户编写的无参数构造函数仅为构造函数的基对象和成员变量调用无参数构造函数。有效的基元类型具有 “不执行任何操作”无参数构造函数。
- 用户编写的复制构造函数仅复制所有的基对象和成员变量。

- 用户编写的复制赋值运算符仅复制所有的基对象和成员变量。
- 用户编写的析构函数可以为空。

某些 C++ 编程手册建议编写类的编程人员始终定义所有的运算符，以便该代码的任何读者都能了解该编程人员没有忘记考虑默认运算符的语义。显然，该建议与以上讨论的优化有冲突。这种冲突的解决方案是在代码中放置注释以表明类正使用默认运算符。

10.4 使用值类

包括结构和联合在内的 C++ 类通过值来传递和返回。对于 Plain-Old-Data (POD) 类，C++ 编译器需要像 C 编译器一样传递结构。这些类的对象是**直接**传递的。对于用户定义复制构造函数的类的对象，编译器需要构造对象的副本，将指针传递到副本，并在返回后销毁副本。这些类的对象是**间接**传递的。编译器也可以选择介于这两个需求之间的类。不过，该选择影响二进制的兼容性，因此编译器对每个类的选择必须保持一致。

对于大多数编译器，直接传递对象可以加快执行速度。这种执行速度的改善对于小值类（例如复数和概率值）来说尤其明显。有时为了改善程序执行效率，您可以设计更可能直接传递而不是间接传递的类。

在兼容模式 (`-compat[=4]`) 中，如果类具有以下任何一条，则间接传递该类：

- 用户定义的构造函数
- 虚函数
- 虚拟基类
- 间接传递的基
- 间接传递的非静态数据成员

否则，类被直接传递。

在标准模式（默认模式）中，如果类具有以下任何一条，则间接传递该类：

- 用户定义的复制构造函数
- 用户定义的析构函数
- 间接传递的基
- 间接传递的非静态数据成员

否则，类被直接传递。

10.4.1 选择直接传递类

尽可能直接传递类：

- 只要可能，就使用默认构造函数，尤其是默认复制构造函数。

- 尽可能使用默认析构函数。默认析构函数不是虚拟的，因此具有默认析构函数的类通常不是基类。
- 避免使用虚函数和虚拟基。

10.4.2 在不同的处理器上直接传递类

C++ 编译器直接传递的类（和联合）与 C 编译器传递结构（或联合）完全相同。不过，C++ 结构和联合在不同的架构上进行不同的传递。

表 10-1 在不同架构上结构和联合的传递

架构	描述
SPARC V7/V8	通过在调用方内分配存储并将指针传递到该存储，传递并返回结构和联合。（也就是说，所有的结构和联合都通过引用传递。）
SPARC V9	不超过 16 个字节（32 个字节）的结构在寄存器中传递。通过在调用方内分配存储并将指针传递到该存储，联合和所有其他结构将被传递并返回。（也就是说，小的结构在寄存器中传递，而联合和大的结构通过引用传递。）因此，小值类与基元类具有相同的传递效率。
x86 平台	结构和联合通过在堆栈上分配空间并将参数复制到堆栈上来传递。通过在调用方的帧中分配临时对象并作为隐式第一个参数传递临时对象的地址，返回结构和联合。

10.5 缓存成员变量

访问成员变量是 C++ 成员函数的通用操作。

编译器必须经常从内存通过 `this` 指针装入成员变量。因为值通过指针装入，所以编译器有时不能决定何时执行第二次装入或以前装入的值是否仍然有效。在这些情况下，编译器必须选择安全但缓慢的方法，在每次访问成员变量时重新装入成员变量。

如下所示，可以通过在局部变量中显式缓存成员变量的值来避免不必要的内存重新装入：

- 声明局部变量并使用成员变量的值初始化该变量。
- 在函数中成员变量的位置使用局部变量。
- 如果局部变量变化，那么将局部变量的最终值赋值到成员变量。不过，如果成员函数在该对象上调用另一个成员函数，那么该优化会产生不可预料的结果。

当值位于寄存器中时，这种优化最有效，而这种情况也与基元类型相同。基于内存的值的优化也会很有效，因为减少的别名使编译器获得了更多的机会来进行优化。

如果成员变量经常通过引用 （显式或隐式）来传递，那么优化可能并没什么效果。

有时，类的目标语义需要成员变量的显式缓存，例如在当前对象和其中一个成员函数参数之间有潜在别名时。例如：

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

会产生不可预料的结果，前提是调用时使用：

```
x*=x;
```

第 11 章

生成多线程程序

本章解释了如何生成多线程程序。此外，还讨论了异常的使用，解释了如何在线程之间共享 C++ 标准库对象，此外还描述了如何在多线程环境中使用传统（旧的） iostream。

关于多线程的更多信息，请参见《多线程编程指南》、《Tools.h++ 用户指南》和《标准 C++ 库用户指南》。

11.1 生成多线程程序

C++ 编译器附带的所有库都是多线程安全的。如果需要生成多线程应用程序，或者需要将应用程序链接到多线程库，那么您必须使用 `-mt` 选项来编译和链接程序。该选项将 `-D_REENTRANT` 传递给预处理程序，并将 `-lthread` 以正确的顺序传递给 `ld`。在兼容模式 (`-compat[=4]`) 下， `-mt` 选项确保了 `libthread` 在 `libC` 之前被链接。在标准模式（默认模式）下， `-mt` 选项确保了 `libthread` 在 `libCrun` 之前被链接。

不要直接用 `-lthread` 链接应用程序，因为这将会引起 `libthread` 以错误的顺序链接。

以下示例显示了当编译和链接分开进行时，生成多线程应用程序的正确方法：

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

以下示例显示了生成多线程应用程序的错误方法：

```
example% CC -c -mt myprog.o
example% CC myprog.o -lthread <- libthread 链接错误
```

11.1.1 表明多线程编译

您可以通过使用 ldd 命令来检查应用程序是否被链接到 libthread：

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>       /usr/lib/libm.so.1
libCrun.so.1 =>    /usr/lib/libCrun.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>       /usr/lib/libc.so.1
libdl.so.1 =>      /usr/lib/libdl.so.1
```

11.1.2 与线程和信号一起使用 C++ 支持库

C++ 支持库 libCrun、libiostream、libCstd 和 libC 都是多线程安全的，但不是 async 安全的。这就是说在多线程应用程序中，支持库中可用的函数不能用于信号处理程序。这样做的话将导致死锁状态。

在多线程应用程序的信号处理程序中使用下列内容是不安全的：

- iostream
- new 和 delete 表达式
- 异常

11.2 在多线程程序中使用异常

当前异常处理的实现方法是多线程安全的；一个线程中的异常不会干预其他线程中的异常。不过，您不能使用异常来进行线程之间的通信；一个线程中抛出的异常不会被其他线程捕获到。

每个线程都可以设置自己的 terminate() 或 unexpected() 函数。在一个线程中调用 set_terminate() 或 set_unexpected() 仅影响该线程的异常。对于任何线程而言，terminate() 的默认函数是 abort()（请参见第 8-1 页的第 8.2 节 “指定运行时错误”）。

11.2.1 线程取消

通过对 pthread_cancel(3T) 调用进行线程取消会导致堆栈中的自动（本地非静态）对象的析构，除非指定 -noex 或 -features=no%except。

pthread_cancel(3T) 使用与异常一样的机制。当一个线程被取消时，本地析构函数与清除例程将交叉执行，该清除例程被用户注册为 pthread_cleanup_push()。在特定的清除例程注册之后，函数调用的本地对象在例程执行前就被销毁了。

11.3 在线程之间共享 C++ 标准库对象

C++ 标准库 （libCstd -library=Cstd）是多线程安全的 （有些语言环境例外），这样可以确保在多线程环境中库内部正常工作。但是，您仍需要将各个线程之间要共享的库对象锁定起来。请参见 setlocale(3C) 和 attributes(5) 手册页。

例如，如果实例化字符串，然后创建新的线程并使用引用将字符串传递给线程。因为要在线程之间显示共享这个字符串对象，所以您必须锁定对于该字符串的写访问。（库提供的用于完成该任务的工具在下文中会有描述。）

另一方面，如果通过值将字符串传递给新的线程，即使两个不同的线程通过 Rogue Wave 的 "write on copy" 技术共享表示，也不必担心锁定。库将自动处理锁定。只有当要使对象显式可用于多线程或在线程之间传递引用，以及使用全局或静态对象时，您才需要锁定。

下文描述了 C++ 标准库内部使用的锁定 （同步）机制，该机制用于确保在多线程下出现正确的行为。

_RWSTDMutex 和 _RWSTDGuard 这两个同步类提供了实现多线程安全的机制。

_RWSTDMutex 类在下列成员函数中提供了与平台无关的锁定机制：

- void acquire() — 自己锁定，或在锁定之前处于阻塞状态。
- void release() — 自己解锁。

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

_RWSTDGuard 类是公用包装类，其中封装有 _RWSTDMutex 类的对象。_RWSTDGuard 对象尝试在其构造函数中获取封装的互斥（抛出 std::exception on error 派生的 ::thread_error 类型的异常），并在析构函数中释放互斥（析构函数从不会抛出异常）。

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

另外，您可以使用宏 _RWSTD_MT_GUARD(mutex)（以前的 _STDGUARD）在多线程生成中有条件地创建 _RWSTDGuard 的对象。该对象保护代码块的其余部分，并在该代码块中定义为可同时被多个线程执行。在单线程生成中，宏扩展到空表达式中。

以下示例说明了这些机制的使用。

```
#include <rw/stdmutex.h>

//
// 多个线程共享的整数。
//
int I;

//
// 用以同步更新为 I 的互斥。
//
_RWSTDMutex I_mutex;

//
// 每次对 I 递增 1。直接使用 _RWSTDMutex。
//

void increment_I ()
{
   I_mutex.acquire();  // 锁定互斥。
   I++;
   I_mutex.release();  // 解锁互斥。
}

//
// 每次对 I 递减 1。使用 _RWSTDGuard。
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex);  // 获取 I_mutex 的锁定。
   --I;
   //
   // 调用处于保护状态的析构函数时， I 的锁定被释放。
   //
}
```

11.4 在多线程环境中使用传统 iostream

本节描述了如何使用 `libC` 和 `libiostream` 库的 `iostream` 类在多线程环境中进行输入输出操作 (I/O)。本节还提供了如何通过从 `iostream` 类派生来扩展库的功能。不过本节并**不是**用 C++ 编写多线程代码的指南。

此处的讨论只适用于旧的 iostream（libC 和 libiostream），但不适用于 libCstd（即新的 iostream，它是 C++ 标准库的一部分）。

iostream 库允许其接口被多线程环境下的应用程序使用，而这些程序在运行支持的 Solaris 操作系统版本时使用多线程功能。如果应用程序使用以前版本库的单线程功能，那么该应用程序不会受到影响。

如果库在线程环境中能正常工作，就将其定义为是多线程安全的。通常，此处的“正确”意味着所有的公用函数都是可重入的。iostream 库提供了对多线程的保护，多线程尝试修改由多个线程共享的对象（即 C++ 类的实例）状态。不过，iostream 对象的多线程安全范围限制在执行对象公用成员函数的周期内。

注 – 应用程序**不会**因为使用 libC 库中的多线程安全对象而自动保证是多线程安全的。仅当应用程序在多线程环境中能按预期执行时，才将其定义为是多线程安全的。

11.4.1 多线程安全的 iostream 库的组织

多线程安全的 iostream 库的组织与其他版本的 iostream 库稍有不同。库的输出接口指的是 iostream 类的公共的和受保护的成员函数以及可用的基类集合，这一点与其他版本的库相同，但各个版本的类分层结构是不同的。详细信息，请参见第 11-11 页的第 11.4.2 节 “对 iostream 库进行接口更改”。

初始核心类以 unsafe_ 前缀来重命名。表 11-1 中列出了 iostream 软件包中的核心类。

表 11-1 iostream 初始核心类

类	描述
stream_MT	多线程安全类的基类。
streambuf	缓冲区的基类。
unsafe_ios	该类包含各种流类通用的状态变量；例如，错误和格式化状态。
unsafe_istream	该类支持 streambuf 检索到的字符序列的有格式和无格式的转换。
unsafe_ostream	该类支持存储在 streambuf 中的字符序列的有格式和无格式的转换。
unsafe_iostream	该类合并unsafe_istream 和unsafe_ostream 类用于双向操作。

每个多线程安全类都是从基类 stream_MT 派生的。每个多线程安全类（除了 streambuf 外）也是从现有 unsafe_ 基类派生的。示例如下：

```
class streambuf:public stream_MT {...};
class ios:virtual public unsafe_ios, public stream_MT {...};
class istream:virtual public ios, public unsafe_istream {...};
```

stream_MT 类提供了要求的互斥 (mutex) 锁定，以便每个 iostream 类都是多线程安全的。此外还提供了用于动态启用和禁用锁定的功能，以便多线程安全的属性可以动态更改。unsafe_ 类中包括 I/O 转换和缓冲区管理的基本功能，加到库中的多线程安全只限于派生类。每个类的多线程安全版本包含了与 unsafe_ base 类相同的受保护的和公共的成员函数。多线程安全版本的类中每个成员函数都像包装器一样锁定对象，调用 unsafe_ base 类中的相同函数，然后解锁对象。

注 – streambuf 类**不是**从不安全的类派生的。streambuf 的公共的和受保护的成员函数是可以通过锁定而重入的。此外同时也提供了带有 _unlocked 后缀的不锁定版本。

11.4.1.1 公共转换例程

iostream 接口中增加了一组可重入的多线程安全的公共函数。用户指定的缓冲区被作为每个函数的附加参数。这些函数如下所述：

表 11-2 多线程安全的可重入公共函数

功能	描述
`char *oct_r (char *buf, int buflen, long num, int width)`	将指针返回到用八进制表示数字的 ASCII 字符串。非零宽度假定为格式化的字段宽度。返回值不保证指向用户提供缓冲区的开始部分。
`char *hex_r (char *buf, int buflen, long num, int width)`	将指针返回到用十六进制表示数字的 ASCII 字符串。非零宽度假定为格式化的字段宽度。返回值不保证指向用户提供缓冲区的开始部分。
`char *dec_r (char *buf, int buflen, long num, int width)`	将指针返回到用十进制表示数字的 ASCII 字符串。非零宽度假定为格式化的字段宽度。返回值不保证指向用户提供缓冲区的开始部分。
`char *chr_r (char *buf, int buflen, long num, int width)`	将指针返回到包含字符 chr 的 ASCII 字符串。如果宽度非零，那么字符串包含了后跟 chr 的 width 空格。返回值不保证指向用户提供缓冲区的开始部分。
`char *form_r (char *buf, int buflen, long num, int width)`	返回由 sprintf 格式化字符串的指针，使用格式字符串 format 和其他剩余的参数。缓冲区必须具有足够的空间以包含格式化的字符串。

注 – 用来确保与早期版本 libC 兼容的 iostream 库的公共转换例程（oct、hex、dec、chr 和 form）都**不是**多线程安全的。

11.4.1.2 使用多线程安全的 libC 库进行编译和链接

生成使用 libC 库的 iostream 类的应用程序以在多线程环境中运行时，请使用 -mt 选项编译和链接应用程序的源代码。该选项将 -D_REENTRANT 传递给预处理程序，并将 -lthread 传递给链接程序。

注 – 使用 -mt （而不是 -lthread）来链接 libC 和 libthread。该选项确保了库的正确链接顺序。不正确的使用 -lthread 会引起应用程序工作不正常。

使用 iostream 类的单线程应用程序不需要特殊的编译器或链接程序选项。默认情况下，编译器用 libC 库链接。

11.4.1.3 多线程安全的 iostream 限制

iostream 库的多线程安全的限制定义意味着大量使用 iostream 的编程用语在使用共享 iostream 对象的多线程环境中是不安全的。

检查错误状态

要多线程安全，必须要对引起错误的 I/O 操作所在的临界区进行错误检查。以下示例说明了如何检查错误：

代码样例 11-1 检查错误状态

```
#include <iostream.h>
enum iostate {IOok, IOeof, IOfail};

iostate read_number(istream& istr, int& num)
{
       stream_locker sl(istr, stream_locker::lock_now);

       istr >> num;

       if (istr.eof()) return IOeof;
       if (istr.fail()) return IOfail;
       return IOok;
}
```

在该示例中，stream_locker 对象 sl 的构造函数锁定了 istream 对象 istr。在 read_number 终止时调用的析构函数 sl 解锁 istr。

获取通过上次未格式化输入操作提取的字符

要成为多线程安全的，必须在互斥使用 istream 对象的线程内，并在包括上次输入操作和 gcount 调用的执行周期内，调用 gcount 函数。以下示例说明了对 gcount 的调用：

代码样例 11-2 调用 gcount

```
#include <iostream.h>
#include <rlocks.h>
void fetch_line(istream& istr, char* line, int& linecount)
{
       stream_locker sl(istr, stream_locker::lock_defer);

       sl.lock(); // 锁定流 istr
       istr >> line;
       linecount = istr.gcount();
       sl.unlock(); // 解锁 istr
       ...
}
```

在该示例中，stream_locker 类的 lock 和 unlock 成员函数定义了程序中的互斥区域。

用户定义的 I/O 操作

要成为多线程安全的，为用户定义类型定义的 I/O 操作必须锁定以定义临界区 （这些类型涉及到单个操作的特定顺序）。以下示例说明了用户定义的 I/O 操作：

代码样例 11-3 用户定义的 I/O 操作

```
#include <rlocks.h>
#include <iostream.h>
class mystream:public istream {

       // 其他定义 ...
       int getRecord(char* name, int& id, float& gpa);
};

int mystream::getRecord(char* name, int& id, float& gpa)
{
       stream_locker sl(this, stream_locker::lock_now);
```

代码样例 11-3 用户定义的 I/O 操作 (续)

```
        *this >> name;
        *this >> id;
        *this >> gpa;

        return this->fail() == 0;
}
```

11.4.1.4 减少多线程安全类的性能开销

在本版本的 libC 库中使用多线程安全类 （即使是在单线程应用程序中），也会导致一定的性能开销。但是您可以使用 libC 中的 unsafe_ 类以避免此开销。

范围转换运算符可以用于执行基 unsafe_ 类的成员函数，例如：

```
        cout.unsafe_ostream::put('4');
```

```
        cin.unsafe_istream::read(buf, len);
```

注 – unsafe_ 类不能在多线程应用程序中安全地使用。

您可以使 cout 和 cin 对象 unsafe，然后使用正常操作来代替使用 unsafe_ 类。这会稍微降低性能。以下示例说明了如何使用 unsafe cout 和 cin：

代码样例 11-4 禁用多线程安全

```
#include <iostream.h>
// 禁用多线程安全
cout.set_safe_flag(stream_MT::unsafe_object);
// 禁用多线程安全
cin.set_safe_flag(stream_MT::unsafe_object);
cout.put('4');
cin.read(buf, len);
```

iostream 对象是多线程安全的时，将提供互斥锁定来保护对象的成员变量。该锁定给仅在单线程环境中执行的应用程序增加了不必要的开销。要提高性能，您可以动态切换 iostream 对象的多线程安全。以下示例使 iostream 对象多线程不安全：

代码样例 11-5 切换到多线程不安全

```
fs.set_safe_flag(stream_MT::unsafe_object);// 禁用多线程安全
        .... 执行各种 i/o 操作
```

您可以在 iostream **没有**被线程共享的代码中安全地使用多线程不安全的流；例如，在只有一个线程的程序中，或在每个 iostream 都是线程专用的程序中。

如果将同步显式插入到程序中，那么也可以在线程共享 iostream 对象的环境中安全使用多线程不安全的 iostreams。以下示例说明了该技术：

代码样例 11-6 在多线程不安全的对象中使用同步

```
generic_lock() ;
fs.set_safe_flag(stream_MT::unsafe_object) ;
... 执行各种 i/o 操作
generic_unlock() ;
```

其中 generic_lock 和 generic_unlock 函数可以是使用诸如互斥、信号量或读取器/写入器锁定等基元的任何同步机制。

注 – LibC 提供的 stream_locker 类是实现这一目的的最佳机制。

更多信息请参见第 11-15 页的第 11.4.5 节 “对象锁定”。

11.4.2 对 iostream 库进行接口更改

本节讨论了使 iostream 库具有多线程安全而进行的接口更改。

11.4.2.1 新增类

下表列出了增加到 libC 接口的新类。

代码样例 11-7 新增类

```
stream_MT
stream_locker
unsafe_ios
unsafe_istream
unsafe_ostream
unsafe_iostream
unsafe_fstreambase
unsafe_strstreambase
```

11.4.2.2 新增类的分层结构

下表列出了增加到 iostream 接口的新增类的分层结构。

代码样例 11-8 新增类的分层结构

```
class streambuf:public stream_MT {...};
class unsafe_ios {...};
class ios:virtual public unsafe_ios, public stream_MT {...};
class unsafe_fstreambase:virtual public unsafe_ios {...};
class fstreambase:virtual public ios, public unsafe_fstreambase
  {...};
class unsafe_strstreambase:virtual public unsafe_ios {...};
class strstreambase:virtual public ios, public unsafe_strstreambase
{...};
class unsafe_istream:virtual public unsafe_ios {...};
class unsafe_ostream:virtual public unsafe_ios {...};
class istream:virtual public ios, public unsafe_istream {...};
class ostream:virtual public ios, public unsafe_ostream {...};
class unsafe_iostream:public unsafe_istream, public unsafe_ostream {...};
```

11.4.2.3 新增函数

下表列出了增加到 iostream 接口的新函数。

代码样例 11-9 新增函数

```
 class streambuf {
 public:
   int sgetc_unlocked();
   void sgetn_unlocked(char *, int);
   int snextc_unlocked();
   nt sbumpc_unlocked();
   void stossc_unlocked();
   int in_avail_unlocked();
   int sputbackc_unlocked(char);
   int sputc_unlocked(int);
   int sputn_unlocked(const char *, int);
   int out_waiting_unlocked();
 protected:
   char* base_unlocked();
   char* ebuf_unlocked();
   int blen_unlocked();
   char* pbase_unlocked();
   char* eback_unlocked();
```

代码样例 11-9 新增函数 (续)

```
    char* gptr_unlocked();
    char* egptr_unlocked();
    char* pptr_unlocked();
    void setp_unlocked(char*, char*);
    void setg_unlocked(char*, char*, char*);
    void pbump_unlocked(int);
    void gbump_unlocked(int);
    void setb_unlocked(char*, char*, int);
    int unbuffered_unlocked();
    char *epptr_unlocked();
    void unbuffered_unlocked(int);
    int allocate_unlocked(int);
};

class filebuf:public streambuf {
public:
 int is_open_unlocked();
 filebuf* close_unlocked();
 filebuf* open_unlocked(const char*, int, int =
   filebuf::openprot);

 filebuf* attach_unlocked(int);
};

class strstreambuf:public streambuf {
public:
 int freeze_unlocked();
 char* str_unlocked();
};


unsafe_ostream& endl(unsafe_ostream&);
unsafe_ostream& ends(unsafe_ostream&);
unsafe_ostream& flush(unsafe_ostream&);
unsafe_istream& ws(unsafe_istream&);
unsafe_ios& dec(unsafe_ios&);
unsafe_ios& hex(unsafe_ios&);
unsafe_ios& oct(unsafe_ios&);

char* dec_r (char* buf, int buflen, long num, int width)
char* hex_r (char* buf, int buflen, long num, int width)
char* oct_r (char* buf, int buflen, long num, int width)
char* chr_r (char* buf, int buflen, long chr, int width)
```

代码样例 11-9 新增函数 (续)

```
char* str_r (char* buf, int buflen, const char* format, int width
    = 0);
char* form_r (char* buf, int buflen, const char* format,...)
```

11.4.3 全局和静态数据

多线程应用程序中的全局和静态数据不会在各个线程之间安全地共享。尽管线程独立执行，但它们在进程中共享对全局和静态对象的访问。如果一个线程修改了这种共享对象，那么进程中的其他线程将观察该更改，使得状态难以维持。在 C++ 中，类对象（类的实例）靠其成员变量的值来维持状态。如果类对象被共享，那么该类对象将易于被其他线程更改。

多线程应用程序使用 `iostream` 库并包含 `iostream.h` 时，默认情况下标准流 `cout`、`cin`、`cerr` 和 `clog` 被定义为全局共享对象。因为 `iostream` 库是多线程安全的，所以该库将在执行 `iostream` 对象的成员函数时保护共享对象的状态不受其他线程的访问或更改。不过，对象的多线程安全范围被限制在执行对象公用成员函数的周期内。例如，

```
int c;
cin.get(c);
```

在 `get` 缓冲区中获得下一个字符，并更新 *ThreadA* 中的缓冲区指针。不过，如果 *ThreadA* 中的下一条指令是另一个 `get` 调用，那么 `libC` 库将不能保证按序返回下一个字符。这是因为，例如 *ThreadB* 可能也会在 *ThreadA* 的两个 `get` 调用的间隙中执行 `get` 调用。

更多关于共享对象和多线程问题的处理策略，请参见第 11-15 页的第 11.4.5 节 “对象锁定”。

11.4.4 序列执行

通常当使用 `iostream` 对象时，I/O 操作的序列必须是多线程安全的。例如，如下所示代码：

```
cout << " Error message:"<< errstring[err_number] << "\n";
```

涉及到 `cout` 流对象的三个成员函数的执行。由于 `cout` 是共享对象，所以必须像临界区那样独立执行序列，使其能在多线程环境中正常工作。要独立对 `iostream` 类对象执行操作序列，您必须使用某些形式的锁定。

libC 库提供了 stream_locker 类来锁定 iostream 对象的操作。关于 stream_locker 类的信息，请参见第 11-15 页的第 11.4.5 节 “对象锁定”。

11.4.5 对象锁定

解决共享对象和多线程的最简单策略是通过确保 iostream 对象被锁定到线程，从而避免此类问题的发生。例如，

- 在线程的函数入口内局部声明对象。
- 在线程特定数据中声明对象。（关于如何使用线程特定的数据，请参见 thr_keycreate(3T) 手册页。）
- 使流对象专用于特定线程。按照约定，对象线程是 private。

不过在许多情况下 （例如默认共享标准流对象），使对象专用于某线程是不可能的，这就需要其他的策略了。

要独立对 iostream 类对象执行操作序列，您必须使用某些形式的锁定。即使对于单线程应用程序，锁定也会增加某些开销。是否增加锁定或使 iostream 对象专用于某个线程取决于为应用程序选择的线程模型：线程是独立的还是协同操作的？

- 如果每个独立的线程使用自己的 iostream 对象来生成或使用数据，那么这些 iostream 对象专用于各自所属的线程并且不需要锁定。
- 如果线程是协同操作的 （即，共享同一个 iostream 对象），那么必须对共享对象进行同步访问，而且必须使用某种形式的锁定以使得序列化操作独立化。

11.4.5.1 stream_locker 类

iostream 库提供了 stream_locker 类来锁定 iostream 对象的一系列操作。因此您可以将动态启用或禁用 iostream 对象的锁定所造成的性能开销降到最低。

stream_locker 类的对象可用于使流对象的操作序列独立化。例如，下例中所示代码尝试查找文件中的某一位置，并读取下一个数据块。

代码样例 11-10 使用锁定操作的示例

```
#include <fstream.h>
#include <rlocks.h>

void lock_example (fstream& fs)
{
    const int len = 128;
    char buf[len];
    int offset = 48;
       stream_locker s_lock(fs, stream_locker::lock_now);
       . . . . .// 打开文件
```

代码样例 11-10 使用锁定操作的示例 (续)

```
        fs.seekg(offset, ios::beg);
        fs.read(buf, len);
}
```

在该示例中，stream_locker 对象的构造函数定义了互斥区域的开始位置。在互斥区域中每次只能执行一个线程。从函数返回后调用的析构函数定义了互斥区域的结束位置。Stream_locker 对象确保了在文件中查找特定偏移和从文件中读取能够自动地同步执行，并且在初始的 *ThreadA* 读取文件之前， *ThreadB* 无法更改文件偏移。

使用 stream_locker 对象的替代方法是显式定义互斥区域。在以下示例中，要使 I/O 操作和随后的错误检查独立化，使用了 vbstream_locker 对象的 lock 和 unlock 成员函数调用。

代码样例 11-11 令 I/O 操作和错误检查独立化

```
{
        ...
        stream_locker file_lck(openfile_stream,
                                stream_locker::lock_defer);
        ....
        file_lck.lock();  // 锁定 openfile_stream
        openfile_stream << "Value:" << int_value << "\n";
        if(!openfile_stream) {
                file_error("Output of value failed\n");
                return;
        }
        file_lck.unlock();  // 解锁 openfile_stream
}
```

更多信息请参见 stream_locker(3CC4) 手册页。

11.4.6 多线程安全类

您可以通过派生新的类来扩展或专用化 iostream 类。如果由派生类实例化的对象被用于多线程环境中，那么这些类必须是多线程安全的。

派生多线程安全类时的注意事项包括：

- 通过保护对象的内部状态避免多线程的修改，使得类对象成为多线程安全的。要实现这一目的，请序列化对成员变量的公共访问，并使用互斥锁定来保护成员函数。
- 使用 stream_locker 对象，令多线程安全基类的成员函数调用序列独立化。
- 通过在 stream_locker 对象定义的临界区中使用 streambuf 的成员函数 _unlocked，避免锁定开销。

- 函数被应用程序直接调用时，锁定 streambuf 类的公共虚函数。这些函数包括：xsgetn、underflow、pbackfail、xsputn、overflow、seekoff 和 seekpos。
- 使用 ios 类中的成员函数 iword 和 pword 来扩展 ios 对象的格式化状态。不过，如果多个线程共享对 iword 或 pword 函数的相同索引，将会发生问题。要使线程成为多线程安全的，请使用相应的锁定方案。
- 锁定返回成员变量的值大于 char 的成员函数。

11.4.7 对象析构

多个线程共享的 iostream 对象被删除之前，主线程必须核实子线程已完成了对共享对象的使用。下例说明了如何安全销毁共享对象。

代码样例 11-12 销毁共享对象

```
#include <fstream.h>
#include <thread.h>
fstream* fp;

void *process_rtn(void*)
{
       // 使用 fp 的子线程体 ...
}

void multi_process(const char* filename, int numthreads)
{
       fp = new fstream(filename, ios::in); // 创建 fstream 对象
                                            // 创建线程之前。
       // 创建线程
       for (int i=0; i<numthreads; i++)
              thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);

           ...
       // 等待线程结束
       for (int i=0; i<numthreads; i++)
              thr_join(0, 0, 0);

       delete fp;                             // 所有线程完成后
       fp = NULL;                            // 删除 fstream 对象。
}
```

11.4.8 示例应用程序

以下代码提供了多线程应用程序的示例，该应用程序以多线程安全方式使用 libC 库的 iostream 对象。

该示例应用程序创建了多达 255 个线程。每个线程读取不同的输入文件，每次读取一行，并且使用标准输出流 cout 将行输出到输出文件。所有线程共享的输出文件用值来标记，该值表明了哪个线程执行输出操作。

代码样例 11-13 以多线程安全方式使用 iostream 对象

```
// 创建标记过的线程数据
// 输出文件的形式为：
//          < 标记 >< 数据字符串 >\n
// 其中标记是无符号字符的整数值。
// 最多允许 255 个线程在该应用程序中运行
// < 数据字符串 > 是任何可以打印的字符
// 因为标记是字符形式的整数值，
// 您需要使用 od 来查看输出文件，建议：

//             od -c out.file |more

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <thread.h>

struct thread_args {
  char* filename;
  int thread_tag;
};

const int thread_bufsize = 256;

// 每个线程的入口例程
void* ThreadDuties(void* v) {
// 获取该线程的参数
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // 打开线程输入文件
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // 每次读取一行
```

代码样例 11-13 以多线程安全方式使用 iostream 对象 (续)

```
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // 锁定 cout 流使得 i/o 操作独立化
    lockout.lock();
  // 标记行并发送到 cout
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  // argv: 1+ 每个线程的文件名列表
   if(argc < 2) {
     cout << "usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// thread_id 的数组
  thread_t created_threads[thread_bufsize];
// 线程入口例程的参数数组
  thread_args thr_args[thread_bufsize];
  int i;
  for(i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// 为线程赋值标记 - 小于 256 的值
    thr_args[i].thread_tag = total_tags++;
// 创建线程
    thr_create(0, 0, ThreadDuties, &thr_args[i],
               THR_SUSPENDED, &created_threads[i]);
  }

  for(i = 0; i < num_threads; i++) {
    thr_continue(created_threads[i]);
  }
  for(i = 0; i < num_threads; i++) {
    thr_join(created_threads[i], 0, 0);
  }
  return 0;
}
```

第 III部分 库

第 12 章

使用库

库提供了在多个应用程序间共享代码的方法，也提供了减小超大型应用程序复杂度的方法。 C++ 编译器使您可以访问各种库。本章说明了如何使用这些库。

12.1 C 库

Solaris 操作环境附带了多个库，安装在 `/usr/lib` 中。这些库大多有 C 接口。其中，`libc` 和 `libm` 库默认由 `CC` 驱动程序链接。如果使用了 `-mt` 选项，库 `libthread` 就会被链接。要链接任何其他的库，请在链接时使用合适的 `-l` 选项。例如，要链接 `libdemangle` 库，请在链接时 `CC` 命令行上传递 `-ldemangle`：

```
example% CC text.c -ldemangle
```

C++ 编译器具有自己的运行时支持库。所有的 C++ 应用程序通过 `CC` 驱动程序链接到这些库。 C++ 编译器还具有其他一些有用的库，如下节所述。

12.2 C++ 编译器提供的库

某些库是和 C++ 编译器一起提供的。这些库中有一些仅在兼容模式 (`-compat=4`) 下可用，有一些仅在标准模式 (`-compat=5`) 下可用，还有一些在两种模式下都可用。 `libgc` 和 `libdemangle` 具有 C 接口，可以在任何模式下链接到应用程序。

下表列出了随 C++ 编译器提供的库，以及可以使用这些库的模式。

表 12-1 C++ 编译器附带的库

库	描述	可用模式
libstlport	标准库的 STLport 实现。	-compat=5
libstlport_dbg	调试模式的 STLport 库	-compat=5
libCrun	C++ 运行时	-compat=5
libCstd	C++ 标准库	-compat=5
libiostream	传统的 iostreams	-compat=5
libC	C++ 运行时，传统的 iostreams	-compat=4
libcsunimath	支持 -xia 选项	-compat=5
libcomplex	complex 库	-compat=4
librwtool	Tools.h++ 7	-compat=4、 -compat=5
librwtool_dbg	启用调试的 Tools.h++ 7	-compat=4、 -compat=5
libgc	垃圾收集	C 接口
libdemangle	还原	C 接口

注 – 请不要重定义或修改 STLport、 Rogue Wave 或 Sun Microsystems C++ 库的任何配置宏。库配置和生成的方式是能够和 C++ 编译器一起工作。 libCstd 和 Tool.h++ 配置为可互操作，所以修改配置宏会导致程序不能编译、不能链接或不能正确运行。

12.2.1 C++ 库描述

以下是这些库中每个库的简单描述。

- libCrun：该库包含了标准模式 （-compat=5）下编译器需要的运行时支持。提供了对 new/delete、异常和 RTTI 的支持。

  libCstd：这是 C++ 标准库。特别之处是，该库包括了 iostreams。如果已有使用传统 iostreams 的源码，并且要使用标准 iostreams，那么您必须修改源码以符合新接口。详细信息请参见 《C++ 标准库参考》联机手册。要访问该手册，请将 Web 浏览器指向：

```
file:/opt/SUNWspro/docs/zh/index.html
```

  如果您的编译器软件没有安装在 /opt 目录中，请通过系统管理员获取系统中的相应路径。

- `libiostream`：这是使用 `-compat=5` 生成的传统 iostream 库。如果已有使用传统 iostream 的源码，而您希望使用标准模式 (`-compat=5`) 编译这些源码，可以使用 `libiostream` 而不必修改源码。使用 `-library=iostream` 获取该库。

注 – 标准库的很大部分取决于使用的标准 iostream。在相同程序中使用传统的 iostream 可能会出现问题。

- `libC`：这是兼容模式 （`-compat=4`）需要的库。该库包含了 C++ 运行时支持和传统 iostream。
- `libcomplex`：该库提供了兼容模式（`-compat=4`）下的复数运算。在标准模式下，可使用 `libCstd` 中的复数运算功能。
- `libstlport`：这是 C++ 标准库的 STLport 实现。通过指定选项 `-library=stlport4`，您可以使用该库而非默认的 `libCstd`。不过，不能在同一程序中同时使用 `libstlport` 和 `libCstd`。您必须使用其中之一编译和链接包括输入库在内的一切项目。
- `librwtool` (`Tools.h++`)：Tools.h++ 是源自 RogueWave 的 C++ 基础类。本发行版本提供了该库的版本 7。该库可用于传统 iostream 形式 (`-library=rwtools7`) 和标准 iostream 形式 (`-library=rwtools7_std`)。关于该库的更多信息，请参见以下联机文档。
  - *Tools.h++ 用户指南*（版本 7）
  - *Tools.h++ 类库参考*（版本 7）

  要访问该文档，请将 Web 浏览器指向：

```
file:/opt/SUNWspro/docs/zh/index.html
```

  如果您的编译器软件没有安装在 `/opt` 目录中，请通过系统管理员获取系统中的相应路径。
- `libgc`：该库用于部署模式或垃圾收集模式。与 `libgc` 库的简单链接会自动且永久地修复程序的内存泄漏。将程序与 `libgc` 库链接时，无需调用 `free` 或 `delete` 而只需正常编程。垃圾收集库对动态加载库具有依存性，因此在链接程序时要指定 `-lgc` 和 `-ldl`。

  附加信息请参见 `gcFixPrematureFrees`(3) 和 `gcInitialize`(3) 手册页。
- `libdemangle`：该库用于还原 C++ 修整名。

12.2.2 访问 C++ 库的手册页

本节描述的与库有关的手册页位于：

- `/opt/SUNWspro/man/man1`
- `/opt/SUNWspro/man/man3`
- `/opt/SUNWspro/man/man3C++`

- /opt/SUNWspro/man/man3cc4

注 – 如果您的编译器软件没有安装在 /opt 目录中，请通过系统管理员获取系统中的相应路径。

要访问这些手册页，请确保 MANPATH 包括了 /opt/SUNWspro/man （或系统上编译器软件的相应路径）。关于设置 MANPATH 的指示，请参见本书前面部分中的第 xxx 页的“访问手册页”。

要访问 C++ 库的手册页，请键入：

```
example% man library-name
```

要访问 C++ 库版本 4.2 的手册页，请键入：

```
example% man -s 3CC4 library-name
```

您也可以通过将浏览器指向以下地址来访问手册页：

```
file:/opt/SUNWspro/docs/zh/index.html
```

12.2.3 默认 C++ 库

某些 C++ 库默认由 CC 驱动程序链接，而其他库需要显式链接。在标准模式下，下列库默认由 CC 驱动程序链接：

```
-lCstd -lCrun -lm -lc
```

在兼容模式 (-compat) 下，下列库是默认链接的：

```
-lC -lm -lc
```

更多信息请参见第 A-41 页的第 A.2.48 节 "-library=l[,l...]"。

12.3 相关的库选项

CC 驱动程序提供了多个选项帮助使用库。

- 使用 -l 选项指定要链接的库。

- 使用 -l 选项指定搜索库的目录。
- 使用 -mt 选项编译和链接多线程代码。
- 使用 -xia 选项链接区间运算库。
- 使用 -xlang 链接 Fortran 运行时库。
- 使用 -library 选项指定 Sun C++ 编译器附带的以下库：
  - libCrun
  - libCstd
  - libiostream
  - libC
  - libcomplex
  - libstlport, libstlport_dbg
  - librwtool, librwtool_dbg
  - libgc

注 – 要使用传统 iostream 形式的 librwtool，请使用 -library=rwtools7 选项。要使用标准 iostream 格式的 librwtool，请使用 -library=rwtools7_std 选项。

同时使用 -library 和 -staticlib 选项指定的库将会静态链接。某些示例：

- 以下命令将 Tools.h++ 版本 7 的传统 iostream 形式与 libiostream 库动态链接起来。

```
example% CC test.cc -library=rwtools7,iostream
```

- 以下命令静态链接了 libgc 库。

```
example% CC test.cc -library=gc -staticlib=gc
```

- 以下命令在兼容模式下编译 test.cc 并静态链接了 libC。因为在兼容模式下 libC 被默认链接，所以您不必使用 -library 选项指定该库。

```
example% CC test.cc -compat=4 -staticlib=libC
```

- 以下命令排除了库 libCrun 和 libCstd，否则这两个库是默认包括在内的。

```
example% CC test.cc -library=no%Crun,no%Cstd
```

默认情况下，CC 根据命令行选项链接不同的系统库集合。如果指定 -xnolib（或 -nolib），CC 只链接那些在命令行上使用 -l 选项显式指定的库。（使用 -xnolib 或 -nolib 时，出现的 -library 选项会被忽略。）

-R 选项允许在可执行文件中生成动态库搜索路径。执行期间，运行时链接程序使用这些路径搜索应用程序所需的共享库。默认情况下（如果编译器安装在标准目录中），CC 驱动程序将 -R/opt/SUNWspro/lib 传递到 ld。可以使用 -norunpath 禁止在可执行程序中生成共享库的默认路径。

12.4 使用类库

通常使用类库有两个步骤：

1. 在源码中包括适当的头文件。
2. 将程序与对象库链接。

12.4.1 iostream 库

C++ 编译器提供了 iostream 的两个实现：

- **传统 iostream**。该术语是指 C++ 4.0、4.0.1、4.1 和 4.2 编译器以及更早基于 cfront 的 3.0.1 编译器附带的 iostream 库。该库没有任何标准，只是很多现有代码使用它。该库在兼容模式下是 libC 的一部分，在标准模式下也可用于 libiostream。
- **标准 iostream**。C++ 标准库 libCstd 的一部分，仅用于标准模式；该库与传统 iostream 库的二进制和源码都不兼容。

如果已有 C++ 源，那么代码可能象以下示例一样使用传统 iostream。

```
// 文件 prog1.cc
#include <iostream.h>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

以下命令在兼容模式下编译，并将 prog1.cc 链接到名为 prog1 的可执行程序中。传统 iostream 库是 libC 的一部分，在兼容模式下是默认链接的。

```
example% CC -compat prog1.cc -o prog1
```

下面的示例使用标准 iostream。

```
// 文件 prog2.cc
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

以下命令将 prog2.cc 编译和链接到名为 prog2 的可执行程序中。该程序在标准模式下编译，而包括标准 iostream 库的 libCstd 默认链接。

```
example% CC prog2.cc -o prog2
```

关于 libCstd 的更多信息，请参见第 13 章。关于 libiostream 的更多信息，请参见第 14 章。

关于编译模式的完整讨论，请参见 《C++ 迁移指南》。

12.4.2 complex 库

标准库提供了模板化的 complex 库，该库与 C++ 4.2 编译器提供的 complex 库类似。如果在标准模式下编译，则必须使用 <complex> 来代替 <complex.h>。不能在兼容模式下使用 <complex>。

在兼容模式下，必须在链接时显式请求 complex 库。在标准模式下， complex 库包括在 libCstd 中，并且是默认链接的。

标准模式没有 complex.h 头文件。在 C++ 4.2 中， "complex" 是类的名称，但是在标准 C++ 中， "complex" 是模板名称。不可能提供可使旧的代码不加修改就可工作的 typedef。因此，为使用复数的 4.2 版编写的代码需要某些简单的编辑，以便使用标准库。例如，以下代码是为 4.2 版编写的，并将在兼容模式下编译。

```
// 文件 ex1.cc （兼容模式）
#include <iostream.h>
#include <complex.h>

int main()
{
    complex x(3,3), y(4,4);
    complex z = x * y;
    cout << "x=" << x << ", y=" << y << ", z=" << z << endl;
}
```

以下示例在兼容模式下编译并链接 ex1.cc，然后执行该程序。

```
example% CC -compat ex1.cc -library=complex
example% a.out
x=(3, 3), y=(4, 4), z=(0, 24)
```

此处将 ex1.cc 重写为 ex2.cc 以便在标准模式下编译：

```
// 文件 ex2.cc（为标准模式重写的 ex1.cc）
#include <iostream>
#include <complex>
using std::complex;

int main()
{
     complex<double> x(3,3), y(4,4);
     complex<double> z = x * y;
     std::cout << "x=" << x << ", y=" << y << ", z=" << z <<
       std::endl;
}
```

以下示例在标准模式下编译并链接重写的 ex2.cc，然后执行该程序。

```
% CC ex2.cc
% a.out
x=(3,3), y=(4,4), z=(0,24)
```

关于使用复数运算库的更多信息，请参见第 15 章。

12.4.3 链接 C++ 库

下表列出了链接 C++ 库的编译器选项。更多信息请参见第 A-41 页的第 A.2.48 节 "-library=l[,l...]"。

表 12-2 链接 C++ 库的编译器选项

库	编译模式	选项
Classic iostream	-compat=4	不需要
	-compat=5	-library=iostream
complex	-compat=4	-library=complex
	-compat=5	不需要
Tools.h++ version 7	-compat=4	-library=rwtools7
	-compat=5	-library=rwtools7，iostream -library=rwtools7_std
Tools.h++ version 7 debug	-compat=4	-library=rwtools7_dbg
	-compat=5	-library=rwtools7_dbg，iostream -library=rwtools7_std_dbg
Garbage collection	-compat=4	-library=gc
	-compat=5	-library=gc
STLport version 4	-compat=5	-library=stlport4
STLport version 4 debug	-compat=5	-library=stlport4_dbg

12.5 静态链接标准库

通过将每个默认库的 -l*lib* 选项传递到链接程序，CC 驱动程序默认链接了包括 libc 和 libm 的多个库的共享版本。（关于兼容模式和标准模式默认库的列表，请参见第 12-4 页的第 12.2.3 节 “默认 C++ 库”。）。

如需静态链接任何默认库，则可以结合 -staticlib 选项使用 -library 选项静态链接 C++ 库。这种替换方法比上文所述的方法简单。例如：

```
example% CC test.c -staticlib=Crun
```

在本示例中，-library 选项没有显式包括在命令中。在这种情况下，-library 选项是不必要的，因为在标准模式下 （默认模式），-library 的默认设置是 Cstd,Crun。

或者，您可以使用 -xnolib 编译器选项。使用 -xnolib 选项时，驱动程序不将任何 -l 选项传递到 ld，所以您必须自己传递这些选项。下例显示了在 Solaris 8 或 Solaris 9 操作环境中如何静态链接 libCrun，以及如何动态链接 libm 和 libc：

```
example% CC test.c -xnolib -lCstd -Bstatic -lCrun -Bdynamic -lm -lc
```

-l 选项的顺序是重要的。 -lCstd, -lCrun 和 -lm 选项出现在 -lc 前面。

某些 CC 选项链接到其他库。也可以使用 -xnolib 禁止这些库链接。例如，使用 -mt 选项会导致 CC 驱动程序将 -lthread 传递到 ld。不过，如果同时使用了 -mt 和 -xnolib，则 CC 驱动程序不会将 -lthread 传递到 ld。更多信息请参见第 A-108 页的第 A.2.142 节 "-xnolib"。关于 ld 的更多信息，请参见 《Linker and Libraries Guide》。

12.6 使用共享库

以下共享库包括在 C++ 编译器中：

- ibCrun.so
- libC.so
- libcomplex.so
- libstlport.so
- librwtool.so
- libgc.so
- libgc_dbg.so
- libCstd.so
- libiostream.so

与程序链接的每个共享对象所出现的消息都记录在生成的可执行文件 （a.out 文件）中，该信息由 ld.so 用于在运行时编辑期间执行动态链接。因为将库代码放入地址空间的工作会延迟，所以使用共享库的程序的运行时行为对环境的改变 （也就是说，将库从一个目录移动到另外一个目录）是很敏感的。例如，如果程序与 /opt/SUNWspro/lib 中的 libcomplex.so.5 链接，而 libcomplex.so.5 后来被移动到 /opt2/SUNWspro/lib 中，那么运行二进制代码时，会产生以下消息：

```
ld.so: libcomplex.so.5: not found
```

通过将环境变量 LD_LIBRARY_PATH 设置到新的库目录，不必重新编译就可以运行旧的二进制代码。

在 C shell 中：

```
example% setenv LD_LIBRARY_PATH \
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
```

在 Bourne shell 中：

```
example$ LD_LIBRARY_PATH=\
/opt2/SUNWspro/release/lib:${LD_LIBRARY_PATH}
example$ export LD_LIBRARY_PATH
```

注 – *release* 是特定于编译器软件的每个发行版本。

LD_LIBRARY_PATH 具有通常使用冒号分隔的一个目录列表。运行 C++ 程序时，动态加载器搜索默认目录之前，会先搜索 LD_LIBRARY_PATH 中的目录。

使用以下示例所示的 ldd 命令，查看可执行文件中哪些库被动态链接：

```
example% ldd a.out
```

因为共享库很少移动，所以一般并不需要该步骤。

注 – 当使用 dlopen 打开共享库后，要使异常工作，必须使用 RTLD_GLOBAL。

关于使用共享库的更多信息，请参见 《链接程序和库指南》。

12.7 替换 C++ 标准库

替换与编译器一起发布的标准库是有风险的，不能保证产生预期的结果。基本操作是禁用编译器提供的标准头文件和库，指定找到新的头文件和库（及库本身的名称）的目录。

编译器支持标准库的 STLport 实现。更多信息请参见第 13-13 页的第 13.3 节 "STLport"。

12.7.1 可以替换的内容

您可以替换大部分标准库及其关联的头文件。替换的库是 libCstd，关联的头文件列在下表中：

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

库的可替换部分由称为 "STL"，加上字符串类、iostream 类及它们的帮助程序类组成。因为这些类和头文件是相互依赖的，所以不能仅替换其中的一部分。如果需要替换任何一部分，就应该替换所有的头文件和所有的 libCstd。

12.7.2 不可以替换的内容

标准头文件 <exception>、<new> 和 <typeinfo> 与编译器本身和 libCrun 紧密相关，且不能可靠替换。库 libCrun 包含了大量编译器依赖的且不能替换的 "helper" 函数。

从 C （<stdlib.h>、<stdio.h>，和 <string.h> 等等）继承而来的 17 个标准头文件与 Solaris 操作环境和基本 Solaris 运行时库 libc 紧密相关，且不能可靠替换。这些头文件的 C++ 版本 （<cstdlib>, <cstdio>, <cstring> 等）和基本 C 版本紧密结合，不能可靠地替换。

12.7.3 安装替换库

要安装替换库，必须先决定替换头文件的位置和 libCstd 的替换库。为方便讨论，假定头文件放置在 /opt/mycstd/include 中，库放置在 /opt/mycstd/lib 中。假定库由 libmyCstd.a 调用。（库名最好以 "lib" 开头。）

12.7.4 使用替换库

对于每一次编译，使用 -I 选项指向安装头文件的位置。此外，使用 -library=no%Cstd 选项可防止查找编译器本身版本的 libCstd 头文件。例如：

```
example% CC -I/opt/mycstd/include -library=no%Cstd...  (编译)
```

编译期间， -library=no%Cstd 选项防止搜索编译器本身版本的这些头文件所在的目录。

对于每个程序或库链接，使用 -library=no%Cstd 选项防止查找编译器本身的 libCstd， -L 选项指向替换库所在目录， -l 选项指定替换库。示例：

```
example% CC -library=no%Cstd -L/opt/mycstd/lib -lmyCstd...（链接）
```

或者，可以直接使用库的完整路径名称，忽略使用 -L 和 -l 选项。例如：

```
example% CC -library=no%Cstd /opt/mycstd/lib/libmyCstd.a...（链接）
```

链接期间， -library=no%Cstd 选项防止链接编译器本身版本的 libCstd。

12.7.5 标准头文件实现

C 具有 17 个标准头文件（<stdio.h>、<string.h>、<stdlib.h> 和其他）。这些头文件作为 Solaris 操作环境的一部分，位于 /usr/include 目录。 C++ 具有这些相同的头文件，另外要求在全局名称空间和 std 名称空间中显示不同声明的名称。在 Solaris 操作环境版本 8 之前的版本中， C++ 编译器提供了本身版本的这些头文件来替换 /usr/include 目录中的头文件。

C++ 也具有每个 C 标准头文件 （<cstdio>、 <cstring>、 <cstdlib> 和其他）的第二个版本，不同声明的名称仅显示在名称空间 std 中。最后， C++ 增加了 32 个自己的标准头文件 （<string>、 <utility>、 <iostream> 和其他）。

标准头文件的明显实现将 C++ 源码中找到的名称用作包括的文本文件的名称。例如，标准头文件 <string>（或 <string.h>）在某些目录中引用名为 string（或 string.h）的文件。这种明显实现有以下缺点：

- 如果没有文件名后缀，则不能搜索头文件或创建头文件的 makefile 规则。
- 如果具有名为 string 的目录或可执行程序，那么会错误地找到该目录或程序，而不是找到标准头文件。
- 在 Solaris 8 操作环境以前的 Solaris 操作环境上，启用 .KEEP_STATE 时，makefile 的默认相关性会导致尝试将标准头文件替换为可执行程序。（没有后缀的文件默认假定为要生成的程序。）

为了解决这些问题，编译器 include 目录包含了与头文件具有相同名称的文件，还有到具有唯一后缀 .SUNWCCh 的符号链接（SUNW 是所有编译器相关软件包的前缀， CC 是 C++ 编译器， h 是头文件的通用后缀）。指定 <string> 后，编译器将其重写到 <string.SUNWCCh> 并搜索该名称。后缀名只会在编译器自己的 include 目录中才能找到。如果这样找到的文件是符号链接 （正常情况下），编译器就会去掉链接，并将结果（这里是 string）用作错误消息和调试器引用的文件名。忽略文件的相关性信息时，编译器使用带后缀的名称。

仅当出现在尖括号中且无需指定任何路径时， 17 种标准 C 头文件和 32 种标准 C++ 头文件的两种格式才会发生名称重写。如果使用引号来代替尖括号指定任何路径组件或其他某些头文件，就不会有重写发生。

下表说明了通常的情况。

表 12-3 头文件搜索示例

源码	编译器搜索	注释
`<string>`	`string.SUNWCCh`	C++ 字符串模板
`<cstring>`	`cstring.SUNWCCh`	C `string.h` 的 C++ 版本
`<string.h>`	`string.h.SUNWCCh`	C `string.h`
`<fcntl.h>`	`fcntl.h`	不是标准 C 或 C++ 头文件
`"string"`	`string`	双引号，不是尖括号
`<../string>`	`../string`	指定的路径

如果编译器未找到 *header*`.SUNWCCh`，那么编译器将会重新搜索以查找 `#include` 指令提供的名称。例如，如果给定指令 `#include <string>`，编译器就会尝试找到名为 `string.SUNWCCh` 的文件。如果搜索失败，编译器就会查找名为 `string` 的文件。

12.7.5.1 替换标准 C++ 头文件

由于在第 12-13 页的第 12.7.5 节 “标准头文件实现”中描述的搜索算法，所以不必提供第 12-12 页的第 12.7.3 节“安装替换库”中所述替换头文件的 `SUNWCCh` 版本。但是会遇到某些上文所述的问题。如果是这样的话，建议为每个无后缀的头文件增加带有后缀 `.SUNWCCh` 的符号链接。也就是说，对于文件 `utility`，您可以运行命令

```
example% ln -s utility utility.SUNWCCh
```

当编译器第一次查找 `utility.SUNWCCh` 时，将会找到它，而不会和其他名为 `utility` 的文件或目录混淆。

12.7.5.2 替换标准 C 头文件

不支持替换标准 C 头文件。如果仍然希望提供标准头文件的自己的版本，那么建议按以下步骤操作：

- 将所有替换头文件放置在一个目录中。
- 在目录中创建每个替换头文件的 `.SUNWCCh` 符号链接。
- 通过在每次调用编译器时使用 `-I`，搜索包含替换头文件的目录。

例如，假定您具有 <stdio.h> 和 <cstdio> 的替换。请将文件 stdio.h 和 cstdio 放置在目录 /myproject/myhdr 中。在该目录中，运行如下命令：

```
example% ln -s stdio.h stdio.h.SUNWCCh
example% ln -s cstdio cstdio.SUNWCCh
```

每次编译时使用 -I/myproject/mydir 选项。

警告：

- 如果要替换任何 C 头文件，就必须成对替换。例如，如果要替换 <time.h>，就也必须替换 <ctime>。
- 替换头文件必须与被替换版本具有相同的效果。这就是说，不同的运行时库（如 libCrun、libC、libCstd、libc 和 librwtool）使用标准头文件中的定义来生成。如果替换文件不匹配，那么程序不能工作。

第 13 章

使用 C++ 标准库

当在默认（标准）模式下编译时，编译器可以访问 C++ 标准指定的整个库。库组件包括了标准模板库 (STL) 和以下组件。

- 字符串类
- 数字类
- 标准版本的流 I/O 类
- 基本内存分配
- 异常类
- 运行时类型信息

术语 STL 没有正式的定义，但是通常理解为包括容器、迭代器以及算法。以下标准库头的子集可以认为包含了 STL。

- `<algorithm>`
- `<deque>`
- `<iterator>`
- `<list>`
- `<map>`
- `<memory>`
- `<queue>`
- `<set>`
- `<stack>`
- `<utility>`
- `<vector>`

C++ 标准库（`libCstd`）基于版本 2 的 RogueWave™ 标准 C++ 库。该库仅在编译器的默认模式 (`-compat=5`) 下可用，使用 `-compat[=4]` 选项时不支持该库。

C++ 编译器也支持 STLport 的标准库实现版本 4.5.3。`libCstd` 仍然是默认库，但 STLport 的产品只是可选的。更多信息请参见第 13-13 页的第 13.3 节 "STLport"。

如果需要使用自己的 C++ 标准库版本代替编译器提供的版本，则可以通过指定 `-library=no%Cstd` 选项来实现。替换与编译器一起发布的标准库是有风险的，不能保证产生预期的结果。有关更多信息请参见第 12-11 页的第 12.7 节 “替换 C++ 标准库”。

关于标准库的详细信息，请参见 《标准 C++ 库用户指南》和 《标准 C++ 类库参考》。本书前面的“阅读本书之前”第 xxxii 页的“访问 Sun Studio 文档”一节中包含了访问该文档的信息。关于 C++ 标准库可用书籍的列表，请参见“阅读本书之前”中的第 xxxv 页的 “其他公司出版的书籍”一节。

13.1 C++ 标准库头文件

表 13-1 列出了完整标准库的头文件以及每个头的简要描述。

表 13-1 C++ 标准库头文件

头文件	描述
<algorithm>	操作容器的标准算法
<bitset>	位的固定大小序列
<complex>	数字类型表示复数
<deque>	支持在端点增加和删除的序列
<exception>	预定义异常类
<fstream>	文件的流 I/O
<functional>	函数对象
<iomanip>	iostream 操纵器
<ios>	iostream 基类
<iosfwd>	iostream 类的前向声明
<iostream>	基本流 I/O 功能
<istream>	输入 I/O 流
<iterator>	遍历序列的类
<limits>	数字类型的属性
<list>	排序的序列
<locale>	国际化支持
<map>	带有键/值对的关联容器
<memory>	专用内存分配器
<new>	基本内存分配和释放
<numeric>	通用的数字操作
<ostream>	输出 I/O 流

表 13-1 C++ 标准库头文件 (续)

头文件	描述
<queue>	支持在头部增加和在尾部删除的序列
<set>	有唯一键值的关联容器
<sstream>	将内存中的字符串用为源或接收器的流 I/O
<stack>	支持在头部增加和删除的序列
<stdexcept>	附加的标准异常类
<streambuf>	iostream 的缓冲区类
<string>	字符序列
<typeinfo>	运行时类型标识
<utility>	比较操作符
<valarray>	用于数字编程的值数组
<vector>	支持随机访问的序列

13.2 C++ 标准库手册页

表 13-2 列出了标准库中每个组件的可用文档。

表 13-2 C++ 标准库手册页

手册页	概述
Algorithms	在容器和序列上执行各种操作的通用算法
Associative_Containers	排序的容器
Bidirectional_Iterators	可以读取和写入，并且可以双向遍历容器的迭代器
Containers	标准模板库 (STL) 集合
Forward_Iterators	可以读取和写入的前移式迭代器
Function_Objects	定义了 operator() 的对象
Heap_Operations	查看 make_heap、pop_heap、push_heap 和 sort_heap 条目
Input_Iterators	只读的前移式迭代器
Insert_Iterators	允许迭代器向容器插入元素而非覆盖容器内元素的迭代器适配器
Iterators	集合遍历和修改的指针泛化。

表 **13-2** C++ 标准库手册页 (续)

手册页	概述
Negators	用于遍历谓词函数对象检测的函数适配器和函数对象
Operators	C++ 标准模板库输出的运算符
Output_Iterators	只写的前移式迭代器
Predicates	返回布尔 (true/false) 值或整型值的函数或函数对象
Random_Access_Iterators	可以读取、写入并随机访问容器的迭代器
Sequences	组织序列集合的容器
Stream_Iterators	包括允许直接在流上使用通用算法的 ostreams 和 istreams 的迭代器功能
__distance_type	决定迭代器所用距离的类型 （已废弃）
__iterator_category	决定迭代器所属的种类 （已废弃）
__reverse_bi_iterator	向后遍历集合的迭代器
accumulate	将一定范围内的所有元素聚集到单个值中
adjacent_difference	输出在一定范围内每个相邻元素对之间的差别的序列
adjacent_find	寻找在序列中相等的第一个相邻元素对
advance	按特定的距离将迭代器向前或者向后移动 （如可能）
allocator	在标准库容器中用于存储管理的默认分配器对象
auto_ptr	一个简单、智能的指针类
back_insert_iterator	用于在集合末端插入项目的插入迭代器
back_inserter	用于在集合末端插入项目的插入迭代器
basic_filebuf	将输入序列或输出序列与文件关联的类
basic_fstream	支持对命名文件的读取和写入，或者与文件描述符关联的设备的读取和写入
basic_ifstream	支持从命名文件读取或者从其他与文件描述符关联的设备读取
basic_ios	一个包含所有流都需要的通用函数的基类
basic_iostream	帮助格式化或者翻译由流缓冲区控制的字符序列
basic_istream	帮助读取或者翻译由流缓冲区控制的序列输入
basic_istringstream	支持从内存中的数组读取 basic_string<charT,traits,Allocator> 类对象
basic_ofstream	支持写入命名文件或者其他与文件描述符关联的设备
basic_ostream	帮助格式化或者写入由流缓冲区控制的序列输出

表 13-2　C++ 标准库手册页 (续)

手册页	概述
basic_ostringstream	支持写入类对象 basic_string<charT,traits,Allocator>
basic_streambuf	用于派生便于字符序列控制的各种流缓冲区的抽象基类
basic_string	处理类似字符实体的模板化类
basic_stringbuf	将输入或者输出序列与任意字符序列关联
basic_stringstream	支持读取或写入 basic_string<charT,traits,Allocator> 类的对象
binary_function	创建二元函数对象的基类
binary_negate	返回二元谓词结果补码的函数对象
binary_search	对容器中的值执行二元搜索
bind1st	用于将值绑定到函数对象的模板化实用程序
bind2nd	用于将值绑定到函数对象的模板化实用程序
binder1st	用于将值绑定到函数对象的模板化实用程序
binder2nd	用于将值绑定到函数对象的模板化实用程序
bitset	用于存储和处理固定大小位序列的模板类和相关函数
cerr	控制对未缓冲的流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stderr 对象关联
char_traits	一个具有类型和用于 basic_string 容器和 iostream 类的运算的特性类
cin	控制从流缓冲区的输入，该缓冲区与 <cstdio> 中声明的 stdin 对象关联
clog	控制对流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stderr 对象关联
codecvt	代码转换侧面
codecvt_byname	一个包含以命名语言环境为基础的代码集转换分类工具的侧面
collate	一个字符串检验、比较和散列侧面
collate_byname	一个字符串检验、比较和散列侧面
compare	返回真或假的二元函数或函数对象
complex	C++ 复数库
copy	复制一定范围内的元素
copy_backward	复制一定范围内的元素

表 13-2 C++ 标准库手册页 (续)

手册页	概述
count	计算容器中满足给定条件的元素的数量
count_if	计算容器中满足给定条件的元素的数量
cout	控制对流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stdout 对象关联
ctype	包括字符分类工具的侧面
ctype_byname	一个包含以命名语言环境为基础的字符分类工具的侧面
deque	一个支持随机访问迭代器并支持在开始和结束位置进行高效插入/删除的序列
distance	计算两个迭代器之间的距离
divides	返回用第一个参数除以第二个参数所得到的结果
equal	比较等式的两个范围
equal_range	在集合中找到最大的子范围，可在该范围中插入一个给定值而无需违反集合排序
equal_to	如果第一个参数与第二个参数相等就返回真的二元函数对象
exception	一个支持逻辑和运行时错误的类
facets	用于封装语言环境功能分类的类系列
filebuf	将输入序列或输出序列与文件关联的类
fill	用给定值初始化一个范围
fill_n	用给定值初始化一个范围
find	在序列中寻找出现的值
find_end	在序列中寻找上次出现的子序列
find_first_of	在序列中寻找在另一个序列中第一次出现的值
find_if	在满足特定谓词的序列中寻找第一次出现的值
for_each	将函数应用于范围内的每个元素
fpos	保持 iostream 类的位置信息
front_insert_iterator	用于在集合起始端插入条目的插入迭代器
front_inserter	用于在集合起始端插入条目的插入迭代器
fstream	支持对命名文件的读取和写入，或者与文件描述符关联的设备的读取和写入
generate	初始化一个具有由值产生器类产生的值的容器
generate_n	初始化一个具有由值产生器类产生的值的容器

表 13-2　C++ 标准库手册页 (续)

手册页	概述
get_temporary_buffer	基于指针的基元，用于处理内存
greater	如果第一个参数比第二个参数大就返回真的二元函数对象
greater_equal	如果第一个参数大于或等于第二个参数就返回真的二元函数对象
gslice	用于表示数组的通用片的数字数组类
gslice_array	用于表示 valarray 的类 BLAS 片的数字数组类
has_facet	用于确定语言环境是否具有给定侧面的函数模板
ifstream	支持从命名文件读取或者从其他与文件描述符关联的设备读取
includes	已排序序列的一系列基本操作
indirect_array	用于表示从 valarray 中所选元素的数字数组类
inner_product	计算两个范围 A 和 B 的内积 A X B
inplace_merge	将两个已排序的序列合并成为一个
insert_iterator	用于将项目插入集合而非覆盖集合的插入迭代器
inserter	用于将项目插入集合而非覆盖集合的插入迭代器
ios	一个包含所有流都需要的通用函数的基类
ios_base	定义成员类型并维护从它继承的类的数据
iosfwd	声明输入/输出库模板类并使之专用于宽字符和微型字符
isalnum	确定字符是字母还是数字
isalpha	确定字符是否为字母
iscntrl	确定字符是否为控制字符
isdigit	确定字符是否为十进制数字
isgraph	确定字符是否为图形字符
islower	确定字符是否为小写形式
isprint	确定字符是否可打印
ispunct	确定字符是否为标点符号
isspace	确定字符是否为空格
istream	帮助读取或者翻译由流缓冲区控制的序列输入
istream_iterator	具有 istreams 迭代器功能的流迭代器
istreambuf_iterator	从流缓冲区读取为其构造的连续字符

表 13-2 C++ 标准库手册页 (续)

手册页	概述
istringstream	支持读取内存中数组的 basic_string<charT,traits,Allocator> 类对象
istrstream	从内存中的数组读取字符
isupper	确定字符是否为大写形式
isxdigit	确定字符是否为十六进制数字
iter_swap	交换两个位置的值
iterator	基本的迭代器类
iterator_traits	返回有关迭代器的基本信息
less	如果第一个参数比第二个参数小就返回真的二元函数对象
less_equal	如果第一个参数小于或等于第二个参数就返回真的二元函数对象
lexicographical_compare	按照字典编排顺序来比较两个范围
limits	参考 numeric_limits
list	支持双向迭代器的序列
locale	包含多态侧面集的本地化类
logical_and	如果两个参数都为真就返回真的二元函数对象
logical_not	如果参数是假就返回真的一元函数对象
logical_or	如果两个参数有一个为真就返回真的二元函数
lower_bound	确定在已排序容器中元素的第一个有效位置
make_heap	创建堆
map	用唯一关键字访问非关键字值的关联容器
mask_array	给出了 valarray 的屏蔽视图的数字数组类
max	查找并返回一对值中的最大值
max_element	查找一个范围中的最大值
mem_fun	与指向成员函数的指针相匹配的函数对象，替代全局函数
mem_fun1	与指向成员函数的指针相匹配的函数对象，替代全局函数
mem_fun_ref	与指向成员函数的指针相匹配的函数对象，替代全局函数
mem_fun_ref1	与指向成员函数的指针相匹配的函数对象，替代全局函数
merge	将两个已排序的序列合并为第三个序列
messages	消息传送侧面

表 13-2　C++ 标准库手册页 (续)

手册页	概述
messages_byname	消息传送侧面
min	查找并返回一对值中的最小值
min_element	查找一个范围中的最小值
minus	返回用第一个参数减去第二个参数所得到的结果
mismatch	比较来自两个序列的元素并且返回首次出现的两个不匹配元素
modulus	返回第一个参数除以第二个参数所得到的余数
money_get	输入的货币格式
money_put	输出的货币格式
moneypunct	货币标点格式
moneypunct_byname	货币标点格式
multimap	用关键字访问非关键字值的关联容器
multiplies	用于返回第一个参数与第二个参数相乘结果的二元函数对象。
multiset	允许快速访问已保存关键字值的关联容器
negate	返回其参数负值的一元函数对象
next_permutation	生成以排序函数为基础的序列的连续置换
not1	对一元谓词函数对象进行求反操作的函数适配器
not2	对一元谓词函数对象进行求反操作的函数适配器
not_equal_to	如果第一个参数与第二个参数不相等就返回 true 的二元函数对象
nth_element	对集合重新排序，以便使在已排序顺序中比第 *n* 个元素小的排到第 *n* 个前面，比第 *n* 个元素大的就排到它后面。
num_get	输入的数字格式
num_put	输出的数字格式
numeric_limits	表示标量类型信息的类
numpunct	数字标点格式
numpunct_byname	数字标点格式
ofstream	支持写入命名文件或者其他与文件描述符关联的设备
ostream	帮助格式化或者写入由流缓冲区控制的序列输出
ostream_iterator	流迭代器允许使用具有 ostreams 和 istreams 的迭代器
ostreambuf_iterator	向从其构造的流缓冲区对象写入连续的字符

表 13-2 C++ 标准库手册页 (续)

手册页	概述
ostringstream	支持写入类对象 basic_string<charT,traits,Allocator>
ostrstream	写入一个在内存中的数组
pair	异类值对的模板
partial_sort	对实体集合排序的模板化算法
partial_sort_copy	对实体集合排序的模板化算法
partial_sum	计算一组值的连续部分的和
partition	将所有满足给定谓词的实体放置在不满足给定谓词的实体后面
permutation	生成以排序函数为基础的序列的连续置换
plus	用于返回第一个参数与第二个参数相加结果的二元函数对象
pointer_to_binary_function	与指向二元函数的指针相匹配的函数对象，替代 binary_function
pointer_to_unary_function	与指向函数的指针相匹配的函数对象类，替代 unary_function
pop_heap	从堆中移出最大的元素
prev_permutation	生成以排序函数为基础的序列的连续置换
priority_queue	像优先队列一样运行的容器适配器
ptr_fun	一个与指向某函数的指针对应的过载函数，替换一个函数
push_heap	将一个新元素放入堆
queue	像队列一样运行的容器适配器 （先入先出）
random_shuffle	集合的随机混洗元素
raw_storage_iterator	使基于迭代器的算法能够将结果存入尚未初始化的内存中
remove	将所需元素移动到容器的前端，并返回一个说明所需元素序列的结束位置的迭代器
remove_copy	将所需元素移动到容器的前端，并返回一个说明所需元素序列的结束位置的迭代器
remove_copy_if	将所需元素移动到容器的前端，并返回一个说明所需元素序列的结束位置的迭代器
remove_if	将所需元素移动到容器的前端，并返回一个说明所需元素序列的结束位置的迭代器
replace	用新值替换集合中的元素

表 13-2　C++ 标准库手册页 (续)

手册页	概述
replace_copy	用新值替换集合中的元素，并将修改过的序列移入结果
replace_copy_if	用新值替换集合中的元素，并将修改过的序列移入结果
replace_if	用新值替换集合中的元素
return_temporary_buffer	基于指针的基元，用于处理内存
reverse	反转集合中元素的顺序
reverse_copy	将集合中元素复制到新集合时反转它们的顺序
reverse_iterator	向后遍历集合的迭代器
rotate	将包含第一个元素到第 middle-1 个元素的部分与包含从中间到最后元素的部分交换
rotate_copy	将包含第一个元素到第 middle-1 个元素的部分与包含从中间到最后元素的部分交换
search	在值（这些值在元素状态时与标明范围内的值相等）的序列中查找子序列
search_n	在值（这些值在元素状态时与标明范围内的值相等）的序列中查找子序列
set	支持唯一关键字的关联容器
set_difference	构建已排序差集的基本设置操作
set_intersection	构建已排序交集的基本设置操作
set_symmetric_difference	构建已排序对称差集的基本设置操作
set_union	构建已排序并集的基本设置操作
slice	表示数组的类 BLAS 片的数字数组类
slice_array	用于表示 valarray 的类 BLAS 片的数字数组类
smanip	用于实现参数化控制器的帮助程序类
smanip_fill	用于实现参数化控制器的帮助程序类
sort	对实体集合排序的模板化算法
sort_heap	将堆转换为已排序的集合
stable_partition	在保持每组中元素的相对顺序的同时，将所有满足给定谓词的实体放在所有不满足给定谓词的实体之前
stable_sort	对实体集合排序的模板化算法
stack	像堆栈一样运行的容器适配器 （后入先出）
streambuf	用于派生便于字符序列控制的各种流缓冲区的抽象基类
string	basic_string<char、 char_traits<char>、allocator<char>> 的 typedef

表 **13-2** C++ 标准库手册页 (续)

手册页	概述
stringbuf	将输入或者输出序列与任意字符序列关联
stringstream	支持读取或写入 basic_string<charT,traits,Allocator> 类对象，位置是内存中的数组
strstream	在内存中读取或者写入一个数组
strstreambuf	将输入序列或者输出序列与微型字符数组（其元素存储任意值）关联
swap	交换值
swap_ranges	将一个位置的值与在其他位置的值交换
time_get	输入的时间格式
time_get_byname	输入的时间格式，以命名语言环境为基础
time_put	输出的时间格式
time_put_byname	输出的时间格式，以命名语言环境为基础
tolower	将字符转换为小写形式
toupper	将字符转换为大写形式
transform	将操作应用到集合中的一系列值并且存储结果
unary_function	创建一元函数对象的基类
unary_negate	返回一元谓词结果补码的函数对象
uninitialized_copy	使用构造从一个范围向另一个位置复制值的算法
uninitialized_fill	使用了在集合中设置值的构造算法的算法
uninitialized_fill_n	使用了在集合中设置值的构造算法的算法
unique	从一个值范围移除连续的重复值并将得到的唯一值放入结果
unique_copy	从一个值范围移除连续的重复值并将得到的唯一值放入结果
upper_bound	确定已排序容器中值的最后一个有效位置
use_facet	用于获取侧面的模板函数
valarray	用于数字操作的优化数组类
vector	支持随机访问迭代器的序列
wcerr	控制对未缓冲的流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stderr 对象关联
wcin	控制从流缓冲区的输入，该缓冲区与 <cstdio> 中声明的 stdin 对象关联

表 13-2　C++ 标准库手册页 (续)

手册页	概述
wclog	控制对流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stderr 对象关联
wcout	控制对流缓冲区的输出，该缓冲区与 <cstdio> 中声明的 stdout 对象关联
wfilebuf	将输入序列或输出序列与文件关联的类
wfstream	支持对命名文件的读取和写入，或者与文件描述符关联的设备的读取和写入
wifstream	支持从命名文件读取或者从其他与文件描述符关联的设备读取
wios	一个包含所有流都需要的通用函数的基类
wistream	帮助读取或者翻译由流缓冲区控制的序列输入
wistringstream	支持读取内存中数组的 basic_string<charT,traits,Allocator> 类对象
wofstream	支持写入命名文件或者其他与文件描述符关联的设备
wostream	帮助格式化或者写入由流缓冲区控制的序列输出
wostringstream	支持写入类对象 basic_string<charT,traits,Allocator>
wstreambuf	用于派生便于字符序列控制的各种流缓冲区的抽象基类
wstring	basic_string<wchar_t、char_traits<wchar_t>、allocator<wchar_t>> 的 typedef
wstringbuf	将输入或者输出序列与任意字符序列关联

13.3 STLport

如果要使用 libCstd 的另一个标准库，请使用标准库的 STLport 实现。您可以使用以下编译器选项关闭 libCstd 并使用替代的 STLport 库：

- -library=stlport4

更多信息请参见第 A-41 页的第 A.2.48 节 "-library=l[,l...]"。

本发行版本包括称为 libstlport.a 的静态归档和称为 libstlport.so 的动态库。

决定是否使用 STLport 实现之前，请先考虑以下信息：

- STLport 是开放源代码产品，并不能保证不同发行版本之间的兼容性。也就是说，使用后续版本的 STLport 编译，可能导致使用 STLport 4.5.3 编译的应用程序中断。使用后续版本的 STLport 编译的二进制文件可能无法与使用 STLport 4.5.3 编译的二进制文件链接。
- `stlport4`、`Cstd` 和 `iostream` 库提供了自己的 I/O 流的实现。使用 `-library` 指定其中的多个库，会导致不确定的程序行为。
- 编译器的后续发行版本可能不包括 STLport4，只包括更新版本的 STLport。编译器选项 `-library=stlport4` 在后续发行版本中可能不能使用，但是可以使用引用更新 STLport 版本的选项来替换该选项。
- Tools.h++ 不支持 STLport。
- STLport 和默认的 `libCstd` 是二进制级不兼容的。如果使用标准库的 STLport 实现，那么您必须使用选项 `-library=stlport4` 编译和链接包括第三方库的所有文件。这意味着不可以同时使用 STLport 实现和 C++ 区间数学库 `libCsunimath`。原因是 `libCsunimath` 使用了默认库头文件编译，而不用 STLport 来编译。
- 如果决定使用 STLport 实现，那么一定要包括代码隐式引用的头文件。允许标准头文件，但不必要包括另一个标准头文件作为实现的一部分。
- 如果使用 `-compat=4` 编译，那么不能使用 STLport 实现。

13.3.1 重新分发和支持的 STLport 库

请参见依照“最终用户对象代码许可协议”的条款重新分发的可执行文件或库的库和对象文件列表的运行时库自述文件。此自述文件的 C++ 节列出该编译器发行版支持的 STLport .so 版本。此自述文件作为已安装产品的一部分。要查看此自述文件的 HTML 版本，请将浏览器指向默认安装目录：

```
file:/opt/SUNWspro/docs/zh/index.html
```

注 – 如果产品软件没有安装在默认目录中，请通过系统管理员获取系统中的相应路径。

因为以下测试示例中的代码将库实现假定为不可移植，所以在该测试示例中不能使用 STLport 编译。特别的一点是，假定 <vector> 或 <iostream> 自动包括了 <iterator>，不过该假定是无效的。

```
#include <vector>
#include <iostream>

using namespace std;

int main()
{
    vector <int> v1 (10);
    vector <int> v3 (v1.size());
    for (int i = 0; i < v1.size (); i++)
      {v1[i] = i; v3[i] = i;}
    vector <int> v2(v1.size ());
    copy_backward (v1.begin (), v1.end (), v2.end ());
    ostream_iterator<int> iter (cout, " ");
    copy (v2.begin (), v2.end (), iter);
    cout << endl;
    return 0;
}
```

要解决此问题，请将 <iterator> 包括在源码中。

第 14 章

使用传统 iostream 库

与 C 类似，C++ 没有内建输入或输出语句。相反，I/O 工具是由库提供的。C++ 编译器提供了 iostream 类的传统实现和 ISO 标准实现。

- 在兼容模式 (-compat[=4]) 下，传统 iostream 类包含在 libC 中。
- 在标准模式 （默认模式）下，传统 iostream 类包含在 libiostream 中。如果具有使用传统 iostream 类的源代码而且要在标准模式下编译它，就请使用 libiostream。要在标准模式下使用传统 iostream 工具，请包括 iostream.h 头文件并使用 -library=iostream 选项进行编译。
- 包含在 C++ 标准库 libCstd 中的标准 iostream 类只能在标准模式下使用。

本章介绍了传统 iostream 库并提供了使用该库的示例。本章未提供对 iostream 库的完整描述。更多详细信息，请参见 iostream 库手册页。要访问传统 iostream 的手册页，请键入：

```
example% man -s 3CC4 name
```

14.1 预定义的 iostream

有四个预定义的 iostream：

- cin，连接到标准输入
- cout，连接到标准输出
- cerr，连接到标准错误
- clog，连接到标准错误

除了 cerr 之外，所有预定义的 iostream 是完全缓冲的。请参见第 14-3 页的第 14.3.1 节“使用 iostream 进行输出”和第 14-6 页的第 14.3.2 节“使用 iostream 进行输入”。

14.2 iostream 交互的基本结构

通过包括 iostream 库，程序可以使用任意数量的输入和输出流。每个流都具有某些源或接收器，如下所示：

- 标准输入
- 标准输出
- 标准错误
- 文件
- 字符数组

流可以被限定到输入或输出，或同时具有输入和输出。iostream 库使用两个处理层来实现这些流。

- 较低层实现了序列，即字符的简单流。这些序列由 streambuf 类或其派生的类实现。
- 较高层对序列执行格式化操作。这些格式化操作由 istream 和 ostream 类实现，而这些类具有从类 streambuf 派生的类型的对象（作为成员）。附加类 iostream 用于执行输入和输出的流。

标准输入、输出和错误由从类 istream 或 ostream 派生的特殊类对象处理。

分别从 istream、ostream 和 iostream 派生的 ifstream、ofstream 和 fstream 类处理文件的输入和输出。

分别从 istream、 ostream 和 iostream 派生的 istrstream、 ostrstream 和 strstream 类处理字符数组的输入和输出。

打开输入或输出流时，请创建其中一种类型的对象，并将流的 streambuf 成员与设备或文件关联。通常用流构造函数执行该关联，因此不能直接处理 streambuf。iostream 库为标准输入、标准输出和错误输出预定义流对象，因此您不必为这些流创建自己的对象。

您可以使用运算符或 iostream 成员函数将数据插入流 （输出）或从流 （输入）提取数据，并且控制插入或提取数据的格式。

如果要插入或提取新的数据类型 （其中一个类），通常需要重载插入和提取运算符。

14.3 使用传统 iostream 库

要使用传统 iostream 库的例程，就必须包括所需库部分的头文件。下表对头文件进行了具体描述。

表 14-1 iostream 例程头文件

头文件	描述
iostream.h	声明 iostream 库的基本功能。
fstream.h	声明文件专用的 iostream 和 streambuf。其中包括 iostream.h。
strstream.h	声明字符数组专用的 iostream 和 streambuf。其中包括 iostream.h。
iomanip.h	声明控制器：插入 iostream 或从中提取的值具有不同的作用。其中包括 iostream.h。
stdiostream.h	（废弃的）声明专用于 stdio FILE 的 iostream 和 streambuf，其中包括 iostream.h。
stream.h	（废弃的）包括 iostream.h、 fstream.h、 iomanip.h 和 stdiostream.h。用于兼容 C++ 1.2 版的旧式流。

通常程序中不需要所有这些头文件，而仅包括所需声明的头文件。在兼容模式 (-compat[=4]) 下，传统 iostream 库是 libC 的一部分，并由 CC 驱动程序自动链接。在标准模式（默认）下， libiostream 包含传统 iostream 库。

14.3.1 使用 iostream 进行输出

使用 iostream 进行输出通常依赖于 iostream 上下文中重载的 左移运算符 (<<)，该运算符称为插入运算符。如果要将值输出到标准输出，就要将该值插入预定义的输出流 cout 中。例如，要将给定值 someValue 发送到标准输出，可以使用以下语句：

```
cout << someValue;
```

插入运算符的重载用于所有内建类型，并且 someValue 表示的值转换为该值正确的输出表示。例如，如果 someValue 是 float 值， << 运算符就会将该值转换为具有小数位数字的正确序列。在输出流上插入 float 值的位置，<< 称为浮点插入器。通常在给定类型 X 的情况下，<< 称为 X 插入器。ios(3CC4) 手册页中讨论了输出的格式以及如何控制输出格式。

iostream 库不支持用户定义的类型。如果您定义了要以自己的方式输出的类型，就必须定义可以正确处理这些类型的插入器 （即，重载 << 运算符）。

<< operator 可以重复应用。要在 cout 上插入两个值，可以使用以下示例中所示的语句：

```
cout << someValue << anotherValue;
```

以上示例的输出在两个值间不显示空格。因此您可能会要按以下方式编写编码：

```
cout << someValue << " " << anotherValue;
```

<< 运算符具有左移运算符（其内建含义）的优先级。对于其他运算符，您可以始终用括号来指定操作的顺序。使用括号来避免优先级问题是个很好的方法。下列四个语句中，前两个是等价的，但后两个不是。

```
cout << a+b;  // + 比 << 具有更高的优先级
cout << (a+b);
cout << (a&y);// << 的优先级比 & 高
cout << a&y;  // 可能是错误：(cout << a) & y
```

14.3.1.1 定义自己的插入运算符

以下示例定义了 string 类：

```
#include <stdlib.h>
#include <iostream.h>


class string {
private:
       char* data;
       size_t size;

public:
       // （这里的函数不相关）

       friend ostream& operator<<(ostream&, const string&);
       friend istream& operator>>(istream&, string&);
};
```

在这种情况下，必须将插入运算符和提取运算符定义为友元，因为 string 类的数据部分是 private。

```
ostream& operator<< (ostream& ostr, const string& output)
{       return ostr << output.data;}
```

以下是与 string 一起使用的重载 operator<< 的定义。

```
cout << string1 << string2;
```

operator<< 将 ostream& （即，对 ostream 的引用）作为它的第一个参数并返回相同的 ostream，这样就有可能将插入合并到一个语句中。

14.3.1.2 处理输出错误

通常情况下，因为 iostream 库用于传送错误，所以重载 operator<< 时不需要检查错误。

错误发生时，发生错误所在的 iostream 进入错误状态。并根据错误的通用种类设置 iostream 状态中的各个位。iostream 定义的插入器忽略尝试将数据插入错误状态下的任何流，这种尝试不会更改 iostream 的状态。

通常，处理错误的推荐方法是定期检查某些中心位置中输出流的状态。如果有错误，就应该以某些方式来处理。本章假定您定义了具有一个字符串并终止程序的 error 函数。error 不是预定义的函数。关于 error 函数的示例请参见第 14-9 页的第 14.3.9 节 “处理输入错误”。您可以使用运算符 ! 来检查 iostream 的状态，该运算符在 iostream 为错误状态时，返回一个非零值。例如：

```
if (!cout) error( "output error");
```

还有另外一种方法来测试错误。ios 类定义了 operator void *()，因此发生错误时该类返回空指针。您可以使用如下语句：

```
if (cout << x) return ; // 如果成功，就返回
```

您还可以使用函数 good，它是 ios 的成员：

```
if ( cout.good() ) return ; // 如果成功，就返回
```

错误位在 enum 中声明：

```
enum io_state {goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80};
```

关于错误函数的详细信息，请参见 iostream 手册页。

14.3.1.3 刷新

对于多数 I/O 库，iostream 经常会聚集输出并将其发送到更大并且更有效的块中。如果您要刷新缓冲，那么只需简单地插入特殊值 flush。例如：

```
cout << "This needs to get out immediately." << flush ;
```

flush 是称为**控制器**的对象的示例。（**控制器**是一个值，它可以插入 iostream 中，这样做会有某些影响但不会引起值的输出。）它真正采用了 ostream& 或 istream& 参数并在执行完参数上的部分操作后返回它的参数。（请参见第 14-15 页的第 14.7 节“控制器”）

14.3.1.4 二进制输出

要以值的原始二进制形式获得输出，请使用下列示例所示的成员函数 write。该示例显示了 x 的原始二进制形式的输出。

```
cout.write((char*)&x, sizeof(x));
```

以上示例将 &x 转换为 char*，这违反了类型规程。这样做通常没有任何害处，但如果 x 的类型是具有指针的类、虚拟成员函数或需要特殊的构造函数操作，就无法正确重复以上示例所写入的值。

14.3.2 使用 iostream 进行输入

使用 iostream 进行输入与进行输出相类似。可以使用提取运算符 >>，按插入所用的方法将提取连贯起来。例如：

```
cin >> a >> b ;
```

该语句从标准输入获得两个值。对其他重载运算符来讲，所用的提取器依赖于 a 和 b 的类型（如果 a 和 b 具有不同的类型，则使用两个不同的提取器。）ios(3CC4) 手册页中详细讨论了输入的格式以及如何控制输入格式。通常，前导空白字符 （空格、换行符、标签、换页等）被忽略。

14.3.3 定义自己的提取运算符

要输入新的类型时，如同重载输出的插入运算符，请重载输入的提取运算符。

类 string 定义了自己的提取运算符，如以下编码示例所示：

代码样例 14-1 string 提取运算符

```
istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, '\n');
    input = holder;
    return istr;
}
```

get 函数从输入流 istr 读取字符，并且在读取到 maxline-1 字符，或遇到新行或文件尾 （不管哪个先发生）之前，将字符存储在 holder 中。然后 holder 中的数据为空的终止。最后， holder 中的字符被复制到目标字符串。

按照约定，提取器转换第一个参数 （此处是 istream& istr）中的字符，然后将字符存储在第二个参数 （通常是引用），最后返回第一个参数。因为提取器会将输入值存储在第二个参数中，所以第二个参数必须是引用。

14.3.4 使用 char* 提取器

此处提及这个预定义的提取器是因为它可能产生问题。使用方法如下：

```
char x[50];
cin >> x;
```

该提取器跳过前导空白，在遇到另一空白字符之前提取字符并将这些字符复制到 x，最后完成具有终止空 (0) 字符的字符串。因为输入会溢出给定的数组，所以要小心操作。

您还必须确保指针指向了分配的存储。例如，下面列出了一个常见的错误：

```
char * p; // 未初始化
cin >> p;
```

因为没有告知存储输入数据的位置，所以会导致程序的终止。

14.3.5 读取任何单一字符

除了使用 char 提取器外，您还可以获得具有 get 成员函数任意一种形式的单一字符。例如：

```
char c ;
cin.get(c); // 如果输入失败，不更改 c

int b;
b = cin.get(); // 如果输入失败，将 b 设置为 EOF
```

注 – 与其他提取器不同的是 char 提取器不跳过前导空白。

以下方法可以只跳过空格，并在制表符、换行符或任何其他字符处停止：

```
int a;
do {
    a = cin.get();
   }
while(a =='');
```

14.3.6 二进制输入

如果需要读取二进制值 （如用成员函数 write 所写的值），那么您可以使用 read 成员函数。以下示例显示了如何使用 read 成员函数来输入 x 的原始二进制形式 （它是先前使用 write 的示例的反向操作）。

```
cin.read((char*)&x, sizeof(x));
```

14.3.7 查看输入

您可以使用 peek 成员函数来查看流中的下一个字符，而不必提取该字符。例如：

```
if (cin.peek() != c) return 0;
```

14.3.8 提取空白

默认情况下，iostream 提取器跳过前导空白。您可以关闭**跳过**标志以防止发生跳过。以下示例先从 cin 关闭了空白跳过，然后再重新打开：

```
cin.unsetf(ios::skipws); // 关闭空白跳过
...
cin.setf(ios::skipws); // 重新打开
```

无论跳过是否启用，都可以使用 iostream 控制器 ws 从 iostream 删除前导空白。以下示例显示了如何从 iostream istr 删除前导空白：

```
istr >> ws;
```

14.3.9 处理输入错误

按照约定，第一个参数为非零错误状态的提取器不能从输入流提取任何数据，且不能清除任何错误位。失败的提取器至少应该设置一个错误位。

对于输出错误，您应该定期检查错误状态，并在发现非零状态时采取某些操作（诸如终止）。！运算符用于测试 iostream 的错误状态。例如，如果输入字母字符用于输入，以下代码就会产生输入错误：

```
#include <stdlib.h>
#include <iostream.h>
void error (const char* message) {
        cerr << message << "\n";
        exit(1);
}
int main() {
        cout << "Enter some characters: ";
        int bad;
        cin >> bad;
        if (!cin) error("aborted due to input error");
        cout << "If you see this, not an error."<< "\n";
       return 0;
}
```

类 ios 具有用于错误处理的成员函数。详细信息请参见手册页。

14.3.10 使用具有 stdio 的 iostream

您可以与 C++ 程序一起使用 stdio，但是在程序内的相同标准流中混合 iostream 和 stdio 时会发生某些问题。例如，如果同时写入 stdout 和 cout，就会发生独立缓冲并产生未预料的结果。如果您既从 stdin 输入也从 cin 输入，问题就会更严重，因为独立缓冲会使输入成为垃圾。

要消除标准输入、标准输出和标准错误中的这种问题，就请在执行输入或输出前使用以下指令：它将所有预定义的 iostream 与相应预定义的 stdio FILE 连接起来。

```
ios::sync_with_stdio();
```

因为在预定义流作为连接的一部分成为无缓冲流时，性能会显著下降，所以该连接不是默认连接。可以在应用到不同文件的同一程序中同时使用 stdio 和 iostream。也就是说，您可以使用 stdio 例程写入到 stdout，也可以写入到附加到 iostream 的其他文件。只要不再尝试从 stdin 读取，就可以打开 stdio FILE 用于输入，还可以从 cin 读取。

14.4 创建 iostream

要读取或写入流而不是预定义的 iostream，就需要创建自己的 iostream。通常，这意味着创建 iostream 库中所定义的类型的对象。本节讨论了可用的各种类型：

14.4.1 处理使用类 fstream 的文件

处理文件与处理标准输入和标准输出相似，类 ifstream、ofstream 和 fstream 分别从类 istream、 ostream 和 iostream 派生。作为派生的类，它们继承了插入和提取运算符 （以及其他成员函数），还有与文件一起使用的成员和构造函数。

包括文件 fstream.h 以使用任何 fstream。如果您只想执行输入，请使用 ifstream，如果只进行输出，请使用 ofstream。如果要同时执行输入和输出流，那么请使用 fstream。将文件名称用作构造函数参数。

例如，将文件 thisFile 复制到文件 thatFile，如下例：

```
ifstream fromFile("thisFile");
if     (!fromFile)
       error("unable to open 'thisFile' for input");
ofstream toFile ("thatFile");
if     ( !toFile )
       error("unable to open 'thatFile' for output");
char c ;
while (toFile && fromFile.get(c)) toFile.put(c);
```

该代码：

- 创建了具有 ios::in 默认模式名为 fromFile 的 ifstream 对象，并将该对象连接到 thisFile。该对象打开 thisFile。
- 如果新 ifstream 对象处于失败状态，则检查它的错误状态并调用 error 函数（该函数必须在程序的其他位置定义）。
- 创建了具有 ios::out 默认模式名为 toFile 的 ofstream 对象，并将该对象连接到 thatFile。
- 按以上步骤检查 toFile 的错误状态。
- 创建了 char 变量用于该变量被传递时存放数据。
- 将 fromFile 的内容复制到 toFile，每次复制一个字符。

注 – 当然这种方法（每次复制一个字符）不适用于复制文件。该代码仅作为使用 fstream 的示例提供。您应该将与输入流关联的 streambuf 插入到输出流中。请参见第 14-19 页的第 14.10 节 "Streambuf" 和手册页 sbufpub(3CC4)。

14.4.1.1 打开模式

该模式由枚举类型 open_mode 的 or-ing 位构造，是类 ios 的公有类型，定义如下：

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
        nocreate=0x20, noreplace=0x40};
```

注 – UNIX 本身不需要 binary 标志，所以提供该标志的目的是与需要该标志的系统相兼容。可移植代码在打开二进制文件时要使用 binary 标志。

您可以打开文件同时用于输入和输出。例如，以下代码打开了同时用于输入和输出的文件 someName，同时将该文件绑定到 fstream 变量 inoutFile。

```
fstream inoutFile("someName", ios::in|ios::out);
```

14.4.1.2 在不指定文件的情况下声明 fstream

您不必指定文件就可以声明 fstream 并在稍后打开文件。例如，下面创建了 ofstream toFile 进行写入。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

14.4.1.3 打开和关闭文件

您可以关闭 fstream，然后使用另一文件打开它。例如，要在命令行上处理提供的文件列表：

```
ifstream infile;
for (char** f = &argv[1]; *f; ++f) {
   infile.open(*f, ios::in);
   ...;
   infile.close();
}
```

14.4.1.4 使用文件描述符打开文件

如果您了解诸如用于标准输出的整数 1 等文件描述符，就可以按如下方式打开：

```
ofstream outfile;
outfile.attach(1);
```

如果通过为 fstream 构造函数之一提供文件名称，或使用 open 函数来打开文件，那么该文件在 fstream 被销毁（被 delete 销毁或超出范围）时会自动关闭。当您将文件 attach 到 fstream 时，文件不会自动关闭。

14.4.1.5 在文件内重新定位

您可以在文件中改变读取和写入的位置。有多个工具可以达到这个目的。

- streampos 是一种可以在 iostream 中记录位置的类型。
- tellg (tellp) 是报告文件位置的 istream (ostream) 成员函数。因为 istream 和 ostream 是 fstream 的父类，所以 tellg 和 tellp 还可以作为 fstream 类的成员函数被调用。
- seekg (seekp) 是查找给定位置的 istream (ostream) 成员函数。
- seek_dir enum 为 seek 的使用指定相对位置。

```
enum seek_dir {beg=0, cur=1, end=2};
```

例如，给定 fstream aFile：

```
streampos original = aFile.tellp(); // 保存当前位置
aFile.seekp(0, ios::end); // 重新定位到文件尾
aFile << x;               // 向文件写入值
aFile.seekp(original);    // 返回原始位置
```

seekg (seekp) 可以使用一个或两个参数。当它具有两个参数时，第一个参数是与 seek_dir 值所标明的位置（也就是第二个参数）相对的位置。例如：

```
aFile.seekp(-10, ios::end);
```

从终点移动 10 个字节

```
aFile.seekp(10, ios::cur);
```

从当前位置向前移 10 个字节。

注 – 在文本流上进行任意寻找是不可移植的，但您总是可以返回到以前保存的 streampos 值。

14.5 iostream 的赋值

iostream 不允许将一个流赋值给另一个流。

复制流对象的问题是有两个可以独立更改的状态信息版本，例如输出文件中指向当前写入位置的指针。因此，某些问题会发生。

14.6 格式控制

手册页 ios(3CC4) 中详细讨论了格式控制。

14.7 控制器

控制器是可以插入到具有特殊作用的 iostream 中或从中提取的值。

参数化控制器是具有一个或多个参数的控制器。

因为控制器是普通的标识符，并用完了所有可能名称，所以 iostream 不为每个可能的函数定义控制器。本章的其他部分讨论了各种控制器和成员函数。

如表 14-2 所示，有 13 个预定义的控制器。使用该表时，要进行以下假设：

- i 具有类型 long。
- n 具有类型 int。
- c 具有类型 char。
- istr 为输入流。
- ostr 为输出流。

表 14-2 iostream 预定义的控制器

	预定义的控制器	描述
1	`ostr << dec, istr >> dec`	进行基于 10 的整型转换。
2	`ostr << endl`	插入换行符 ('\n') 并调用 `ostream::flush()`。
3	`ostr << ends`	插入空 (0) 字符。这在处理 strstream 时很有用。
4	`ostr << flush`	调用 `ostream::flush()`。
5	`ostr << hex, istr >> hex`	进行基于 16 的整型转换。
6	`ostr << oct, istr >> oct`	进行基于 8 的整型转换。
7	`istr >> ws`	找到非空白字符（留在 istr 中的字符）之前，提取空白字符 （跳过空白）。
8	`ostr << setbase(n), istr >> setbase(n)`	将转换基数设置为 n（限于 0、8、10 和 16）。
9	`ostr << setw(n), istr >> setw(n)`	调用 `ios::width(n)`。将字段宽度设置为 n。
10	`ostr << resetiosflags(i), istr >> resetiosflags(i)`	根据 i 中的位设置，清除标志位向量。

表 **14-2** iostream 预定义的控制器 (续)

	预定义的控制器	描述
11	`ostr << setiosflags(i), istr >> setiosflags(i)`	根据 i 中的位设置，设置标志位向量。
12	`ostr << setfill(c), istr >> setfill(c)`	将填充字符 （用于填充字段）设置为 c。
13	`ostr << setprecision(n), istr >> setprecision(n)`	将浮点精度设置为 n 个数字。

要使用预定义的控制器，就必须在程序中包括文件 iomanip.h。

您可以定义自己的控制器。控制器共有两个基本类型：

- 无格式控制器，使用 istream&、ostream& 或 ios& 参数操作流并返回控制器的参数。
- 参数化控制器，使用 istream&、ostream& 或 ios& 参数及一个附加变量（参数）来操作流并返回该控制器的流参数。

14.7.1 使用无格式控制器

无格式控制器是具有如下功能的函数：

- 执行到流的引用
- 以某种方式操作流
- 返回控制器的参数

由于移位运算符使用了为 iostream 预定义的函数 （指向它的指针），所以可以将函数放入输入或输出运算符的序列。移位运算符调用函数，而不尝试读取或写入值。将 tab 插入到 ostream 中 tab 控制器的示例如下所示：

```
ostream& tab(ostream& os) {
               return os <<'\t';
              }
...
cout << x << tab << y;
```

详细描述实现以下操作的方法：

```
const char tab = '\t';
...
cout << x << tab << y;
```

下面示例显示了无法用简单常量来实现的代码。假设您要打开和关闭输入流的空白跳过。那么您可以分别调用 ios::setf 和 ios::unsetf 打开和关闭 skipws 标志，或定义两个控制器。

```
#include <iostream.h>
#include <iomanip.h>
istream& skipon(istream &is) {
       is.setf(ios::skipws, ios::skipws);
       return is;
}
istream& skipoff(istream& is) {
       is.unsetf(ios::skipws);
       return is;
}
...
int main()
{
      int x,y;
      cin >> skipon >> x >> skipoff >> y;
      return 1;
}
```

14.7.2 参数化控制器

包括在 iomanip.h 中的其中一个参数化控制器名为 setfill。setfill 设置用于填写字段宽度的字符。该操作按照下例所示实现：

```
// 文件 setfill.cc
#include <iostream.h>
#include <iomanip.h>

// 私有控制器
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
// 公共 applicator
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

参数化控制器的实现分为两部分：

- **控制器**。它使用一个额外的参数。在前一个编码示例中，它使用了额外的 int 参数。由于未给这个控制器函数定义移位运算符，所以您无法将它放至输入或输出操作序列中。相反，您必须使用辅助函数 applicator。
- *applicator*。它调用该控制器。 applicator 是全局函数，您会为它生成在头文件中可用的原型。通常控制器是文件中的静态函数，该文件包含了 applicator 的源代码。只有 applicator 可以调用该控制器，如果您将控制器设置为静态，就要使控制器名称始终位于全局地址空间之外。

在头文件 iomanip.h 中定义了多个类。每个类都保存一个控制器函数的地址和一个参数的值。手册页 manip(3CC4) 中描述了 iomanip 类。前一个示例使用了处理 ios 的 smanip_int 类。因为该类和 ios 一起使用，所以也可以和 istream 与 ostream 一起使用。前一示例还使用了类型为 int 的第二个参数。

applicator 创建并返回类对象。在前一编码示例中，类对象是 smanip_int，其中包含了 applicator 的控制器和 int 参数。 iomanip.h 头文件定义了该类的移位运算符。当 applicator 函数 setfill 出现在输入或输出操作的序列中时，applicator 函数被调用并返回类。移位运算符作用于该类，以调用具有参数值 （存储在类中）的控制器函数。

在下例中，控制器 print_hex:

- 将输出流设置成十六进制模式。
- 将 long 值插入到流中。
- 恢复流的转换模式。

使用类 omanip_long 的原因是该编码示例仅用于输出，而且代码操作的是 long 而不是 int:

```
#include <iostream.h>
#include <iomanip.h>
static ostream& xfield(ostream& os, long v) {
        long save = os.setf(ios::hex, ios::basefield);
        os << v;
        os.setf(save, ios::basefield);
        return os;
    }
omanip_long print_hex(long v) {
       return omanip_long(xfield, v);
   }
```

14.8 Strstream: 数组的 iostream

请参见 strstream(3CC4) 手册页。

14.9 Stdiobuf：stdio 文件的 iostream

请参见 stdiobuf(3CC4) 手册页。

14.10 Streambuf

iostream 是两部分（输入或输出）系统的格式部分。系统的其他部分由 streambuf 组成，处理无格式字符流的输入和输出。

一般情况下可以通过 iostream 来使用 streambuf，因此不必担心 streambuf 的细节。例如，要提高效率，避免错误处理或格式化内建的 iostream 时，可以直接使用 streambuf。

14.10.1 和 Streambuf 一起使用

streambuf 由流或字符序列和一个或两个指向该序列的指针组成。每个指针都指向两个字符间。（实际上，指针无法指向字符间，但可以按这种方式考虑指针。）streambuf 指针共有两个类型：

- *put* 指针，指向下一字符要存储的位置之前
- *get* 指针，指向下一个要获取的字符之前

streambuf 可以具有这两个指针中的一个或全部。

14.10.1.1 指针位置

可以使用多种方法来操作指针的位置和序列的内容。操作时两个指针是否会移动依赖于所使用 streambuf 的种类。通常情况下，对于类似队列的 streambuf， get 和 put 指针可以独立移动；而对于类似文件的 streambuf， get 和 put 指针始终一起移动。 strstream 是类似队列流的一个例子；而 fstream 是类似文件流的例子。

14.10.2 使用 Streambuf

您从未创建过实际的 streambuf 对象，只创建了类 streambuf 所派生的类的对象。具体示例是 filebuf 和 strstreambuf，手册页 filebuf(3CC4) 和 ssbuf(3) 对其进行了分别的描述。高级用户可能想从 streambuf 派生自己的类，这样就可以为指定设备提供接口或基本缓冲以外的内容。手册页 sbufpub(3CC4) 和 sbufprot(3CC4) 讨论了如何完成该操作。

除创建自己的特殊种类的 streambuf 外，您可能还想通过访问与 iostream 相关的 streambuf 来访问公用成员函数（如上引用的手册页所述）。此外，每个 iostream 都具有使用 streambuf 指针的预定义的插入器和提取器。插入或提取 streambuf 时，整个流被复制。

这里还列出了之前所讨论的另一种执行文件复制的方法，其中省略了清晰的错误检查：

```
ifstream fromFile("thisFile");
ofstream toFile ("thatFile");
toFile << fromFile.rdbuf();
```

按照前面所述的方式打开输入和输出文件。每个 iostream 类都具有成员函数 rdbuf，该函数将指针返回到与其相关的 streambuf 对象。对于 fstream 的情况， streambuf 对象是类型 filebuf。与 fromFile 相关的整个文件被复制到（插入到）与 toFile 相关的文件。最后一行也可以改写为：

```
fromFile >> toFile.rdbuf();
```

然后源文件被提取到目标中。两种方法是完全等同的。

14.11 iostream 手册页

许多 C++ 手册页阐述了 iostream 库的详细内容。下表概述了每个手册页中的内容。

要访问传统 iostream 库手册页，请键入：

```
example% man -s 3CC4 name
```

表 14-3　iostream 手册页概述

手册页	概述
filebuf	详细说明类 filebuf 的公用接口，该类从 streambuf 派生并专用于文件。关于从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 和 sbufprot(3CC4) 手册页。通过类 fstream 使用 filebuf 类。
fstream	详细说明类 ifstream、ofstream 和 fstream 的专用成员函数，该类用于使用文件的 istream、ostream 和 iostream 的专用版本。
ios	详细说明类 ios 的各个部分，该类作为 iostream 的基类。该类也包含了所有流公共的状态数据。
ios.intro	介绍并概述了 iostream。
istream	详细说明了以下内容： • 类 istream 的成员函数，这些函数支持从 streambuf 获取的字符解释。 • 输入格式化 • 定位描述为部分 ostream 类的函数。 • 某些相关函数 • 相关控制器
manip	描述 iostream 库中定义的输入和输出控制器。
ostream	详细说明了以下内容： • 类 ostream 的成员函数，这些函数支持写入到 streambuf 的字符解释。 • 输出格式化 • 定位描述为部分 ostream 类的函数 • 某些相关函数 • 相关控制器
sbufprot	描述了为从类 streambuf 派生的类编码的编程人员所需的接口。因为部分公用函数未在 sbufprot(3CC4) 手册页中进行讨论，所以另请参阅 sbufpub(3CC4) 手册页。
sbufpub	详细说明类 streambuf 的公用接口，尤其是 streambuf 的公用成员函数。该手册页包含了直接处理 streambuf 类对象所需的信息，或是找到从 streambuf 派生的类继承的函数所需的信息。如果您要从 streambuf 派生类，另请参阅 sbufprot(3CC4) 手册页。
ssbuf	详细说明类 strstreambuf 的专用的公用接口，该类从 streambuf 派生并专用于处理字符数组。关于从类 streambuf 继承的函数的详细信息，请参见 sbufpub(3CC4) 手册页。
stdiobuf	包含了对类 stdiobuf 的最基本描述，该类从 streambuf 派生并专用于处理 stdio FILE。关于从类 streambuf 继承的函数详细信息，请参见 sbufpub(3CC4) 手册页。
strstream	详细说明 strstream 的专用成员函数，这些函数由 iostream 所派生类的集合实现并专用于处理字符数组。

14.12 iostream 术语

iostream 库描述了通常使用的术语，与一般编程的术语相类似，但却具有专门的含义。下表定义了这些术语，与讨论 iostream 库时的用法相同。

表 14-4 iostream 术语

iostream 术语	定义
缓冲	该词具有两个含义，一个专用于 iostream 软件包，而另一个通常适用于输入和输出。 当特指 iostream 库时，缓冲是由类 streambuf 定义的类型的对象。 通常，缓冲是一个内存块，用于将字符高效传输到输出的输入。对于已缓冲的 I/O，缓冲已满或被强制刷新之前，字符的实际传输会延迟。 无缓冲的缓冲是指在其中没有以上所定义的通用缓冲的 streambuf。本章避免使用引用 streambuf 的术语缓冲。但手册页和其他 C++ 文档还用术语缓冲来表示 streambuf。
提取	从 iostream 获得输入的过程。
Fstream	专用于文件的输入或输出流。特指以 courier 字体打印时类 iostream 所派生的类。
插入	发送输出到 iostream 中的过程。
iostream	通常为输入或输出流。
iostream 库	该库的实现需要包括 iostream.h、 fstream.h、 strstream.h、 iomanip.h 和 stdiostream.h。因为 iostream 是面向对象的库，所以您应该扩展该库。因此，处理 iostream 库的某些工作还未实现。
流	通常情况下指 iostream、 fstream、 strstream 或用户定义的流。
Streambuf	包含字符序列的缓冲，其中字符具有 put 或 get 指针（或兼有）。以 courier 字体打印时，它表示特定类。否则，通常指类 streambuf 或从 streambuf 派生的类的对象。任何流对象包含了 streambuf 所派生类型的一个对象或指向对象的指针。
Strstream	专用于使用字符数组的 iostream。是指以 courier 字体打印时的特定类。

第 15 章

使用复数运算库

复数是由**实**部和**虚**部组成的数字。例如：

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

在简并条件下，`0 + 3i` 完全是虚数，通常写为 `3i`，而 `5 + 0i` 完全是实数，通常写为 `5`。您可以使用 `complex` 数据类型来表示复数。

注 – 复数运算库 (`libcomplex`) 只用于兼容模式 (`-compat[=4]`)。在标准模式 （默认模式）中， C++ 标准库 `libCstd` 包括具有简单功能的复数类。

15.1 复数库

复数运算库将复数数据类型实现为新的数据类型并提供：

- 运算符
- 数学库函数 （为内建数字类型定义）
- 扩展 （用于允许复数输入和输出的 iostream）
- 错误处理机制

复数还可以表示为**绝对值** （或**幅度**）和**参数** （或**角度**）。该库提供了在实部虚部 （笛卡尔）表示和幅度角度 （极性）表示间进行转换的函数。

数字**复共轭**的虚部中符号相反。

15.1.1 使用复数库

要使用复数库，请将头文件 complex.h 包括在程序中，然后编译并用 -library=complex 选项进行链接。

15.2 类型 complex

复数运算库定义了一个类：complex 类。类 complex 的对象可以容纳一个单一复数。复数被构造为两部分：

- 实部
- 虚部

```
class complex {
    double re, im;
};
```

类 complex 的对象值是一对 double 值。第一个值表示实部，第二个值表示虚部。

15.2.1 类 complex 的构造函数

complex 有两个构造函数。它们的定义是：

```
complex::complex(){re=0.0; im=0.0;}
complex::complex(double r, double i = 0.0) {re=r; im=i;}
```

如果不通过指定参数来声明复数变量，那么会使用第一个构造函数且变量被初始化，这时复数变量的两部分都为 0。下列示例创建了一个其实部和虚部均为 0 的复数变量。

```
complex aComp;
```

您可以给定一个或两个参数。无论是以上哪种情况，都将使用第二个构造函数。只给定一个参数时，视为实部和虚部的值的参数被设置为 0。例如：

```
complex aComp(4.533);
```

用下列值创建一个复数变量：

```
4.533 + 0i
```

如果给定了两个值，第一个值被视为实部的值，而第二个值被视为虚部的值。例如：

```
complex aComp(8.999, 2.333);
```

用下列值创建一个复数变量：

```
8.999 + 2.333i
```

您还可以使用复数运算库所提供的 polar 函数来创建复数 （请参见第 15-4 页第 15.3 节的 “数学函数”）。 polar 函数根据给定的极性坐标幅度和角度创建复数值。

complex 类型没有析构函数。

15.2.2 算法运算符

复数运算库定义了所有基本的算法运算符。具体来说，以下运算符按一般方法和普通的优先级工作：

```
+   -   /   *   =
```

减运算符 (-) 具有其通常的二元和一元含义。

此外，您可以按通常的方法使用以下运算符：

- 加法赋值运算符 (+=)
- 减法赋值运算符 (-=)
- 乘法赋值运算符 (*=)
- 除法赋值运算符 (/=)

但是，若将以上四个运算符用于表达式，则不产生任何值。例如，下列表达式无法进行运算：

```
complex a, b;
...
if ((a+=2)==0) {...}; // 非法
b = a *= b; // 非法
```

您还可以按照常规含义使用等号 (==) 和不等号 (!=) 。

将运算表达式中的实数和复数混合时，C++ 使用复数运算符函数并将实数转换为复数。

15.3 数学函数

复数运算库提供了许多数学函数。一些是专用于复数的，而其余的则是标准 C 数学库中函数的复数版本。

全部这些函数为每个可能的参数产生结果。如果函数无法产生数学上可接受的结果，它就调用 `complex_error` 并返回一些适用的值。特别的一点是，函数会尽量避免实际溢出，而改为调用具有消息的 `complex_error`。下表描述了复数运算库函数的提示。

注 – `sqrt` 和 `atan2` 函数的实现是按照 C99 csqrt 附录 G 规范进行的。

表 15-1 复数运算库函数

复数运算库函数	描述
`double abs(const complex)`	返回复数的幅度。
`double arg(const complex)`	返回复数的角度。
`complex conj(const complex)`	返回其参数的复共轭。
`double imag(const complex&)`	返回复数的虚部。
`double norm(const complex)`	返回其参数幅度的平方。较 `abs` 快，但更容易产生溢出。用于比较幅度。
`complex polar(double mag, double ang=0.0)`	执行一对表示复数幅度和角度的极性坐标，并返回对应的复数。
`double real(const complex&)`	返回复数的实部。

表 15-2 复数数学函数和三角函数

复数运算库函数	描述
`complex acos(const complex)`	返回余弦为其参数的角度。
`complex asin(const complex)`	返回正弦为其参数的角度。
`complex atan(const complex)`	返回正切为其参数的角度。
`complex cos(const complex)`	返回其参数的余切。
`complex cosh(const complex)`	返回其参数的双曲余弦。
`complex exp(const complex)`	计算 `e**x`，其中 `e` 为自然对数的基础， `x` 为 `exp` 的给定参数。
`complex log(const complex)`	返回其参数的自然对数。
`complex log10(const complex)`	返回其参数的常用对数。

表 15-2　复数数学函数和三角函数 (续)

复数运算库函数	描述

`complex pow(double b, const complex exp)`
`complex pow(const complex b, int exp)`
`complex pow(const complex b, double exp)`
`complex pow(const complex b, const complex exp)`
	使用两个参数：`pow(`*b, exp*`)`。这就使 *b* 具有了 *exp* 的幂。

`complex sin(const complex)`	返回其参数的正弦。
`complex sinh(const complex)`	返回其参数的双曲正弦。
`complex sqrt(const complex)`	返回其参数的平方根。
`complex tan(const complex)`	返回其参数的正切。
`complex tanh(const complex)`	返回其参数的双曲正切。

15.4　错误处理

复数库具有以下对错误处理的定义：

```
extern int errno;
class c_exception {...};
int complex_error(c_exception&);
```

外部变量 `errno` 是 C 库中的全局错误状态。`errno` 可以执行标准头 `errno.h` 中所列的值 （请参见手册页 `perror`(3)）。没有任何函数将 `errno` 设置为零，但是有许多函数会将它设置为其他值。

要分辨特定运算是否失败：

1. **请在运算前将 `errno` 设置为零。**
2. **测试运算。**

函数 `complex_error` 引用 `c_exception` 的类型并被下列复数运算库函数调用：

- `exp`
- `log`
- `log10`
- **双曲正弦**
- **双曲余弦**

complex_error 的默认版本返回零值。这个零值的返回意味着发生了默认的错误处理。您可以提供自己的执行其他错误处理的替换函数 complex_error。手册页 cplxerr(3CC4) 中描述了错误处理。

手册页 cplxtrig(3CC4) 和 cplxexp(3CC4) 描述了默认的错误处理。在下表中也对其进行了总结。

表 15-3 复数运算库函数的默认错误处理

复数运算库函数	默认错误处理汇总
exp	如果发生溢出，请将 errno 设置为 ERANGE 并返回一个极大的复数。
log, log10	如果参数为零，请将 errno 设置为 EDOM 并返回一个极大的复数。
sinh, cosh	如果参数的虚部产生溢出，则返回一个零复数。如果实部产生溢出，则返回一个极大的复数。无论是以上哪种情况，都将 errno 设置为 ERANGE。

15.5 输入和输出

如下列示例所示，复数运算库为复数成员提供了默认的**提取程序**和**插入程序**。

```
ostream& operator<<(ostream&, const complex&); // 插入程序
istream& operator>>(istream&, complex&); // 提取程序
```

关于提取程序和插入程序的基本信息，请参见第 14-2 页第 14.2 节的 “iostream 交互的基本结构”和第 14-3 页第 14.3.1 节的 “使用 iostream 进行输出”。

对输入来讲，复数提取程序 >> 从输入流提取一对数值 （在圆括号中由逗号分隔）并将它们读入一个复数对象。第一个值被视为实部的值，而第二个值被视为虚部的值。例如，给定的声明和输入语句：

```
complex x;
cin >> x;
```

以及输入 (3.45, 5)， x 的值与 3.45 + 5.0i 的值相等。对插入程序来讲反向为真。给定的 complex x(3.45, 5)， cout<<x 输出 (3.45, 5)。

输入通常由括号中的一对数值 （由逗号分隔）组成，也可选择空格。如果您提供一个单一数值 （具有或不具有括号和空格），那么提取程序会将数值的虚部设置为零。不要将符号 i 包括在输入文本中。

插入程序会插入括号中实部和虚部的值 （由逗号分隔）。它不包括符号 i。这两个值作为 double 处理。

15.6 混合模式运算

类型 complex 在设计上适用于混合模式表达式中的内建运算类型。运算类型默认转换为类型 complex，而且算法运算符和多数数学函数都有 complex 版本。例如：

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

其中表达式 b+i 是混合模式。通过构造函数 complex::complex(double,double=0) 整型 i 被转换为类型 complex，该整型首先是被转换为类型 double。结果是被 double y 除，所以 y 也被转换为 complex 并使用了复数除法运算。商是类型 complex，所以会调用复数正弦例程，产生另一 complex 结果，等等。

但是，并非所有的数学运算符和转换都是暗示的 （即使定义）。例如从数学角度，复数未较好排序，只能比较等式。

```
complex a, b;
a == b; // 正确
a != b; // 正确
a <  b; // 错误：操作符 < 不能应用于类型 complex
a >= b; // 错误：操作符 >= 不能应用于类型 complex
```

类似的，由于未明确定义概念，所以无法将类型 complex 自动转换为其他类型。您可以指定是否需要实部、虚部或幅度。

```
complex a;
double f(double);
f(abs(a)); // 正确
f(a);      // 错误：f(complex) 不匹配
```

15.7 效率

complex 类的设计主要侧重于效率。

最简单的函数声明为 inline，用于消除函数调用的开销。

在函数不同时就会提供函数的多个开销版本。例如，pow 函数的版本使用了类型 double， int 和 complex 的指数，因为前者的计算更简单。

当您包含 complex.h 时，也会自动包含标准 C 数学库的头文件 math.h。然后 C++ 开销规则就会产生类似于下面的表达式效率评估：

```
double x ;
complex x = sqrt(x);
```

在该示例中，调用了标准数学函数 sqrt(double)，且将结果转换为类型 complex，而不是先转换为类型 complex 再调用 sqrt(complex)。该结果转向重载解决规则的外部，正好是您所希望的结果。

15.8 复数手册页

复数运算库的剩余文档由下表所列的手册页组成：

表 15-4 类型 complex 的手册页

手册页	概述
cplx.intro(3CC4)	对复数运算库的一般性介绍
cartpol(3CC4)	笛卡尔函数和极性函数
cplxerr(3CC4)	错误处理函数
cplxexp(3CC4)	指数、对数和平方根函数
cplxops(3CC4)	算法运算符函数
cplxtrig(3CC4)	三角函数

第 16 章

生成库

本章解释了如何生成您自己的库。

16.1 了解库

库具有两点好处。首先，它们提供了在多个应用程序间共享代码的方法。如果您有要共享的代码，则可以创建一个具有该代码的库，并将该库链接到需要这些代码的应用程序。其次，库提供了降低大型应用程序复杂性的方法。这类应用程序可以将相对独立的部分生成为库并进行维护，因此减轻编程人员在其他部分工作的负担。

简单来说，生成一个库意味着创建 `.o` 文件 （通过使用 `-c` 选项编译代码）并使用 `CC` 命令将 `.o` 文件组合到库中。可以生成两种库，静态 （归档）库和动态 （共享）库。

对于静态 （归档）库，在链接时库中的对象会链接到程序的可执行文件中。只有库中这些应用程序所需的 `.o` 文件被链接到可执行文件。静态（归档）库的名称通常以 `.a` 后缀结束。

对于动态 （共享）库，库中的对象不会链接到程序的可执行文件，而是由链接程序告知可执行文件，程序要取决于库。执行该程序时，系统会装入程序所需的动态库。如果使用同一动态库的两个程序同时执行，那么操作系统在程序间共享这个动态库。动态 （共享）库的名称以 `.so` 后缀结束。

动态链接共享库较静态链接归档库有多个优势：

- 可执行文件较小。
- 在运行时，代码的有效部分可在程序间共享，这样就可以降低内存使用量。
- 库可以在运行时替换，无需重新链接应用程序。（动态链接共享库的主要机制是使程序能够利用 Solaris 操作系统的多项改进的功能，而无需重新链接和分发程序。）
- 可以使用 `dlopen()` 函数调用在运行时装入共享库。

但动态库也具有一些缺点：

- 运行时链接有执行时间成本。
- 使用动态库进行程序的分发可能会要求同时分发该程序所使用的库。
- 将共享库移动到一个不同的位置就可以阻止系统查找该库并执行程序。（环境变量 LD_LIBRARY_PATH 可以帮助克服这个问题。）

16.2 生成静态（归档）库

生成静态（归档）库的机制与生成可执行文件相似。对象 (.o) 集合文件可以使用 CC 的 -xar 选项合并到一个单一的库中。

您可以直接使用 CC -xar 来生成静态（归档）库，而无需使用 ar 命令。C++ 语言通常要求编译器所维护的信息比传统 .o 文件所提供的信息更多，尤其是模板实例。-xar 选项确保所有必要信息都包括在库中，其中包括模板实例。由于 make 无法确定哪些是实际创建和引用的模板文件，所以在通常的编程环境下，您可能无法实现该操作。如果没有所需的 CC -xar，所引用的模板实例就可能不包括在库中。例如：

```
% CC -c foo.cc # 编译主文件，创建模板对象。
% CC -xar -o foo.a foo.o # 将所有对象收集到库中。
```

-xar 标志使 CC 创建一个静态（归档）库。-o 指令用于命名新创建的库。编译器检查命令行上的对象文件，交叉引用模板系统信息库中的对象文件，并将这些用户对象文件所需的模板（包括主对象文件本身）增加到归档中。

注 – 使用 -xar 标志来创建或更新现有归档。不要用它来维护归档。-xar 选项等同于 ar -cr。

最好在每个 .o 文件中只具有一个函数。如果您要链接归档，那么在需要该特定 .o 文件中的符号时，归档的整个 .o 文件都会被链接到应用程序。每个 .o 文件具有一个函数就可以确保只有这些应用程序所需的符号将与归档链接。

16.3 生成动态（共享）库

动态（共享）库的生成方法与静态（归档）库相同，不同之处是您在命令行上使用的是 -G 而不是 -xar。

您不应该直接使用 ld。使用静态库时，如果您使用的是模板， CC 命令就可以确保模板系统信息库中所有必要的模板实例都包括在库中。动态库中链接到应用程序的所有静态构造函数在执行 main() **之前**调用，所有静态析构函数在 main() 退出**之后**调用。如果使用 dlopen() 打开共享库，则所有的静态构造函数在 dlopen() 执行，而所有的静态析构函数在 dlclose() 执行。

您应该使用 CC -G 来生成动态库。当您使用 ld （链接编辑器）或 cc （C 编译器）来生成动态库时，异常可能不工作且库中所定义的全局变量不被初始化。

要生成动态（共享）库，就必须创建可重定位对象文件，做法是通过用 CC 的 -Kpic（或 -KPIC）选项来编译每个对象。然后您就可以生成一个具有这些可重定位对象文件的动态库。如果您遇到奇怪的链接失败，则可能忘记了用 -Kpic 或 -KPIC 编译部分对象。

要生成名为 libfoo.so 的 C++ 动态库 （该库包含源文件 lsrc1.cc 和 lsrc2.cc 的对象），请键入：

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

-G 选项指定动态库的结构。 -o 选项指定库的文件名。 -h 选项指定共享库的名称。 -Kpic 选项指定与位置无关的对象文件。

注 – CC -G 命令不将任何 -l 选项传递到 ld。如果您要使共享库具有对另一共享库的依存性，就必须在命令行上传递必需的 -l 选项。例如，如果您要使共享库具有对 libCrun.so 的依存性，就必须在命令行上传递 -lCrun。

16.4 生成包含异常的共享库

对于包含 C++ 代码的程序，永远不要使用 -Bsymbolic，而是要使用链接程序映射文件。使用 -Bsymbolic 时，在不同模块中的引用可以绑定到被假设为全局对象内容的不同副本中。

异常机制依赖对地址的比较。如果您具有某项内容的两个副本，它们的地址就不等同且异常机制可能失败，这是由于异常机制依赖对假设为唯一地址内容的比较。

使用 dlopen() 打开共享库时，必须将 RTLD_GLOBAL 用于异常才能工作。

16.5 生成专用的库

在组织生成一个仅供内部使用的库时，可以使用不建议在一般情况下使用的选项来生成这个库。具体来讲，库不需要兼容系统的应用程序二进制接口 (ABI)。例如，可以使用 `-fast` 选项来编译库，以改善该库在已知架构上的性能。同样地，可以使用 `-xregs=float` 选项来编译库以提高其性能。

16.6 生成公用的库

在组织生成一个供其他公司使用的库时，库的管理、平台的一般性以及其他问题就变得尤为重要。一个用于检验库是否为公用的简单测试就是询问应用程序编程人员是否可以轻松地重新编译该库。公用库的生成应该与系统的应用程序二进制接口 (ABI) 兼容。通常，这意味着应该避免任何特定于处理器的选项。（例如，不要使用 `-fast` 或 `-xtarget`。）

SPARC ABI 为应用程序保留了一些专用寄存器。对 V7 和 V8 来讲，这些寄存器是 `%g2`、`%g3` 和 `%g4`。而对 V9 来讲，这些寄存器是 `%g2` 和 `%g3`。由于多数编译用于应用程序，所以在默认情况下，为了提高程序的性能， C++ 编译器将这些寄存器作为临时寄存器使用。但是，对公用库中寄存器的使用通常不兼容于 SPARC ABI。当生成公用库时，请使用 `-xregs=no%appl` 选项编译所有对象，以确保未使用应用程序寄存器。

16.7 生成具有 C API 的库

如果要生成的库是用 C++ 编写的，且可与 C 程序共同使用，那么就必须创建 C API（应用程序编程接口）。要执行该操作，请先将所有已输出函数标记为 `extern "C"`。注意，只有在全局函数中才能够完成该操作，在成员函数中不行。

如果 C 接口的库需要 C++ 运行时的支持，同时您也用 cc 进行链接，则在使用该 C 接口的库时，还必须用 `libC`（兼容模式）或 `libCrun`（标准模式）链接应用程序。（如果 C 接口的库不需要 C++ 运行时支持，就没必要用 `libC` 或 `libCrun` 链接。）归档库与共享库的链接步骤是不同的。

提供**归档**的 C 接口库时，必须提供如何使用该库的相关指令。

- 如果 C 接口的库是使用**标准模式**下的 `CC`（默认）生成的，那么在使用该 C 接口的库时请将 `-lCrun` 增加到 `cc` 命令行。
- 如果 C 接口的库是使用**兼容模式**下的 `CC (-compat)` 生成的，那么在使用该 C 接口的库时请将 `-lC` 增加到 `cc` 命令行。

提供**共享**的 C 接口库时，必须在生成库的时候创建对 `libC` 或 `libCrun` 的依存。如果共享库具有正确的依存性，您就无需在使用该库时将 `-lC` 或 `-lCrun` 增加到命令行。

- 如果在**兼容模式**下 (`-compat`) 生成 C 接口库，那么在生成该库时请将 `-lC` 增加到 `CC` 命令行。
- 如果在**标准模式**下 （默认）生成 C 接口库，那么在生成该库时请将 `-lCrun` 增加到 `CC` 命令行。

如果您要删除对 C++ 运行库的任何依存性，就必须在库的源码中使用以下编码规则：

- 不要使用任何形式的 `new` 或 `delete`，除非您提供了自己的相应版本。
- 不要使用异常。
- 不要使用运行时类型信息 (RTTI)。

16.8 使用 dlopen 从 C 程序访问 C++ 库

要执行该操作，请在生成共享库时将 `-compat=4` 的 `-lC` 或 `-compat=5` 的 `-lCrun` 增加到命令行。例如：

```
example% CC -G -compat=4...-lC
example% CC -G -compat=5...-lCrun
```

如果共享库使用了异常且不具有对 C++ 运行库的依存性，则 C 程序可能会出现无规律的行为。

注 – 使用 `dlopen()` 打开共享库时， `RTLD_GLOBAL` 必须用于异常以便进行工作。

第 IV部分 附录

附录 A

C++ 编译器选项

本附录详细介绍了 C++ 编译器的命令行选项。描述的特性适用于除了说明以外的所有平台；基于 SPARC 系统的 Solaris 操作系统特有的特性标识为 *SPARC*，基于 x86 系统的 Solaris 操作系统特有的特性标识为 *x86*。

下表显示了典型选项语法格式的示例。

表 A-1 选项语法格式示例

语法格式	示例
`-option`	`-E`
`-option`*value*	`-I`*pathname*
`-option=`*value*	`-xunroll=4`
`-option` *value*	`-o` *filename*

本手册前面的 “阅读本书之前”中列出的印刷约定在这一节中用于描述单个选项。

圆括号、大括号、括号、管道字符和省略号是选项说明中使用的**元字符**，而不是选项自身的一部分。

A.1 选项信息的结构

为了帮助您查找信息，编译器选项说明被分为以下几个子节。如果一个选项被其他选项取代或与其他选项一致，就请参见其他选项的说明以获取完整的详细信息。

表 A-2 选项子节

子节	内容
选项定义	紧跟在每个选项之后的简短定义。（该类无标题。）
值	如果选项具有一个或多个值，则本节将定义每个值。
默认	如果选项具有主默认值或辅助默认值，则在此处进行声明。 如果未指定选项，则主默认值为有效选项值。例如，如果未指定 `-compat`，则默认值为 `-compat=5`。 如果指定了选项但不给定任何值，则辅助默认值为有效选项值。例如，如果 `-compat` 已指定但没有值，则默认值为 `-compat=4`。
扩展	如果选项具有宏扩展，则将在本节中显示。
示例	如果要举例说明选项，则在此处给出所需示例。
交互	如果选项与其他选项进行交互，则在此处讨论它们的关系。
警告	如果有对选项使用的提醒（例如可能产生不期望的行为的操作），则在此处说明。
另请参见	本节包含到其他选项或文档中更多的信息的引用。
“替换为”“与 ... 相同”	如果选项已废弃且已被其他选项替换，则在此处说明替换的选项。以后的发行版本可能不支持这种方式描述的选项。 如果有两个选项具有相同的含义和目的，则在此处引用首选项。例如，“与 `-xO` 相同”表示 `-xO` 为首选项。

A.2 选项参考

A.2.1 -386

x86：与 `-xtarget=386` 相同。**该选项仅用于向后兼容。**

A.2.2 -486

x86：与 -xtarget=486 相同。**该选项仅用于向后兼容。**

A.2.3 -a

与 -xa 相同。

A.2.4 -B*binding*

指定 用于链接的库绑定是 symbolic、dynamic （共享）还是 static （非共享）。

您可以在命令行上多次使用 -B 选项。该选项被传递到链接程序 ld。

注 – 很多系统库在 Solaris 64 位编译环境中只能作为动态库使用。因此，请勿将 -Bstatic 用作命令行的最后一个切换开关。

值

binding 必须是下列值之一：

binding 的值	含义
dynamic	指定链接编辑器查找 lib*lib*.so （共享）文件，如果未找到，则查找 lib*lib*.a （静态，非共享）文件。当链接需要共享库绑定时，请使用该选项。
static	指定链接编辑器只查找 lib*lib*.a （静态，非共享）文件。当链接需要非共享库绑定时，请使用该选项。
符号	如果可能，则强制在共享库中解析符号 （即使符号已经在别处定义）。请参见 ld(1) 手册页。

（在 -B 和 *binding* 值之间不允许有空格。）

默认

如果未指定 -B，则假定为 -Bdynamic。

交互

要静态链接 C++ 默认库，请使用 `-staticlib` 选项。

`-Bstatic` 和 `-Bdynamic` 选项会影响默认提供的库链接。要确保默认库是动态链接的，则最后所使用的 `-B` 应该是 `-Bdynamic`。

在 64 位环境下，许多系统库只可以用作共享动态库。其中包括 `libm.so` 和 `libc.so`（而不提供 `libm.a` 和 `libc.a`）。因此，在 64 位 Solaris 操作系统下， `-Bstatic` 和 `-dn` 可能会产生链接错误。这些情况下应用程序必须与动态库链接。

示例

以下编译器命令链接了 `lib`*foo*`.a`（即使 `lib`*foo*`.so` 存在），所有其他库都被动态链接:

```
example% CC a.o -Bstatic -lfoo -Bdynamic
```

警告

对于包含 C++ 代码的程序，永远不要使用 -Bsymbolic，而是要使用链接程序映射文件。

使用 -Bsymbolic 时，在不同模块中的引用可以绑定到被假设为全局对象的不同副本中。

异常机制依赖对地址的比较。如果您具有某项内容的两个副本，它们的地址就不等同且异常机制可能失败，这是由于异常机制依赖对假设为唯一地址内容的比较。

如果您使用 -B*binding* 选项以不同的步骤进行编译和链接，就必须在链接步骤中包括该选项。

另请参见

`-nolib`, `-staticlib`, `ld`(1)、第 12-9 页的第 12.5 节 “静态链接标准库”和《链接程序和库指南》

A.2.5 -c

仅编译；产生对象文件 `.o` ，但不进行链接。

该选项指定 `CC` 驱动程序禁止用 `ld` 进行链接，并为每个源文件生成一个 `.o` 文件。如果只在命令行上指定一个源文件，就可以用 `-o` 选项来显式命名对象文件。

示例

输入 **cc -c x.cc** 时，生成 x.o 对象文件。

而输入 **cc -c x.cc -o y.o** 时，则生成 y.o 对象文件。

警告

当编译器为输入文件 (.c, .i) 生成对象代码时，编译器总是在工作目录中生成 .o 文件。如果禁止链接步骤，则不会删除 .o 文件。

另请参见

-o *filename*、-xe

A.2.6 -cg{89|92}

与 -xcg{89|92} 相同。

A.2.7 -compat[={4|5}]

设置编译器的主发行版本兼容模式。该选项控制 __SUNPRO_CC_COMPAT 和 __cplusplus 宏。

C++ 编译器有两个主要模式。兼容模式接受 4.2 编译器所定义的 ARM 语义和语言。标准模式接受符合 ANSI/ISO 标准的构造。由于 ANSI/ISO 标准在名称粉碎、虚函数表布局和其他 ABI 详细信息中强制进行显著的不兼容的更改，所以这两个模式是互相不兼容的。这两个模式由以下值中所示的 -compat 选项进行区分。

值

-compat 选项可以具有以下值。

值	含义
-compat=4	（兼容模式）设置语言和二进制使其与 4.0.1、4.1 和 4.2 编译器兼容。将 __cplusplus 和 __SUNPRO_CC_COMPAT 预处理程序宏分别设置为 1 和 4。
-compat=5	（标准模式）设置语言和二进制使其与 ANSI/ISO 标准模式兼容。将 __cplusplus 和 __SUNPRO_CC_COMPAT 预处理程序宏分别设置为 199711L 和 5。

默认

如果未指定 -compat 选项，则假定为 -compat=5。

如果仅仅指定了 -compat，则假定为 -compat=4。

交互

您无法在兼容模式下 (-compat[=4]) 使用标准库。

不支持对具有下列任何选项的 -compat[=4] 的使用。

- -Bsymbolic
- -features=[no%]strictdestrorder
- -features=[no%]tmplife
- -library=[no%]iostream
- -library=[no%]Cstd
- -library=[no%]Crun
- -library=[no%]rwtools7_std
- -xarch=native64、-xarch=generic64、-xarch=v9、-xarch=v9a 或 -xarch=v9b

不支持对具有下列任何选项的 -compat=5 的使用。

- -Bsymbolic
- +e
- features=[no%]arraynew
- features=[no%]explicit
- features=[no%]namespace
- features=[no%]rtti
- library=[no%]complex
- library=[no%]libC
- -vdelx

警告

生成共享库时，不要使用 -B*symbolic*。

另请参见

《C++ 迁移指南》

A.2.8 +d

不扩展 C++ 内联函数。

按照 C++ 语言规则， C++ 内联函数是一个函数，对于该函数以下语句之一为真。

- 该函数的定义使用了 `inline` 关键字，
- 该函数在类定义内部定义 （不仅是声明）
- 该函数是编译器生成的类成员函数

按照 C++ 语言规则，编译器可以选择是否将调用实际内联到内联函数。C++ 编译器将调用内联到内联函数，除非：

- 函数过于复杂，
- 已选定 `+d` 选项，或者
- 已选定 `-g` 选项

示例

默认情况下，编译器可以在以下代码示例中内联函数 `f()` 和 `memf2()`。此外，该类具有编译器可以内联的由编译器生成的默认构造函数和析构函数。当您使用 `+d` 时，编译器不会内联 `f()` 和 `C::mf2()` （即构造函数和析构函数）。

```
inline int f() {return 0;} // 可以被内联
class C {
  int mf1(); // 不内联，除非稍后出现内联定义
  int mf2() {return 0;} // 可以被内联
};
```

交互

指定 `-g` 调试选项时，该选项自动打开。

`-g0` 调试选项不打开 `+d`。

`+d` 选项对使用 `-xO4` 或 `-xO5` 时所执行的自动内联无效。

另请参见

`-g0`、`-g`

A.2.9 -D[ ]*name*[=*def*]

将宏符号 *name* 定义到预处理程序。

使用该选项与在源码开始位置包括 `#define` 指令等效。您可以使用多个 `-D` 选项。

值

下表显示了预定义的宏。您可以在如 `#ifdef` 这样的预处理程序条件下使用这些值。

表 A-3 预定义的宏

类型	宏名称	说明
SPARC 和 x86	`__ARRAYNEW`	如果启用了操作符 `new` 和 `delete` 的“数组”形式，则定义 `__ARRAYNEW`。更多信息，请参见 `-features=[no%]arraynew`。
	`_BOOL`	如果类型 `bool` 已启用，则定义 `_BOOL`。有关更多信息，请参见 `-features=[no%]bool`。
	`__BUILTIN_VA_ARG_INCR`	用于 `varargs.h`、`stdarg.h` 和 `sys/varargs.h`. 中的关键字 `__builtin_alloca`、`__builtin_va_alist` 和 `__builtin_va_arg_incr`。
	`__cplusplus`	
	`__DATE__`	
	`__FILE__`	
	`__LINE__`	
	`__STDC__`	设置为 0（零）
	`__sun`	
	`sun`	请参见**交互**。
	`__SUNPRO_CC=0x580`	`__SUNPRO_CC` 的值表示编译器的发行版本编号
	`__SUNPRO_CC_COMPAT=4` **或** `__SUNPRO_CC_COMPAT=5`	请参见第A-5页的第A.2.7节 "`-compat[={4|5}]`"
(SPARC)	`__SUN_PREFETCH=1`	
	`__SVR4`	
	`__TIME__`	

表 **A-3** 预定义的宏 (续)

类型	宏名称	说明
	__' *uname –s*′_′*uname –-r*\'	在 -D__SunOS_5_7 和 -D__SunOS_5_8 中时， *uname -s* 是 uname -s 的输出，而 *uname -r* 是具有无效字符的 uname -r 的输出，如用下划线替换的句号 (.)。
	__unix	
	unix	请参见**交互**。
(SPARC)	__sparc	
	sparc	请参见**交互**。
(SPARC) v9	__sparcv9	只限于 64 位编译模式
x86	__i386	
	i386	请参见**交互**。
UNIX	_WCHAR_T	

如果您不使用 *=def*，则 *name* 被定义为 1。

交互

使用 +p，则不定义 sun、 unix、 sparc 和 i386。

另请参见

-U

A.2.10 -d{y|n}

允许或不允许整个可执行文件的动态库。

该选项被传递到 ld。

该选项只能在命令行出现一次。

值

值	含义
-dy	在链接编辑器中指定动态链接。
-dn	在链接编辑器中指定静态链接。

默认

如果未指定 -d 选项，则假定为 -dy。

交互

在 64 位环境下，许多系统库只可以用作共享动态库。其中包括 libm.so 和 libc.so（而不提供 libm.a 和 libc.a）。因此，在 64 位 Solaris 操作系统下， -Bstatic 和 -dn 可能会产生链接错误。这些情况下应用程序必须与动态库链接。

警告

如果将此选项与动态库结合使用，将导致致命错误。大多数系统库仅作为动态库时可用。

另请参见

ld(1)、《链接程序和库指南》

A.2.11 -dalign

-dalign 等价于 -xmemalign=8s。有关更多信息，请参见第 A-104 页的第 A.2.139 节 "-xmemalign=ab"。

警告

如果您用 -dalign 编译一个程序单元，就要使用 -dalign 来编辑程序的所有单元，否则会产生意外的结果。

A.2.12 -dryrun

显示但不编译驱动程序所生成的子命令。

该选项指示驱动程序 CC 显示但不执行编译驱动程序所构造的子命令。

A.2.13 -E

对源文件运行预处理程序但不进行编译。

指示 CC 驱动程序仅在 C++ 源文件上运行预处理程序，并将结果发送到 stdout（标准输出）。不执行任何编译，不生成任何 .o 文件。

该选项使预处理程序输入行号信息包括在输出中。

示例

该选项用于确定预处理程序所进行的更改。例如，以下程序 foo.cc 生成了代码样例 A-2 中所示的输出 .

代码样例 A-1 预处理程序示例程序 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main() {
  int x;
  x=power(2, 10);
}
```

代码样例 A-2 使用 -E 选项的 foo.cc 的预处理程序输出

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power ( int , int ) ;


int main() {
int x;
x = power ( 2 , 10 ) ;
}
```

警告

使用模板时，从该选项的输出不支持作为 C++ 编译器的输入。

另请参见

`-P`

A.2.14 +e{0|1}

在兼容模式 (`-compat[=4]`) 下控制虚拟表的生成。在标准模式下 （默认模式）无效并被忽略。

值

`+e` 选项可以具有以下值。

值	含义
0	禁止虚拟表的生成并创建对所需虚拟表的外部引用。
1	为所有定义的具有虚函数的类创建虚拟表。

交互

使用该选项进行编译时，也可以使用 `-features=no%except` 选项。否则，编译器会为用于异常处理的内部类型生成虚拟表。

如果模板类具有虚函数，就可能无法确保编译器生成全部所需的虚拟表而不复制这些表。

另请参见

《C++ 迁移指南》

A.2.15 -erroff[=*t*]

该命令禁止出现 C++ 编译器警告消息，而不会影响错误消息。该选项适用于所有警告消息，无论 `-errwarn` 是否指定这些消息以导致非零退出状态。

值

t 是一个以逗号分隔的列表，该列表包括以下各项中的一个或多个：*tag*、no%*tag*、%all、%none。顺序是很重要的；例如 %all,no%*tag* 禁止了除 *tag* 以外的所有警告消息。下表列出了 -erroff 值：

表 A-4 -erroff 值

值	含义
tag	禁止由该 *tag* 指定的警告消息。使用 -errtags=yes 选项可以显示消息的标记。
no%*tag*	启用由该 *tag* 指定的警告消息。
%all	禁止所有警告消息。
%none	启用所有警告消息 （默认）。

默认

默认为 -erroff=%none。指定 -erroff 与指定 -erroff=%all 相同。

示例

例如， -erroff=*tag* 会禁止由该标记指定的警告消息。另一方面， -erroff=%all，no%*tag* 会禁止除 *tag* 指定的消息之外的所有警告消息。

使用 -errtags=yes 选项可以显示警告消息的标记。

警告

使用 -erroff 选项只可以禁止来自 C++ 编译器前端的警告消息，该警告消息会在使用 -errtags 选项时显示一个标记。

另请参见

-errtags、-errwarn

A.2.16 -errtags[=*a*]

显示 C++ 编译器前端每个警告消息的消息标记，使用 -erroff 选项可以禁止警告消息，而使用 -errwarn 选项则可能产生致命的错误。

值和默认

a 可以是 yes 或 no。默认为 -errtags=no。指定 -errtags 与指定 -errtags=yes 相同。

警告

来自 C++ 编译器驱动程序和编译系统其他组件的消息没有错误标记，因此无法使用 -erroff 来禁止，使用 -errwarn 也不会产生致命错误。

另请参见

-erroff、-errwarn

A.2.17 -errwarn[=*t*]

利用 -errwarn 来使 C++ 编译器在给定警告消息处于失败的状态下退出。

值

t 是一个以逗号分隔的列表，该列表包括以下各项中的一个或多个：*tag*、no%*tag*、%all、%none。顺序很重要，例如如果出现除 *tag* 之外的任何警告， %all,no%*tag* 会使 cc 带致命状态退出。

下表详细列出了 -errwarn 值：

表 A-5 -errwarn 值

值	含义
tag	如果该 *tag* 指定的消息以警告消息的形式出现，就会使 CC 带致命状态退出。如果未发出 *tag*，则不会产生这种情况。
no%*tag*	如果 *tag* 指定的消息只以警告消息的形式出现，就要防止 CC 带致命状态退出。如果 *tag* 指定的消息未出现，则不会产生这种情况。为了避免在发生警告消息时导致 cc 带致命状态退出，就要使用该选项来还原以前用 *tag* 或 %all 指定的警告消息。
%all	如果发生任何警告消息，就会使 CC 带致命状态退出。%all 可以后跟 no%*tag* 以避免该行为的特定警告消息。
%none	防止任何引起 CC 带致命状态退出的警告消息 （该发出任何警告消息时）。

默认

默认为 -errwarn=%none。如果单独指定 -errwarn，就等效于 -errwarn=%all。

警告

-errwarn 选项只可以指定来自 C++ 编译器前端的警告消息来使编译器在失败状态下退出，其中的警告消息会在使用 -errtags 选项时显示标记。

因为编译器错误检查的增强和功能的增加，C++ 编译器生成的警告消息在不同发行版本中各有不同。使用 -errwarn=%all 进行编译而不会产生错误的代码，在编译器下一个发行版本中编译时也可能出现错误。

另请参见

-erroff、-errtags

A.2.18 -fast

此选项是一个宏，可有效用作调节可执行程序以达到最大运行时性能的起始点。 -fast 是一个宏，可随编译器发行版本的升级而更改，并能扩展为目标平台特定的选项。使用 -# 选项或 -xdryrun 检查 -fast 的扩展选项，并将 -fast 的相应选项合并到现行的可执行程序调节进程中。

该选项是一个宏，选择编译选项的组合用于在编译代码的机器上优化执行速度。

扩展

该选项通过扩展到以下编译选项，为大量应用程序提供了几乎最高的性能。

表 A-6 -fast 扩展

选项	SPARC	x86
-fns	X	X
-fsimple=2	X	-
-ftrap=%none	X	X
-nofstore	-	X
-xlibmil	X	X
-xlibmopt	X	X

表 **A-6** -fast 扩展 (续)

选项	SPARC	x86
-xmemalign	X	-
-xO5	X	X
-xtarget=native	X	X
-xbuiltin=%all	X	X

交互

-fast 宏扩展到可能影响其他指定选项的编译选项中。例如，在以下命令中，-fast 宏的扩展包括了将 -xarch 还原为 32 位架构选项的 -xtarget=native。

错误：

```
example% CC -xarch=v9 -fast test.cc
```

正确：

```
example% CC -fast -xarch=v9 test.cc
```

查看每个选项的描述以确定可能的交互操作。

代码生成选项、优化级别、内建函数的优化和内联模板文件的使用可以用后续选项来覆盖 （请参见示例）。指定的优化级别将覆盖以前所设置的优化级别。

-fast 选项包括 -fns -ftrap=%none ；即，该选项关闭了所有捕获。

示例

以下编译器命令生成了 -xO3 的优化级别。

```
example% CC -fast -xO3
```

以下编译器命令产生了 -xO5 的优化级别。

```
example% CC -xO3 -fast
```

警告

如果单独进行编译和链接，`-fast` 选项就必须出现在编译命令和链接命令中。

`-fast` 选项所编译的代码不可移植。例如，在 UltraSPARC II 系统中无法执行在 UltraSPARC III 系统中用以下命令生成的二进制文件。

```
example% CC -fast test.cc
```

不要将该选项用于依赖 IEEE 标准浮点运算的程序，否则可能会产生不同的数字结果、过早的程序终止或意外的 SIGFPE 信号。

在以前的 SPARC 发行版本中， `-fast` 宏扩展到了 `-fsimple=1`。而现在扩展到 `-fsimple=2`。

在以前的发行版本中， `-fast` 宏扩展到了 `-xO4`。而现在扩展到 `-xO5`。

注 – 在以前的 SPARC 发行版本中，`-fast` 宏选项包括了 `-fnonstd`，而现在的版本则不包括。非标准浮点模式不用 `-fast` 初始化。请参见《数值计算指南》， `ieee_sun`(3M)。

另请参见

`-fns`、 `-fsimple`、 `-ftrap=%none`、 `-xlibmil`, `-nofstore`、 `-xO5`、 `-xlibmopt`、 `-xtarget=native`

A.2.19 `-features=`*a*`[,`*a*`...]`

启用/禁用在逗号分隔的列表中命名的各种 C++ 语言功能。

值

在兼容模式 (-compat[=4]) 和标准模式 （默认模式）下 *a* 都可以具有以下值：

表 A-7 兼容模式和标准模式的 -features 值

a 的值	含义
%all	所有 -features 选项对指定的模式 （兼容模式或标准模式）都有效。
[no%]altspell	[不] 识别替换标记拼写 （如替换 "&&" 的 "and"）。在兼容模式下默认为 no%altspell，而标准模式下默认为 altspell。
[no%]anachronisms	[不] 允许过时的构造。禁用时 （即 -features=no%anachronisms），不允许任何过时构造。默认为 anachronisms。
[no%]bool	[不] 允许 bool 类型和文字。启用时，宏 _BOOL=1。未启用时，不定义宏。在兼容模式下默认为 no%bool，在标准模式下默认为 bool。
[no%]conststrings	[不] 放置只读内存中的文字字符串。在兼容模式下默认为 no%conststrings，在标准模式下默认为 conststrings。
[no%]except	[不] 允许 C++ 异常。 C++ 异常被禁用时 （即 -features=no%except），接受但忽略了函数的抛出规范，编译器不生成异常代码。注意，通常保留关键字 try、 throw 和 catch。请参见第 8-2 页的第 8.3 节 “禁用异常”。默认为 except。
[no%]export	[不] 识别关键字 export。在兼容模式下默认为 no%export，在标准模式下默认为 export。
[no%]extensions	[不] 允许其他 C++ 编译器通常接受的非标准代码。关于使用 -features=extensions 选项时编译器接受的无效代码的说明，请参见第 4 章。默认为 no%extensions。
[no%]iddollar	[不] 允许 $ 符号作为非初始化标识符字符。默认为 no%iddollar。
[no%]localfor	[不] 在 for 语句中使用新的局部作用域规则。在兼容模式下默认为 no%localfor，在标准模式下默认为 localfor。
[no%]mutable	[不] 识别关键字 mutable。在兼容模式下默认为 no%mutable，在标准模式下默认为 mutable。

表 A-7　兼容模式和标准模式的 -features 值 (续)

a 的值	含义
[no%]split_init	[不] 将非局部静态对象的初始化函数放入个别函数中。使用 -features=no%split_init 时，编译器将所有初始化函数放入一个函数中。使用 -features=no%split_init 在可能占用编译时间的情况下将代码大小最小化。默认为 split_init。
[no%]transitions	[不] 允许 ARM 语言构造，该构造会在标准 C++ 下产生问题，且可能使程序无法按预期工作或以后可能不能编译。使用 -features=no%transitions 时，编译器会将这种情况视为错误。在标准模式下使用 -features=transitions 时，编译器会发出关于这些构造的警告，而不发出错误消息。兼容模式 (-compat[=4]) 下使用 -features=transitions 时，编译器仅在指定 +w 或 +w2 的情况下才显示关于这些构造的警告。以下构造被认为是转换错误：在使用模板之后再重定义模板，忽略在模板定义时所需的 typename 指令，隐式声明类型 int。在以后的发行版本中可能会更改转换错误的集合。默认为 transitions。
%none	关闭指定模式中可以关闭的所有功能。

在标准模式 （默认模式）下， *a* 可以具有以下附加值：

表 A-8　仅用于标准模式的 -features 值

a 的值	含义
[no%]strictdestrorder	[不] 遵循由 C++ 标准指定的要求，该要求关于具有静态存储持续时间对象的析构顺序。默认为 strictdestrorder。
[no%]tmplife	[不] 清除由位于全表达式结尾处的表达式 （在 ANSI/ISO C++ 标准中定义）创建的临时对象。(-features=no%tmplife 生效时，多数临时对象会在块的结尾处清除。）默认为 no%tmplife。

在兼容模式 (-compat[=4]) 下， *a* 可以具有以下附加值。

表 A-9 仅用于兼容模式的 -features 值

a 的值	含义
[no%]arraynew	[不] 识别运算符 new 和 delete 的数组形式 （如 operator new [ ] (void*))。启用时，宏 __ARRAYNEW=1。未启用时，不定义宏。默认为 no%arraynew。
[no%]explicit	[不] 识别关键字 explicit。默认为 no%explicit。
[no%]namespace	[不] 识别关键字 namespace 和 using。默认为 no%namespace。 -features=namespace 的目的是帮助将代码转换到标准模式。通过启用该选项，将这些关键字作为标识符使用时就会出现错误消息。关键字识别选项使您能够使用增加的关键字，而不必在标准模式下进行编译。
[no%]rtti	[不] 允许运行时类型信息 (RTTI)。要使用 dynamic_cast<> 和 typeid 运算符，就必须启用 RTTI。默认为 no%rtti。对于 -compat=4 mode，默认值为 no%rtti。否则，默认值为 -features=rtti，并且不允许使用选项 -features=no%rtti。

注 – [no%]castop 设置可以与为 C++ 4.2 编译器所编写的 makefiles 兼容，但不影响编译器版本 5.0、5.1、5.2 和 5.3。新型的类型转换（const_cast、dynamic_cast、reinterpret_cast 和 static_cast）始终会被识别但无法被禁用。

默认

如果未指定 -features，则做出以下假定：

- 兼容模式 (-compat[=4])

```
-features=%none,anachronisms,except,split_init,transitions
```

- 标准模式 （默认模式）

```
-features=%all,no%iddollar,no%extensions,no%tmplife
```

交互

该选项会累积而不覆盖。

在标准模式 （默认）中以下选项与标准库和头文件不兼容：

- no%bool
- no%except
- no%mutable
- no%explicit

在兼容模式 (-compat[=4]) 下，-features=transitions 选项无效，除非您指定了 +w 选项或 +w2 选项。

警告

指定 -features=%all 或 -features=%none 时务必要小心。 features 的设置可能会随每个编译器发行版和补丁程序而发生变化。从而可能会有一些您不希望获得的程序行为发生。

使用 -features=tmplife 选项时，程序的行为可能更改。测试程序是否可以使用 -features=tmplife 选项是测试程序可移植性的一种方法。

在兼容模式 (-compt=4) 下，编译器在默认情况下假定 -features=split_init。如果使用 -features=%none 选项来关闭其他功能，就会发现需要通过使用 -features=%none,split_init 将初始化函数的分离部分转换到独立的函数中。

另请参见

第 4 章 和 《C++ 迁移指南》

A.2.20 -filt[=*filter*[,*filter*...]]

控制编译器通常应用于链接程序和编译器错误消息的过滤。

值

filter 必须是下列值之一：

表 A-10 `-filt` 值

filter 的值	含义
`[no%]errors`	[不] 显示链接程序错误消息的 C++ 解释。链接程序的诊断信息被直接提供到其他工具时，可以禁止这种解释。
`[no%]names`	[不] 还原 C++ 粉碎的链接程序名称。
`[no%]returns`	[不] 还原函数的返回类型。禁止该类型的还原可以使您更快速地识别函数的名称，但请注意联合变体返回的部分函数只在返回类型上有区别。
`[no%]stdlib`	[不] 在链接程序和编译器错误消息中简化标准库的名称。这能帮助您更容易识别出标准库模板类型的名称。
`%all`	等效于 `-filt=errors,names,returns,stdlib`。这是默认行为。
`%none`	等效于 `-filt=no%errors,no%names,no%returns,no%stdlib`。

默认

如果未指定 `-filt` 选项或指定没有任何值的 `-filt`，那么编译器假定 `-filt=%all`。

示例

以下示例显示了使用 `-filt` 选项编译该代码的效果。

```
// filt_demo.cc
class type {
public:
    virtual ~type(); // 未提供定义
};

int main()
{
    type t;
}
```

编译代码时如果不使用 -filt 选项，编译器就会假定 -filt=errors,names,returns,stdlib 并显示标准输出。

```
example% CC filt_demo.cc
Undefined                       first referenced
 symbol                             in file
type::~type()          filt_demo.o
type::__vtbl           filt_demo.o
[ 提示：试图检查是否已定义了类的类型的第一个非内联、非纯虚函数 ]

ld: fatal: Symbol referencing errors. No output written to a.out
```

以下命令禁止还原 C++ 粉碎链接程序名称，并禁止链接程序错误的 C++ 解释。

```
example% CC -filt=no%names,no%errors filt_demo.cc
Undefined                       first referenced
 symbol                             in file
__1cEtype2T6M_v_                    filt_demo.o
__1cEtypeG__vtbl_                   filt_demo.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

现在考虑以下代码：

```
#include <string>
#include <list>
int main()
{
    std::list<int> l;
    std::string s(l); // 此处出错
}
```

以下是指定 -filt=no%stdlib 时的输出内容：

```
Error: Cannot use std::list<int, std::allocator<int>> to
initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

以下是指定 -filt=stdlib 时的输出内容：

```
Error: Cannot use std::list<int> to initialize std::string.
```

交互

当您指定 no%names 时，无论是 returns 还是 no%returns 都是无效的。也就是说，以下选项是相同的：

- -filt=no%names
- -filt=no%names,no%returns
- -filt=no%names,returns

A.2.21 -flags

与 -xhelp=flags 相同。

A.2.22 -fnonstd

使硬件自陷可用于浮点溢出，除以零，并可用于无效运算异常。产生的结果被转换为 SIGFPE 信号，如果程序没有 SIGFPE 处理程序，它就会终止并转储内存 （除非将核心转储大小限制到零）。

SPARC：此外，-fnonstd 还选择 SPARC 非标准浮点。

默认

如果未指定 -fnonstd，则 IEEE 754 浮点算术异常不终止程序并逐步下溢。

扩展

x86：-fnonstd 扩展到 -ftrap=common。

SPARC：-fnonstd 扩展到 -fns -ftrap=common。

另请参见

-fns、 -ftrap=common、《数值计算指南》。

A.2.23 -fns[={yes|no}]

- SPARC：启用/禁用 SPARC 的非标准浮点模式。

  -fns=yes （或 -fns）会在程序开始执行时启用非标准浮点模式。

该选项提供了一种切换非标准或标准浮点模式的方法，后跟包括 -fns 的某些其他宏选项 （如 -fast）。

在某些 SPARC 设备上，非标准浮点模式会禁用 “渐进下溢”，这会导致微小的结果被刷新为零而不是产生低于正常的值，还会导致低于正常的操作数被默置为零。

在不支持硬件逐步下溢和低于正常的数的 SPARC 设备上，-fns=yes（或 -fns）可以显著提高某些程序的性能。

- (*x86*) 选择 SSE 刷新为零模式以及 （如果可用）反向规格数为零模式。

  此选项导致将次正规结果刷新为零。如果可用的话，此选项还导致将次正规操作数视为零。

  此选项未采用 SSE 或 SSE2 指令集的传统 x86 浮点运算。

值

-fns 选项可以具有以下值。

表 A-11 -fns 值

值	含义
是	选择非标准浮点模式
否	选择标准浮点模式

默认

如果未指定 -fns，则不自动启用非标准浮点模式。标准 IEEE 754 浮点计算发生 （即渐进下溢）。

如果仅指定了 -fns，则假定 -fns=yes。

示例

在以下示例中，-fast 扩展到了多个选项，其中一个是 -fns=yes ，即选择非标准浮点模式。随后 -fns=no 选项覆盖初始设置，并选择浮点模式。

```
example% CC foo.cc -fast -fns=no
```

警告

非标准模式启动时，浮点运算可以产生不符合 IEEE 754 标准要求的结果。

如果先使用 -fns 选项编译一个例程，就要使用 -fns 选项来编译程序的所有例程；否则，就可能获得意外的结果。

该选项仅在 SPARC 设备上有效，并且仅在编译主程序时才能有效使用。在 x86 设备上，此选项被忽略。

在程序中出现通常由 IEEE 浮点自陷处理程序管理的浮点错误时，使用 -fns=yes （或 -fns）选项可能会生成以下消息：

另请参见

《数值计算指南》、 ieee_sun(3M)

A.2.24 -fprecision=*p*

x86：设置非默认的浮点精度模式。

-fprecision 选项设置浮点控制字中的舍入精度模式位。这些位控制基本算术运算（加、减、乘、除和平方根）结果的舍入精度。

值

p 必须是下列值之一：

表 A-12 -fprecision 值

***p* 的值**	**含义**
单精度	舍入到 IEEE 单精度值。
双精度	舍入到 IEEE 双精度值。
extended	舍入到最大可用精度。

如果 *p* 为 single 或 double，则该选项会使舍入精度模式在程序开始执行时分别设置为 single 或 double 精度。如果 *p* 是未使用的 extended 或 -fprecision 选项，则舍入精度模式会保留 extended 精度。

single 精度舍入模式会将结果舍入到 24 个有效位，而 double 精度舍入模式会将结果舍入到 53 个有效位。在默认的 extended 精度模式下，结果被舍入到 64 个有效位。该模式只控制在寄存器中结果的舍入精度，而不影响范围。寄存器中所有的结果都使用了各种已扩展的双精度格式来舍入。不过，存储在内存中的结果既舍入到目标格式的范围也舍入到目标格式的精度。

float 类型的名义精度为 single。 long double 类型的名义精度为 extended。

默认

未指定 -fprecision 选项时，默认的舍入精度模式为 extended。

警告

此选项仅对 x86 设备且仅在编译主程序时使用才有效。在 SPARC 设备上，该选项被忽略。

A.2.25 -fround=*r*

启动时设置有效的 IEEE 舍入模式。

该选项将 IEEE 754 舍入模式设置为：

- 编译器可以将其用于计算常量表达式
- 程序初始化期间在运行时建立

该含义与 ieee_flags 子例程的含义相同，可用于更改运行时的模式。

值

r 必须是下列值之一：

表 A-13 -fround 值

r 的值	含义
nearest	舍入到最接近的数字并转变为偶数。
tozero	舍入到零。
negative	舍入到负无穷大。
positive	舍入到正无穷大。

默认

未指定 -fround 选项时，默认的舍入模式为 -fround=nearest。

警告

如果先使用 -fround=*r* 编译一个例程，就还要使用相同的 -fround=*r* 选项来编译程序的所有例程；否则就会获得意外的结果。

只有编译主程序时该选项才有效。

A.2.26 -fsimple[=*n*]

选择浮点优化首选项。

该选项允许优化器简化关于浮点运算的假定。

值

如果存在 *n*，则必须是 0、 1 或 2。

表 A-14 -fsimple 值

n 的值	含义
0	不允许简化假定。保持严格的 IEEE 754 一致性。
1	允许保守简化。产生的代码与 IEEE 754 不完全一致，但多数程序所产生的数值结果没有更改。 使用 -fsimple=1 时，优化器可能要进行以下假定： • 进程初始化之后，不更改 IEEE754 的默认舍入/自陷模式。 • 除产生潜在浮点异常的计算不能删除外，产生不可视结果的计算都可以删除。 • 使用无穷大或 NaNs 作为操作数的计算需要将 NaNs 传送到它们的结果中，即 x*0 可以用零替换。 • 计算不依赖于零的符号。 使用 -fsimple=1 时，如果与误差或异常无关，则不允许优化器完全优化。具体来讲，运行时将舍入模式保存为常量时，不能用产生不同结果的计算来替换浮点计算。
2	允许主动的浮点优化。由于在舍入过程中进行了更改，所以该优化会使许多程序产生不同的数值结果。例如，允许优化器在指定循环中使用 x*z 来替换 x/y 的所有计算，其中要确保 x/y 在循环 z=1/y 中至少运算一次，并且在循环的执行期间 y 和 z 的值为常量值。

默认

如果未指定 -fsimple，则编译器使用 -fsimple=0。

如果指定了 -fsimple 但未给定 *n* 任何值，则编译器使用 -fsimple=1。

交互

-fast 暗示了 -fsimple=2。

警告

该选项可以破坏 IEEE 754 的一致性。

另请参见

`-fast`

《Techniques for Optimizing Applications:High Performance Computing》，由 Rajat Garg 和 Ilya Sharapov 撰写，详细解释了优化对精度的影响。

A.2.27 `-fstore`

x86：在以下值为真时，该选项使编译器将浮点表达式或函数的值转换为赋值左侧的类型，而不是将该值保留在寄存器中：

- 将表达式或函数分配到变量。
- 表达式被转换为较短的浮点类型。

要关闭该选项，请使用 `-nofstore` 选项。

警告

由于误差和截断，结果可能会与寄存器值所生成的结果不同。

另请参见

`-nofstore`

A.2.28 `-ftrap=`*t*`[,`*t*`...]`

设置 IEEE 捕获模式启动时有效，但不安装 `SIGFPE` 处理程序。您可以使用 `ieee_handler`(3M) 或 `fex_set_handling`(3M) 启用自陷，并同时安装 SIGFPE 处理程序。如果指定多个值，则按从左到右顺序处理列表。

值

t 可以是下列值之一：

表 A-15 `-ftrap` 值

t 的值	含义
`[no%]division`	[不] 在除以零时自陷。
`[no%]inexact`	[不] 在结果不精确时自陷。
`[no%]invalid`	[不] 在无效操作上自陷。
`[no%]overflow`	[不] 在溢出上自陷。
`[no%]underflow`	[不] 在下溢上自陷。
`%all`	在所有以上内容中自陷。
`%none`	不在以上任何内容中自陷。
`common`	在无效、除以零和溢出时自陷。

注意，选项的 `[no%]` 形式只用于修改 `%all` 和 `common` 值的含义，且必须用于其中的一个值，如以下示例所示。选项自身的 `[no%]` 形式不会显式导致禁用特定的自陷。

默认

如果您不指定 `-ftrap`，则编译器假定 `-ftrap=%none`。

示例

`-ftrap=%all,no%inexact` 意味着设置所有陷阱，但 inexact 除外。

警告

如果先使用 `-ftrap=`*t* 编译一个例程，就还要使用相同的 `-ftrap=`*t* 选项来编译程序的所有例程；否则就会获得意外的结果。

使用 `-ftrap=inexact` 自陷时要小心操作。只要浮点值不能精确表示，就会使用 `-ftrap=inexact` 产生自陷。例如，以下语句就会产生这种情况：

```
x = 1.0 / 3.0;
```

只有编译主程序时该选项才有效。请小心使用该选项。如果您希望启用 IEEE 自陷，请使用 `-ftrap=common`。

另请参见

ieee_handler(3M)、 fex_set_handling(3M) 手册页。

A.2.29 -G

生成动态共享库而不是可执行文件。

默认情况下，在命令行指定的所有源文件使用 -xcode=pic13 来编译。

生成使用模板的共享库时，多数情况下有必要将这些模板函数包括在共享库中，而这些函数已在模板数据库中实例化。使用该选项可以将这些模板自动增加到所需的共享库中。

如果通过指定 -G 以及其他必须在编译时和链接时指定的编译器选项来创建共享对象，请确保在与生成的共享对象编译和链接时也指定这些相同的选项。

创建共享对象时，使用 -xarch=v9 编译的所有对象文件还必须使用第 A-82 页的第 A.2.114 节 "-xcode=a" 中建议的某个显式 -xcode 值进行编译。

交互

如果未指定 -c （仅编译选项），以下选项就会被传递到链接程序。

- -dy
- -G
- -R

警告

不要使用 ld -G 来生成共享库，而要使用 CC -G。 CC 驱动程序自动将多个选项传递到 C++ 所需的 ld 中。

使用 -G 选项时，编译器不将任何默认 l 选项传递到 -ld 选项。如果您要使共享库具有对另一共享库的依赖性，就必须在命令行上传递必需的 -l 选项。例如，如果您要使共享库具有对 libCrun 的依赖性，就必须在命令行上传递 -lCrun。

另请参见

-dy、-Kpic、-xcode=pic13、 -ztext、ld(1) 手册页第 16-2 页的第 16.3 节 “生成动态 （共享）库”。

A.2.30 -g

生成使用 dbx(1) 或调试程序进行调试或者使用性能分析器 analyzer(1) 进行分析所需的附加符号表信息。

指示编译器和链接程序准备进行调试和性能分析的文件或程序。

其任务包括：

- 在对象文件和可执行文件的符号表中生成 *stabs* 详细信息。
- 生成某些 “helper 函数”，调试程序可以调用这些函数来实现自身某些特性。
- 禁用函数的内联生成。
- 禁用优化的某些级别。

交互

如果与 -xO*level*（或与 -O 等效的选项）一起使用该选项，那么将会获得有限的调试信息。有关更多信息，请参见第 A-110 页的第 A.2.145 节 "-xOlevel"。

如果您所用选项的优化级别为 -xO4 或更高，编译器就会提供全部优化的最多的符号信息。

指定该选项时， +d 选项被自动指定。

注 – 在以前的发行版中，此选项强制编译器在默认情况下使用递增链接程序 (ild)，而不使用用于编译器的仅链接调用的链接程序 (ld)。也就是说，使用 -g 时，只要您使用编译器来链接对象文件，编译器的默认行为就是自动调用 ild 而不是 ld，除非您在命令行上指定了 -G 或源文件。情况不再是这样，增量式链接程序不再可用。

要使用性能分析器的所有功能，请使用 -g 选项进行编译。某些性能分析特性不需要 -g 时，您必须使用 -g 进行编译，以查看已注释的源码、某些函数级别信息和编译器注释消息。有关详细信息，请参见 analyzer(1) 手册页和 《程序性能分析工具》中的 “编译您的程序以进行数据收集和分析”。

使用 -g 生成的注释消息描述了编译器在编译程序时进行的优化和转换。使用 er_src(1) 命令来显示与源代码交叉的消息。

警告

如果单独进行编译和链接程序，那么将 -g 选项包括在一步骤中并将该选项从其他步骤中排除不会影响程序的正确性，但这样会影响调试程序的能力。没有使用 -g （或 -g0）编译但使用 -g （或 -g0）链接的任何模块将不能正常调试。注意，通常必须使用 -g 选项 （或 -g0 选项）来编译包含函数 main 的模块才可用于调试。

另请参见

+d、-g0、-xs、analyzer(1) 手册页，er_src(1) 手册页，ld(1) 手册页，*使用* dbx（关于 stab 的详细信息）调试程序，《程序性能分析工具》。

A.2.31 -g0

为调试编译并链接，但不禁用内联。

此选项与 -g 相同，但 +d 被禁用且 dbx 无法步入内联函数。

如果指定 -g，并且优化级别为 -xO3 或更低，则编译器提供最佳效果符号信息。尾部调用优化和后端内联被禁用。

另请参见

+d、 -g、《使用 dbx 调试程序》

A.2.32 -H

打印包含文件的路径名称。

在标准错误输出（stderr）中，该选项打印当前编译中包含的每个 #include 文件的路径名称 （每行一个名称）。

A.2.33 -h[ ]*name*

将名称 *name* 分配到生成的动态共享库。这是一个加载器选项，传递到 ld。通常，-h 后的名称应该与 -o 后的名称完全相同。在 -h 和 *name* 之间的空格是可选的。

编译时的加载器将指定名称分配到正在创建的共享动态库中，并将该名称作为库的内部名称记录在库文件中。如果没有 -h*name* 选项，在库文件中就没有记录内部名称。

每个可执行文件都具有所需的共享库文件列表。当运行时链接程序将库链接到可执行文件中时，链接程序将内部名称从库复制到所需共享库文件的列表中。如果没有共享文件的内部名称，链接程序就复制共享库文件的路径。

示例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

A.2.34 -help

与 -xhelp=flags 相同。

A.2.35 -I*pathname*

将 *pathname* 增加到 #include 文件搜索路径。

该选项将 *pathname* 增加到用于搜索具有相对文件名称的 #include 文件 （不以斜杠开头的文件）的目录列表。

编译器按此顺序搜索带引号的包括文件 （#include "foo.h" 形式）。

1. 在包含源码的目录中
2. 在使用 -I 选项命名的目录 （如果有）中
3. 在编译器提供的 C++ 头文件、 ANSI C 头文件和特定目的文件的 include 目录中
4. 在 /usr/include 目录中

编译器按此顺促搜索带尖括号的包括文件 （#include <foo.h> 形式）。

1. 在使用 -I 选项命名的目录 （如果有）中
2. 在编译器提供的 C++ 头文件、 ANSI C 头文件和特定目的文件的 include 目录中
3. 在 /usr/include 目录中

注 – 如果拼写与标准头文件的名称相匹配，也可参见第 12-13 页的第 12.7.5 节“标准头文件实现”。

交互

-I- 选项使您能够覆盖默认的搜索规则。

如果指定了 -library=no%Cstd，那么编译器在其搜索路径中就不包括编译器提供的与 C++ 标准库相关的头文件。请参见第 12-11 页的第 12.7 节 “替换 C++ 标准库”。

如果未使用 -pti*path*，编译器就会在 -I*pathname* 中查找模板文件。

请使用 -I*pathname* 而不要使用 -pti*path*。

该选项会累积而不覆盖。

警告

勿将编译器安装区域 /usr/include、 /lib 或 /usr/lib 指定为搜索目录。

另请参见

-I-

A.2.36 -I-

将包含文件的搜索规则更改为：

对于 #include "foo.h" 形式的包含文件，按以下顺序搜索目录：

1. 使用 -I 选项 （包括在 -I- 前后）命名的目录
2. 编译器提供的 C++ 头文件、 ANSI C 头文件和专用文件的目录中
3. /usr/include 目录

对于 #include <foo.h> 形式的包含文件，按以下顺序搜索目录。

1. 使用出现在 -I- 之后的 -I 选项命名的目录
2. 编译器提供的 C++ 头文件、 ANSI C 头文件和专用文件的目录中
3. /usr/include 目录

注 – 如果包含文件的名称与标准头文件的名称相匹配，也可参见第 12-13 页的第 12.7.5 节 “标准头文件实现”。

示例

以下示例显示了编译 prog.cc 时使用 -I- 的结果。

```
prog.cc       #include "a.h"
              #include <b.h>
              #include "c.h"

c.h           #ifndef _C_H_1
              #define _C_H_1
              int c1;
              #endif

inc/a.h       #ifndef _A_H
              #define _A_H
              #include "c.h"
              int a;
              #endif

inc/b.h       #ifndef _B_H
              #define _B_H
              #include <c.h>
              int b;
              #endif

inc/c.h       #ifndef _C_H_2
              #define _C_H_2
              int c2;
              #endif
```

以下命令显示了为 #include "foo.h" 形式的包含语句搜索当前目录 （包含文件的目录）的默认行为。在 inc/a.h 中处理 #include "c.h" 语句时，编译器包括来自 inc 子目录的 c.h 头文件。在 prog.cc 中处理 #include "c.h" 语句时，编译器包括了源于包含 prog.cc 目录的 c.h 文件。注意，-H 选项指示编译器打印包含文件的路径。

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

下一条命令显示了 -I- 选项的结果。编译器处理 #include "foo.h" 形式的语句时，并不首先在包含的目录中查找。相反，编译器按照目录在命令行上出现的顺序，搜索通过 -I 选项命名的目录。在 inc/a.h 中处理 #include "c.h" 语句时，编译器包括了 ./c.h 头文件，而没有包括 inc/c.h 头文件。

```
example% CC -c -I. -I- -Iinc -H prog.cc
inc/a.h
        ./c.h
inc/b.h
        inc/c.h
./c.h
```

交互

-I- 出现在命令行时，编译器不再搜索当前目录，除非该目录显式列于 -I 指令中。该结果甚至还适用于 #include "foo.h" 形式的包括语句。

警告

只有命令行上的第一个 -I- 会引起描述的行为。

勿将编译器安装区域 /usr/include、 /lib 或 /usr/lib 指定为搜索目录。

A.2.37 -i

告知链接程序 ld(1) 忽略任何 LD_LIBRARY_PATH 设置。

A.2.38 -inline

与 -xinline 相同。

A.2.39 -instances=*a*

控制模板实例的放置和链接。

值

a 必须是下列值之一：

表 A-16 `-instances` 值

a **的值**	**含义**
`extern`	将全部所需示例放置到 comdat 部分的模块系统信息库并赋予全局链接。（如果系统信息库中的实例过期，就会被重新实例化。） **请注意：**如果以不同的步骤编译和链接并为编译步骤指定了 `-instance=extern`，则也必须为链接步骤指定 `-instance=extern`。
显式	将显式实例化的实例放置到当前对象文件中并赋予全局链接。不生成其他任何所需实例。
全局	将全部所需的实例放置到当前对象文件中并赋予全局链接。
`semiexplicit`	将显式实例化的实例放置到当前对象文件中并赋予全局链接。将显式实例所需的全部实例放置到当前对象文件中并赋予全局链接。不生成其他任何所需实例。
`static`	**请注意：**`-instances=static` 已过时。没有任何理由再使用 `-instances=static`，因为 `-instances=global` 现在向您提供了所有静态的优点而没有它的缺点。以前编译器中提供的该选项用于克服此编译器版本中不存在的问题。 将全部所需的实例放置到当前对象文件中并赋予静态链接。

默认

如果未指定 `-instances`，则假定 `-instances=global`。

另请参见

第 7-2 页的第 7.2.4 节 “模板实例的放置和链接”。

A.2.40 `-instlib=`*filename*

使用该选项以避免在库 （共享或静态）和当前对象中生成重复的模板实例。一般来说，如果程序与函数库共享大量的实例，则可以试试 `-instlib=`*filename* 选项，看看编译时间是否会减少。

值：

使用 *filename* 参数以指定包含现有模块实例的库 （已知的）。 filename 参数必须包含正斜杠 "/" 字符。与当前目录相关的路径要使用点斜杠 "./"。

默认：

`-instlib=`*filename* 选项不是默认选项，只有在指定该选项之后才能使用。该选项可被多次指定和累积。

示例：

假定 `libfoo.a` 和 `libbar.so` 库实例化与源文件 `a.cc` 共享的大量模板实例。增加 `-instlib=`*filename* 并指定库可以通过避免冗余，帮助减少编译时间。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

交互：

使用 `-g` 进行编译时，如果用 `-instlib=`*file* 指定的库没有用 `-g` 来编译，那么这些模板实例是不可调试的。解决方法是在使用 `-g` 时避免 `-instlib=`*file*。

警告

如果使用 `-instlib` 指定库，就必须与该库链接。

另请参见：

`-template`、`-instances`、`-pti`

A.2.41 -KPIC

SPARC：与 `-xcode=pic32` 相同。

x86：与 `-Kpic` 相同。

生成共享库时使用该选项编译源文件。对全局数据的每个引用都生成为全局偏移表中指针的非关联化。每个函数调用都通过过程链接表在 pc 相对地址模式中生成。

A.2.42 -KPIC

SPARC：与 -xcode=pic13 相同。

x86：使用与位置无关的代码进行编译。

生成共享库时使用该选项编译源文件。对全局数据的每个引用都生成为全局偏移表中指针的非关联化。每个函数调用都通过过程链接表在 pc 相对地址模式中生成。

A.2.43 -keeptmp

保留编译时创建的临时文件。

加上 -verbose=diags，该选项就可用于调试。

另请参见

-v、 -verbose

A.2.44 -L*path*

将 *path* 增加到目录列表以搜索库。

该选项被传递到 ld。 *path* 命名的目录先于编译器提供的目录搜索。

交互

该选项会累积而不覆盖。

警告

勿将编译器安装区域 /usr/include、 /lib 或 /usr/lib 指定为搜索目录。

A.2.45 -l*lib*

将 library lib*lib*.a 或 lib*lib*.so 增加到链接程序的库搜索列表。

该选项被传递到 ld。正常的库具有诸如 lib*lib*.a 或 lib*lib*.so 的名称，其中 lib 和 .a 或 .so 部分是必需的。应该使用该选项指定 *lib* 部分。将尽可能多的库输入到在单一命令行上，这些库按照 -L*dir* 指定的顺序搜索。

请在对象文件名之后使用该选项。

交互

该选项会累积而不覆盖。

在源码和对象的列表**之后**放入 -l*x* 可以确保按照正确顺序搜索这些库。

警告

要确保正确的库链接顺序，必须使用 -mt 而不是 -lthread 与 libthread 链接。

另请参见

-L*dir*、-mt、第 12 章 和 《Tools.h++ Class Library Reference》

A.2.46 -libmieee

与 -xlibmieee 相同。

A.2.47 -libmil

与 -xlibmil 相同。

A.2.48 -library=*l*[,*l*...]

将指定的 CC 提供的库放入编译和链接中。

值

对于兼容模式 (`-compat[=4]`)，*l* 必须是下列值之一。

表 A-17 用于兼容模式的 `-library` 值

l 的值	含义
`[no%]f77`	已过时。**应该使用** `-xlang=f77`。
`[no%]f90`	已过时。应该使用 `-xlang=f90`。
`[no%]f95`	已过时。应该使用 `-xlang=f95`。
`[no%]rwtools7`	[不] 使用传统 iostream `Tools.h++` 版本 7。
`[no%]rwtools7_dbg`	[不] 使用可调试的 `Tools.h++` 版本 7。
`[no%]complex`	[不] 使用复数运算的 `libcomplex`。
`[no%]interval`	已过时。不要使用。使用 `-xia`。
`[no%]libC`	[不] 使用 `libC` C++ 支持库。
`[no%]gc`	[不] 使用 `libgc` 垃圾收集。
`[no%]sunperf`	[不] 使用 Sun Performance Library™
`%none`	仅使用 `libC` C++ 库。

在标准模式 （默认模式）下，*l* 必须是下列值之一：

表 A-18 用于标准模式的 `-library` 值

l 的值	含义
`[no%]f77`	已过时。**应该使用** `-xlang=f77`。
`[no%]f90`	已过时。应该使用 `-xlang=f90`。
`[no%]f95`	已过时。应该使用 `-xlang=f95`。
`[no%]rwtools7`	[不] 使用传统 iostream `Tools.h++` 版本 7。
`[no%]rwtools7_dbg`	[不] 使用可调试的 `Tools.h++` 版本 7。
`[no%]rwtools7_std`	[不] 使用标准 iostream `Tools.h++` 版本 7。
`[no%]rwtools7_std_dbg`	[不] 使用可调试的标准 iostream `Tools.h++` 7。
`[no%]interval`	已过时。不要使用。使用 `-xia`。
`[no%]iostream`	[不] 使用 `libiostream` 传统 iostream 库。
`[no%]Cstd`	[不] 使用 `libCstd` C++ 标准库。[不] 包括编译器提供的 C++ 标准库头文件。
`[no%]Crun`	[不] 使用 `libCrun` C++ 运行库。
`[no%]gc`	[不] 使用 `libgc` 垃圾收集。

表 A-18　用于标准模式的 -library 值 (续)

I 的值	含义
[no%]stlport4	[不] 使用 STLport 的标准库实现版本 4.5.3 代替默认的 libCstd。关于使用 STLport 实现的更多信息，请参见第 13-13 页的第 13.3 节 "STLport"。
[no%]stlport4_dbg	[不] 使用 STLport 可调试的库。
[no%]sunperf	[不] 使用 Sun Performance Library £
%none	仅使用 libCrun C++ 库。

默认

- **兼容模式 (-compat[=4])**
  - 如果未指定 -library，则假定为 -library=libC。
  - 除非 libC 库使用 -library=no%libC 被特别排除，否则始终包括该库。
- **标准模式 （默认模式）**
  - 除非 libCstd 库使用 -library=%none、-library=no%Cstd 或 -library=stlport4 被特别排除，否则始终包括该库。
  - 始终包括 libCrun 库。

无论是标准模式还是兼容模式，始终包括 libm 和 libc 库，即使指定了 -library=%none。

示例

要在没有任何 C++ 库 （除 libCrun）的标准模式下进行链接，请使用：

```
example% CC -library=%none
```

要在标准模式下包括传统 iostream Rogue Wave tools.h++ 库，请使用：

```
example% CC -library=rwtools7,iostream
```

要在标准模式下包括标准 iostream Rogue Wave tools.h++ 库，请使用：

```
example% CC -library=rwtools7_std
```

要在兼容模式下包括传统 iostream Rogue Wave `tools.h++` 库，请使用：

```
example% CC -compat -library=rwtools7
```

交互

使用 `-library` 指定库时，正确的 `-I` 路径在编译期间设置。正确的 `-L`,`-Y P`, `-R` 路径和 `-l` 选项在链接期间设置。

该选项会累积而不覆盖。

在使用区间运算库时，必须包括以下库之一：`libC`、`libCstd` 或 `libiostream`。

`-library` 选项的使用可以确保指定库的 `-l` 选项按正确顺序发送。例如，`-l` 选项按照 `-library=rwtools7,iostream` 和 `-library=iostream,rwtools7` 的 `-lrwtool -liostream` 顺序传递到 `ld`。

指定的库在系统支持库链接之前链接。

不能在相同命令行上使用 `-library=sunperf` 和 `-xlic_lib=sunperf`。

不能在相同命令行上使用 `-library=stlport4` 和 `-library=Cstd`。

每次只能使用一个 Rogue Wave 工具库，而且不能使用任何具有 `-library=stlport4` 的 Rogue Wave 工具库。

当在标准模式（默认模式）下包括传统 iostreams Rogue Wave 工具库时，还必须包括 `libiostream`（请参见《C++ 迁移指南》）。仅在标准模式下可以使用标准 iostreams Rogue Wave 工具库。以下命令示例显示了 Rogue Wave `tools.h++` 库选项的有效和无效的使用。

```
% CC -compat -library=rwtools7 foo.cc          <-- 有效
% CC -compat -library=rwtools7_std foo.cc      <-- 无效

% CC -library=rwtools7,iostream foo.cc         <-- 有效，传统 iostream
% CC -library=rwtools7 foo.cc                  <-- 无效

% CC -library=rwtools7_std foo.cc              <-- 有效，标准 iostream
% CC -library=rwtools7_std,iostream foo.cc     <-- 无效
```

如果既包括了 `libCstd` 也包括了 `libiostream`，就必须注意不要在程序中使用 iostreams 的新的和旧的形式（如 `cout` 和 `std::cout`）来访问同一文件。如果从传统和标准 iostreams 代码访问同一文件，那么在相同的程序中混合标准 iostreams 和传统 iostreams 可能会出现问题。

既不链接 `libC` 也不链接 `libCrun` 的程序不能使用 C++ 语言的所有特性。

如果指定了 -xnolib，则忽略 -library。

警告

如果单独进行编译和链接，那么出现在编译命令中的 -library 选项的集合必须出现在链接命令中。

stlport4、Cstd 和 iostream 库提供了自己的 I/O 流的实现。使用 -library 指定其中的多个库，会导致不确定的程序行为。关于使用 STLport 实现的更多信息，请参见第 13-13 页的第 13.3 节 "STLport"。

库的集合不稳定，会因不同的发行版本而变化。

另请参见

-I、-l、-R、-staticlib、-xia、-xlang、-xnolib、第 12 章、第 13 章、第 14 章、第 2-14 页的第 2.7.3.3 节 “和标准库头文件一起使用 make”、《Tools.h++ 用户指南》、《Tools.h++ 类库参考》、《标准 C++ 类库参考》、《C++ 区间运算编程参考》。

关于使用 -library=no%cstd 选项以启用自身 C++ 标准库的信息，请参见第 12-11 页的第 12.7 节 “替换 C++ 标准库”。

A.2.49 -mc

从对象文件的 .comment 部分删除重复的字符串。如果字符串包含空格，就必须使用引号将该字符串括入。使用 -mc 选项时，会调用 mcs -c 命令。

A.2.50 -migration

解释了获取关于为编译器的早期版本生成的迁移源代码信息的位置。

注 – 在下一发行版本中该选项会取消。

A.2.51 -misalign

SPARC：允许内存中包含未对齐数据，否则会生成错误。如以下代码所示：

```
char b[100];
int f(int * ar) {
return *(int *) (b +2) + *ar;
}
```

该选项通知编译器程序中的某些数据未正确对齐。因此，非常保守的装入和存储必须用于会不对齐的任何数据，即每次一个字节。使用该选项会显著降低运行时性能。性能降低的程度与应用程序有关。

交互

在 SPARC 平台上使用 `#pragma pack` 包装比类型的默认对齐更紧密的对齐时，必须为程序的编译和链接指定 `-misalign` 选项。

未对齐数据通过在运行时 `ld` 提供的自陷机制来处理。如果优化标志 （`-xO{1|2|3|4|5}` 或等效的标志）与 `-misalign` 选项一起使用，则用于对齐未对齐数据的附加指令插入到生成的对象文件中，而且不生成运行时未对齐自陷。

警告

如果可能，请不要链接程序的对齐部分和未对齐部分。

如果在不同的步骤中执行编译和链接，那么 `-misalign` 选项必须同时出现在编译命令和链接命令中。

A.2.52 -mr[,*string*]

从对象文件的 `.comment` 部分删除所有的字符串，并在提供了 *string* 之后将 *string* 放置到该节。如果字符串包含空格，就必须使用引号将字符串括入。使用该选项时，命令 `mcs -d` [`-a` *string*] 被调用。

交互

指定 `-S`、 `-xsbfast` 或 `-sbfast` 时，该选项无效。

A.2.53 -mt

编译和链接多线程代码。

该选项：

- 将 -D_REENTRANT 传递到预处理程序
- 按照正确的顺序将 -lthread 传递到 ld
- 在标准模式 （默认模式）下，确保 libthread 在 libCrun 之前链接。
- 在兼容模式 (-compat) 下，确保 libthread 在 libC 之前链接。

如果应用程序或库是多线程的，则 -mt 选项是必须的。

警告

要确保正确的库链接顺序，必须使用该选项而不是 -lthread 来与 libthread 链接。

如果正使用 POSIX 线程，就必须用 -mt 和 -lpthread 选项链接。由于 libCrun （标准模式）和 libC （兼容模式）需要多线程应用程序的 libthread，所以 -mt 选项是必须的。

如果单独进行编译和链接且使用 -mt 编译，就要确保也使用 -mt 链接，否则会产生意外的结果，如以下示例所示。

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

如果使用 C++ 对象来混合并行的 Fortran 对象，链接行就必须指定 -mt 选项。

另请参见

-xnolib、第 11 章、《多线程编程指南》、《链接程序和库指南》

A.2.54 -native

与 -xtarget=native 相同。

A.2.55 -noex

与 -features=no%except 相同。

A.2.56 -nofstore

x86：禁用表达式的强制精度。

该选项不将浮点表达式或函数的值强制为赋值左侧的类型，但会在以下任一条件为真时将值保留在寄存器中：

- 将表达式或函数分配到变量

  或

- 表达式或函数被转换为较短的浮点类型

另请参见

`-fstore`

A.2.57 -nolib

与 `-xnolib` 相同。

A.2.58 -nolibmil

与 `-xnolibmil` 相同。

A.2.59 -noqueue

禁用许可证队列。

如果没有可用的许可证，该选项就会在无队列请求和编译的情况下返回。用于测试 makefile 的非零状态返回。

A.2.60 -norunpath

不将共享库的运行时搜索路径生成到可执行文件中。

如果可执行文件使用共享库，编译器通常会生成将运行时链接程序指向这些共享库的路径。要完成该操作，编译器将 `-R` 选项传递到 `ld`。路径取决于安装编译器的目录。

建议用该选项生成提交到客户（这些客户的程序使用的共享库具有不同路径）的可执行文件。

交互

如果使用编译器安装区域（默认位置为 /opt/SUNWspro/lib）下的任何共享库，并且还使用 -norunpath，那么就应该在链接时使用 -R 选项或在运行时设置环境变量 LD_LIBRARY_PATH 来指定共享库的位置。该操作使运行时链接程序可以找到共享库。

A.2.61 -O

-O 宏现在扩展为 -xO3 而不是 -xO2。

更改默认设置可以使运行时性能更佳。但是，对于依赖于所有自动被视为 volatile 的变量的程序， -xO3 可能不适用。可能做出此假定的典型程序包括设备驱动程序，以及实现其自己的同步基元的较旧的多线程应用程序。解决方法是使用 -xO2 而不是 -O 进行编译。

A.2.62 -O*level*

与 -xO*level* 相同。

A.2.63 -o *filename*

将输出文件或可执行文件的名称设置为 *filename*。

交互

编译器必须存储模板实例时，会将模板实例存储到输出文件目录中的模板系统信息库中。例如，以下命令将对象文件写入 ./sub/a.o，并将模板实例写入 ./sub/SunWS_cache 内包含的系统信息库中。

```
example% CC -o sub/a.o a.cc
```

编译器从对应于编译器读取的对象文件的模板系统信息库读取。例如，以下命令从 ./sub1/SunWS_Cache 和 ./sub2/SunWS_cache 读取，并且如果需要，则写入 ./SunWS_cache。

```
example% CC sub1/a.o sub2/b.o
```

有关更多信息，请参见第 7-5 页的第 7.4 节 “模板系统信息库”。

警告

filename 必须具有通过编译产生该文件类型的合适后缀。由于 CC 驱动程序不覆盖源文件，所以该文件与源文件不是相同的文件。

A.2.64 +p

忽略非标准预处理程序断言。

默认

如果 +p 未出现，则编译器识别非标准预处理程序断言。

交互

如果使用了 +p，就不定义以下宏。

- sun
- unix
- sparc
- i386

A.2.65 -P

仅预处理源码；不编译。（输出具有 .i 后缀的文件。）

该选项不在输出中包括预处理程序类型的行号信息。

另请参见

-E

A.2.66 -p

已废弃，请参见第 A-120 页的 "-xpg" 。

A.2.67 -pentium

x86：用 -xtarget=pentium 替换。

A.2.68 -pg

与 -xpg 相同。

A.2.69 -PIC

SPARC：与 -xcode=pic32 相同。

x86：与 -Kpic 相同。

A.2.70 -pic

SPARC：与 -xcode=pic13 相同。

x86：与 -Kpic 相同。

A.2.71 -pta

与 -template=wholeclass 相同。

A.2.72 -pti*path*

为模板源文件指定附加搜索目录。

该选项是对 -I*pathname* 所设置的正常搜索路径的替换方法。如果使用了 -pti*path* 选项，编译器就会在该路径上查找模板定义文件，并忽略 -I*pathname* 选项。

使用 -I*pathname* 选项代替 -pti*path* 可以产生较少的混淆。

交互

该选项会累积而不覆盖。

另请参见

-I*pathname*

A.2.73 -pto

与 -instances=static 相同。

A.2.74 -ptr

该选项已废弃并被编译器忽略。

警告

即使编译器忽略了 -ptr 选项，您仍应该从所有编译命令中删除 -ptr，因为在后续发行版本中重用该选项时会产生不同的行为。

另请参见

关于系统信息库目录的信息，请参见第 7-5 页的第 7.4 节 “模板系统信息库”。

A.2.75 -ptv

与 -verbose=template 相同。

A.2.76 -Qoption *phase option*[,*option*...

将 *option* 传递到编译**阶段**。

要传递多个选项，按照逗号分隔列表的顺序指定它们。可以重新排序使用 -Q 给组件传递的选项。驱动程序识别的选项将按正确顺序排列。对于驱动程序已识别的选项，请不要使用 -Q。例如， C++ 编译器识别链接程序 (ld) 的 -z 选项。如果发出如下命令：

```
CC -G -zallextract mylib.a -zdefaultextract ...// 正确
```

则将 -z 选项按顺序传递给链接程序。但是，如果指定了如下命令：

```
CC -G -Qoption ld -zallextract mylib.a -Qoption ld
-zdefaultextract ...// 错误
```

则可能会重新排序 -z 选项，从而给出错误的结果。

值

phase 必须具有下列值之一：

表 A-19 -Qoption 值

SPARC	x86
ccfe	ccfe
iropt	cg386
cg	codegen
CClink	CClink
ld	ld

示例

在下列命令行中，当 CC 驱动程序调用 ld 时， -Qoption 会将 -i 和 -m 选项传递到 ld。

```
example% CC -Qoption ld -i,-m test.c
```

警告

请注意避免无法预料的结果。例如，

```
-Qoption ccfe -features=bool,iddollar
```

被解释为

```
-Qoption ccfe -features=bool -Qoption ccfe iddollar
```

正确的用法为

```
-Qoption ccfe -features=bool,-features=iddollar
```

A.2.77 -qoption *phase option*

与 -Qoption 相同。

A.2.78 -qp

与 -p 相同。

A.2.79 -Qproduce *sourcetype*

使 CC 驱动程序生成类型 *sourcetype* 的输出。

Sourcetype 后缀在下面定义。

表 A-20 -Qproduce 值

后缀	含义
.i	ccfe 的预处理的 C++ 源码
.o	对象文件代码生成器
.s	cg 的汇编程序源码

A.2.80 -qproduce *sourcetype*

与 -Qproduce 相同。

A.2.81 -R*pathname*[:*pathname...*

将动态库搜索路径生成到可执行文件中。

该选项被传递到 ld。

默认

如果未出现 -R 选项，该库会搜索记录在输出对象并根据目标架构指令传递到运行时链接程序的路径。其中的目标架构指令由 -xarch 选项（-xarch 未出现时，假定为 -xarch=generic）指定。

-xarch 值	默认库搜索路径
v9、v9a 或 v9b	*install-directory*/SUNWspro/lib/v9
所有其他值	*install-directory*/SUNWspro/lib

在默认的安装中， *install-directory* 是 /opt。

交互

该选项会累积而不覆盖。

如果定义了 LD_RUN_PATH 环境变量且指定了 -R 选项，则扫描 -R 的路径而忽略 LD_RUN_PATH 的路径。

另请参见

-norunpath、《链接程序和库指南》

A.2.82 -readme

与 -xhelp=readme 相同。

A.2.83 -S

编译并仅生成汇编代码。

该选项使 CC 驱动程序编译程序并输出汇编源文件，而不汇编程序。汇编源文件名称具有 .s 后缀。

A.2.84 -s

从可执行文件去掉符号表。

该选项从输出可执行文件中删除所有符号信息。该选项被传递到 ld。

A.2.85 -sb

用 -xsb 替换。

A.2.86 -sbfast

与 -xsbfast 相同。

A.2.87 -staticlib=*l*[,*l*...]

表明由 -library 选项（包括其默认设置）、-xlang 选项和 -xia 选项指定的 C++ 库中，哪些是静态链接的。

值

l 必须是下列值之一：

表 A-21 -staticlib 值

***l* 的值**	**含义**
[no%] *library*	[不] 静态链接 *library*。 *library* 的有效值包括了 -library（除 %all 和 %none 之外）的全部有效值、-xlang 的全部有效值和 interval （要与 -xia 配合使用）。
%all	静态链接 -library 选项所指定的所有库、-xlang 选项所指定的所有库，并且在命令行指定了 -xia 时，还会静态链接区间库。
%none	不静态链接任何 -library 选项和 -xlang 选项所指定的库。如果在命令行指定了 -xia，则不静态链接区间库。

默认

如果未指定 -staticlib，则假定为 -staticlib=%none。

示例

以下命令行静态链接 libCrun，原因是 Crun 是 -library 的默认值。

```
example% CC -staticlib=Crun （正确）
```

不过，以下命令行不链接 libgc，原因是只有用 -library 选项显式指定才链接 libgc。

```
example% CC -staticlib=gc （不正确）
```

要静态链接 libgc，请使用以下命令：

```
example% CC -library=gc -staticlib=gc （正确）
```

使用以下命令， librwtool 库被动态链接。因为 librwtool 不是默认库且未使用 -library 选项来选择它，所以 -staticlib 无效：

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 （不正确）
```

该命令静态链接了 librwtool 库：

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 （正确）
```

该命令将动态链接 Sun 性能库，原因是 -library=sunperf 必须与 -staticlib=sunperf 配合使用以便 -staticlib 选项可以有效链接这些库：该命令静态链接 Sun

```
example% CC -xlic_lib=sunperf -staticlib=sunperf （不正确）
```

性能库：

```
example% CC -library=sunperf -staticlib=sunperf （正确）
```

交互

该选项会累积而不覆盖。

除默认隐式选择的 C++ 库之外，-staticlib 选项仅适用于使用 -xia 选项、-xlang 选项和 -library 选项显式选择的 C++ 库。在兼容模式 (-compat=[4]) 下，默认选择 libC。在标准模式 （默认模式）下，默认选择了 Cstd 和 Crun。

使用 -xarch=v9、-xarch=v9a 或 -xarch=v9b （或等效的 64 位架构选项）时，某些 C++ 库不能用作静态库。

警告

library 允许值的集合不稳定，会因不同的发行版本而变化。

使用 -xarch=v9、-xarch=v9a 或 -xarch=v9b（或等效的 64 位体系结构选项）时，某些库不能用作静态库。

选项 -staticlib=Crun 和 -staticlib=Cstd 不能用于 64 位 Solaris x86 平台。Sun 建议不要在任何平台上静态链接这些库。在某些情况下，静态链接可能会使程序无法正常运行。

另请参见

-library、第 12-9 页的第 12.5 节 “静态链接标准库”

A.2.88 -sync_stdio=[yes|no]

如果您的运行时性能由于 C++ iostreams 和 C stdio 之间的同步而下降，请使用此选项。仅当您在相同的程序中使用 iostreams 写入 cout 以及使用 stdio 写入 stdout 时，才需要同步。C++ 标准要求同步，因此默认情况下 C++ 编译器打开同步。但是，不使用同步时，应用程序性能通常更佳。如果您的程序既不写入 cout 也不写入 stdout，则可以使用选项 -sync_stdio=no 关闭同步。

默认：

如果未指定 -sync_stdio，编译器将其设置为 -sync_stdio=yes。

示例：

考虑以下示例：

```
#include <stdio.h>
#include <iostream>
int main()
{
   std::cout << "Hello ";
   printf("beautiful ");
   std::cout << "world!";
   printf("\n");
}
```

使用同步时，程序自行打印在一行中：

```
Hello beautiful world!
```

不使用同步时，输出会变得杂乱。

警告：

此选项仅对链接可执行文件有效，对库无效。

A.2.89 -temp=*path*

为临时文件定义目录。

该选项设置目录的路径名称，用于存储在编译过程中生成的临时文件。

另请参见

-keeptmp

A.2.90 -template=*opt*[,*opt*...

启用/禁用各种模板选项。

值

opt 必须是下列值之一：

表 A-22 `-template` 值

opt 的值	含义
`[no%]extdef`	[不] 在独立的源文件中搜索模板定义。
`[no%]geninlinefuncs`	[不] 生成未引用的内联成员函数用于显式实例化的类模板。
`[no%]wholeclass`	[不] 实例化整个模板类，而不仅仅是使用的这些函数。必须至少引用类的一个成员，否则编译器不实例化该类的任何成员。

默认

如果未指定 `-template` 选项，则假定为 `-template=no%wholeclass,extdef`。

示例

请考虑以下代码：

```
example% cat Example.cc
     template <class T> struct S {
               void imf() {}
               static void smf() {}
     };

     template class S <int>;

     int main() {
     }
example%
```

指定 -template=geninlinefuncs 时，即使在该程序中不调用 S 的两个成员函数，也会在对象文件中生成这两个函数。

```
example% CC -c -template=geninlinefuncs Example.cc
example% nm -C Example.o

Example.o:

[Index] Value Size Type  Bind  Other Shndx Name
[5]       0   0    NOTY  GLOB  0     ABS   __fsr_init_value
[1]       0   0    FILE  LOCL  0     ABS   b.c
[4]      16   32   FUNC  GLOB  0     2     main
[3]     104   24   FUNC  LOCL  0     2     void S<int>::imf()
                                           [__1cBS4Ci_Dimf6M_v_]
[2]     64    20   FUNC  LOCL  0     2     void S<int>::smf()
                                           [__1cBS4Ci_Dsmf6F_v_]
```

另请参见

第 7-2 页的第 7.2.2 节 “整个类实例化”、第 7-7 页的第 7.5 节 “模板定义搜索”

A.2.91 -time

与 -xtime 相同。

A.2.92 -U*name*

删除预处理程序符号 *name* 的初始化定义。

该选项会删除宏符号 *name* 的所有初始化定义，该符号在命令行上通过 -D 创建，包括了 CC 驱动程序隐式放置的内容。该选项对任何其他预定义的宏和源文件中的宏定义都没有影响。

要查看 CC 驱动程序置于命令行上的 -D 选项，请将 -dryrun 选项增加到命令行。

示例

以下命令取消了预定义符号 __sun 的定义。foo.cc 中的预处理程序语句（例如 #ifdef(__sun)）会判断出该符号是未定义的。

```
example% CC -U__sun foo.cc
```

交互

您可以在命令行上指定多个 -U 选项。

所有的 -U 选项都在任何 -D 选项出现之后处理。也就是说，如果在命令行上为 -D 和 -U 指定了相同的 *name*，则 *name* 是未定义的，这与选项出现的顺序无关。

另请参见

-D

A.2.93 -unroll=*n*

与 -xunroll=*n* 相同。

A.2.94 -V

与 -verbose=version 相同。

A.2.95 -v

与 -verbose=diags 相同。

A.2.96 -vdelx

已过时，不使用。

仅兼容模式 (-compat[=4])：

对于使用 delete[] 的表达式，该选项生成对运行时库函数 _vector_deletex_ 的调用，而不生成对 _vector_delete_ 的调用。函数 _vector_delete_ 使用两个参数：要删除的指针以及每个数组元素的大小。

函数 _vector_deletex_ 的行为与 _vector_delete_ 的行为相同，但前者使用第三个参数：类的析构函数的地址。第三个参数不用于该函数，而是供第三方供应商使用。

默认

该编译器为使用 delete[] 的表达式生成对 _vector_delete_ 的调用。

警告

这是在以后的发行版本中要删除的废弃选项。除非您已从第三方供应商购买了某些软件且供应商推荐使用该选项，否则请不要使用该选项。

A.2.97 -verbose=*v*[,*v*_]

控制编译器详细内容。

值

v 必须是下列值之一：

表 A-23 -verbose 值

v 的值	含义
[no%]diags	[不] 为每个编译传递打印命令行。
[no%]template	[不] 打开模板实例化 verbose 模式 （有时称为 “检验”模式）。verbose 模式显示编译过程中实例化的每个阶段。
[no%]version	[不] 指定 CC 驱动程序打印调用程序的名称和版本编号。
%all	调用以上所有内容。
%none	-verbose=%none 与 -verbose=no%template,no%diags,no%version 相同。

默认

如果未指定 -verbose，则假定为 -verbose=%none。

交互

该选项会累积而不覆盖。

A.2.98 +w

标识可能产生不可预料结果的代码。如果函数过大而无法内联或声明的程序元素未被使用，那么 +w 选项不再生成警告。该警告不指定源码中的真正问题，因此不适合某些开发环境。将这些警告从 +w 删除就可以在这些环境下更主动的使用 +w。这些警告仍然可以与 +w2 选项一起使用。

该选项生成如下关于有问题构造的其他警告：

- 不可移植
- 可能出错
- 低效

默认

如果未指定 +w，则编译器警告构造极可能出现问题。

交互

某些 C++ 标准头文件在使用 +w 编译时会产生警告。

另请参见

-w、 +w2

A.2.99 +w2

发出由 +w 提供的所有警告，以及关于技术违规的附加警告。这类技术违规可能是无害的，但可能会降低系统的最大可移植性。

+w2 选项不再发出关于依赖于实现系统头文件中构造的警告。因为系统头文件是实现，所以发出警告是不合适的。从 +w2 删除这些警告可以更主动地使用该选项。

警告

某些 Solaris 操作系统和 C++ 标准头文件在使用 +w2 编译时会产生警告。

另请参见

+w

A.2.100 -w

禁止大多数警告消息。

该选项使编译器**不**打印警告消息。不过某些警告（尤其是严重的记时错误）不能被禁止。

另请参见

+w

A.2.101 -Xm

与 -features=iddollar 相同。

A.2.102 -xa

为文件配置生成代码。

如果在编译时进行设置，则 TCOVDIR 环境变量会指定覆盖 (.d) 文件所在的目录如果该变量未设置，则覆盖 (.d) 文件仍与 source 文件保存在同一目录中。

使用该选项是为了向后兼容旧的覆盖文件。

交互

-xprofile=tcov 选项和 -xa 选项在单一可执行文件中兼容。也就是说，您可以链接程序，其中包含了用 -xprofile=tcov 编译过的某些文件和用 -xa 编译的其他文件。您不能同时用这两个选项来编译单一文件。

-xa 选项与 -g 不兼容。

警告

如果单独进行编译和链接且使用 -xa 编译，就要确保也使用 -xa 进行链接，否则会产生意外的结果。

另请参见

`-xprofile=tcov`、`tcov`(1) 手册页、《程序性能分析工具》。

A.2.103 `-xalias_level[=n]`

(SPARC) 指定以下命令时， C++ 编译器可以执行基于类型的别名分析和优化：

- `-xalias_level[=n]`

  其中 *n* 是 `any`、`simple` 或 `compatible`。

- `-xalias_level=any`

  在此分析级别上，编译器假定任何类型都可以为其他类型起别名。不过尽管只是假定，但还是可以执行某些优化。

- `-xalias_level=simple`

  编译器假定简单的类型没有别名。具体来说就是具有动态类型的存储对象，这些类型是以下简单类型之一：

char	short int	long int	浮点
signed char	unsigned short int	unsigned long int	双精度
unsigned char	int	long long int	长双精度
wchar_t	unsigned int	unsigned long long int	枚举类型
数据指针类型	函数指针类型	数据成员指针类型	函数成员指针类型

  只能通过以下类型的左值访问：

  - 对象的动态类型
  - 对象动态类型的 `constant` 或 `volatile` 限定版本，与对象动态类型相对应的带符号或不带符号的类型
  - 与对象动态类型的 `constant` 或 `volatile` 限定版本相对应的带符号或不带符号的类型
  - 在其成员（包括递归的子集成员或包含的联合）中包括上述类型的聚集或联合类型
  - char 或 unsigned char type

- `-xalias_level=compatible`

  编译器假定布局不兼容类型没有别名。存储对象只能通过以下类型的左值访问：

  - 对象的动态类型
  - 对象动态类型的 `constant` 或 `volatile` 限定版本，与对象动态类型相对应的带符号或不带符号的类型

- 与对象动态类型的 constant 或 volatile 限定版本相对应的带符号或不带符号的类型
- 在其成员 （包括递归的子集成员或包含的联合）中包括上述类型的聚集或联合类型
- 对象动态类型的基类类型 （可能用 constant 或 volatile 限定）
- char 或 unsigned char type。

编译器假定所有引用的类型都与相应存储对象的动态类型是布局兼容的。两种类型在以下情况下是布局兼容的：

- 如果两个类型相同，则这两个类型就是布局兼容的类型。
- 如果两个类型仅在 constant 或 volatile 限定中不同，则这两个函数是布局兼容的类型。
- 每个存在的有符号整数类型对应 （但是不相同）一个无符号整数类型。这些对应的类型是布局兼容的。
- 如果两个枚举类型具有相同的基础类型，则它们是布局兼容的。
- 如果两个 Plain Old Data (POD) 结构类型具有相同数量的成员，并且对应的成员（按顺序）具有布局兼容的类型，那么这两个结构类型是布局兼容的。
- 如果两个 POD 联合类型具有相同数量的成员，并且对应的成员 （按顺序）具有布局兼容的类型，那么这两个联合类型是布局兼容的。

在某些情况下具有存储对象动态类型的引用可能是非布局兼容的：

- 如果 POD 联合包含了两个或两个以上共享通用初始序列的 POD 结构，且 POD 联合对象当前包含了其中一个 POD 结构，就可以检查任何 POD 结构的通用初始部分。对包含一个或多个初始成员的序列来说，如果相应成员具有布局兼容类型（适用于位字段）和相同宽度，则两个 POD 结构共享一个通用初始序列。
- 指向 POD 结构对象的指针 （使用 reinterpret_cast 适当转换）将指向该结构的初始成员，而如果该成员是位字段则指向该结构所在的单元。

默认

如果未指定 -xalias_level，则编译器将选项设置为 -xalias_level=any。如果指定了 -xalias_level 但未提供任何值，则编译器将选项设置为 -xalias_level=compatible。

交互

编译器在 -xO2 及低于它的优化级别下不执行基于类型的别名分析。

警告

如果正使用 reinterpret_cast 或等效的旧式类型转换，那么程序就可能违反分析的假定。此外，联合类型也违反了分析的假定，如以下示例所示。

```
union bitbucket{
  int i;
  float f;
};

int bitsof(float f){
bitbucket var;
var.f=3.6;
return var.i;
}
```

A.2.104 -xar

创建归档库。

生成使用模板的 C++ 归档文件时，多数情况下有必要将这些模板函数包括在该归档文件中。这些函数已在模板数据库中被实例化。使用该选项可以将这些模板自动增加到所需的归档文件中。

值

指定 -xar 调用 ar -c-r 并重新创建归档文件。

示例

以下命令行归档包含在库和对象文件中的模板函数。

```
example% CC -xar -o libmain.a a.o b.o c.o
```

警告

不在命令行上从模板数据库增加 .o 文件。

不直接使用 ar 命令生成归档文件。使用 CC -xar 以确保模板实例化自动包括在归档文件中。

另请参见

ar(1)、第 16 章

A.2.105 -xarch=*isa*

指定目标指令集架构 (*ISA)*。

该选项将编译器生成的代码限制到特定指令集架构的指令。该选项不保证使用任何目标特定的指令。不过，使用该选项会影响二进制程序的可移植性。

SPARC 值

表 A-24 给出了 SPARC 平台上每个 -xarch 关键字的详细信息。

表 A-24 SPARC 平台的 -xarch 值

isa 的值	含义
generic	**在多数系统上生成高性能的 32 位对象二进制文件**。这是默认行为。该选项在多数处理器上使用高性能的最佳指令集，同时不降低处理器的主要性能。如果需要，在新的发行版本中会调整“最佳”指令集的定义。目前，它等效于 -xarch=v7。
generic64	**在多数 64 位平台架构上生成高性能的 64 位对象二进制文件**。该选项在具有 64 位内核的 Solaris 操作系统中使用高性能的最佳指令集，同时不降低操作环境的性能。如果需要，在新的发行版本中会调整“最佳”指令集的定义。目前，它等效于 -xarch=v9。
native	**在本系统上生成高性能的 32 位对象二进制文件**。这是 -fast 选项的默认值。编译器在处理器运行的系统上选择适当的设置。
native64	**在本系统上生成高性能的 64 位对象二进制文件**。编译器在处理器运行的系统上为生成系统的 64 位二进制选择适当的设置。
v7	**编译用于 SPARC-V7 ISA**。（已废弃）当前的 Solaris 操作系统不再支持 SPARC V7 体系结构，并且使用此选项编译的程序在当前平台上的运行速度较慢。
v8a	**编译用于 SPARC-V8 ISA V8a 版本**。根据定义，V8a 意味着 V8 ISA，但不包括 fsmuld 指令。该选项在 V8a ISA 上使编译器生成高性能代码。 示例：基于 microSPARC I 芯片架构的任何系统
v8	**编译用于 SPARC-V8 ISA**。在 V8 架构上使编译器生成高性能代码。 示例：SPARCstation 10

表 **A-24** SPARC 平台的 -xarch 值 (续)

isa 的值	含义
v8plus	**编译用于 SPARC-V9 ISA 的 V8plus 版本。** 这是默认设置。根据定义， V8plus 意味着 V9 ISA，但只限于由 V8plus ISA 规范所定义的 32 位子集，而不包括可视化指令集 (VIS) 和特定实现的 ISA 扩展。 • 该选项在 V8plus ISA 上使编译器生成高性能代码。 • 产生的对象代码是 SPARC-V8+ ELF32 格式且只在 Solaris UltraSPARC 环境下执行 （不能在 V7 或 V8 处理器上运行）。 示例：基于 UltraSPARC 芯片架构的任何系统
v8plusa	**编译用于 SPARC-V9 ISA 的 V8plusa 版本。** 根据定义， V8plusa 意味着 V8plus 架构加可视化指令集 (VIS) 版本 1.0 和 UltraSPARC 扩展。 • 该选项使编译器在 UltraSPARC 架构上生成高性能代码，但只限于 V8plus 规范定义的 32 位子集。 • 产生的对象代码是 SPARC-V8+ ELF32 格式且只在 Solaris UltraSPARC 环境下执行 （不能在 V7 或 V8 处理器上运行）。 示例：基于 UltraSPARC 芯片架构的任何系统
v8plusb	**编译用于具有 UltraSPARC III 扩展的 SPARC-V8plus ISA V8plusb 版本。**使编译器为 UltraSPARC 架构生成对象代码，加上具有 UltraSPARC III 扩展的可视化指令集 (VIS) 版本 2.0。 • 产生的对象代码是 SPARC-V8+ ELF32 格式且只在 Solaris UltraSPARC III 环境下执行。 • 用该选项进行编译时要在 UltraSPARC III 架构上使用高性能的最佳指令集。

表 **A-24** SPARC 平台的 -xarch 值 (续)

isa 的值	含义
v9	**编译用于 SPARC-V9 ISA**。在 V9 SPARC 架构上使编译器生成高性能代码。 • 产生的 .o 对象文件是 ELF64 格式且只能与相同格式的其他 SPARC-V9 对象文件链接。 • 产生的可执行文件只能在 UltraSPARC 处理器上运行，该处理器运行具有 64 位内核的 64 位 Solaris 操作系统。 • 在启用 64 位的 Solaris 系统下编译时，只有 -xarch=v9 可用。
v9a	**编译用于具有 UltraSPARC 扩展的 SPARC-V9 ISA**。 将可视化指令集 (VIS) 和 UltraSPARC 处理器的特定扩展增加到 SPARC-V9 ISA，并使编译器在 V9 SPARC 架构上生成高性能代码。 • 产生的 .o 对象文件是 ELF64 格式且只能与相同格式的其他 SPARC-V9 对象文件链接。 • 产生的可执行文件只能在 UltraSPARC 处理器上运行，该处理器运行具有 64 位内核的 64 位 Solaris 操作系统。 • 在启用 64 位的 Solaris 操作系统下编译时，只有 -xarch=v9a 可用。
v9b	**编译用于具有 UltraSPARC III 扩展的 SPARC-V9 ISA**。 将 UltraSPARC III 扩展和 VIS 版本 2.0 增加到 SPARC-V9 ISA 的 V9a 版本。用该选项进行编译时要在 Solaris UltraSPARC III 环境下使用高性能的最佳指令集。 • 产生的对象代码是 SPARC-V9 ELF64 格式且只能与相同格式的其他 SPARC-V9 对象文件链接。 • 产生的可执行文件只能在 UltraSPARC III 处理器上运行，该处理器运行具有 64 位内核的启用 64 位的 Solaris 操作系统。 • 在启用 64 位的 Solaris 操作系统下编译时，只有 -xarch=v9a 可用。

另请注意：

- SPARC 指令集体系结构 V8 和 V8a 均是二进制兼容的。
- 由 v8plus 和 v8plusa 编译的对象二进制文件 (.o) 可以同时链接和执行，但必须在与 SPARC V8plusa 兼容的平台上运行。
- 由 v8plus、v8plusa 和 v8plusb 编译的对象二进制文件 (.o) 可以同时链接和执行，但必须在与 SPARC V8plusb 兼容的平台上运行。
- -xarch 值 v9、v9a 和 v9b 只能在 UltraSPARC 64 位 Solaris 操作系统中使用。
- 由 generic64、native64、v9 和 v9a 编译的对象二进制文件 (.o) 可以同时链接和执行，但必须在与 SPARC V9a 兼容的平台上运行。
- 由 generic64、native64、v9、v9a 和 v9b 编译的对象二进制文件 (.o) 可以同时链接和执行，但必须在与 SPARC V9b 兼容的平台上运行。

对于任何特定选择，生成的可执行文件在早期架构中可能运行更缓慢。此外，虽然在多数指令集架构中都可以使用四精度 （REAL*16 和 long double）浮点指令，但编译器不在它生成的代码中使用这些指令。

x86 值：

表 A-25 给出了 x86 平台上每个 -xarch 标志的详细信息。

表 A-25 x86 平台的 -xarch 值

isa 的值	含义
386	在该发行版本中 generic 和 386 是等效的。
amd64	为 64 位 Solaris x86 平台进行编译。
amd64a	将 AMD 扩展（3DNow!、3DNow! 扩展和 MMX 扩展）添加到 AMD64 体系结构中，并生成 64 位 ELF 格式二进制文件。
generic	编译以便在多数系统上达到较高的性能。这是默认设置。该选项在多数处理器上使用高性能的最佳指令集，同时不降低处理器的主要性能。如果需要，在新的发行版本中会调整 “最佳”指令集的定义。
generic64	在多数 64 位平台体系结构上生成高性能的 64 位对象二进制文件。 该选项在具有 64 位内核的 Solaris 操作系统中使用高性能的最佳指令集，同时不降低操作环境的性能。如果需要，在新的发行版本中会调整“最佳”指令集的定义。
native	为了在此系统上获得良好性能而进行编译。这是 -fast 选项的默认值。编译器为当前正编译的系统处理器选择适当的设置。
pentium_pro	将指令集限制到 pentium_pro 体系结构。
pentium_proa	将 AMD 扩展 （3DNow!、 3DNow! 扩展和 MMX 扩展）添加到 32 位 pentium_pro 体系结构中。
sse	将 SSE 指令集添加到 pentium_pro 体系结构。
ssea	将 AMD 扩展 （3DNow!、 3DNow! 扩展和 MMX 扩展）添加到 32 位 pentium_pro 体系结构中。
sse2	将 SSE2 指令集添加到 pentium_pro 体系结构。
sse2a	将 AMD 扩展 （3DNow!、 3DNow! 扩展和 MMX 扩展）添加到 32 位 pentium_pro 体系结构中。

警告

为了在兼容 Solaris x86 SSE 或 SSE2 平台上运行而将 -xarch 设置为 sse (Pentium 3) 或 sse2 (Pentium 4) 以进行编译的程序只能在支持 SSE 或 SSE2 的平台上运行。

将 -xarch 设置为 pentium_proa 以进行编译的程序必须在支持 AMD 3DNow! 和 3DNow! 扩展的平台上运行。

通过将 -xarch 设置为 ssea 或 sse2a 而编译的程序必须在支持 AMD 3DNow!、3DNow! 扩展的平台以及支持 SSE 或 SSE2 的平台上运行。

从 Solaris 9 4/04 开始的操作系统发行版本在兼容 Pentium 3 或 4 的平台上支持 SSE/SSE2。早期版本的 Solaris 操作系统不支持 SSE/SSE2。

类似地，使用 `-xarch=amd64` 为 Solaris x86 AMD64 平台编译的程序必须在支持 AMD 64 位体系结构的平台上运行。请注意， AMD64 体系结构支持 SSE/SSE2。

使用 `-xarch=amd64a` 编译的程序必须在支持 AMD 64 位体系结构以及 AMD 3DNOW! 和 AMD 3DNow! 扩展的平台上运行。

操作系统将对使用这些专用 `-xarch` 硬件标志编译和生成的程序二进制文件进行检查，看其是否在适当启用的硬件上运行。这种验证适用于在 Solaris 10 操作系统上运行的 Sun Studio 11。

在 Solaris 10 以前的系统上，不执行任何验证，而是由用户来负责确保将使用这些标志生成的对象部署在相应的硬件上。

如果在不支持相应的功能或指令集扩展的平台上运行使用这些 `-xarch` 标志编译的程序，则会在不出现任何显式警告消息的情况下发生“非法指令”错误、段故障或错误结果。这一警告还会扩展到采用 .il 内联汇编语言函数或 `__asm` 汇编程序代码（采用 SSE、SSE2、 AMD 64 和 AMD 3DNow! 指令以及 AMD 3DNow! 扩展）的程序。

如果在单独的步骤中编译和链接，请始终使用编译器和相同的 `-xarch` 设置进行链接，以确保链接正确的启动例程。

x86 上的运算结果可能与 SPARC 上的结果不同，这是由于 x86 80 字节浮点寄存器造成的。要最大限度地减少这些差异，请使用 `-fstore` 选项，或者使用 `-xarch=sse2` 进行编译 （如果硬件支持 SSE2 ）。

SPARC 默认设置

目前， C++ 编译器生成其代码的默认体系结构是 `v8plus` (UltraSPARC)。以后的发行版将取消对 `v7` 的支持。

新的默认设置可以使几乎所有当前使用的计算机的运行时性能更佳。但是，在默认情况下，设计用于在 UltraSPARC 之前的计算机上进行部署的应用程序将不再在那些计算机上执行。使用 `-xarch=v8` 编译可以确保这些应用程序在那些计算机上执行。

如果要在 `v8` 系统上部署，则必须在每个编译器命令行以及任何链接时命令上显式指定选项 `-xarch=v8`。提供的系统库将可以在 `v8` 体系结构上运行。

如果要在 `v7` 系统上部署，则必须在每个编译器命令行以及任何链接时命令上显式指定选项 `-xarch=v7`。提供的系统库将使用 `v8` 指令集。对于此发行版来说，唯一支持 `v7` 的操作系统是 Solaris 8 OS 发行版。遇到 `v8` 指令时，Solaris 8 操作系统会在软件中解释指令。程序会运行，但性能将下降。

x86 默认设置

对于 x86，-xarch 默认设置为 generic。请注意，x86 上的 -fast 扩展为 -xarch=native。该选项将编译器生成的代码限制到特定指令集架构的指令。该选项不保证使用任何目标 - 特定的指令。不过，使用该选项会影响二进制程序的可移植性。

如果单独进行编译和链接，请确保在两个步骤中为 -xarch 指定了相同的值。

交互

尽管该选项可单独使用，但它是 -xtarget 选项的扩展部分并且可用于覆盖由特定 -xtarget 选项设置的 -xarch 值。例如，-xtarget=ultra2 可扩展到 -xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1。在下面的命令中，-xarch=v8plusb 覆盖了由 -xtarget=ultra2 的扩展设置的 -xarch=v8plusa。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

-compat[=4] 不能与 -xarch=generic64、 -xarch=native64、 -xarch=v9、-xarch=v9a 或 -xarch=v9b 同时使用。

警告

如果带优化使用该选项，那么在指定架构上适当选择就可以提供高性能的可执行文件。但如果选择不当就会导致性能的严重降级，或导致在预定目标平台上无法执行二进制程序。

如果单独进行编译和链接，请确保在两个步骤中为 -xarch 指定了相同的值。

A.2.106 -xautopar

注 – 该选项不接受 OpenMP 并行化指令， Sun 特定的 MP pragma 已过时并且不再受支持。有关该标准指令的迁移信息，请参见 《OpenMP API 用户指南》。

(SPARC) 为多个处理器打开自动并行化。进行依赖性分析 （分析循环以了解迭代间数据依赖性）和循环重构。如果优化级别不是 -xO3 或更高级别，则将优化级别提高到 -xO3 并发出警告。

如果要执行自己的线程管理，请不要使用 -xautopar。

要使执行速度更快，则该选项需要多处理器系统。在单处理器系统上，最终的二进制程序通常运行较慢。

要确定您使用的处理器数量，请使用 psrinfo 命令：

```
% psrinfo
0       on-line since 01/12/95 10:41:54
1       on-line since 01/12/95 10:41:54
3       on-line since 01/12/95 10:41:54
4       on-line since 01/12/95 10:41:54
```

要请求多个处理器，请设置 PARALLEL 环境变量。默认值为 1。

- 请求的处理器数量不能超出可用的处理器数量。
- PARALLEL 环境变量的值不能大于单用户计算机上的处理器的数量。在多用户计算机上， PARALLEL 环境变量值应该小于处理器数量，以避免计算机过载。

如果您使用 -xautopar 并在**一个**步骤中进行编译和链接，则链接将自动包括微任务库和线程安全 C 运行时库。如果您使用 -xautopar 并分别在**单独的**步骤中进行编译和链接，则您还必须链接 -xautopar。

另请参见

第 A-113 页的第 A.2.146 节 "-xopenmp[=i]"

A.2.107 -xbinopt={prepare|off}

(SPARC) 指示编译器准备二进制文件以随后进行优化、转换和分析，请参见 binopt (1)。该选项可用于生成可执行文件或共享对象。如果在单独的步骤中进行编译，则 -xlinkopt 必须同时出现在编译和链接步骤中：

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

如果某些源代码不能用于编译，则仍然可以使用该选项来编译代码的其余部分。在后面的创建最终二进制文件的链接步骤中必须使用该选项。在此类情况下，只有通过该选项编译的代码才能进行优化、转换或分析。

默认

默认值为 -xbinopt=off 。

交互

该选项必须用于 -xO1 或更高的优化级别才有效。在使用此选项进行生成时，二进制文件大小略有增加。

使用 -xbinopt=prepare 和 -g 编译会将调试信息包括在内，从而增加了可执行文件的大小。

A.2.108 -xbuiltin[={%all|%none}]

启用或禁用标准库调用的最佳优化。

默认情况下，编译器会将标准库头文件中声明的函数视为普通函数。不过，编译器会将某些函数识别为“内部”或“内建”。视为内建函数时，编译器可以生成更有效的代码。例如，编译器可以识别无副作用的函数，且通常为给定的相同输入返回相同的输出。编译器可将部分函数直接生成为内联函数。有关如何阅读对象文件中的编译器注释以确定编译器实际为哪些函数进行替换的说明，请参见 er_src(1) 手册页。

-xbuiltin=%all 选项会请求编译器识别尽可能多的内建标准函数。所识别函数的确切列表在不同的编译器代码生成器版本中各不相同。

-xbuiltin=%none 选项会生成默认编译器行为，编译器对内建函数不执行任何优化。

默认

如果未指定 -xbuiltin 选项，则编译器假定 -xbuiltin=%none。

如果仅指定了 -xbuiltin，则编译器假定 -xbuiltin=%all。

交互

宏 -fast 的扩展包括了 -xbuiltin=%all。

示例

下面的编译器命令请求标准库调用的特殊处理。

```
example% CC -xbuiltin -c foo.cc
```

下面的编译器命令请求不对标准库调用进行特别处理。注意，宏 -fast 的扩展包括了 -xbuiltin=%all。

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

A.2.109 -xcache=*c*

SPARC：定义优化器使用的缓存属性。

该选项指定了优化器可以使用的缓存属性，不保证使用每个特定的缓存属性。

注 – 该选项可单独使用，但它是 -xtarget 选项的部分扩展，主要用途是覆盖 -xtarget 选项提供的值。

此发行版本引入了一个可选属性 [/ti]，它设置了可以共享缓存的线程数。

值

c 必须是下列值之一：

表 A-26 -xcache 的值

c **的值**	**含义**
generic	该默认值指示了编译器在多数 x86 和 SPARC 处理器上使用缓存属性来获得高性能，而不会降低任何处理器的性能。 如果需要，在新的发行版本中会调整最佳定时属性。
native	设置在主机环境中最佳性能的参数。
s1/*l1*/*a1*[/*t1*]	定义级别 1 缓存属性
s1/*l1*/*a1*[/*t1*]:*s2*/*l2*/*a2*[/*t2*]	定义级别 1 和 2 缓存属性
s1/*l1*/*a1*[/*t1*]:*s2*/*l2*/*a2*[/*t2*]:*s3*/*l3*/*a3*[/*t3*]	定义级别 1、2 和 3 缓存属性

缓存属性的定义 *si/li/ai/ti* 如下：

属性	**定义**
si	*i* 级别的数据缓存**大小**以千字节为单位
li	*i* 级别的数据缓存**行大小**以字节为单位
ai	*i* 级别的数据缓存**关联**

例如， *i*=1 指定了级别 1 缓存属性 *s1/l1/a1*。

默认

如果未指定 `-xcache` ，则默认假定为 `-xcache=generic`。该值指示了编译器在多数 SPARC 处理器上使用缓存属性来获得高性能，而不降低任何处理器的性能。

如果没有指定 `t` 值，则默认值为 1。

示例

`-xcache=16/32/4:1024/32/1` 指定了：

级别 **1** 缓存具有	级别 **2** 缓存具有
16 千字节	1024 千字节
32 字节行大小	32 字节行大小
4 方向关联	指示映射关联

另请参见

`-xtarget=`*t*

A.2.110 `-xcg[89|92]`

已废弃，不要使用此选项。当前的 Solaris 操作系统软件不再支持 SPARC V7 体系结构。使用此选项编译生成的代码在当前的 SPARC 平台中运行较慢。请改用 `-O`，并使用 `-xarch`、`-xchip` 和 `-xcache` 的编译器默认设置。

A.2.111 `-xchar[=`*o*`]`

有些系统将字符定义成无符号类型，该选项使得迁移这些系统上的代码变得容易。如果不是从这样的系统中迁移，最好不要使用该选项。只有那些依赖字符类型符号的程序才需要重写，它们要改写成显式指定有符号或者无符号。

值

您可以用下列值之一来替换 *o*：

表 A-27 `-xchar` 值

值	含义
`signed`	将声明为字符的字符常量和变量视为带符号的。这会影响编译的代码的行为，而不影响库例程的行为。
`s`	等价于 `signed`
`unsigned`	将声明为字符的字符常量和变量视为无符号的。这会影响编译的代码的行为，而不影响库例程的行为。
`u`	等效于 `unsigned`

默认

如果未指定 `-xchar`，则编译器假定 `-xchar=s`。

如果指定了 `-xchar` 但未指定值，则编译器假定 `-xchar=s`。

交互

`-xchar` 选项会更改用 `-xchar` 编译的代码中类型 char 的值范围。该选项不更改任何系统例程或头文件中类型 char 的范围。具体来讲，指定选项时不更改 `limits.h` 定义的 `CHAR_MAX` 和 `CHAR_MIN` 的值。因此， `CHAR_MAX` 和 `CHAR_MIN` 不再表示无格式字符中可编码的值的范围。

警告

如果使用 `-xchar`，则在将字符与预定义的系统宏进行比较时要特别小心，原因是宏中的值可能已有符号。任何返回错误代码而且可以用宏来访问错误代码的例程通常是这样的。错误代码一般是负值，因此在将字符与宏中的值进行比较时，结果始终为假。负数永远不等于无符号类型的值。

强烈建议不要使用 `-xchar` 为从库输出的任何接口编译例程。 Solaris ABI 将类型 char 指定为带符号的，并且系统库也按此指定。还未使用系统库对将 char 指定为无符号的效果进行广泛测试。如果不使用该选项，那么请修改您的代码使其不依赖于类型 char 是否带符号。类型 char 的符号因不同的编译器和操作系统而不同。

A.2.112 -xcheck[=*i*]

SPARC：使用 -xcheck=stkovf 进行编译将增加对单线程程序中的主线程的栈溢出的运行时检查，并对多线程程序中的从属线程栈的栈溢出执行运行时检查。如果检测到堆栈溢出，则生成 SIGSEGV。如果应用程序需要以与处理其他地址空间违规不同的方法，来处理堆栈溢出引起的 SIGSEGV，那么请参见 sigaltstack(2)。

值

i 必须是下列值之一：

表 A-28 -xcheck 值

值	含义
%all	执行全部检查。
%none	不执行检查。
stkovf	打开堆栈溢出检查。
no%stkovf	关闭堆栈溢出检查。

默认

如果未指定 -xcheck，则编译器默认为 -xcheck=%none。

如果指定没有任何参数的 -xcheck，编译器默认为 -xcheck=%none。

在命令行上 -xcheck 选项不进行累积。编译器按照上次出现的命令设置标志。

A.2.113 -xchip=*c*

指定供优化器使用的目标处理器。

-xchip 选项通过指定目标处理器来指定定时属性。该选项会影响：

- 指令的顺序 （即调度）
- 编译器使用分支的方法
- 语义上等价于其他指令时使用的指令

注 – 该选项可单独使用，但它是 -xtarget 选项的部分扩展，主要用途是覆盖 -xtarget 选项提供的值。

值

c 必须是下列值之一：

表 A-29 `-xchip` 值

平台	*c* 的值	使用定时属性优化
SPARC	`generic`	可以在多数 SPARC 处理器上获得高性能
	`native`	可以在运行编译器的系统上获得高性能
	`old`	早于 SuperSPARC 处理器的处理器
	`super`	SuperSPARC 处理器
	`super2`	SuperSPARC II 处理器
	`micro`	microSPARC 处理器
	`micro2`	microSPARC II 处理器
	`hyper`	hyperSPARC 处理器
	`hyper2`	hyperSPARC II 处理器
	`powerup`	Weitek PowerUp 处理器
	`ultra`	UltraSPARC 处理器
	`ultra2`	UltraSPARC II 处理器
	`ultra2e`	UltraSPARC IIe 处理器
	`ultra2i`	UltraSPARC IIi 处理器
	`ultra3`	UltraSPARC III 处理器
	`ultra3cu`	UltraSPARC III Cu 处理器
	`ultra3i`	UltraSparc IIIi 处理器。
x86	`generic`	多数 x86 处理器
	`386`	Intel 386 处理器
	`486`	Intel 486 处理器
	`pentium`	Intel Pentium 处理器
	`pentium_pro`	Intel Pentium Pro 处理器
	`pentium3`	Intel Pentium 3 处理器
	`pentium4`	Intel Pentium 4 处理器

默认

在多数 SPARC 处理器中 generic 为默认值，指示编译器使用最佳定时属性以获得高性能，而不降低任何处理器的性能。

A.2.114 -xcode=*a*

SPARC：指定代码地址空间。

注 – 您应该通过指定 -xcode=pic13 或 -xcode=pic32 来生成共享对象。也可以使用 -xarch=v9 -xcode=abs64 和 -xarch=v8、-xcode=abs32 生成可用共享对象，但它们效率很低。用 -xarch=v9、 -xcode=abs32 或 -xarch=v9、 -xcode=abs44 生成的共享对象无法工作。

值

a 必须是下列值之一：

表 A-30 -xcode 值

a 的值	含义
abs32	生成快速但有范围限制的 32 位绝对地址。代码 + 数据 + bss 的大小被限制为 2**32 字节。
abs44	SPARC：生成具有适当速度和范围的 44 位绝对地址。代码 + 数据 + bss 的大小被限制为 2**44 字节。只适用于 64 位架构：-xarch={v9|v9a|v9b}。使用动态 （共享）库时不要使用此值。
abs64	SPARC：生成缓慢但无范围限制的 64 位绝对地址。只适用于 64 位架构：-xarch={v9|v9a|v9|generic64|native64}
pic13	生成快速但有范围限制的位置独立代码（小模型）。等价于 -Kpic。允许在 32 位架构上最多引用 2**11 个唯一的外部符号，而在 64 位架构上可以最多引用 2**10 个。
pic32	生成缓慢但无范围限制的位置独立代码（大模型）。等价于 -KPIC。允许在 32 位架构上最多引用 2**30 个唯一的外部符号，而在 64 位架构上可以最多引用 2**29 个。

要确定使用 -xcode=pic13 还是 -xcode=pic32，请通过使用 elfdump -c 查看全局偏移表 (GOT) 的大小（有关详细信息，请参见 elfdump(1) 手册页）以及段标题 sh_name: .got。 sh_size 值是 GOT 的大小。如果 GOT 小于 8,192 字节，请指定 -xcode=pic13，否则指定 -xcode=pic32。

通常，确定应该如何使用 -xcode 时应遵循以下准则：

- 如果是生成可执行文件，则不应该使用 -xcode=pic13 或 -xcode=pic32。
- 如果是生成仅用于链接到可执行文件的归档库，则不应该使用 -xcode=pic13 或 -xcode=pic32。
- 如果是生成共享库，请先使用 -xcode=pic13，一旦 GOT 大小超过了 8,192 字节，则使用 -xcode=pic32。
- 如果是生成用于链接到共享库的归档库，则只应该使用 -xcode=pic32。

默认

对于 SPARC V8 和 V7 处理器，默认为 -xcode=abs32。

而对于 SPARC 和 UltraSPARC 处理器，当您使用 -xarch={v9|v9a|v9b|generic64|native64} 时，默认为 -xcode=abs64。

在 SPARC 上存在两个具有 -xcode=pic13 和 -xcode=pic32 的名义性能成本：

- 取决于设置为指向表 (_GLOBAL_OFFSET_TABLE_) 的寄存器的入口（该表用于访问共享库全局或静态变量的表），用 -xcode=pic13 或 -xcode=pic32 编译的例程会执行一些附加指令。
- 对全局或静态变量的每次访问都会通过 _GLOBAL_OFFSET_TABLE_ 涉及附加间接内存引用。如果使用 -xcode=pic32 完成编译，则在每个全局和静态内存引用中有两个附加指令。

考虑到以上成本时，由于共享库代码的效果，使用 -xcode=pic13 和 -xcode=pic32 会显著降低系统内存要求。编译 -xcode=pic13 或 -xcode=pic32 的共享库中的每个代码页面可以由使用该库的每个进程共享。只要共享库中的代码页面包含了一个非 pic（即绝对）内存引用，该页面就是不可共享的，而且每次执行使用该库的程序时必须创建该页面的副本。

要辨别是否已经使用 -xcode=pic13 或 -xcode=pic32 编译了 .o 文件，最简单的方法是使用 nm 命令：

```
% nmfile.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

包含位置独立代码的 .o 文件还包含了对 _GLOBAL_OFFSET_TABLE_ 未解决的外部引用，该引用由字母 U 指示。

要决定是使用 -xcode=pic13 还是使用 -xcode=pic32，请用 nm 来标识库中使用或定义的不同全局和静态变量的编号。如果 _GLOBAL_OFFSET_TABLE_ 的大小小于 8,192 字节，就可以使用 -Kpic。否则，必须使用 -xcode=pic32。

警告

在不同的步骤中编译和链接时，在编译步骤和链接步骤中必须使用相同的 -xarch 选项。

A.2.115 -xcrossfile[=*n*]

SPARC：启用跨源文件的优化和内联处理。 -xcrossfile 在编译期间工作并且仅涉及出现在编译命令中的文件。请参考下列命令行示例：

```
example% CC -xcrossfile -xO4 -c f1.cc f2.cc
example% CC -xcrossfile -xO4 -c f3.cc f4.cc
```

跨模块优化发生在文件 f1.cc 和 f2.cc 之间，以及 f3.cc 和 f4.cc 之间。在 f1.cc 和 f3.cc 或 f4.cc 之间不发生优化。

值

n 必须是下列值之一：

表 A-31 -xcrossfile 值

n 的值	含义
0	不执行跨文件优化或跨文件内联。
1	执行多个源文件之间的优化和内联处理。

通常，在命令行上编译器的分析范围限制到每个独立的文件。例如，传递 -xO4 选项时，自动内联被限制到同一源文件中定义和引用的子程序。

编译器使用 -xcrossfile 或 -xcrossfile=1 可以分析在命令行上命名的所有文件，结果就好像这些文件被连接到单一的源文件中。

默认

如果未指定 -xcrossfile，则假定 -xcrossfile=0 且不执行任何跨文件优化或内联。

-xcrossfile 与 -xcrossfile=1 相同。

交互

-xcrossfile 选项只有与 -xO4 或 -xO5 一起使用时才有效。

警告

这种编译所产生的文件由于可能的内联所以是相互依赖的，而且在将这些文件链接到程序时必须作为一个单元使用。如果更改了任何一个例程并重新编译了文件，则必须重新编译所有的文件。因此，使用该选项会影响 makefile 的结构。

另请参见

`-xldscope`

A.2.116 -xdebugformat=[stabs|dwarf]

如在 “DWARF 调试信息格式”中指定的那样，编译器将调试器信息从 stabs 格式迁移到 dwarf 格式。在此发行版本中，默认设置为 `-xdebugformat=stabs`。

如果要维护读取调试信息的软件，现在您可以使用此选项将工具从 stabs 格式转换为 dwarf 格式。

使用此选项可作为一种为移植工具而存取新格式的方法。除非您要维护读取调试器信息的软件，或者特定工具要求使用这些格式之一的调试器信息，否则没有必要使用此选项。

表 A-32 `-xdebugformat` 标志

值	含义
`stabs`	`-xdebugformat=stabs` 生成的调试信息采用 stabs 标准格式。
`dwarf`	`-xdebugformat=dwarf` 生成的调试信息采用 dwarf 标准格式。

如果未指定 `-xdebugformat`，则编译器假定 `-xdebugformat=stabs`。该选项需要一个参数。

此选项影响使用 `-g` 选项记录的数据的格式。即使在没有使用 `-g` 的情况下记录少量调试信息，此选项仍可控制其信息格式。因此，即使不使用 `-g`，`-xdebugformat` 仍有影响。

`dbx` 和性能分析软件理解 stabs 和 dwarf 格式，因此使用此选项对所有工具的功能性并无影响。

注 – 这是过渡性接口，因此发行版本之间会发生更改而不兼容，即使在发行版本更新较少时也是如此。 stabs 或 dwarf 格式的任何特定字段或值的详细资料也在不断改进。

有关详细信息，另请参见 `dumpstabs`(1) 和 `dwarfdump`(1) 手册页。

A.2.117 -xdepend=[yes|no]

(SPARC) 分析循环以了解迭代间数据依赖性并执行循环重构。

循环重构包括循环交换、循环合并、标量替换和“死”数组赋值消除。如果优化级别不是 -xO3 或更高级别，则编译器将优化级别提高到 -xO3 并发出警告。

如果您未指定 -xdepend，则默认值为 -xdepend=no，该值表示编译器不分析循环以了解数据依赖性。如果您指定 -xdepend，但未指定参数，则编译器将此选项设置为 -xdepend=yes，该值表示编译器分析循环以了解数据依赖性。

依赖性分析可能有助于单处理器系统。然而，如果要在单处理器系统上使用 -xdepend，则您不应使用 -xautopar 或 -xexplicitpar。如果使用了二者之一，则对多处理器系统执行 -xdepend 优化。

A.2.118 -xdumpmacros[=*value*[,*value*...]]

要查看宏在程序中如何工作时，请使用这个选项。该选项提供了诸如宏定义、取消定义的宏和宏用法实例的信息，并按宏的处理顺序将输出打印到标准错误 (stderr)。-xdumpmacros 选项在整个文件结尾之前或在 dumpmacros 或 end_dumpmacros pragma 覆盖该选项之前都是有效的。请参见第 B-6 页的第 B.2.5 节 "#pragma dumpmacros"。

值

您可以用以下参数代替 *value*：

表 **A-33** -xdumpmacros 值

值	含义
[no%]defs	[不] 打印所有宏定义
[no%]undefs	[不] 打印所有宏取消定义
[no%]use	[不] 打印关于所用宏的信息
[no%]loc	[不] 打印 defs、 undefs 和 use 的位置 （路径名称和行号）
[no%]conds	[不] 打印在条件指令中宏的使用信息
[no%]sys	[不] 打印系统头文件中所有宏的定义、取消定义的宏和宏的使用信息
%all	将选项设置为 -xdumpmacros=defs,undefs,use,loc,conds,sys。该参数最好与其他参数的 [no%] 形式配合使用。例如， -xdumpmacros=%all,no%sys 将从输出排除系统头文件宏，但仍然提供所有其他宏的信息。
%none	不打印任何宏信息

这些选项值会累积，因此指定 -xdumpmacros=sys -xdumpmacros=undefs 与指定 -xdumpmacros=undefs,sys 具有相同的效果。

注 – 子选项 loc、 conds 和 sys 是 defs、 undefs 和 use 选项的限定符。 loc、 conds 和 sys 本身并不会产生任何效果。例如，-xdumpmacros=loc,conds,sys 是无效的。

默认

如果指定了没有任何参数的 -xdumpmacros，则意味着指定的是 -xdumpmacros=defs,undefs,sys。如果未指定 -xdumpmacros，则默认为 -xdumpmacros=%none。

示例

如果使用选项 -xdumpmacros=use,no%loc，则使用的每个宏名称只打印一次。不过，要了解更多详细信息，请使用选项 -xdumpmacros=use,loc，这样每次使用宏时都可以打印位置和宏的名称。

请参考以下文件 t.c：

```
example% cat t.c
#ifdef FOO
#undef FOO
#define COMPUTE(a, b)  a+b
#else
#define COMPUTE(a,b)  a-b
#endif
int n = COMPUTE(5,2);
int j = COMPUTE(7,1);
#if COMPUTE(8,3) + NN + MM
int k = 0;
#endif
```

以下示例显示了基于 defs、 undefs、 sys 和 loc 参数的文件 t.c 的输出。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_7 1
#define __SUNPRO_CC 0x570
#define unix 1
#define sun 1
#define sparc 1
#define __sparc 1
#define __unix 1
#define __sun 1
#define __BUILTIN_VA_ARG_INCR 1
#define __SVR4 1
#define __SUNPRO_CC_COMPAT 5
#define __SUN_PREFETCH 1
#define FOO 1
#undef FOO
#define COMPUTE(a, b) a + b

example% CC -c -xdumpmacros=defs,undefs,loc -DFOO -UBAR t.c
命令行: #define __SunOS_5_7 1
命令行: #define __SUNPRO_CC 0x570
命令行: #define unix 1
命令行: #define sun 1
命令行: #define sparc 1
命令行: #define __sparc 1
命令行: #define __unix 1
命令行: #define __sun 1
命令行: #define __BUILTIN_VA_ARG_INCR 1
命令行: #define __SVR4 1
命令行: #define __SUNPRO_CC_COMPAT 5
命令行: #define __SUN_PREFETCH 1
命令行: #define FOO 1
命令行: #undef BAR
t.c, line 2:#undef FOO
t.c, line 3:#define COMPUTE(a, b) a + b
```

以下示例说明了 use、 loc 和 conds 参数如何报告文件 t.c 中的宏的行为：

```
example% CC -c -xdumpmacros=use t.c
used macro COMPUTE

example% CC -c -xdumpmacros=use,loc t.c
t.c, line 7:used macro COMPUTE
t.c, line 8:used macro COMPUTE

example% CC -c -xdumpmacros=use,conds t.c
used macro FOO
used macro COMPUTE
used macro NN
used macro MM

example% CC -c -xdumpmacros=use,conds,loc t.c
t.c, line 1:used macro FOO
t.c, line 7:used macro COMPUTE
t.c, line 8:used macro COMPUTE
t.c, line 9:used macro COMPUTE
t.c, line 9:used macro NN
t.c, line 9:used macro MM
```

请参考文件 y.c：

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

以下是基于 y.c 中的宏从 -xdumpmacros=use,loc 产生的输出：

```
example% CC -c -xdumpmacros=use,loc y.c
y.c, line 4:used macro Z
y.c, line 4:used macro Y
y.c, line 4:used macro X
```

另请参见

要覆盖 -xdumpmacros 的作用域时，请使用 dumpmacros pragma 和 end_dumpmacros pragma。

A.2.119 -xe

仅检查语法和语义错误。指定 -xe 时，编译器不生成任何对象代码。-xe 的输出被指向 stderr。

如果不需要编译器生成对象文件，就可以使用 -xe 选项。例如，如果要使用删除代码的方法来确定导致错误消息的原因，使用 -xe 可以加速编辑、编译周期。

另请参见

-c

A.2.120 -xF[=*v*[,*v*...]]

使链接程序能对函数和变量重组以达到最优。

该选项指示编译器将函数和（或）数据变量放置到单独的分段中，这使链接程序（使用链接程序的 -M 选项指定的映射文件中的方向）能将这些段重组以优化程序性能。通常，该优化仅在缺页时间构成程序运行时间的一大部分时才有效。

重组变量有助于解决对运行时性能产生负面影响的以下问题：
- 在内存中存放位置很近的无关变量会造成缓存和页的争用。
- 在内存中存放位置很远的相关变量会造成过大的工作集。
- 未用到的弱变量副本会造成过大的工作集，从而降低有效数据密度。

重组变量和函数以优化性能，需要执行以下操作：

1. 使用 -xF 进行编译和链接。
2. 关于如何生成函数的映射文件请遵循 《程序性能分析工具》手册的说明，而关于如何生成数据的映射文件请遵循 《链接程序和库指南》中的说明。
3. 通过使用链接程序的 -M 选项与新的映射文件重新链接。
4. 在分析器下重新执行以验证是否增强。

值

v 可以是下列值中的一个或多个：

表 A-34 `-xF` 值

值	含义
[no%]func	[不] 将函数分段到单独的段中。
[no%]gbldata	[不] 将全局数据 （具有外部链接的变量）分段到单独的段中。
[no%]lcldata	[不] 将局部数据 （具有内部链接的变量）分段到单独的段中。
%all	分段函数、全局数据和局部数据。
%none	不分段。

默认

如果未指定 `-xF`，则默认为 `-xF=%none`。如果指定了没有参数的 `-xF`，则默认为 `-xF=%none,func`。

交互

使用 `-xF=lcldata` 可以禁止某些地址计算优化，因此如果实验证明它是有效的，则只能使用该标志。

另请参见

`analyzer`(1)、 `debugger`(1)、 `ld`(1) 手册页

A.2.121 -xhelp=flags

显示了对每个编译器选项的简要描述。

A.2.122 -xhelp=readme

显示了联机 `readme` 文件的目录。

在环境变量 `PAGER` 中指定的命令标明了 `readme` 文件的页数。如果未设置 `PAGER`，则默认的分页命令为 `more`。

A.2.123 -xia

SPARC：链接合适的区间运算库并设置适当的浮点环境。

注 – C++ 区间运算库与 Fortran 编译器中实现的区间运算相兼容。

扩展

`-xia` 选项是扩展到 `-fsimple=0 -ftrap=%none -fns=no -library=interval` 的宏。如果使用区间并通过为 `-fsimple`、 `-ftrap`、 `-fns` 或 `-library` 指定不同的标志来覆盖 `-xia` 设置的值，则可能导致编译器出现不正确的行为。

交互

要使用区间运算库，请包括 `<suninterval.h>`。

在使用区间运算库时，必须包括以下库之一：`libC`、 `Cstd` 或 `iostreams`。关于包括这些库的详细信息请参见 `-library`。

警告

如果使用了区间并为 `-fsimple`、 `-ftrap` 或 `-fns` 指定了不同的值，那么您的程序可能会产生错误行为。

C++ 区间运算处于实验阶段且正在改进。不同的发行版本具有不同的功能。

另请参见

C++ Interval Arithmetic Programming Reference, *Interval Arithmetic Solves Nonlinear Problems While Providing Guaranteed Results* (`http://www.sun.com/forte/info/features/intervals.html`), `-library`

A.2.124 -xinline[=*func_spec*[,*func_spec*...]]

指定在 `-xO3` 或更高级别上优化器可以内联哪些用户编写的例程。

值

func_spec 必须是下列值之一。

表 A-35 `-xinline` 值

func_spec 值	含义
`%auto`	在 `-xO4` 或更高的优化级别上启用自动内联此参数告知优化器它可以内联所选择的函数。注意，如果没有 `%auto` 规范，则在命令行上使用 `-xinline=[no%]`*func_name*... 指定显式内联时，自动内联正常关闭。
func_name	强烈请求优化器内联函数。如果未将函数声明为 extern "C"，则必须粉碎 *func_name* 的值。您可以在可执行文件上使用 `nm` 命令来查找粉碎的函数名称。对于已声明为 extern "C" 的函数，编译器不粉碎名称。
`no%`*func_name*	如果将列表上的 `no%` 作为例程的前缀，则会禁止对该例程的内联。关于 *func_name* 的粉碎名的规则也适用于 `no%`*func_name*。

只有使用 `-xcrossfile[=1]`，正在编译文件中的例程才会考虑链接。优化器决定适合内联的例程。

默认

如果未指定 `-xinline` 选项，则编译器假设 `-xinline=%auto`。

如果指定了没有参数的 `-xinline=`，则不内联任何函数，这与优化的级别无关。

示例

要在禁用函数 （该函数声明了 `int foo()`）的内联时启用自动内联，请使用

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

要强烈请求声明为 `int foo()` 的函数的内联，并将所有其他函数作为内联的候选，请使用

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

要强烈请求声明为 int foo() 的函数的内联，且不允许任何其他函数的内联，请使用

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

交互

-xinline 选项对低于 -xO3 优化级别的函数无效。若在 -xO4 或更高级别上，则优化器决定哪些函数应该内联，而无需指定 -xinline 选项。若在 -xO4 或更高级别上，编译器也尝试决定哪些函数内联后可以提高性能。

如果出现以下任一情况，则不内联例程。不会省略任何警告。

- 优化级别低于 -xO3
- 无法找到例程
- 内联无益或不安全
- 源码不在正被编译的文件中，或者如果使用 -xcrossfile[=1]，则源码不在命令行上所命名的文件中

警告

如果使用 -xinline 强制内联一个函数，实际上就会降低性能。

另请参见

-xldscope

A.2.125 -xipo[={0|1|2}]

执行过程间的优化。

-xipo 选项通过调用过程间分析传递，来执行部分程序的优化。与 -xcrossfile 不同的是 -xipo 按链接步骤对所有对象文件执行优化，而且优化不只限于编译命令上的源文件。但是，正如同 -xcrossfile，使用 -xipo 执行的整个程序优化不包括汇编 (.s) 源文件。

-xipo 选项特别适用于编译和链接较大的多文件应用程序。用该标志编译的对象文件具有在这些文件内编译的分析信息，这些信息实现了在源码和预编译的程序文件中的过程间分析。不过，分析和优化只限于用 -xipo 编译的对象文件，而不扩展到库的对象文件或库。

值

-xipo 选项可以具有以下值。

表 A-36 -xipo 值

值	含义
0	不执行过程间的优化
1	执行过程间的优化
2	执行过程间的别名分析和内存分配及布局的优化，以提高缓存的性能

默认

如果未指定 -xipo，则假定为 -xipo=0。

如果仅指定了 -xipo，则假定为 -xipo=1。

示例

以下示例在相同的步骤中编译和链接。

```
example% CC -xipo -xO4 -o prog  part1.cc part2.cc part3.cc
```

优化器在三个源文件之间执行交叉文件内联。该操作在链接的最后一步完成，因此源文件的所有编译不必全在单一的编译中完成，而且这些源文件的编译会覆盖许多独立的编译，其中每个编译都会指定 -xipo 选项。

以下示例在不同的步骤中编译和链接。

```
example% CC -xipo -xO4 -c part1.cc part2.cc
example% CC -xipo -xO4 -c part3.cc
example% CC -xipo -xO4 -o prog  part1.o part2.o part3.o
```

在编译步骤中创建的对象文件具有在这些文件内编译的附加分析信息，这样就可以在链接步骤中执行交叉文件优化。

交互

-xipo 选项要求的最低优化级别为 -xO4。

在同一编译器命令行上不能同时使用 -xipo 选项和 -xcrossfile 选项。

警告

在不同的步骤中编译和链接时，必须在两个步骤中都指定 -xipo 才是有效的。

未使用 -xipo 编译的对象可以与使用 -xipo 编译的对象自由链接。

即使使用 -xipo 进行编译时，库也不参与交叉文件的过程间分析，如以下示例所示。

```
example% CC -xipo -xO4 one.cc two.cc three.cc
example% CC -xar -o mylib.a one.o two.o three.o
...
example% CC -xipo -xO4 -o myprog main.cc four.cc mylib.a
```

在该示例中，在 one.cc、two.cc 和 three.cc 之间，main.cc 和 four.cc 之间执行过程间的优化，而不在 main.cc 或 four.cc 和 mylib.a 中的例程之间执行过程间优化。（第一个编译可能生成有关未定义符号的警告，但仍可执行过程间优化，因为过程间优化是编译和链接的一个步骤。）

由于在多个文件之间执行优化所需的附加信息，-xipo 选项生成了更大的对象文件。不过，该附加信息不会成为最后可执行的二进制文件的一部分。可执行程序大小的增加都是由于执行的附加优化导致的。

A.2.125.1 何时不使用 -xipo=2 过程间分析

在链接步骤中使用对象文件集合时，编译器试图执行整个程序分析和优化。对于该对象文件集合中定义的任何函数或子例程 foo()，编译器作出以下两个假定：

1. **foo() 无法在运行时被在该对象文件集合之外定义的其他例程显式调用。**
2. **来自该对象文件集合中的任何例程的 foo() 调用将不受在该对象文件集合以外定义的不同版本的 foo() 的干预。**

如果假定 1 对于给定的应用程序不成立，则不要使用 -xipo=2 编译

如果假定 2 不成立，则既不要使用 -xipo=1 也不要使用 -xipo=2 编译。

例如，考虑使用您自己的源版本干预函数 malloc() 并使用 -xipo=2 编译。然后，任何库中引用与您的代码链接的 malloc() 的所有函数也必须使用 -xipo=2 编译，并且它们的对象文件将不需要参与链接步骤。由于这对于系统库不大可能，因此不要使用 -xipo=2 编译您的版本的 malloc。

另举一例，假设您使用在两个不同的源文件中的两个外部调用 foo() 和 bar() 生成共享库。然后，假设 bar() 调用 foo()。如果有可能在运行时干预 foo()，则不要使用 -xipo=1 或 -xipo=2 编译 foo() 或 bar() 的源文件。否则，foo() 会内联到 bar() 中，从而导致生成错误的结果。

另请参见

`-xjobs`

A.2.126 `-xjobs=`*n*

指定 `-xjobs` 选项用来设定编译器创建多少个进程来完成它的任务。在多 cpu 机器上，该选项可以减少生成时间。目前，`-xjobs` 只在与 `-xipo` 一起使用时才有效。如果指定 `-xjobs=`*n*，过程间调用优化器在编译不同的文件时会用 *n* 作为它能启动的代码生成器实例的最大数。

值

必须带值来指定 `-xjobs`。否则会发出错误诊断并使编译终止。

通常， *n* 的安全值等于 1.5 乘以可用处理器的数量。由于上下文在产生的作业间切换的开销，使用等于可用处理器数量整数倍的值会降低性能。此外，如果使用很大的数值会耗尽系统资源 （如交换空间）。

默认

出现最合适的实例之前， `-xjobs` 的多重实例在命令行上会互相覆盖。

示例

以下示例在有两个处理器的系统上进行编译时比不使用 `-xjobs` 选项而使用相同命令进行编译时更为快速。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

A.2.127 `-xlang=`*language*[`,`*language*]

包含适当的运行库，并确保指定语言的正确运行时环境。

值

language 必须是 `f77`、 `f90`、 `f95` 或 `c99`。

f90 与 f95 参数是等价的。c99 参数为使用 cc -xc99=%all 编译并使用 \f3CC\f1 链接的对象调用 ISO 9899:1999 C 编程语言行为。

交互

-xlang=f90 和 -xlang=f95 选项表明了 -library=f90，而 -xlang=f77 选项表明了 -library=f77。不过，要进行混合语言链接只有 -library=f77 和 -library=f90 选项是不够的，因为只有 -xlang 选项才能确保正确的运行时环境。

要决定在混合语言链接中使用的驱动程序，请使用下列语言分层结构：

1. C++
2. Fortran 95 （或 Fortran 90）
3. Fortran 77
4. C 或 C99

将 Fortran 95、 Fortran 77 和 C++ 对象文件链接在一起时，请使用最高级语言的驱动程序。例如，使用下列 C++ 编译器命令来链接 C++ 和 Fortran 95 的对象文件。

```
example% CC -xlang=f95 ...
```

要链接 Fortran 95 和 Fortran 77 的对象文件，请使用如下所示的 Fortran 95 驱动程序。

```
example% f95 -xlang=f77 ...
```

在相同的编译器命令中不能同时使用 -xlang 选项和 -xlic_lib 选项。如果正使用 -xlang 并且要在 Sun 性能库中进行链接，那么请使用 -library=sunperf。

警告

不要同时使用 -xnolib 和 -xlang。

如果使用 C++ 对象来混合并行的 Fortran 对象，链接行就必须指定 -mt 标志。

另请参见

-library, -staticlib

A.2.128 -xldscope={*v*}

指定 -xldscope 选项更改用于定义外部符号的默认链接程序作用域。由于更好的隐藏了实现，所以对默认的更改会产生更快速更安全的共享库和可执行文件。

值

v 必须是下列值之一：

表 A-37 -xldscope 值

值	含义
全局	全局链接程序作用域是限制最少的链接程序作用域。对符号的所有引用都绑定到定义符号的第一个动态装入模块中的定义。该链接程序作用域是外部符号的当前链接程序作用域。
符号	符号链接程序作用域比全局链接程序作用域具有更多的限制。将对链接的动态装入模块内符号的所有引用绑定到模块内定义的符号。在模块外部，符号也显示为全局符号。该链接程序作用域对应于链接程序选项 -Bsymbolic。尽管不能与 C++ 库一起使用 -Bsymbolic，但是可以使用不会引起问题的 -xldscope=symbolic。关于链接程序的更多信息，请参见 ld(1)。
hidden	隐藏链接程序作用域具有比符号和全局链接程序作用域更高的限制。将动态装入模块内的所有引用绑定到该模块内的定义。符号在模块外部是不可视的。

默认

如果未指定 -xldscope，则编译器假定 -xldscope=global。如果指定了没有任何值的 -xldscope，编译器就会产生错误。出现最合适的实例之前，该选项的多重实例在命令行上会互相覆盖。

警告

如果要使客户端覆盖库中的函数，就必须确保该库生成期间函数不生成内联。如果使用 -xinline 指定函数名称，如果在可以自动内联的 -xO4 或更高级别进行编译，如果使用内联说明符，或如果正使用跨文件的优化，那么编译器内联函数。

例如，假定库 ABC 具有默认的分配器函数，该函数可用于库的客户端，也可在库的内部使用：

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

如果在 -xO4 或更高级别生成库，则编译器内联库组件中出现的对 ABC_allocator 的调用。如果库的客户端要用自定义的版本替换 ABC_allocator，则在调用 ABC_allocator 的库组件中不能进行该替换。最终程序将包括函数的不同版本。

生成库时，用 __hidden 或 __symbolic 说明符声明的库函数可以内联生成。假定这些库函数不被客户端覆盖。请参见第 4-2 页的第 4.2 节 “线程局部存储”。

用 __global 说明符声明的库函数不应内联声明，并且应该使用 -xinline 编译器选项来防止内联。

另请参见

-xinline、 -xO、 -xcrossfile

A.2.129 -xlibmieee

在异常情况下，使 libm 返回数学例程的 IEEE 754 值。

libm 的默认行为是兼容 XPG。

另请参见

《数值计算指南》

A.2.130 -xlibmil

内联选定 libm 库例程用于优化。

注 – 该选项不影响 C++ 内联函数。

对于部分 libm 库例程存在可用的内联模板。该选项为当前使用的浮点选项和平台选择这些内联模板，生成执行速度最快的可执行文件。

交互

-fast 选项中隐含了这个选项。

另请参见

-fast、《数值计算指南》

A.2.131 -xlibmopt

使用优化数学例程的库。使用此选项时，必须通过指定 -fround=nearest 使用默认的舍入模式。

该选项使用了优化性能的数学例程库，并且通常会生成运行速度更快的代码。这样生成的代码可能与普通数学库生成的代码稍有不同，不同之处通常在最后一位上。

该库选项在命令行上的顺序并不重要。

交互

-fast 选项中隐含了这个选项。

另请参见

-fast、 -xnolibmopt、 -fround

A.2.132 -xlic_lib=sunperf

已过时，不使用。请改为指定 -library=sunperf。有关详细信息，请参见第 A-41 页的 "-library=l[,l...]" 。

A.2.133 -xlicinfo

编译器无提示地忽略该选项。

A.2.134 -xlinkopt[=*level*]

指示编译器在对象文件中的任何优化上生成的可执行文件或动态库上，执行链接时优化。这些优化在链接时通过分析对象二进制代码执行。虽然未重写对象文件，但产生的可执行代码可能与初始对象的代码不同。

必须至少在部分编译命令中使用 -xlinkopt，才能使 -xlinkopt 在链接时有效。优化器仍可以在未使用 -xlinkopt 进行编译的对象二进制文件上执行部分受限的优化。

-xlinkopt 优化出现在编译命令行上的静态库代码，但会跳过出现在命令行上的共享（动态）库代码而不对其进行优化。当生成共享库时 （使用 -G 编译），也可以使用 -xlinkopt。

值

级别必须将执行的优化级别设置为 0、 1 或 2。各优化级别如下所示：

表 A-38 -xlinkopt 值

链接优化器设置	行为
0	禁用链接优化器。（这是默认情况。）
1	在链接时根据控制流分析执行优化，其中包括指令缓冲着色和分支优化。
2	在链接时执行附加的数据流分析，其中包括终止的代码排除和地址计算简化。

如果在不同的步骤中编译，则 -xlinkopt 必须同时出现在编译和链接步骤中：

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

注意，仅当链接编译器时才使用级别参数。在以上示例中，即使编译对象二进制文件时使用的是隐含的级别 1，链接优化器的级别仍然是 2。

默认

不用级别参数来指定的 -xlinkopt 隐含了 -xlinkopt=1。

交互

当编译整个程序并且使用性能分析反馈时，该选项才最有效。配置会显示代码中最常用和最少用的部分并相应地指导优化器集中其努力方向。这对大型应用非常重要，因为链接时执行代码地优化放置可以减少指令的高速缓存缺失。一般来说，编译如下所示：

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

关于使用性能分析反馈的详细信息，请参见第A-128页的第A.2.157节 "-xprofile=p"。

警告

使用 -xlinkopt 编译时，请不要使用 -zcompreloc 链接程序选项。

注意，使用该选项编译会略微延长链接的时间，也会增加对象文件的大小，但可执行文件的大小保持不变。使用 -xlinkopt 和 -g 编译会将调试信息包括在内，从而增加了可执行文件的大小。

A.2.135 -xM

在命名的 C++ 程序上只运行预处理程序，同时请求该程序生成 makefile 依赖性并将结果发送到标准输出 （关于 make 文件和依赖的详细信息，请参见 make(1)）。

示例

例如：

```
#include <unistd.h>
void main(void)
{}
```

生成的输出如下：

```
e.o:e.c
e.o:/usr/include/unistd.h
e.o:/usr/include/sys/types.h
e.o:/usr/include/sys/machtypes.h
e.o:/usr/include/sys/select.h
e.o:/usr/include/sys/time.h
e.o:/usr/include/sys/types.h
e.o:/usr/include/sys/time.h
e.o:/usr/include/sys/unistd.h
```

另请参见

make(1S)（关于 makefile 和依赖性的详细信息）

A.2.136 -xM1

该选项除了不报告 /usr/include 头文件的依赖性，也不报告编译器提供的头文件的依赖性之外，与 -xM 相同。

A.2.137 -xMerge

SPARC：将数据段和文本段合并。

除非使用 ld -N 进行链接，否则对象文件中的数据为只读并在进程间共享。

另请参见

ld (1) 手册页

A.2.138 -xmaxopt[=*v*]

此命令将 pragma opt 的级别设置为指定的级别。*v* 是 off、1、2、3、4、5 之一。默认值为 -xmaxopt=off，该值导致 pragma opt 被忽略。如果指定 -xmaxopt 而不提供参数，则等价于指定 -xmaxopt=5。

如果同时指定了 -xO 和 -xmaxopt，则使用 -xO 设置的优化级别不能超过 -xmaxopt 值。

A.2.139 -xmemalign=*ab*

(SPARC) 使用 -xmemalign 选项用来控制编译器对数据对齐的假设。通过控制可能会出现非对齐内存访问的代码和出现非对齐内存访问时的处理程序，可以更轻松的将程序移植到 SPARC。

指定最大的假定内存对齐和未对齐的数据访问行为。在 *a*（对齐）和 *b*（行为）之间必须有一个值。*a* 指定了最大假定内存对齐，而 *b* 指定了未对齐的内存访问行为。

对于可在编译时决定对齐的内存访问，编译器会为数据对齐生成适当的装入/存储指令序列。

对于不能在编译时决定对齐的内存访问，编译器必须假定一个对齐以生成所需的装入/存储序列。

如果运行时的实际数据对齐小于指定的对齐，则未对齐的访问尝试（内存读取或写入）生成一个陷阱。对陷阱的两种可能响应是

- 操作系统将陷阱转换为 SIGBUS 信号。如果程序无法捕捉到信号，则程序终止。即使程序捕捉到信号，未对齐的访问尝试仍将无法成功。
- 操作系统通过翻译未对齐的访问并将控制返回给程序 （仿佛访问已成功正常结束）来处理陷阱。

值

下表列出了 -xmemalign 的对齐和行为的值

表 A-39 -xmemalign 的对齐和行为值

a		*b*	
1	假定最多 1 字节对齐。	i	解释访问并继续执行。
2	假定最多 2 字节对齐。	s	产生信号 SIGBUS。
4	假定最多 4 字节对齐。	f	仅限 -xarch=v9 变体：SIGBUS，否则解释访问并继续执行。对于所有其他 -xarch 值， f 标志等效于 i。
8	假定最多 8 字节对齐。		
16	假定最多 16 字节对齐		

默认

以下默认值仅适用于 -xmemalign 选项不出现时：

- xmemalgin=8i，对于所有 v8 体系结构。
- xmemalign=8s，对于所有 v9 体系结构。

在出现 -xmemalign 选项但未给出任何值时，默认值为：

- xmemalign=1i，对于所有 -xarch 值。

示例

下表说明了如何使用 -xmemalign 来处理不同的对齐情况。

表 A-40 -xmemalign 示例

命令	情况
-xmemalign=1s	大量未对齐访问导致了自陷处理非常缓慢。
-xmemalign=8i	在发生错误的代码中存在偶然的、有目的的、未对齐的访问。

表 **A-40** -xmemalign 示例

命令	情况
-xmemalign=8s	程序中应该没有任何未对齐访问。
-xmemalin=2s	要检查可能的奇数字节访问。
-xmemalign=2i	要检查可能的奇数字节访问并要程序工作。

A.2.140 -xmodel=[*a*]

通过使用 -xmodel 选项，编译器可以为 Solaris x86 平台修改 64 位对象的格式，且只应为此类对象编译指定此选项。

仅当还指定了 -xarch=generic64、-xarch=amd64 或 -xarch=amd64a 时，该选项才有效。

必须是以下值之一：

表 **A-41** -xmodel 标志

值	含义
small	此选项为小模型生成代码，在此模型中，在链接时已知所执行代码的虚拟地址，并且已知所有符号均位于范围在 0 至 2^31 - 2^24 - 1 之间的虚拟地址。
kernel	为内核模型生成代码，在此模型中，将所有符号定义为位于 2^64 - 2^31 和 2^64 - 2^24 之间的范围内。
medium	为中模型生成代码，在此模型中，没有假定有关数据部分的符号引用的范围。文本部分的大小和地址与小代码模型具有相同的限制。

该选项不是累积的，因此编译器根据命令行中最右侧的 -xmodel 实例来设置模型值。

如果未指定 -xmodel，则编译器假定 -xmodel=small。如果指定不带参数的 -xmodel，则会出现错误。

不必使用此选项来编译所有转换单元。只要确保可以找到要访问的对象就可以编译选定的文件。

A.2.141 -xnativeconnect[=*i*]

注 – 这是 “功能终止”声明。在以后的 Sun Studio 发行版本中，可能会删除该选项。

要将接口信息包括在对象文件及后续共享库内部时，请使用 -xnativeconnect 选项，这样共享库就可以与用 Java™ 编程语言 （Java 代码）编写的代码接口。当使用 -G 生成共享库时，也必须包括 -xnativeconnect。

当用 -xnativeconnect 编译时，可以提供最大的外部可视本地代码接口。“本地连接器工具” (Native Connector Tool， NCT) 能够自动生成 Java 代码和 Java 本地接口 (JNI) 代码。一起使用 -xnativeconnect 和 NCT 可以使 C++ 共享库中的函数能从 Java 代码进行调用。关于如何使用 NCT 的更多信息请参见联机帮助。

值

i 必须是下列值之一：

表 A-42 -xnativeconnect 值

值	含义
%all	生成 -xnativeconnet 各选项所描述的全部不同数据。
%none	不生成 -xnativeconnet 各选项所描述的任何不同数据。
[no%]inlines	强制生成已引用内联函数的行外实例。它提供具有外部可视方法的本机连接器来调用内联函数。调用点中的函数普通内联不受其影响。
[no%]interfaces	强制生成二进制接口描述符 (Binary Interface Descriptors, BIDS)

默认

- 如果未指定 -xnativeconnect，则编译器将选项设置为 -xnativeconnect=%none。
- 如果仅指定了 -xnativeconnect，则编译器将选项设置为 -xnativeconnect=inlines,interfaces。
- 该选项不累积。编译器使用指定的最后一个设置。例如，如果您指定了：

```
CC -xnativeconnect=inlines first.o -xnativeconnect=interfaces
second.o -O -G -o library.so
```

则编译器将选项设置为 -xnativeconnect=no%inlines,interfaces。

警告

这是 “功能终止”声明。在以后的 Sun Studio 发行版本中，可能会删除该选项。

如果计划使用 -xnativeconnect，不要用 -compat=4 进行编译。记住，如果指定没有参数的 -compat，则编译器将其设置为 -compat=4。如果未指定 -compat，编译器将其设置为 -compat=5。您还可以通过 -compat=5 将其显式设置为兼容模式。

A.2.142 -xnolib

禁止链接默认系统库。

通常（不含该选项）情况下，C++ 编译器会链接多个系统库以支持 C++ 程序。-l*lib* 选项使用这个选项来链接没有传递到 ld 的默认系统库。

通常情况下，编译器按照以下顺序链接系统支持库：

- 在标准模式 （默认模式）下：

```
-lCstd -lCrun -lm -lc
```

- 在兼容模式 (-compat) 下：

```
-lC -lm -lc
```

-l 选项的顺序非常重要。-lm 选项必须出现在 -lc 之前。

注 – 如果指定了 -mt 编译器选项，编译器通常就在链接 -lm 之前对 -lthread 进行链接。

要决定在默认情况下链接的系统支持库，请使用 -dryrun 选项进行编译。例如，以下命令的输出：

```
example% CC foo.cc -xarch=v9 -dryrun
```

在输出中包括了以下内容：

```
-lCstd -lCrun -lm -lc
```

示例

对于符合 C 应用程序二进制接口的基本编译（即只支持 C 所需的 C++ 程序），请使用：

```
example% CC -xnolib test.cc -lc
```

要将 libm 静态链接到具有通用架构指令集合的单线程应用程序，请使用：

- 标准模式下：

```
example% CC -xnolib test.cc -lCstd -lCrun -Bstatic -lm \
-Bdynamic -lc
```

- 兼容模式下：

```
example% CC -compat -xnolib test.cc -lC -Bstatic -lm \
-Bdynamic -lc
```

交互

使用 -xarch=v9, -xarch=v9a 或 -xarch=v9b 链接时，某些诸如 libm.a 和 libc.a 的静态系统库不可用。

如果指定了 -xnolib，就必须按给定的顺序手动链接所有需要的系统支持库。必须最后链接系统支持库。

如果指定了 -xnolib，则忽略 -library。

警告

许多 C++ 语言特性需要使用 libC（兼容模式下）或 libCrun（标准模式下）。

系统支持库的集合不稳定，会因不同的发行版本而更改。

另请参见

-library、-staticlib、-l

A.2.143 -xnolibmil

在命令行上取消 -xlibmil。

将该选项与 -fast 一起使用以覆盖对优化数学库的链接。

A.2.144 -xnolibmopt

不使用数学例程库。

示例

在命令行上 -fast 选项之后使用该选项，如下例所示：

```
example% CC -fast -xnolibmopt
```

A.2.145 -xO*level*

指定优化级别，注意大写字母 O 要后跟数字 1、2、3、4 或 5。通常，程序执行速度取决于优化的级别。优化级别越高，运行时性能越好。不过，较高的优化级别会延长编译时间并生成较大的可执行文件。

某些情况下，-xO2 的执行可能比其他优化级别好，而 -xO3 也可能胜过 -xO4。尝试用每个级别编译以查看您是否会遇到这种少见的情况。

如果优化器运行时内存不足，则会尝试在较低的优化等级上重试当前过程来恢复。优化器在 -xO*level* 选项中指定的初始级别中恢复随后的过程。

有五个级别可以与 -xO 一起使用。以下几节描述了在 SPARC 平台和 x86 平台上如何操作这些级别。

值

在 SPARC 平台上：

- -xO1 只执行最小量的优化 (peephole)，也称为 postpass，即汇编级优化。如果使用 -xO2 或 -xO3 导致了过多的编译时间，或者您所使用的交换空间不足，那么可以使用 -xO1。
- -xO2 执行基本的局部和全局优化，其中包括：
  - 感应变量消除
  - 局部和全局的通用子表达式消除

- 代数运算简化
- 复制传播
- 常量传播
- 非循环变体优化
- 寄存器分配
- 基本块合并
- 尾部递归消除
- 终止代码消除
- 尾部调用消除
- 复杂表达式扩展

  该级别不优化外部变量或间接变量的引用或定义。

  -O 选项与 -xO2 选项等价。

- 除了在 -xO2 级别执行优化外， -xO3 还会优化外部变量的引用和定义。该级别不跟踪指针赋值的结果。编译 volatile 没有正确保护的设备驱动程序或编译修改信号处理程序内外部变量的程序时，请使用 -xO2。通常，如果不将该级别与 -xspace 选项组合，就会增加代码的大小。
- -xO4 除执行 -xO3 的优化之外，还自动内联包含在同一文件中的函数。自动内联通常会提高执行速度，但有时却会使速度变得更慢。通常，如果不将该级别与 -xspace 选项组合，就会增加代码的大小。
- -xO5 生成最高级别的优化。它只适用于占用大量计算机时间的小部分程序。该级别采用了占用更多编译时间或无法在某种程度上减少执行时间的优化算法。如果使用性能分析反馈执行该级别上的优化，则更容易提高性能。请参见第 A-128 页的第 A.2.157 节 "-xprofile=p"。

在 x86 平台上：

- -xO1 执行基本优化。其中包括代数运算简化、寄存器分配、基本块合并、终止代码和存储消除以及 peephole 优化。
- -xO2 执行局部通用子表达式消除、局部复制传播、常量传播、尾部递归消除和级别 1 执行的优化。
- -xO3 执行全局通用子表达式的消除、全局的复制传播和常量传播、循环长度简约、感应变量消除、循环变体优化和级别 2 执行的优化。
- -xO4 自动内联包含在同一文件中的函数和级别 3 执行的优化。这种自动内联通常会提高执行速度，但有时却会使速度变得更慢。该级别还释放了通用的框架指针注册 (ebp)。通常该级别会增加代码的大小。
- -xO5 生成最高级别的优化。该级别采用了占用更多编译时间或无法在某种程度上减少执行时间的优化算法。

交互

如果您使用 -g 或 -g0，且优化级别是 -xO3 或更低，那么编译器会尽力提供几乎全部优化的符号信息。尾部调用优化和后端内联被禁用。

如果您使用 -g 或 -g0，且优化级别是 -xO4 或更高，那么编译器会尽力提供全部优化的符号信息。

使用 -g 进行调试时不禁止 -xO **级别**，但 -xO **级别**会以某些方法限制 -g。例如， -xO *级别*的选项会降低调试的公用性，因此无法显示 dbx 中的变量，但仍可使用 dbx where 命令获取符号回溯。更多信息请参见 《使用 dbx 调试程序》。

-xcrossfile 选项只有与 -xO4 或 -xO5 一起使用时才有效。

-xinline 选项对低于 -xO3 优化级别的函数无效。若在 -xO4 级别上，则优化器决定哪些函数应该内联，而与是否指定 -xinline 选项无关。在 -xO4 级别，编译器还尝试决定哪些函数内联后可以提高性能。如果使用 -xinline 强制内联一个函数，实际上就会降低性能。

默认

默认为不优化。不过，只有不指定优化级别时才使用默认。如果指定了优化级别，则没有任何选项可以关闭优化。

如果尝试避免设置优化级别，那么请不要指定任何隐含优化级别的选项。例如， -fast 是设置优化级别为 -xO5 的宏选项。隐含优化级别的所有其他选项给出优化已设置的警告消息。不使用任何优化来编译的一种方法是从命令行删除所有选项或创建指定优化级别的文件。

警告

如果在 -xO3 或 -xO4 级别上优化很大的过程 （一个过程具有数千行代码），则编译器会需要非常多的内存。在这些情况下，机器的性能就会降低。

为了防止性能的降低，请使用 limit 命令来限制单一进程可使用的虚拟内存大小 （请参见 csh(1) 手册页）。例如，将虚拟内存限制为 16MB：

```
example% limit datasize 16M
```

如果虚拟内存达到 16 MB 的数据空间，该命令会使优化器尝试恢复。

限制不能大于机器总的可用交换空间，而且要足够的小以允许在大型编译的过程中机器可以正常使用。

数据大小的最佳设置取决于要求的优化程度、真实内存和可用虚拟内存的大小。

要查找实际的交换空间，请键入：**swap -l**

要查找实际的真实内存，请键入：**dmesg | grep mem**

另请参见

-xldscope -fast、 -xcrossfile=*n*、 -xprofile=*p*、 csh(1) 手册页

A.2.146 -xopenmp[=*i*]

SPARC：使用 -xopenmp 选项启用具有 OpenMP 指令的显式并行。实现包括了一组源代码指令、运行时库例程和环境变量。

值

下表列出了 *i* 的值：

表 A-43 -xopenmp 值

i 的值	含义
parallel	启用 OpenMP pragma 的识别。低于 -xopenmp=parallel 的最小优化级别是 -x03。如果需要，编译器就将较低级别的优化更改为 -x03，并发出警告。
stubs	禁用对 OpenMP pragma 的识别，链接到存根库例程且不更改优化级别。如果应用程序对 OpenMP 运行时库例程执行显式调用，并且要编译该应用程序使其能串行执行，就请使用此选项。 -xopenmp=stubs 命令还定义了 _OPENMP 预处理程序标记。
none	禁用对 OpenMP pragma 的识别，不更改程序的优化级别且不预定义任何预处理程序标记。

默认

如果未指定 -xopenmp，则编译器将选项设置为 -xopenmp=none。

如果指定了没有参数的 -xopenmp，则编译器将选项设置为 -xopenmp=parallel。

警告

在将来的发行版本中， -xopenmp 的默认可能更改。可以通过显式指定适当的优化来避免警告消息。

如果单独进行编译和链接，那么也可以在链接步骤中指定 -xopenmp。这在编译包含 OpenMP 指令的库时尤其重要。

另请参见

关于生成多进程应用程序的 OpenMP Fortran 95、 C 和 C++ 应用程序编程接口 (API) 的完整摘要，请参见 《OpenMP API 用户指南》。

A.2.147 -xpagesize=*n*

设置堆栈和堆首选的页面大小。

值

n 值必须是以下项之一：8K、64K、512K、4M、32M、256M、2G、16G 或 default。

您必须为目标平台指定有效的页面大小。如果您不指定有效的页面大小，则运行时请求就会被忽略。

在 Solaris 操作系统上，可以使用 getpagesize(3C) 命令来确定页面中的字节数。Solaris 操作系统不保证支持页面大小的请求。您可以使用 pmap(1) 或 meminfo(2) 来决定目标平台的页面大小。您可以使用 pmap(1) 或 meminfo(2) 来决定目标平台的页面大小。

注 – 该功能只在 Solaris 8 操作系统中可用。在 Solaris 8 软件上不链接使用该选项编译的程序。

默认

如果指定 -xpagesize=default，则 Solaris 操作系统设置页面大小。

扩展

该选项其实是代表 -xpagesize_heap 和 -xpagesize_stack 的宏。这两个选项接受与 -xpagesize 参数相同的参数：8K、64K、512K、4M、32M、256M、2G、16G 或 default。可以通过指定 -xpagesize 来使用相同的值设置这两个选项，或使用不同的值对其进行分别指定。

警告

只有在编译时和链接时使用 -xpagesize 选项，该选项才是有效的。

另请参见

使用该选项进行编译与使用等价的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等价的选项运行 Solaris 命令 ppgsz(1) 具有相同的效果。详细信息请参见 Solaris 手册页。

A.2.148 -xpagesize_heap=*n*

在内存中为堆设置页面大小。

值

n 可以是 8K、64K、512K、4M、32M、256M、2G、16G 或 default。您必须为目标平台指定有效的页面大小。如果不指定有效的页面大小，运行时该请求就会被忽略。

在 Solaris 操作系统上，可以使用 getpagesize(3C) 命令来确定页面中的字节数。Solaris 操作系统不保证支持页面大小的请求。您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页面大小。

注 – 该功能只在 Solaris 8 操作系统中可用。在 Solaris 8 环境不链接使用该选项编译的程序。

默认

如果指定 -xpagesize_heap=default，则 Solaris 操作系统设置页面大小。

警告

除非在编译和链接时使用，否则 -xpagesize_heap 选项无效。

另请参见

使用该选项进行编译与使用等价的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等价的选项运行 Solaris 命令 ppgsz(1) 具有相同的效果。详细信息请参见 Solaris 手册页。

A.2.149 -xpagesize_stack=*n*

在内存中为堆栈设置页面大小。

值

n 可以是 8K、64K、512K、4M、32M、256M、2G、16G 或 default。您必须为目标平台指定有效的页面大小。如果不指定有效的页面大小，运行时该请求就会被忽略。

在 Solaris 操作系统上，可以使用 getpagesize(3C) 命令来确定页面中的字节数。Solaris 操作系统不保证支持页面大小的请求。您可以使用 pmap(1) 或 meminfo(2) 来确定目标平台上的页面大小。

注 – 该功能只在 Solaris 8 操作系统中可用。在 Solaris 8 软件上不链接使用该选项编译的程序。

默认

如果指定 -xpagesize_stack=default，则 Solaris 操作系统设置页面大小。

警告

除非在编译和链接时使用，否则 -xpagesize_heap 选项不会生效。

另请参见

使用该选项进行编译与使用等价的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等价的选项运行 Solaris 命令 ppgsz(1) 具有相同的效果。详细信息请参见 Solaris 手册页。

A.2.150 -xpch=*v*

该编译器选项激活了预编译头文件特性。预编译头文件的作用是减少源代码共享同一组包含文件的应用程序的编译时间，而且这些包含文件往往有大量的源代码。编译器首先从一个源文件收集一组头文件信息，然后在重新编译该源文件或者其他有同样头文件的源文件时就可以使用这些收集到的信息。编译器收集的信息存储在预编译头文件中。要使用这个特性，需要指定 -xpch 和 -xpchstop 选项，并同时使用 #pragma hdrstop 指令。

另请参见：

- 第 A-119 页的第 A.2.151 节 "-xpchstop=file"
- 第 B-8 页的第 B.2.8 节 "#pragma hdrstop"

创建预编译头文件

在指定 -xpch=*v* 时，*v* 可以是 collect:*pch_filename* 或 use:*pch_filename*。首次使用 -xpch 时，必须指定 collect 模式。指定 -xpch=collect 的编译命令只能指定一个源文件。在以下示例中，-xpch 选项根据源文件 a.cc 创建名为 myheader.Cpch 的预编译头文件：

```
CC -xpch=collect:myheader a.cc
```

有效预编译头文件的名称通常具有后缀 .Cpch。在指定 *pch_filename* 时，后缀可以由您亲自增加或由编译器增加。例如，如果指定 CC -xpch=collect:foo a.cc，则预编译头文件称为 foo.Cpch。

在创建预编译头文件时，请选取包含所有源文件之间包括文件通用序列的源文件，预编译头文件与这些源文件一起使用。包括文件的共同序列在这些源文件之间必须是一样的。切记，在 collect 模式中只有一个源文件名称的值是合法的。例如，CC -xpch=collect:foo bar.cc 是有效的，而由于 CC -xpch=collect:foo bar.cc foobar.cc 指定了两个源文件，所以它是无效的。

使用预编译头文件

指定 -xpch=use:*pch_filename* 使用预编译头文件。您可以将包括文件同一序列中任意数量的源文件指定为用于创建预编译头文件的源文件。例如，在 use 模式中的命令可类似于以下命令：CC -xpch=use:foo.Cpch foo.c bar.cc foobar.cc。

如果下列情况为真，就只应使用现有的预编译头文件。如果以下任意情况都为假，就应该重新创建预编译头文件：

- 用于访问预编译头文件的编译器与创建预编译头文件的编译器相同。编译器的一个版本创建的预编译头文件可能无法用于另一版本（包括安装的修补程序产生的差异）。
- 除 -xpch 选项之外，用 -xpch=use 指定的编译器选项必须与创建预编译头文件时指定的选项相匹配。
- 用 -xpch=use 指定的包括头文件的集合与创建预编译头文件时指定的头文件集合是相同的。
- 用 -xpch=use 指定的包括头文件的内容与创建预编译头文件时指定的包括头文件的内容是相同的。
- 当前目录（即发生编译并尝试使用给定预编译头文件的目录）与创建预编译头文件所在的目录相同。
- 在用 -xpch=collect 指定的文件中预处理指令（包括 #include 指令）的初始序列，与在用 -xpch=use 指定的文件中预处理指令的序列相同。

要共享多个源文件间的预编译头文件，这些源文件就必须作为标记的初始序列共享包括文件的共同集合。该标记的初始序列称为可用前缀。可用前缀必须在使用相同预编译头文件的所有源文件中解释一致。

源文件的可用前缀只能包含注释和以下任意预处理程序指令：

#include

#if/ifdef/ifndef/else/elif/endif

#define/undef

#ident （如果相等，按原样传递）

#pragma （如果相等）

以上任何指令都可以引用宏。#else、#elif 和 #endif 指令必须在可用前缀内匹配。

在共享预编译头文件的每个文件的可用前缀中，每个相应的 #define 和 #undef 指令必须引用相同的符号 （例如每个 #define 必须引用同一个值）。这些指令在每个可用前缀中出现的顺序也必须相同。每个相应 pragma 也必须相同，且必须按相同顺序出现在共享预编译头文件的所有文件中。

融入预编译头文件的头文件一定不要违反以下约束。这里没有定义对违反任意这些约束的程序的编译结果。

- 头文件一定不要包含函数和变量定义。
- 头文件一定不要使用 __DATE__ 和 __TIME__。使用预处理宏会产生无法预料的结果。
- 头文件一定不要包含 #pragma hdrstop。
- 头文件一定不要在可用前缀中使用 __LINE__ 和 __FILE__。头文件可以在包括的头文件中使用 __LINE__ 和 __FILE__。

如何修改 make 文件

以下是要将 -xpch 融入生成中而对 make 文件进行修改的几种可能方法。

- 可以通过使用 make 和 dmake 的辅助 CCFLAGS 变量和 KEEP_STATE 功能来使用隐式 make 规则。预编译头文件在独立的步骤中产生。

```
.KEEP_STATE :
CCFLAGS_AUX = -O etc
CCFLAGS = -xpch=use:shared $(CCFLAGS_AUX)
shared.Cpch:foo.cc
     $(CCC) -xpch=collect:shared $(CCFLAGS_AUX) foo.cc
a.out :foo.o ping.o pong.o
     $(CCC) foo.o ping.o pong.o
```

您还可以定义自己的编译规则，而无需尝试使用辅助 CCFLAGS。

```
.KEEP_STATE :
.SUFFIXES:.o .cc
%.o : %.cc shared.Cpch
        $(CCC) -xpch=use:shared $(CCFLAGS) -c $<
shared.Cpch:foo.cc
        $(CCC) -xpch=collect:shared $(CCFLAGS) foo.cc -xe
a.out :foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

- 您可以将预编译头文件作为常规编译的副作用，而无需使用 KEEP_STATE，但该方法要求显式编译命令。

```
shared.Cpch + foo.o:foo.cc bar.h
        $(CCC) -xpch=collect:shared foo.cc $(CCFLAGS) -c
ping.o:ping.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared ping.cc $(CCFLAGS) -c
pong.o:pong.cc shared.Cpch bar.h
        $(CCC) -xpch=use:shared pong.cc $(CCFLAGS) -c
a.out :foo.o ping.o pong.o
        $(CCC) foo.o ping.o pong.o
```

A.2.151 -xpchstop=*file*

使用 -xpchstop=*file* 选项指定在使用 -xpch 选项创建预编译头文件时要考虑的最后一个包含文件。在命令行上使用 -xpchstop 等价于将 hdrstop pragma 放置在第一个包括指令之后，该指令引用由 cc 命令指定的每个源文件中的文件。

在以下示例中， -xpchstop 选项指定了预编译头文件的可用前缀以 projectheader.h 的包括结束。因此， privateheader.h 不是可用前缀的一部分。

```
example% cat a.cc
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% CC -xpch=collect:foo.Cpch a.cc -xpchstop=projectheader.h
-c
```

另请参见

-xpch, pragma hdrstop

A.2.152 -xpg

-xpg 选项编译自身文件配置代码，为使用 gprof 的文件配置收集数据。该选项调用运行时记录机制，该机制会在程序正常终止时生成 gmon.out 文件。

注 – -xprofile 与 -xpg 相比较而言，指定 -xpg 更佳。这两个选项没有准备或使用由另一个选项提供的数据。

可以在 64 位 Solaris 平台上使用 prof(1) 或 gprof(1) 或者在 32 位 Solaris 平台上仅使用 gprof 来生成性能分析数据，这些性能分析数据包含近似的用户 CPU 时间。这些时间是从主要可执行文件中的例程以及共享库（在链接可执行文件时，被指定为链接程序参数）中的例程的 PC 样例数据（请参见 pcsample(2)）中获得的。尚未对其他共享库（使用 dlopen(3DL) 启动进程后打开的库）进行性能分析。

在 32 位 Solaris 系统上，使用 prof(1) 生成的性能分析数据仅限于可执行文件中的例程。通过将可执行文件与 -xpg 相链接并使用 gprof(1)，可以分析 32 位共享库。

Solaris 10 软件不包含使用 -p 编译的系统库。因此，在 Solaris 10 平台上收集的性能分析数据不包含系统库例程的调用计数。

警告

如果单独进行编译和链接且使用了 -xpg 进行链接，就要确保使用 -xpg 进行链接。

另请参见

-xprofile=*p*、analyzer(1) 手册页、《性能分析器》手册。

A.2.153 -xport64 [=(*v*)]

使用该选项可以帮助您调试移植到 64 位环境的代码。具体来说，该选项会对以下情况提示警告信息：类型的截断（包括指针），符号扩展以及对位包装的更改。将代码从诸如 V7 的 32 位体系结构移植到诸如 V9 的 64 位体系结构时，这些更改是很通常的。

值

下表列出了 *v* 的有效值：

表 A-44 `-xport64` 值

v 的值	含义
否	将代码从 32 位环境移植到 64 位环境时，不会生成与该代码移植有关的任何警告。
implicit	只生成隐式转换的警告。显式类型转换出现时不生成警告。
full	将代码从 32 位环境移植到 64 位环境时，生成了与该代码移植有关的所有警告。其中包括对 64 位值的截断警告、根据 ISO 值的保存规则对 64 位的符号扩展，以及对位字段包装的更改。

默认

如果未指定 `-xport64`，则默认为 `-xport64=no`。如果指定了 `-xport64` 但不指定标志，则默认为 `-xport64=full`。

示例

本节提供了可以导致类型截断、符号扩展和对位包装更改的代码示例。

检查 64 位值的截断

在移植到诸如 V9 的 64 位架构时，数据可能会被截断。截断可能会因赋值（初始化时）或显式类型转换而隐式地发生。两个指针的差异在于 typedef `ptrdiff_t`，它在 32 位模式下是 32 位整数类型，而在 64 位模式下是 64 位整数类型。将较长的整数类型截断为较小的整型类型会生成警告，如下示例所示。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; // 警告

example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test1.c
"test1.c", line 3:Warning:Conversion of 64-bit type value to "int"
causes truncation.
1 Warning(s) detected.
example%
```

当显式转换导致数据截断时，请使用 -xport64=implicit 禁用 64 位编译模式下的截断警告。

```
example% CC -c -xarch=v9 -Qoption ccfe -xport64=implicit test1.c
"test1.c", line 3:Warning:Conversion of 64-bit type value to "int"
causes truncation.
1 Warning(s) detected.
example%
```

在移植到 64 位架构过程中出现的另一个常见问题是指针的截断。该问题在 C++ 中始终是错误。指定 -xport64 时，诸如将指针转换为引起这种截断的整型的操作，可能会在 V9 中导致错误诊断。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) ( ((unsigned int)p) & 0xFF ); // -xarch=v9 错误
  return 0;
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test2.c
"test2.c", line 3:Error: Cannot cast from char* to unsigned.
1 Error(s) detected.
example%
```

检查符号扩展

您还可以使用 -xport64 选项来检查这种情况：标准 ISO C 值保留的规则允许在无符号整数类型的表达式中进行有符号整数值的符号扩展。这种符号扩展会产生细微的运行时错误。

```
example% cat test3.c
int i= -1;
void promo(unsigned long l) {}

int main() {
    unsigned long l;
    l = i;  // 警告
    promo(i);       // 警告
}
example% CC -c -xarch=v9 -Qoption ccfe -xport64=full test3.c
"test3.c", line 6:Warning:Sign extension from "int" to 64-bit
integer.
"test3.c", line 7:Warning:Sign extension from "int" to 64-bit
integer.
2 Warning(s) detected.
```

检查位字段包装的更改

使用 -xport64 生成对长位字段的警告。出现这种位字段时，位字段的包装可能会显著更改。在成功移植到 64 位架构之前，依赖于假定的任何程序都需要重新检查，该假定与包装位字段的方法有关。

```
example% cat test4.c
#include <stdio.h>

union U {
   struct S {
       unsigned long b1:20;
       unsigned long b2:20;
   } s;

   long buf[2];
} u;

int main() {
   u.s.b1 = 0XFFFFF;
   u.s.b2 = 0XFFFFF;
   printf(" u.buf[0] = %lx u.buf[1] = %lx\n", u.buf[0], u.buf[1]);
   return 0;
}
example%
```

V9 中的输出：

```
example% u.buf[0] = fffffffff000000  u.buf[1] = 0
```

V7 中的输出：

```
example% u.buf[0] = fffff000  u.buf[1] = fffff000
example% CC -c -xarch=v9 -Qoption ccfe -xport64 test4.c
"test4.c", line 5:Warning:64-bit type bitfield may change bitfield
packing within structure or union.
"test4.c", line 6:Warning:64-bit type bitfield may change bitfield
packing within structure or union.
2 Warning(s) detected.
example%
```

警告

注意，只有在 64 位模式下通过指定诸如 -arch=generic64 或 -xarch=v9 选项进行编译时才生成警告。

另请参见

第 A-69 页的第 A.2.105 节 "-xarch=isa".

A.2.154 -xprefetch[=*a*[,*a*...]]

SPARC：在支持预取的架构上启用预取指令，这些架构包括 UltraSPARC II （-xarch=v8plus、v8plusa、v9plusb、v9、v9a 或 v9b）

a 必须是下列值之一：

表 A-45 -xprefetch 值

值	含义
auto	启用预取指令的自动生成
no%auto	禁用预取指令的自动生成
显式	*(SPARC)* 启用显式预取宏
no%explicit	*(SPARC)* 禁用显式预取宏
latx:*factor*	按照指定的因子调整编译器假定的预取到装入和预取到存储的延迟。您只能将此标志与 -xprefetch=auto 结合使用。该因子必须是正浮点数或整数。
是	已废弃，不使用。而是使用 -xprefetch=auto,explicit。
否	已废弃，不使用。而是使用 -xprefetch=no%auto,no%explicit。

使用 -xprefetch, -xprefetch=auto 和 -xprefetch=yes，编译器就可以将预取指令自由插入到它生成的代码中。该操作会提高支持预取的架构的性能。

如果正在较大的多处理器上运行计算密集的代码，您会发现使用 -xprefetch=latx:*factor* 有很多优点。该选项指示代码生成器按照指定的因子调节在预取及其相关的装入或存储之间的默认延迟时间。

预取延迟是执行预取指令和预取数据在缓存中可用时间之间的硬件延迟。编译器决定放置使用预取数据的预取指令和装入或存储指令的距离时，假定预取延迟值。

注 – 在预取和装入之间假定的延迟可能与在预取和存储之间假定的延迟不同。

编译器在多个机器和应用程序间调整预取机制以获得最佳性能。这种调整并非总能达到最优。对于内存密集的应用程序，尤其是要在较大的多处理器上运行的应用程序，您可以通过增加预取延迟值获得更高的性能。要增加值，请使用大于 1 的因子。在 .5 和 2.0 之间的值最有可能提供最高的性能。

对于具有完全位于外部缓存内的数据集的应用程序，您可以通过减小预取延迟值来获得更高的性能。要减小值，请使用小于 1 的因子。

要使用 -xprefetch=latx:*factor* 选项，请在开始时使用接近 1.0 的因子值并对应用程序运行性能测试。然后适当增加或减小该因子，并再次运行性能测试。获得最优性能之前，可以不断调整因子并运行性能测试。以很小的增量逐渐增加或减小因子时，前几步中不会看到性能差异，然后会突然出现差异，最后再趋于稳定。

默认

如果未指定 -xprefetch，则假定为 -xprefetch=no%auto,explicit。

如果仅指定了 -xprefetch，则假定为 -xprefetch=auto,explicit。

只有使用没有任何参数或具有 auto 或 yes 参数的 -xprefetch 进行显式覆盖时，才假定 no%auto 的默认。例如，-xprefetch=explicit 与 -xprefetch=explicit,no%auto 相同。

只有在使用 no%explicit 的参数或 no 的参数进行显式覆盖时，才假定 explicit 的默认。例如， -xprefetch=auto 与 -xprefetch=auto,explicit 相同。

如果启用了诸如使用 -xprefetch 或 -xprefetch=yes 的自动预取，但不指定延迟因子，则假定为 -xprefetch=latx:1.0。

交互

该选项会累积而不覆盖。

sun_prefetch.h 头文件提供了指定显式预取指令的宏。这些预取可能位于对应于宏出现位置的可执行文件中。

要使用显式预取指令，就必须在正确的架构中包括 sun_prefetch.h，或者从编译器命令中排除 -xprefetch 或使用 -xprefetch、 -xprefetch=auto,explicit、-xprefetch=explicit 或 -xprefetch=yes。

如果您调用了宏并包括了 sun_prefetch. 头文件，但传递了 -xprefetch=no%explicit 或 -xprefetch=no，那么显式预取将不会出现在可执行文件中。

只有自动预取启用时才可以使用 latx:*factor*。即，只有像在 -xprefetch=yes, latx:*factor* 中一样将它与 yes 或 auto 配合使用，才不会忽略 latx:*factor*。

警告

显式预取只应在度量支持的特殊环境下使用。

因为编译器在多个机器和应用程序之间调节预取机制以获得最优性能，所以当性能测试指示性能明显提高时，应该仅使用 -xprefetch=latx:*factor*。假定的预取延迟在不同发行版本中是不同的。因此，无论何时切换到不同的发行版本，强烈建议重新测试延迟因子对性能的影响。

A.2.155 -xprefetch_auto_type=*a*

(SPARC) 其中 *a* 是 [no%]indirect_array_access。

使用此选项可以确定编译器是否以为直接内存访问生成预取的相同方式为由选项 -xprefetch_level 指示的循环生成间接预取。

如果不指定 -xprefetch_auto_type 的设置，编译器将把它设置为 -xprefetch_auto_type=no%indirect_array_access。

类似 -xdepend、-xrestrict 和 -xalias_level 的选项可以影响计算候选间接预取的主动性，进而影响由于更好的内存别名歧义消除信息而发生的自动间接插入的主动性。

A.2.156 -xprefetch_level[=*i*]

与使用 -xprefetch=auto 一样，使用新的 -xprefetch_level=*i* 选项来控制自动插入的预取指令的攻击性。编译器更加主动，换句话说，引入了更多更高 -xprefetch_level 级别的预取。

-xprefetch_level 的合适值取决于应用程序中缓存缺失的数量。较高的 -xprefetch_level 值有可能提高具有大量缓存缺失的应用程序的性能。

值

i 必须是 1、 2 或 3 之一。

表 A-46 -xprefecth_level 值

值	含义
1	启用预取指令的自动生成。
2	超出 -xprefetch_level=1 目标的目标附加循环用于预取插入。附加预取可以在 -xprefetch_level=1 插入的预取之外插入。
3	超出 -xprefetch_level=2 目标的目标附加循环用于预取插入。附加预取可以在 -xprefetch_level=2 插入的预取之外插入。

默认

当您指定 -xprefetch=auto 时，默认值为 -xprefetch_level=1。

交互

只有在支持预取的平台上（v8plus、 v8plusa、 v9、 v9a、 v9b、 generic64、 native64）并且使用 -xprefetch=auto 和 3 或更高的优化级别 （-xO3）对该选项进行编译时该选项才有效。

A.2.157 -xprofile=*p*

使用该选项要先收集并保存执行频率的数据，这样您才能使用随后运行的数据以提高性能。只有将优化指定为 -xO2 或更高级别时，该选项才有效。

为编译器提供运行时的性能反馈增强了使用高优化级别 （如 -xO5）进行编译的效果。为了生成运行时的性能反馈，就必须使用 -xprofile=collect 编译，然后运行不是典型数据集的可执行文件，最后用最高的优化级别并使用 -xprofile=use 重新编译。

对多线程应用程序来讲，性能分析数据集合是安全的。也就是说，对执行自身多任务 (-mt) 的程序进行文件配置会产生准确的结果。只有将优化指定为 -xO2 或更高级别时，该选项才有效。

值

p 必须是下列值之一：

- collect[:*name*]

优化器使用 -xprofile=use 收集并保存执行频率，以便将来使用。编译器生成测量语句执行频率的代码。

name 是被分析的程序的名称。 *name* 是可选的并 （如果未为其指定名称）被假定为 a.out。

在运行时，使用 -xprofile=collect:*name* 编译的程序会创建子目录 *name*.profile 来保存运行时的反馈信息。数据将写入该子目录下的文件 feedback 中。可以使用 $SUN_PROFDATA 和 $SUN_PROFDATA_DIR 环境变量来更改反馈信息的位置。更多信息请参见**交互**一节。

如果多次运行程序，那么执行频率数据会累积在 feedback 文件中，也就是说以前运行的输出不会丢失。

如果单独进行编译和链接，请确保使用 -xprofile=collect 编译的任何对象文件也使用 -xprofile=collect 进行链接。

- use[:*name*]

  使用生成并保存在 feedback 文件中的执行频率数据来优化程序，该数据由先前执行使用 -xprofile=collect 编译的程序生成并保存。

  name 是被分析的可执行文件的名称。*name* 是可选的并（如果未为其指定名称）被假定为 a.out。

  除了从 -xprofile=collect 更改为 -xprofile=use 的 -xprofile 选项之外，源文件和其他编译器选项必须与用于编译的源文件和编译器选项完全相同，其中编译创建了依次生成 feedback 文件的编译程序。编译器的相同版本必须既用于收集生成也用于使用生成。如果使用 -xprofile=collect:*name* 进行编译，则相同的程序名称 *name* 必须出现在优化编译：-xprofile=use:*name* 中。

  在使用 -xprofile=collect 编译对象文件时，对象文件及其性能分析数据之间的关联取决于对象文件的 UNIX 路径名。在某些情况下，编译器将不会关联目标文件和其性能分析数据：由于以前没有使用 -xprofile=collect 编译目标文件，因此目标文件没有性能分析数据，程序中的目标文件未使用 -xprofile=collect 进行链接，从未执行过该程序。

  如果先前使用了 -xprofile=collect 在不同目录中编译对象文件，并且该对象文件与使用 -xprofile=collect 编译的其他对象文件共享公共基名，但却无法通过它们包含的目录名称进行唯一的标识，那么编译器也会发生混淆。在这种情况下，即使对象文件具有文件配置数据，使用 -xprofile=use 重新编译对象文件时，编译器也不能在反馈目录中找到该对象文件。

  所有这些情况都能使编译器释放对象文件及其文件配置数据之间的关联。因此，在您指定 -xprofile=use 时，如果对象文件具有文件配置数据但编译器无法将其与对象文件的路径名相关联，那么请使用 -xprofile_pathmap 选项来标识正确的目录。请参见第 A-131 页的第 A.2.159 节 "-xprofile_pathmap"。

- tcov

  使用新型的 tcov 进行基本块覆盖分析。

  该选项是 tcov 的新型基本块文件配置。该选项具有与 -xa 选项类似的功能，但可以为在头文件中具有源代码或使用 C++ 模板的程序正确收集数据。代码指令与 -xa 选项的代码指令类似，但不再生成 .d 文件。相反却生成单一文件，并且该文件的名

称是按照最后的可执行文件命名的。例如，如果程序在 /foo/bar/myprog.profile 的外部运行，那么数据文件将存储到 /foo/bar/myprog.profile/myprog.tcovd 中。

在运行 tcov 时，必须将其传递到 -x 选项，以强制它使用新型数据。如果不传递 -x，则 tcov 使用默认的旧式 .d 文件并产生意外的输出。

与 -xa 选项不同， TCOVDIR 环境变量在编译时无效。不过，在程序运行时可以使用环境变量的值。

交互

-xprofile=tcov 和 -xa 选项在单一可执行文件中相互兼容。也就是说，您可以链接程序，其中程序包含了用 -xprofile=tcov 编译过的某些文件以及用 -xa 编译的其他文件。您不能同时用这两个选项来编译单一文件。

如果由于使用 -xinline 或 -xO4 才使函数内联，那么 -xprofile=tcov 生成的代码覆盖报告是不可靠的。

您可以设置环境变量 $SUN_PROFDATA 和 $SUN_PROFDATA_DIR，来控制使用 -xprofile=collect 编译的程序放置文件配置数据的位置。如果不设置这些变量，则文件配置数据会被写入当前目录中的 *name*.profile/feedback （*name* 是可执行文件的名称或是在 -xprofile=collect:*name* 标志中指定的名称）。如果设置了这些变量，则 -xprofile=collect 数据会被写入到 $SUN_PROFDATA_DIR/$SUN_PROFDATA。

$SUN_PROFDATA 和 $SUN_PROFDATA_DIR 环境变量类似地控制着用 tcov 编写的文件配置数据文件的路径和名称。更多信息请参见 tcov(1) 手册页。

警告

如果单独进行编译和链接，-xprofile 选项就必须出现在编译命令和链接命令中。虽然在一个步骤中包括 -xprofile 并从其他步骤排除不会影响程序的正确性，但这样将无法进行文件配置。

另请参见

-xa、 tcov(1) 手册页、《程序性能分析工具》。

A.2.158 -xprofile_ircache[=*path*]

(SPARC) 使用 -xprofile_ircache[=*path*] 及 -xprofile=collect|use，通过重用**收集**阶段保存的编译数据，减少 use 阶段的编译时间。

在编译大程序时，由于中间数据的保存，使得使用阶段的编译时间大大减少。注意，所保存的数据会占用相当大的磁盘空间。

在使用 -xprofile_ircache[=*path*] 时， *path* 会覆盖保存缓存文件的位置。默认情况下，这些文件会作为对象文件保存在同一目录下。 collect 和 use 阶段出现在两个不同目录中时，指定路径才是有用的。以下是命令的典型序列：

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out     // 运行收集反馈数据
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

A.2.159 -xprofile_pathmap

(SPARC) 当您也指定 -xprofile=use 命令时，请使用 -xprofile_pathmap=*collect_prefix*:*use_prefix* 选项。当以下两种情况都为真且编译器无法找到用 -xprofile=use 编译的对象文件的文件配置数据时，请使用 -xprofile_pathmap。

- 使用 -xprofile=use 编译对象文件所在的目录与先前使用 -xprofile=collect 编译对象文件所在的目录不同。
- 对象文件会共享文件配置中的公共基名，但却可以根据它们在不同目录中的位置互相区分。

collect-prefix 是目录树的 UNIX 路径名的前缀，在该目录树中使用 -xprofile=collect 编译对象文件。

use-prefix 是目录树的 UNIX 路径名的前缀，在该目录树中使用 -xprofile=use 编译对象文件。

如果指定了 -xprofile_pathmap 的多个实例，那么编译器将按照这些实例出现的顺序对其进行处理。在标识了匹配的 *use-prefix* 或发现最后指定的 *use-prefix* 与对象文件路径名不匹配之前，每个由 -xprofile_pathmap 的实例指定的 *use-prefix* 都会与对象文件路径名进行比较。

A.2.160 -xregs=*r*[,*r*...]

控制临时寄存器的使用。

如果编译器可以使用更多的寄存器用于临时存储（临时寄存器），那么编译器将能生成速度更快的代码。该选项使得附加临时寄存器可用，而这些附加寄存器通常是不适用的。

值

r 必须是下列值之一：每个值的含义都取决于 `-xarch` 的设置。

表 A-47 `-xregs` 值

r 的值	含义
`[no%]appl`	*(SPARC)* [不] 允许编译器生成将应用程序寄存器用为临时寄存器的代码。应用程序寄存器是： g2, g3, g4 (v8a, v8, v8plus, v8plusa, v8plusb) g2, g3 (v9, v9a, v9b) 强烈建议使用 `-xregs=no%appl` 编译所有系统软件和库。系统软件（包括共享库）必须为应用程序保留这些寄存器值。这些值的使用将由编译系统控制，而且在整个应用程序中必须保持一致。 关于 SPARC 指令集的更多信息，请参见第 A-69 页的第 A.2.105 节 "`-xarch=isa`"。 在 SPARC ABI 中，这些寄存器被描述为**应用程序**寄存器。因为需要更少的 `load` 和 `store` 指令，所以使用这些寄存器可以提高性能。不过，这样使用会与将寄存器用于其他目的的程序发生冲突。
`[no%]float`	*(SPARC)*[不] 允许编译器生成将浮点寄存器用为整数值的临时寄存器的代码。使用浮点值可能会用到与该选项无关的这些寄存器。如果希望您的代码独立于对浮点寄存器的所有引用，则需要使用 `-xregs=no%float` 并确保您的代码不会以任何方式使用浮点类型。
`[no%]frameptr`	*(x86)* [不] 允许编译器将帧指针寄存器（IA32 上的 %ebp、AMD64 上的 %rbp）用作未分配的被调用方保存寄存器。 如果将此寄存器用作被调用方保存寄存器，则可以提高程序运行时性能。但是，它也会降低检查和跟踪栈的某些工具的性能。这种栈检查功能对系统性能测量和调节至关重要。因此，使用这种优化可以提高本地系统性能，但会降低全局系统性能。 • 为事后分析而转储栈的工具（如性能分析器）将无法工作。 • 调试器（例如 `adb`、`mdb`、`dbx`）将无法转储栈或直接弹出栈帧。 • 最新的帧丢失帧指针之前，`dtrace` 性能分析实用程序将无法收集栈上的任何帧的信息。 • Posix `pthread_cancel` 将无法尝试找到清除处理程序。 • C++ 异常无法传播给 C 函数。 丢失了帧指针的 C 函数调用在 C 函数中引发异常的 C++ 函数时，C++ 异常中会发生失败。函数接受函数指针（例如，`qsort`）或全局函数（例如 `malloc`）被干预时，通常会发生此类调用。 上面列出的后两种后果可能会影响应用程序的正常操作。大多数应用程序代码不会遇到这些问题。但是，使用 `-xO4` 开发的库需要详细记录客户端使用限制的文档。 注意：如果还指定了 `-xpg`，则编译器会忽略 `-xregs=frameptr` 并发出警告。

默认

SPARC 默认值为 -xregs=appl,float。

除非在 -xregs=frameptr 中指定 -fast 或 -xO5 优化，否则 x86 默认值为 -xregs=no%frameptr。

示例

要使用所有可用的临时寄存器编译应用程序，请使用 -xregs=appl,float。

要编译对上下文切换敏感的非浮点代码，请使用 -xregs=no%appl,no%float。

另请参见

SPARC V7/V8 ABI， SPARC V9 ABI

A.2.161 -xrestrict[=*f*]

(SPARC) 将指针值函数函数参数视为限定指针。*f* 必须是以下值之一：

表 A-48 -xrestrict 值

值	含义
%all	整个文件中的所有指针参数均被视为限定的。
%none	文件中没有指针参数被视为限定的。
%source	只有在主源文件中定义的函数是限定的。在包含文件中定义的函数不是限定的。
fn[,*fn*...]	用逗号隔开的一个或多个函数名称的列表。如果指定函数列表，则编译器将指定的函数中的指针参数视为限定的；有关详细信息，请参见第 A-134 页的第 A.2.161.1 节 “限定指针”一节。

该命令行选项可用于其自身，但最佳效果是与优化同时使用。例如，命令：

```
%CC -xO3 -xrestrict=%all prog.cc
```

将文件 prog.c 中的所有指针参数视为限定指针。命令：

```
%CC -xO3 -xrestrict=agc prog.cc
```

将文件 prog.c 中函数 agc 中的所有指针参数视为限定指针。

默认值为 %none；指定 -xrestrict 等价于指定 -xrestrict=%source。

A.2.161.1 限定指针

为使编译器有效地执行循环的并行执行任务，需要确定特定左值是否指定不同的存储区域。别名是其存储区域相同的左值。由于需要分析整个程序，因此确定对象的两个指针是否为别名是一个困难而费时的过程。考虑下面的函数 vsq()：

代码样例 A-3 带两个指针的循环

```
void vsq(int n, double * a, double * b) {
       int i;
       for (i=0; i<n; i++) {
              b[i] = a[i] * a[i];
       }
}
```

如果编译器知道指针 a 和 b 访问不同的对象，它可以并行化循环不同迭代的执行。如果通过指针 a 和 b 访问的对象中存在重叠，则编译器并行执行循环将会不安全。在编译时，编译器并不能通过简单地分析 vsq() 来获悉 a 和 b 访问的对象是否重叠；编辑器需要分析整个程序才能获取此信息。

限定指针用来指定哪些指定不同对象的指针，以便编译器可以执行指针别名分析。以下是函数参数声明为限定指针的函数 vsq() 的示例：

```
void vsq(int n, double * restrict a, double * restrict b)
```

指针 a 和 b 声明为限定指针，因此编译器知道 a 和 b 指向不同的存储区域。有了此别名信息，编译器就能够并行化循环。

关键字 restrict 是类型限定符，类似于 volatile，并且仅限定指针类型。当您使用 -xc99=%all（不包括使用 -Xs 的情况）时，restrict 被识别为关键字。在某些情况下，您可能不希望更改源代码。您可以通过使用以下命令行选项，指定将指针赋值的函数参数视为限定指针：

```
-xrestrict=[func1,…,funcn]
```

如果指定函数列表，则指定的函数中的指针参数将被视为限定的；否则，整个 C 文件中的所有指针参数均被视为限定的。例如，-xrestrict=vsq 是限定带关键字 restrict 的函数 vsq() 的第一个示例中给定的指针 a 和 b。

正确使用 `restrict` 至关重要。如果指针被限定为限定指针而指向不同的对象，编译器会错误地并行化循环而导致不确定的行为。例如，假定函数 `vsq()` 的指针 `a` 和 `b` 指向重叠的对象，如 `b[i]` 和 `a[i+1]` 是同一对象。如果 `a` 和 `b` 未被声明为限定指针，循环将以串行执行。如果 `a` 和 `b` 被错误地限定为限定指针，则编译器会并行化循环的执行，但这不安全，原因是 `b[i+1]` 应在计算 `b[i]` 之后才会计算。

A.2.162 -xs

允许在用 `dbx` 调试时不包括对象 (`.o`) 文件。

该选项导致所有调试信息被复制到可执行程序中。这对 dbx 的性能或程序的运行时间性能影响较小，但需要更多磁盘空间。

A.2.163 -xsafe=mem

SPARC：允许编译器假定不违反内存保护。

该选项允许编译器使用 SPARC V9 架构中的无故障装入指令。

交互

该选项只有与 `-xO5` 优化共同使用，且指定了 `-xarch=v8plus`、`v8plusa`、`v8plusb`、`v9`、 `v9a` 或 `v9b` 时才有效。

警告

因为错误 （如地址未对齐或段违规）发生时无故障装入不产生自陷，所以该选项只能用于不发生这些错误的程序。因为很少的程序会导致基于内存的自陷，所以您可以安全地将该选项用于大多数程序。请不要将该选项用于显式依赖基于内存自陷的程序来处理异常情况。

A.2.164 -xsb

已过时，不使用。源浏览器功能已废弃。

A.2.165 -xsbfast

已过时，不使用。源浏览器功能已废弃。

A.2.166 -xspace

SPARC：不允许增加代码大小的优化。

A.2.167 -xtarget=*t*

为指令集和优化指定目标平台。

向编译器提供目标计算机硬件的准确描述有助于提高某些程序的性能。当程序性能很重要时，目标硬件的正确说明会是非常重要的。在较新的 SPARC 处理器上运行时这一点尤其重要。不过，对多数程序和较旧的 SPARC 处理器来讲，性能的获取是可以忽略的，而通用说明已经足够了。

-xtarget 的每个特定值都会扩展到 -xarch、-xchip 和 -xcache 选项值的特定集合。使用 -xdryrun 选项在运行的系统上决定 -xtarget=native 的扩展。关于值的信息请参见表 A-50 和表 A-51。

例如，-xtarget=sun4/15 等效于：-xarch=v8a -xchip=micro -xcache=2/16/1。

注 – 在该平台上进行编译时，特定主机平台的 -xtarget 扩展不能扩展到与 -xtarget=native 相同的 -xarch、 -xchip 或 -xcache 设置上。

值

对于 SPARC 平台：

在 SPARC 平台上， *t* 必须是下列值之一：

表 A-49 SPARC 平台的 -xtarget 值

t 的值	含义
native	在主机系统上获取最佳性能。编译器生成为主机系统优化的代码。它决定了运行编译器的计算机的可用架构、芯片和缓存属性。
native64	在主机系统上获取 64 位对象二进制文件的最佳性能。编译器生成为主机系统优化的 64 位对象二进制文件。它决定了运行编译器的计算机的可用 64 位架构、芯片和缓存属性。

表 A-49　SPARC 平台的 -xtarget 值 (续)

t 的值	含义
generic	获取通用架构、芯片和缓存的最佳性能。 编译器将 -xtarget=generic 扩展到： -xarch=generic -xchip=generic -xcache=generic。 这是默认值。
generic64	在大多数 64 位平台架构上设置 64 位对象二进制文件的最佳性能的参数。
platform-name	获取指定平台的最佳性能。从表 A-50 选择 SPARC 平台名称。

下表详细列出了 -xtarget SPARC 的平台名称及其扩展。

表 A-50　-xtarget 的 SPARC 平台扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
cs6400	v8plusa	super	16/32/4:2048/64/1
entr150	v8plusa	ultra	16/32/1:512/64/1
entr2	v8plusa	ultra	16/32/1:512/64/1
entr2/1170	v8plusa	ultra	16/32/1:512/64/1
entr2/1200	v8plusa	ultra	16/32/1:512/64/1
entr2/2170	v8plusa	ultra	16/32/1:512/64/1
entr2/2200	v8plusa	ultra	16/32/1:512/64/1
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8plusa	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4

表 **A-50** -xtarget 的 SPARC 平台扩展 (续)

-xtarget=	-xarch	-xchip	-xcache
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1

表 **A-50** -xtarget 的 SPARC 平台扩展 (续)

-xtarget=	-xarch	-xchip	-xcache
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1

表 **A-50** -xtarget 的 SPARC 平台扩展 (续)

-xtarget=	-xarch	-xchip	-xcache
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1

表 A-50　-xtarget 的 SPARC 平台扩展 (续)

-xtarget=	-xarch	-xchip	-xcache
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/512/1
ultra3cu	v8plusa	ultra3cu	64/32/4:8192/512/2
ultra3i	v8plusa	ultra3i	64/32/4:1024/64/4
ultra3iplus	v8plusa	ultra3iplus	64/32/4:4096/64/4
ultra4	v8plusa	ultra4	64/32/4:8192/64/4
ultra4plus	v8plusa	ultra4plus	64/32/4:2048/64/4:32768/64/4**
ultra3i	v8plusa	ultra3i	64/32/4:1024/64/4

** 有关与 UltraSPARC IVplus 和 UltraSPARC T1 芯片关联的缓存属性的说明，请参见第 A-77 页的 "-xcache=c" 。

下表列出了 Intel 架构的 -xtarget 值：

表 A-51　x86 架构上的 -xtarget 扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
386*			
486*			
opteron	sse2	opteron	64/64/2:1024/64/16
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic
pentium3	sse	pentium3	16/32/4:256/32/4
pentium4	sse2	pentium4	8/64/4:256/128/8

* 已废弃。应该使用 -xtarget=generic。有关已废弃选项的完整列表，请参见第 3-7 页的第 3.3.8 节 “废弃的选项”。

注 – 新的 -xtarget=opteron 选项并不自动生成 64 位代码。它扩展为 -xarch=sse2、 -xchip=opteron 和 -xcache=64/64/2:1024/64/16，而产生 32 位代码。您必须在 -xtarget 后面 （右侧）指定 -xarch=amd64 以编译 64 位代码。

默认

在 SPARC 和 x86 设备上，如果未指定 松 target，则假定 -xtarget=generic。

扩展

-xtarget 选项是一个宏，它允许出现在商业购买平台上的 -xarch、 -xchip 和 -xcache 组合的快速简单的说明。 -xtarget 的唯一含义位于它的扩展中。

示例

-xtarget=sun4/15 表示 -xarch=v8a -xchip=micro -xcache=2/16/1。

交互

SPARC V9 架构的编译通过 -xarch=v9|v9a|v9b 选项表示。设置 -xtarget=ultra 或 ultra2 是不必要的或不足够的。如果指定了 -xtarget，则 -xarch=v9、 v9a 或 v9b 选项必须出现在 -xtarget 之后。例如：

```
-xarch=v9 -xtarget=ultra
```

扩展到以下内容并将 -xarch 值还原到 v8。

```
-xarch=v9 -xarch=v8 -xchip=ultra -xcache=16/32/1:512/64/1
```

正确的方法是在 -xtarget 之后指定 -xarch。例如：

```
-xtarget=ultra -xarch=v9
```

警告

在不同的步骤中编译和链接时，必须在编译步骤和链接步骤中使用相同的 -xtarget 设置。

A.2.168 -xthreadvar[=*o*]

(SPARC) 指定 -xthreadvar 来控制线程局部变量的实现。将该选项与 __thread 声明说明符配合使用，以利用编译器线程局部存储功能。用 __thread 说明符声明线程变量之后，请指定 -xthreadvar 启用具有动态（共享）库中位置独立代码（非 PIC 代码）的线程局部存储。关于如何使用 __thread 的更多信息，请参见第 4-2 页的第 4.2 节“线程局部存储”。

值

o 必须是下列值之一：

表 A-52 -xthreadvar 的值

***r* 的值**	**含义**
[no%]dynamic	[不] 编译动态装入的变量。使用 -xthreadvar=no%dynamic 时对线程变量访问明显加快，但是不能在动态库中使用对象文件。也就是说，您只能在可执行文件中使用对象文件。

默认

如果未指定 -xthreadvar，则编译器所用的默认取决于是否启用位置独立的代码。如果启用了位置独立的代码，则选项被设置为 -xthreadvar=dynamic。如果禁用了位置独立的代码，则选项被设置为 -xthreadvar=no%dynamic。

如果指定了 -xthreadvar 但未指定任何参数，则选项被设置为 -xthreadvar=dynamic。

交互

在 Solaris 软件的不同版本上使用线程变量需要命令行上的不同选项。

- 在 Solaris 8 软件中，使用 __thread 的对象必须使用 -mt 来编译，而且必须使用 -mt -L/usr/lib/lwp -R/usr/lib/lwp 进行链接。
- 在 Solaris 9 软件中，使用 __thread 的对象必须使用 -mt 来编译和链接。

警告

如果动态库中存在非位置独立的代码，那么就必须指定 -xthreadvar。

链接程序不支持在动态库中与非 PIC 代码等价的线程变量。由于非 PIC 线程变量要快很多，所以它应该作为可执行文件的默认设置。

另请参见

-xcode、 -KPIC、 -Kpic

A.2.169 -xtime

使 cc 驱动程序报告各种编译步骤的执行时间。

A.2.170 -xtrigraphs[={yes|no}]

根据 ISO/ANSI C 标准的定义启用或禁用对三字母序列的识别。

如果源代码具有包含问号 (?) 的文字串 （编译器将其解释为三字符序列），那么您可以使用 -xtrigraph=no 子选项关闭对三字符序列的识别。

值

您可以指定以下两个 -xtrigraphs 值的其中一个：

表 A-53 -xtrigraphs 值

值	含义
是	启用整个编译单元三字母序列的识别
否	禁用整个编译单元三字母序列的识别

默认

如果不在命令行上包括 -xtrigraphs 选项，则编译器假定 -xtrigraphs=yes。

如果仅指定了 -xtrigraphs，则编译器假定 -xtrigraphs=yes。

示例

请参考以下名为 trigraphs_demo.cc 的示例源文件。

```
#include <stdio.h>

int main()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

如果使用 -xtrigraphs=yes 编译该代码，则输出：

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as (]
```

如果使用 -xtrigraphs=no 编译该代码，则输出：

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

另请参见

关于三字母的信息，请参见 《C 用户指南》一章中关于 ANSI/ISO C 的转换。

A.2.171 -xunroll=*n*

启用在可能的场合下解开循环。

该选项指定编译器是否优化 （解开）循环。

值

当 *n* 为 1 时，建议编译器不要解开循环。

当 *n* 为大于 1 的整数时， -unroll=*n* 使编译器解开循环的次数为 *n*。

A.2.172 -xustr={ascii_utf16_ushort|no}

如果在对象文件中代码包含要被编译器转换成 UTF-16 字符串的字符串文字，则可以使用该选项。如果不指定该选项，编译器既不生成、也不识别 16 位的字符串文字。该选项使 U"*ASCII_string*" 字符串文字处理成无符号短整数的数组。因为这样的字符串还不属于任何标准，所以该选项的作用是使非标准 C++ 得以识别。

不是所有文件都必须使用该选项编译。

值

如果需要支持使用 ISO10646 UTF-16 字符串文字的国际化程序，那么就要指定 `-xustr=ascii_utf16_ushort` 选项。通过指定 `-xustr=no`，可以关闭编译器识别 U"*ASCII_string*" 字符串文字。该选项在命令行上最右侧的实例覆盖了先前的所有实例。

可以指定 `-xustr=ascii_ustf16_ushort`，而无需同时指定 U"*ASCII_string*" 字符串文字。这样执行时不会出现错误。

默认

默认为 `-xustr=no`。如果指定了没有参数的 `-xustr`，则编译器不接受该选项，相反发出一个警告。如果 C 或 C++ 标准定义了语法的含义，则默认是可以更改的。

示例

以下示例显示了在引号中前置为 U 的字符串文字，还显示了指定 `-xustr` 的命令行。

```
example% cat file.cc
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() {return U"fun"};
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

警告

不支持十六位的字符文字。

A.2.173 -xvector[=a]

允许自动生成向量库函数调用和/或生成 SIMD（Single Instruction Multiple Data, 单指令多数据）指令。使用此选项时，必须通过指定 -fround=nearest 使用默认的舍入模式。

a 等效于以下值：

表 A-54 -xvector 标志

值	含义
[no%]lib	如果循环内的数学库调用可转换为对等价向量数学例程的单个调用，则允许 [禁止] 编译器进行此类转换。此类转换可提高那些循环计数较大的循环的性能。
[no%]simd	允许 [禁止] 编译器直接使用本机 x86 SSE SIMD 指令来提高某些循环的性能。仅当目标体系结构支持 SIMD 指令时，编译器才会接受此开关。例如，必须指定 -xarch=amd64、 -xarch=amd64a 或 -xarch=generic64。在指定 -xvector=simd 时，还必须指定 -xO3 或更高的优化级别以及 -xdepend。
是	在以后的发行版本中，该选项可能已过时。请改为指定 -xvector=lib。
否	在以后的发行版本中，该选项可能已过时。请改为指定 -xvector=none。

默认

默认值为 -xvector=%none。如果指定 -xvector，但没有提供标志，则编译器假定 -xvector=lib。

交互

如果在以前没有指定 -xdepend 的情况下在命令行上使用 -xvector，则 -xvector 将会触发 -xdepend。如果未指定优化级别或优化级别低于 -x03，则 -xvector 选项还会将优化级别提高到 -x03。

在装入步骤中，编译器包含 libmvec 库。如果您以单独的命令编译和链接，请确保在链接 CC 命令中使用 -xvector。

A.2.174 -xvis[={yes|no}]

(SPARC) 在使用 VIS™ 指令集软件开发者工具包 (VSDK) 中定义的汇编语言模板时，可以使用 -xvis=[yes|no] 命令。

VIS 指令集是 SPARC v9 指令集的扩展。尽管 UltraSPARC 是 64 位处理器，但在很多情况下数据都限制在 8 位或 16 位范围内，特别是多媒体应用程序中。 VIS 指令可以用一条指令处理 4 个 16 位数据，这个特性使得处理诸如图像、线性代数、信号处理、音频、视频以及网络等新媒体的应用程序的性能大大提高。

默认

默认为 `-xvis=no`。指定 `-xvis` 等价于指定 `-xvis=yes`。

另请参见

关于 VSDK 的更多信息，请参见 `http://www.sun.com/processors/vis/`。

A.2.175 -xwe

通过返回非零的退出状态，将所有警告转换成错误。

A.2.176 -Y*c*,*path*

指定组件 *c* 的位置的新路径。

如果已指定组件的位置，则组件的新路径名称为 path/component_name。该选项被传递到 ld。

值

c 必须是下列值之一：

表 A-55 `-Y` 标志

值	含义
P	更改 `cpp` 的默认目录。
0	更改 `ccfe` 的默认目录。
a	更改 `fbe` 的默认目录。
2 *(SPARC)*	更改 `iropt` 的默认目录。
c *(SPARC)*	更改 `cg` 的默认目录。
O *(SPARC)*	更改 `ipo` 的默认目录。

表 **A-55** `-Y` 标志 (续)

值	含义
k	更改 `CClink` 的默认目录。
l	更改 `ld` 的默认目录。
f	更改 `c++filt` 的默认目录。
m	更改 `mcs` 的默认目录。
u *(x86)*	更改 `ube` 的默认目录。
i *(x86)*	更改 `ube_ipa` 的默认目录。
h *(x86)*	更改 `ir2hf` 的默认目录。
A	指定目录以搜索所有编译器组件。如果路径中找不到组件，搜索将转至编译器所安装的目录。
P	将路径添加到默认库搜索路径。将在默认库搜索路径之前搜索此路径。
S	更改启动目标文件的默认目录。

交互

您可以在命令行指定多个 `-Y` 选项。如果对任何一个组件应用了多个 `Y` 选项，则保留最后一个选项。

另请参见

《Solaris 链接程序和库指南》

A.2.177 `-z[ ]`*arg*

链接编辑器选项。更多信息请参见 `ld`(1) 手册页和 《Solaris 链接程序和库指南》。

附录 B

Pragma

本附录描述了 C++ 编译器 pragma。*pragma* 是编译器指令，允许您为编译器提供附加信息。该信息可以更改您所控制的编译详细信息。例如， `pack` pragma 影响了结构内的数据布局。编译器 pragma 也称为**指令**。

预处理程序关键字 `pragma` 是 C++ 标准的一部分，但 pragma 的形式、内容和含义对每个编译器是不同的。 C++ 标准不定义任何 pragma。

注 – 依赖于 pragma 的代码是不可移植的。

B.1 Pragma 形式

C++ 编译器 pragma 的各种形式如下所示：

```
#pragma keyword
#pragma keyword ( a [ , a ] ...)[ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

变量 *keyword* 指定了特定的指令； *a* 表示参数。

B.1.1 将函数作为 Pragma 参数进行重载

本附录中列出了几个将函数名称作为参数的 pragma。如果重载该函数，则 pragma 使用其前面的函数声明作为其参数。考虑以下示例：

```
int bar(int);
int foo(int);
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

在本示例中，`foo` 指 `foo(double)`，即在 pragma 前面的 `foo` 声明；而 `bar` 指 `bar(int)`，即唯一声明的 `bar`。现在，请考虑以下示例，在该示例中再次重载了 `foo`：

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

在本示例中，`bar` 指 `bar(int)`，即唯一声明的 `bar`。然而，pragma 并不知道要使用哪个 `foo` 版本。要更正此问题，必须将 pragma 直接放在您希望 pragma 使用的 `foo` 定义的后面。

以下 pragma 使用本节中介绍的选择方法：

- `does_not_read_global_data`
- `does_not_return`
- `does_not_write_global_data`
- `no_side_effect`
- `opt`
- `rarely_called`
- `returns_new_memory`

B.2 Pragma 引用

本节描述了 C++ 编译器能识别的 pragma 关键字。

- `align`

  使参数变量内存对齐到指定数目的字节，并覆盖默认。

- `does_not_read_global_data`

  断言函数的指定列表不直接或间接读取全局数据。

- `does_not_return`

向编译器断言，调用的指定函数不返回。

- `does_not_write_global_data`

  断言函数的指定列表不直接或间接写入全局数据。

- `dump_macros`

  提供关于在代码中使用宏的信息。

- `end_dumpmacros`

  标记 `dump_macros` pragma 的结束。

- `fini`

  将指定函数标记为完成函数。

- `hdrstop`

  标识预编译头文件的可用源码前缀的结束。

- `ident`

  将指定字符串放置在可执行文件的 `.comment` 部分。

- `init`

  将指定函数标记为初始化函数。

- `no_side_effect`

  表示函数不更改任何持续状态。

- `pack (`*n*`)`

  控制结构偏移的布局。*n* 的值为数字－ 0、 1、 2、 4 或 8，指定了针对任何结构成员的最坏情况的对齐。

- `rarely_called`

  告知编译器很少调用指定的函数。

- `returns_new_memory`

  断言每个命名的函数返回新分配的内存的地址，并且该指针的别名与任何其他指针的别名不同。

- `unknown_control_flow`

  指定了违反程序调用的通用控制流属性的例程列表。

- `weak`

  定义弱符号绑定。

B.2.1 #pragma align

```
#pragma align integer(variable [,variable...])
```

使用 `align` 使列出的变量内存与 *integer* 字节对齐，并覆盖默认。请遵循以下限制：

- *integer* 必须是大于 1 小于 128 且幂为 2 的整数，有效的值为：1、2、4、8、16、32、64 和 128。
- *variable* 是全局或静态变量，但不可以是局部或类成员变量。
- 如果指定的对齐比默认小，就使用默认。
- Pragma 行必须显示在所涉及的变量的声明之前，否则该行被忽略。
- 在 pragma 行上涉及但不在下面 pragma 行的代码中声明的任何变量都被忽略。以下示例中的变量是正确声明的。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

`#pragma align` 在名称空间内部使用时，必须使用修整名称。例如，以下代码中的 `#pragma align` 语句就是无效的。要纠正该问题，请将 `#pragma align` 语句中的 `a`、`b` 和 `c` 用它们的修整名称替换。

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

B.2.2 #pragma does_not_read_global_data

```
#pragma does_not_read_global_data(funcname [, funcname])
```

该 pragma 断言了指定的例程不直接或间接读取全局数据。允许对调用这些例程的代码进行更好的优化。具体来讲，赋值语句或存储可以围绕这样的调用移动。

指定函数的原型被声明后，该 pragma 才可用。如果全局访问的断言不为真，那么程序的行为就是未定义的。

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节“将函数作为 Pragma 参数进行重载”。

B.2.3 #pragma does_not_return

```
#pragma does_not_return(funcname [, funcname])
```

该 pragma 向编译器断言，调用指定例程不会返回。这样编译器可以执行与假定一致的优化。例如，寄存器生命周期在允许更多优化的调用点终止。

如果指定的函数不返回，程序的行为就是未定义的。

指定函数的原型被声明后，该 pragma 才是可用的，如以下示例所示：

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节 “将函数作为 Pragma 参数进行重载”。

B.2.4 #pragma does_not_write_global_data

```
#pragma does_not_wrtie_global_data(funcname [, funcname])
```

该 pragma 断言了指定的例程列表不直接或间接写入全局数据。允许对调用这些例程的代码进行更好的优化。具体来讲，赋值语句或存储可以围绕这样的调用移动。

指定函数的原型被声明后，该 pragma 才可用。如果全局访问的断言不为真，那么程序的行为就是未定义的。

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节 “将函数作为 Pragma 参数进行重载”。

B.2.5 #pragma dumpmacros

```
#pragma dumpmacros (value[,value...])
```

要查看宏在程序中如何工作时，请使用该 pragma。该 pragma 提供了诸如宏定义、取消定义和用法实例的信息，并按宏的处理顺序将输出打印到标准错误 (stderr)。dumpmacros pragma 在文件结束或遇到 #pragma end_dumpmacro 后生效。请参见第 B-7 页的第 B.2.6 节 "#pragma end_dumpmacros"。您可以用以下参数代替 *value*：

值	含义
defs	打印所有宏定义
undefs	打印所有取消定义的宏
use	打印关于使用的宏的信息
loc	打印位置 （路径名称和行号）以及 defs、 undefs 和 use
conds	打印在条件指令中使用的宏的信息
sys	打印系统头文件中所有宏的定义、取消定义和使用的信息

注 – 子选项 loc、conds 和 sys 是 defs、undefs 和 use 选项的限定符。loc、conds 和 sys 本身并不会产生任何效果。例如， #pragma dumpmacros=loc,conds,sys 是无效的。

dumpmacros pragma 与命令行选项具有相同的效果，不过 pragma 会覆盖命令行选项。请参见第 A-86 页的第 A.2.118 节 "-xdumpmacros[=value[,value...]]"。

dumpmacros pragma 不嵌套，因此处理 #pragma end_dumpmacros 时，以下代码行停止打印宏信息：

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros pragma 的效果是累积性的。以下代码行

```
#pragma dumpmacros(defs, undefs)
#pragma dumpmacros(loc)
```

具有和以下行相同的效果

```
#pragma dumpmacros(defs, undefs, loc)
```

如果使用选项 `#pragma dumpmacros=use,no%loc`，那么使用的每个宏名称仅打印一次。如果使用选项 `#pragma dumpmacros=use,loc`，那么每次使用宏时都打印位置和宏名称。

B.2.6 #pragma end_dumpmacros

```
#pragma end_dumpmacros
```

该 pragma 标记 `dumpmacros pragma` 的结束，并停止打印关于宏的信息。如果 `dumpmacros` pragma 之后不使用 `end_dumpmacros` pragma，那么 `dumpmacros` pragma 在文件结束之前继续生成输出。

B.2.7 #pragma fini

```
#pragma fini (identifier[,identifier...])
```

使用 `fini` 将 *identifier* 标记为完成函数。这些函数应为 `void` 类型，不接受任何参数，并且当程序在程序控制下终止或从内存删除包含的共享对象时被调用。和初始化函数一样，完成函数按链接编辑器处理的顺序执行。

在源文件中，`#pragma fini` 中指定的函数在该文件中的静态析构函数之后执行。在 pragma 中使用标识符之前，请先声明这些标识符。

B.2.8 #pragma hdrstop

将 hdrstop pragma 嵌入源文件头文件中以标识可用源码前缀的结束。例如，考虑以下文件：

```
example% cat a.cc
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.cc
#include "a.h"
#include "b.h"
#include "c.h"
```

可用源码前缀在 c.h 处结束，因此可以在每个文件中的 c.h 后插入 #pragma hdrstop。

#pragma hdrstop 只能显示在源文件 （用 CC 命令指定）的可用前缀的结尾部分。不在任何包括文件中指定 #pragma hdrstop。

请参见第A-116页的第A.2.150节 "-xpch=v" 和第A-119页的第A.2.151节 "-xpchstop=file"。

B.2.9 #pragma ident

```
#pragma ident string
```

使用 ident 将 *string* 放置在可执行文件的 .comment 节。

B.2.10 #pragma init

```
#pragma init(identifier[,identifier...])
```

使用 `init` 将 *identifier* 标记为初始化函数。这些函数应为 `void` 类型，不接受任何参数，并且在执行开始构造程序内存映像时调用。将共享对象带到内存中的操作时，在程序启动或某些动态装入操作 （例如 `dlopen()`）期间，执行共享对象中的初始化函数。调用到初始化函数的唯一顺序就是链接编辑器静态和动态处理该函数的顺序。

在源文件中， `#pragma init` 中指定的函数在该文件中的静态构造函数之后执行。在 pragma 中使用标识符之前，请先声明这些标识符。

B.2.11 #pragma no_side_effect

```
#pragma no_side_effect(name[,name...])
```

使用 `no_side_effect` 以表示函数不更改任何持续状态。Pragma 声明了命名的函数不具有任何副作用。这意味着函数将返回仅依赖于传递参数的结果。此外，函数和后面的函数调用：

- 不读取或写入调用点的调用者中可视的程序状态的任何部分。
- 不执行 I/O。
- 不更改调用点不可视程序状态的任何部分。

编译器执行优化时可以使用该信息。

如果函数具有副作用，执行调用该函数的程序的结果是未定义的。

name 参数指定了当前转换单元内函数的名称。 Pragma 必须与函数在相同的作用域，并且必须在函数声明之后出现。 pragma 必须在函数定义之前。

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节 “将函数作为 Pragma 参数进行重载”。

B.2.12 #pragma opt

```
#pragma opt level (funcname[, funcname])
```

funcname 指定当前转换单元内部定义的函数的名称。 *level* 的值指定命名的函数的优化级别。您可以指定优化级别 0、1、2、3、4、5。您可以通过将 *level* 设置为 0 来关闭优化。必须在 pragma 之前使用原型或空参数列表声明函数。 pragma 则必须对要优化的函数进行定义。

pragma 中列出的任何函数的优化级别被降低为值 `-xmaxopt`。当 `-xmaxopt=off` 时，忽略 pragma。

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节“将函数作为 Pragma 参数进行重载”。

B.2.13 #pragma pack(*n*)

```
#pragma pack([n])
```

使用 pack 以影响结构成员的包装。

如果存在，*n* 就必须是 0 或幂为 2 的数。不为 0 的值指示编译器，数据类型使用较小的 *n* 字节对齐和平台的自然对齐。例如，以下指令使得定义在指令之后（和随后的 pack 指令之前）的所有结构成员对齐不严格超过 2 字节的边界，即使正常对齐是 4 或 8 字节边界。

```
#pragma pack(2)
```

n 为 0 或省略时，成员对齐还原为自然对齐值。

如果 *n* 的值等于或大于平台上最严格的对齐时，指令具有自然对齐的效果。下表显示了每个平台最严格的对齐。

表 B-1 平台上最严格的对齐

平台	最严格的对齐
x86	4
SPARC 通用、 V7、 V8、 V8a、 V8plus、 V8plusa、 V8plusb	8
SPARC V9、 V9a、 V9b	16

pack 指令应用到下一个 pack 指令之前的所有结构定义。如果在具有不同包装的不同转换单元中定义了相同的结构，那么程序会因某种原因而失败。具体来将，不应该在包括定义预编译库接口的头文件之前使用 pack 指令。推荐用法是将 pack 指令放置在包装结构之前的程序代码中，并在结构之后放置 #pragma pack()。

在 SPARC 平台上使用 #pragma pack 包装比类型的默认对齐更紧密的对齐时，必须为程序的编译和链接指定 -misalign 选项。下表显示了整型数据类型的存储大小和默认对齐。

表 B-2 存储大小和默认对齐字节数

类型	SPARC V8 大小，对齐	SPARC V9 大小，对齐	x86 大小，对齐
bool	1, 1	1, 1	1, 1
char	1, 1	1, 1	1, 1
short	2, 2	2, 2	2, 2
wchar_t	4, 4	4, 4	4, 4
int	4, 4	4, 4	4, 4
long	4, 4	8, 8	4, 4
float	4, 4	4, 4	4, 4
double	8, 8	8, 8	8, 4
long double	16, 8	16, 16	12, 4
指向数据的指针	4, 4	8, 8	4, 4
指向函数的指针	4, 4	8, 8	4, 4
指向成员数据的指针	4, 4	8, 8	4, 4
指向成员函数的指针	8, 4	16, 8	8, 4

B.2.14 #pragma rarely_called

```
#pragms rarely_called(funcname[, funcname])
```

该 pragma 提示编译器，指定的函数很少被调用。这样编译器就可以在这种例程的调用点上执行性能分析反馈式优化，而无需性能分析数据收集阶段的开销。因为该 pragma 只是建议，所以编译器不执行基于该 pragma 的任何优化。

只有声明指定的函数原型之后， #pragma rarely_called 预处理程序指令才是可用的。以下是 #pragma rarely_called 的示例：

```
extern void error (char *message);
#pragma rarely_called(error)
```

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节 “将函数作为 Pragma 参数进行重载”。

B.2.15 #pragma returns_new_memory

```
#pragma returns_new_memory(name[,name...])
```

该 pragma 断言了每个命名的函数返回新分配内存的地址，并且指针不具有到其他任何指针的别名。该信息允许优化器更好地跟踪指针值并明确内存位置。这样可以改善调度和管线操作。

如果该断言为假，那么执行调用该函数的程序的结果是未定义的。

name 参数指定了当前转换单元内函数的名称。 Pragma 必须与函数在相同的作用域，并且必须在函数声明之后出现。 pragma 必须在函数定义之前。

如果函数被重载，那么该 pragma 应用到最近定义的函数。如果最近定义的函数不具有相同的标识符，那么程序出现错误。

有关 pragma 如何将重载的函数名称视为参数的详细说明，请参见第 B-2 页的第 B.1.1 节 “将函数作为 Pragma 参数进行重载”。

B.2.16 #pragma unknown_control_flow

```
#pragma unknown_control_flow(name[,name...])
```

使用 `unknown_control_flow` 指定违反程序调用的通用控制流属性的例程列表。例如，从对任何其他例程的任意调用可以遇到对 `setjmp()` 的调用后的语句。该语句由对 `longjmp()` 的调用遇到。

因为这种例程使标准流程图分析无效，调用它们的例程不能安全地优化，所以要禁用优化器来编译这些例程。

如果函数名称被重载，那么会选择最近声明的函数。

B.2.17 #pragma weak

```
#pragma weak name1 [ = name2]
```

使用 weak 定义弱全局符号。该 pragma 主要在源文件中用于生成库。链接程序在不能解决弱符号时不会发出警告。

weak pragma 可以用以下两种形式之一来指定符号：

- **字符串形式**。字符串必须是 C++ 变量或函数的修整名称。无效修整名称引用的行为是不可预测的。后端可以或不可以产生无效修整名称引用的错误。无论是否产生错误，使用无效修整名称时后端的行为都是不可预测的。
- **标识符形式**。标识符必须是在编译单元中以前声明的 C++ 函数的明确标识符。标识符形式不能用于变量。前端 (ccfe) 遇到无效的标识符引用时将会产生错误消息。

#pragma weak *name*

在形式 #pragma weak *name* 中，指令生成了 *name* 弱符号。如果链接程序没有找到 *name* 的符号定义，将不会出现错误消息，也不会出现符号的多个弱定义的错误消息。链接程序仅执行第一个遇到的定义。

如果另一个编译单元具有函数或变量的强定义，那么 *name* 将链接到该函数或变量。如果没有 *name* 的强定义，那么链接符号将具有值 0。

以下指令将 ping 定义为弱符号。如果链接程序无法找到名为 ping 的符号定义，那么不会生成任何错误消息。

```
#pragma weak ping
```

#pragma weak *name1* = *name2*

在 #pragma weak *name1* = *name2* 形式中，符号 *name1* 成为对 *name2* 的弱引用。如果 *name1* 不在其他地方定义，那么 *name1* 将具有值 *name2*。如果 *name1* 在其他地方定义，那么链接程序使用该定义并忽略对 *name2* 的弱引用。以下指令指示链接程序解决对 bar 的任何引用，前提是在程序中的任何位置定义它，否则解决对 foo 的引用。

```
#pragma weak bar = foo
```

在标识符形式中， *name2* 必须在当前编译单元中声明和定义。例如：

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

使用字符串形式时，符号不需要预先声明。如果以下示例中的 `_bar` 和 `bar` 都是 `extern "C"`，那么函数不需要声明。不过， `bar` 必须在相同的对象中定义。

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

重载函数

使用标识符形式时，必须在 pragma 位置的作用域中正好有一个具有指定名称的函数。尝试与重载函数一起使用 `#pragma weak` 的标识符形式是错误的。例如：

```
int bar(int);
float bar(float);
#pragma weak bar         // 错误，不明确的函数名称
```

要避免错误，请使用字符串形式，如以下示例所示。

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // 使浮点 bar(int) 为弱符号
```

更多信息，请参见 《Solaris 链接程序和库指南》。

术语表

ABI 请参见*应用程序二进制接口*。

ANSI C 美国国家标准学会定义的 C 编程语言。 ANSI C 与 ISO 定义相同。请参见 *ISO*。

ANSI/ISO C++ 美国国家标准学会和 ISO C++ 编程语言标准。请参见 *ISO*。

cfront C++ 到 C 的编译器程序，可以将 C++ 转换为 C 源代码，然后用标准 C 编译器编译。

ELF 文件 编译器生成的可执行和链接格式文件。

ISO 国际标准化组织。

K&R C Brian Kernighan 和 Dennis Ritchie 在 ANSI C 之前开发的实际上的 C 编程语言标准。

locale 地理位置和/或语言唯一的一组约定，例如日期、时间和货币格式。

pragma 指示编译器执行特定操作的编译器预处理程序指示或特殊的注释。

stab 在对象代码中生成的符号表条目。相同的格式被用在 `a.out` 文件和 ELF 文件中以包含调试信息。

stack 数据存储方法，通过该方法数据可以增加到堆栈顶部或从堆栈顶部删除数据，采用的是后进先出策略。

VTABLE（虚函数表） 编译器为包含虚函数的每个类创建的表。

绑定 将函数调用与特定函数定义关联。更一般的说来，将名称与特定的实体关联。

编译器选项 更改编译器行为的指令。例如，`-g` 选项告知编译器生成调试器的数据。同义字：**标志**、**开关**。

变量 标识符命名的数据项。每个变量都具有类型 （例如 `int` 或 `void`）和作用域。另见*类变量*，*实例变量*，*局部变量*。

标志 请参见*编译器选项*。

捕获 为了执行其他操作，而对诸如程序执行等操作的截获。截获引起微处理器操作的临时中止，并将程序控制转交给另一个源。

成员函数 是函数而非数据定义或类型定义的类元素。

抽象方法 不包含实现的方法。

抽象类 包含一个或多个抽象方法并因此不能被实例化的类。定义抽象类的目的是为了通过实现抽象方法，使其他类可以扩展抽象类并使其固定。

递增链接程序 通过仅将更改后的 `.o` 文件链接到前一个可执行文件来创建新的可执行文件的链接程序。

动态绑定 在运行时函数调用到函数体的连接。有虚函数时才需动态绑定。也称为**后续绑定，运行时绑定**。

动态类型 由具有不同声明类型的指针或引用访问的对象的实际类型。

动态类型转换 将指针或引用从声明的类型转换到与引用到的动态类型一致的任何类型的安全方法。

多继承 直接源于多个基类的派生类的继承。

多态 引用到对象的指针或引用的动态类型与声明的指针或引用类型不同的能力。

多线程 在单或多处理器系统上开发并行应用程序的软件技术。

二进制兼容性 链接对象文件的能力，对象文件由某一个发行版本编译，而使用另一个不同发行版本的编译器。

范围 操作或定义应用的范围。

方法 在某些面向对象的语言中，成员函数的另外一个名称。

符号 表示某些程序实体的名称或标签。

符号表 程序编译时显示的所有标识符、程序中标识符的位置和属性的列表。编译器使用该表来解释标识符的使用。

构造函数 在创建类对象时编译器自动调用的特殊类成员函数，这样可以确保对象实例变量的初始化。构造函数必须始终具有与该函数所属的类相同的名称。请参见*析构函数*。

关键字 在编程语言中具有唯一含义，并且仅在该语言的专用上下文中使用的字。

函数多态 请参见*函数重载*。

函数模板 允许编写可以稍后用作模型或模式的单一函数 （用于编写相关函数）的机制。

函数原型 描述函数与程序其他部分接口的声明。

函数重载 将相同的名称但不同的参数类型和数字赋予不同的函数。也称为**函数多态**。

后续绑定 请参见*动态绑定*。

基类 请参见**继承**。

继承 面向对象编程的一个功能，使得编程人员可以从现有类 （基类）派生新的类 （派生类）。有以下三种继承：公共的、受保护的和专用的。

静态绑定 在编译期间函数调用到函数体的连接。也称为**预先绑定**。

局部变量 在块内已知的数据项，但块外代码不可访问。例如，在方法内定义的任何变量都是局部变量，在方法外部无法使用。

开关 请参见*编译器选项*。

类 由命名的数据元素 （可以是不同类型的数据元素）和可以和数据一起执行的一组操作组成的用户定义数据类型。

类变量 作为一个整体与特定类关联但与类的特定实例不关联的数据项。类变量在类定义中定义。类变量也称为静态字段。另见*实例变量*。

类模板 描述一组类或相关数据类型的模板。

类型 使用符号方法的描述。基本类型是 `integer` 和 `float`。所有其他类型都是从这些基本类型构造的，构造方法有：将基本类型收集到数组或结构中，或增加诸如指针或常量属性等的修饰符。

链接程序 连接对象代码和库以形成完整的可执行程序的工具。

幂等 头文件属性，在一个转换单元中包括多次与包括一次具有相同效果。

名称空间 控制全局名称的作用域的机制，做法是允许全局空间划分为独立的唯一命名的作用域。

名称修整 在 C++ 中，大量函数可以共享相同的名称，因此仅用名称并不能很好的区分不同的函数。编译器通过名称修整解决了这个问题 - 为函数创建由函数名称和参数的某些组合组成的唯一名称来启用类型安全链接。名称修整也称为**名称装饰**。

模板数据库 包含需要处理并实例化模板 （程序需要）的所有配置文件的目录。

模板选项文件 由用户提供的文件，其中包含模板编译选项、源码位置和其他信息。模板选项文件现在已过时，不应该使用。

模板专门化 类模板成员函数的专用实例，用于默认不能处理给出的足够类型时覆盖默认的实例化。

内联函数 用实际函数代码替换函数调用的函数。

派生的类 请参见**继承**。

实例变量 与特定对象关联的任何数据项。类的每个实例具有在类中定义的实例变量的自身副本。实例变量也称为字段。另见*类变量*。

实例化 C++ 编译器从模板创建可用的函数或对象的过程。

数据成员 是**数据** 而非函数或类型定义的类元素。

数据类型 允许表示诸如字符、整数或浮点数等的机制。类型决定了分配到变量的存储以及能在变量上执行的操作。

数组 内存中连续存储一组单一数据类型值的数据结构。每个值可以按在数组中的位置访问。

损坏 请参见*名称修整*。

析构函数 类对象被销毁或运算符 `delete` 应用到类指针后，编译器自动调用的特殊类成员函数。析构函数必须始终具有与该函数所属的类相同的名称，该类前有一个 (~)。请参见**构造函数**。

选项 请参见*编译器选项*。

异常 在防止程序继续的程序正常流中出现的错误。某些错误的原因包括了内存枯竭或被零除。

异常处理 设计用于截取并防止错误的错误恢复过程。在程序执行期间，如果检测到同步错误，那么程序的控制返回到在执行初期注册的异常处理程序，并且忽略包含错误的代码。

异常处理程序 用于处理错误而专门编写的代码，以及异常发生时为已注册的处理程序自动调用的代码。

应用程序二进制接口 编译的应用程序和运行应用程序的操作系统之间的二进制系统接口。

优化 改善编译器生成的对象代码执行效率的过程。

右值 位于赋值运算符右侧的变量。右值可以读取而不能被更改。

预先绑定 请参见*静态绑定*。

运算符重载 使用同一运算符表示法产生不同结果的能力。函数重载的特殊形式。

运行时绑定 请参见*动态绑定*。

运行时类型标识 (RTTI) 提供标准方法以让程序在运行时决定对象类型的机制。

重载	将相同的名称赋予多个函数或运算符。
子例程	函数。在 Fortran 中，子例程是不返回值的函数。
左值	指定在内存中存储变量数据值位置的表达式，以及显示在赋值运算符左侧的变量实例。

索引

C

D

E

F

G

H

J

J

K

L

M

N

O

P

T

U

V

W

X

Y

Z