Sun Studio 12 Update 1 ： C++ 用户指南

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

文件号码 821–0389
2009 年 9 月

090904@22749

目录

示例

前言

本《Sun Studio 12 Update 1 C++ 用户指南》介绍了 Sun Studio C++ 编译器 **CC** 的环境和命令行选项。

本指南的目标读者是具有 C++ 语言和 Solaris 或 Linux 操作环境的工作经验、并希望了解如何有效使用 Sun Studio C++ 编译器的应用程序开发者。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	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。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表列出了 C shell、Bourne shell 和 Korn shell 的缺省 UNIX® 系统提示符和超级用户提示符。

表 P–2 shell 提示符

shell	提示符
C shell 提示符	`machine_name%`
C shell 超级用户提示符	`machine_name#`
Bourne shell 和 Korn shell 提示符	`$`
Bourne shell 和 Korn shell 超级用户提示符	`#`

受支持的平台

此 Sun™ Studio 发行版支持使用 SPARC 和 x86 系列处理器体系结构的系统：UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T。可从以下位置获得硬件兼容性列表，在列表中可以查看您正在使用的 Solaris 操作系统版本支持的系统：`http://www.sun.com/bigadmin/hcl`。这些文档中给出了平台类型间所有实现的区别。

在本文档中，与 x86 相关的术语的含义如下：

- "x86" 泛指 64 位和 32 位的 x86 兼容产品系列。
- "x64" 指出了有关 AMD64 或 EM64T 系统的特定 64 位信息。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

有关受支持的系统，请参阅硬件兼容性列表。

访问 Sun Studio 文档

可以访问以下位置的文档：

- 可以从位于 `http://developers.sun.com/sunstudio/documentation` 的文档索引页中获取文档。
- IDE 所有组件的联机帮助可通过 IDE 中的“帮助”菜单以及许多窗口和对话框上的“帮助”按钮获取。
- 性能分析器的联机帮助可通过性能分析器中的“帮助”菜单以及许多窗口和对话框上的“帮助”按钮获取。

可以通过 Internet 访问 `docs.sun.com` Web 站点 (`http://docs.sun.com`) 阅读、打印和购买 Sun Microsystems 的各种手册。

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。

采用易读格式的文档

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。可以按照下表所述找到文档的易读版本。

文档类型	易读版本的格式和位置
手册	HTML，位于 `http://docs.sun.com`
自述文件	HTML，位于开发者门户网站 `http://developers.sun.com/sunstudio/documentation/ss12u1/`
手册页	HTML，位于开发者门户网站 `http://developers.sun.com/sunstudio/documentation/ss12u1/`
联机帮助	HTML，可通过 IDE 中的“帮助”菜单和“帮助”按钮获取
发行说明	HTML，位于 `http://docs.sun.com`

开发者资源

访问 `http://developers.sun.com/sunstudio` 查看下列经常更新的资源：

- 有关编程技术和最佳做法的文章
- 软件文档以及随软件一起安装的文档的更正信息
- 指导您使用 Sun Studio 工具逐步完成开发任务的教程
- 有关支持级别的信息
- 用户论坛
- 可下载的代码示例
- 新技术预览

Sun Studio 门户是 Sun Developer Network Web 站点 (`http://developers.sun.com`) 上面向开发者的众多其他资源之一。

联系技术支持

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 URL 向 Sun 提交您的意见：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 821-0389-10。

第 1 部分

C++ 编译器

◆ ◆ ◆ 第 1 章

1 C++ 编译器

本章提供了有关当前 Sun Studio C++ 编译器的一般信息：

1.1 Sun Studio 12 Update 1 C++ 5.10 编译器的新特性和新功能

本节简要介绍在 Sun Studio 12 Update 1 C++ 5.10 编译器发行版中引入的 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。

- 如果应用程序代码中包含带有参数或返回值（使用 `_m128/_m64` 数据类型）的函数，则在 x86 平台的 Solaris OS 上或 Linux OS 上，编译器创建的目标文件与以前的编译器版本不兼容。使用 `.il` 内联函数文件、汇编程序代码或调用这些函数的 `asm` 内联语句的用户也需要注意这种不兼容性。
- 为 x86 处理器 `woodcrest`、`penryn`、`nehalem`、`core2` 和 SPARC 处理器 `ultraT2plus` 及 `sparc64vii` 新增了 `-xtarget` 值。
- 为 x86 体系结构 `ssse3`、`sse4_1`、`sse4_2` 和 SPARC 体系结构 `sparcima` 新增了 `-xarch` 值。
- 为 SPARC 处理器 `sparc64vii`、`ultraT2plus` 新增了 `–xchip` 值。为 x86 处理器 `core2`、`penryn`、`nehalem` 新增了 `–xchip` 值。
- `-xprofile=collect` 和 `-xprofile=use` 选项为动态链接的多线程应用程序的文件配置提供了改进的支持。
- `-xcrossfile=1` 选项成为 `-xipo=1` 选项的别名。`-xcrossfile=0` 选项不再起任何作用。具体来说，`-xcrossfile=1` 和 `-xcrossfile=0` 等效于 `-xipo=1`。
- 在 Solaris 平台上，`-xpec[=yes|no]` 选项会生成可重新编译的 PEC 二进制文件，以用于自动调优系统 (Automatic Tuning System, ATS)。
- `-Y` 选项不接受 `i` 作为参数。
- 在 SPARC 平台上，现在针对 `x03` 或更高优化级别隐式启用 `-xdepend` 选项，`-fast` 选项的扩展中不再包括此选项。

- OpenMP 3.0 的支持包括 `libmtsk` 库。缺省情况下，OpenMP 程序将与此库链接，而不是与 Solaris OS 中的 `libmtsk` 库链接。
- `-xannotate[=yes|no]`（仅限 SPARC 平台）指示编译器创建二进制文件，之后这些文件可以由 `binopt(1)` 之类的二进制文件修改工具进行转换。
- 现在，在 x86 Solaris 平台上支持 `-xia`（区间运算）选项。
- 现在，在 x86 Solaris 平台和 Linux 平台上支持 `-xipo_archive` 选项。
- `-Qoption` 选项不再接受 `ube_ipa` 作为参数。
- `-fast` 选项的扩展现在包括 `-D_MATHERR_ERRNO_DONTCARE`。
- 现在支持用于显示并行化警告消息的 `-xvpara` 选项。
- `-sb`、`-sbfast`、`-xsb` 和 `-xsbfast` 选项现已过时，默认忽略。
- 现在，在优化级别为 `-O` 或 `-xO` 的情况下，指定 `-g` 选项时，只要不同时指定 `+d`，编译器就会内联代码。
- 现在支持 pragma `must_have_frame`。
- 在标准 C++ 中，switch 语句中的 case 标签只能有一个关联值。Sun Studio C++ 编译器允许在某些编译器中出现扩展，称为 case 范围。
- 编译器通常在 `/tmp` 目录中创建临时文件。可以通过设置 `TMPDIR` 环境变量指定另一个目录。
- 现在支持函数的以下属性：`__attribute__((const))` `__attribute__((constructor))` `__attribute__((destructor))`
- 现在仅对 `struct` 类型和 `enum` 类型支持变量的以下属性：`__attribute__((packed))`
- 现在支持通用字符名称。
- 现在支持循环 pragma。
- 现在支持用户定义的宏 variadic 参数名称。
- 添加了用来指定预处理程序 include 文件的 `-include` *filename* 选项。

1.2 x86 特殊注意事项

针对 x86 Solaris 平台进行编译时，有一些重要问题值得注意。

传统的 Sun 样式的并行 pragma 在 x86 上不可用。而改用 OpenMP。有关将传统并行化指令转换为 OpenMP 的信息，请参见《Sun Studio 12 Update 1： OpenMP API 用户指南》。

`xarch` 设置为 `-sse`、`sse2`、`sse2a`、`sse3` 或更高时编译的程序只能在提供这些扩展和功能的平台上运行。

从 Solaris 9 4/04 开始的 Solaris OS 发行版在 Pentium 4 兼容的平台上支持 SSE/SSE2。早期版本的 Solaris OS 不支持 SSE/SSE2。如果所运行的 Solaris OS 不支持由 `-xarch` 选定的指令集，则编译器无法为该指令集生成链接代码。

如果在不同的步骤中进行编译和链接，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。

在 x86 上得到的数值结果可能与在 SPARC 上得到的结果不同，这是由 x86 80 位浮点寄存器造成的。为了最大限度减少这些差异，请使用 `-fstore` 选项或使用 `-xarch=sse2` 进行编译（如果硬件支持 SSE2）。

Solaris 和 Linux 之间的数值结果也可能不同，因为内部数学库（例如，`sin(x)`）并不相同。

1.3 针对 64 位平台进行编译

使用 `–m32` 选项可针对 ILP32 32 位模型进行编译。使用 `–m64` 选项可针对 LP64 64 位模型进行编译。

ILP32 模型指定 C++ 语言的 int、long 和 pointer 数据类型的宽度均为 32 位。LP64 模型指定 long 和 pointer 数据类型均为 64 位。Solaris 和 Linux OS 还支持 LP64 内存模型下的大型文件和大型数组。

如果使用 `–m64` 进行编译，则生成的可执行文件仅能在运行 64 位内核的 Solaris OS 或 Linux OS 下的 64 位 UltraSPARC 或 x86 处理器上运行。64 位对象的编译、链接和执行只能在支持 64 位执行的 Solaris 或 Linux OS 上进行。

1.4 二进制兼容验证

在 Solaris 系统上，从 Sun Studio 11 开始，Sun Studio 编译器编译的程序二进制文件都标记了体系结构硬件标志（表示由编译的二进制文件采用的指令集）。在运行时，会检查这些标记标志以确认二进制文件是否可在尝试在其上执行的硬件上运行。

如果程序不包含这些体系结构硬件标志，或者如果平台没有启用适当的功能或指令集扩展，则运行此程序可能会导致段故障或错误结果，且不会显示任何显式警告消息。

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（利用 SSE、SSE2、SSE2a、SSE3 和更新的指令及扩展）的程序。

1.5 标准一致性

C++ 编译器 (`CC`) 支持 C++ ISO 国际标准 ISO IS 14882:2003 **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”是指这些标准的这些版本之外的功能。

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。

1.6 C++ 自述文件

C++ 编译器的自述文件列出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

可以在 Sun Developer Network Sun Studio 门户网站 `http://developers.sun.com/sunstudio/documentation` 上找到此 Sun Studio 发行版及早期发行版的 C++ 编译器自述文件。

1.7 手册页

联机手册 (man) 页提供了关于命令、函数、子例程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，手册页参考都以主题名称和手册节编号表示：通过 `man CC` 访问 `CC(1)`。例如，可在 `man` 命令中使用 `-s` 选项来访问 `ieee_flags(3M)` 指示的其他部分：

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

1.8 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 (Extended UNIX Characte, EUC) 的字符集，在该字符集中，字符串内每个空字节是一个空字符，而字符串内每个 ascii 值为 "/" 的字节是一个 "/" 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，以后链接到可执行文件、静态（归档）库 (`.a`) 文件（使用 `–xar`）或动态（共享）库 (`.so`) 文件中
- 将目标文件或库文件（或两者）链接或重链接成可执行文件
- 在运行时调试处于启用状态的情况下编译可执行文件 (`-g`)
- 使用运行时语句或过程级文件配置 (`-pg`) 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。有关命令行选项的完整参考，请参见第 193 页中的 “16.8 使用 dlopen 从 C 程序访问 C++ 库”。

注 – 本章中的命令行示例说明了 `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 C++ 源文件（后缀为表 2–1 中所列有效后缀之一）
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，并且在缺省情况下编译可执行程序生成文件 a.out。要启动该程序，请在命令提示符下键入可执行文件的名称 a.out。

传统的方法是，UNIX 编译器为可执行文件命名 a.out。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。因此，改用 -o 编译器选项来指定可执行输出文件的名称，如以下示例所示：

```
example% CC– o greetings greetings.cc
```

在此示例中， -o 选项通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。）

也可以在每次编译后使用 mv 命令来为缺省的 a.out 文件重命名。无论是哪种方法，都可以通过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

本章其余部分讨论了 CC 命令使用的约定、编译器源代码行指令和其他有关编译器的使用问题。

2.2.1 命令语法

编译器命令行的一般语法如下所示：

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

选项是前缀为短划线 (–) 或加号 (+) 的选项关键字。某些选项带有参数。

通常，编译器选项的处理顺序是从左到右，从而允许有选择地覆盖宏选项（包含其他选项的选项）。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：

- 所有链接程序选项和 `-features`、`–I -l`、`–L`、`-library`、`–pti`、`–R`、`-staticlib`、`-U`、`-verbose`、`-xdumpmacros` 和 `-xprefetch` 选项都会累积，但它们不会覆盖。
- 所有 `–U` 选项都在所有 `–D` 选项之后处理。

源文件、目标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，在启用了运行时调试的情况下，使用 `CC` 编译两个源文件（`growth.C` 和 `fft.C`）来生成名为 `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

命令行上附加在文件名后面的后缀确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++	以 C++ 源文件编译，将目标文件放在当前目录中；目标文件的缺省名称是源文件名称加上 `.o` 后缀。
`.C`	C++	操作与 `.c` 后缀相同。
`.cc`	C++	操作与 `.c` 后缀相同。
`.cpp`	C++	操作与 `.c` 后缀相同。
`.cxx`	C++	操作与 `.c` 后缀相同。
`.c++`	C++	操作与 `.c` 后缀相同。
`.i`	C++	将预处理程序输出文件作为 C++ 源文件处理。操作与 `.c` 后缀相同。
`.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`	目标文件	将目标文件传递到链接程序。
`.a`	静态（归档）库	将目标库名传递到链接程序。

表 2–1 C++ 编译器识别的文件名称后缀 （续）

后缀	语言	操作
.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。编译器编译的单个源文件和其直接或间接支持的任何文件一起统称为**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

缺省情况下，该编译器不使用高速缓存。只有指定了 `-instances=extern` 时，它才使用高速缓存。如果编译器使用高速缓存，它会检查高速缓存目录的版本，并在遇到高速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：

```
Template Database at ./SunWS_cache is incompatible with
this compiler
```

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。

升级编译器时，最好清除缓存。可对包含模板高速缓存目录（大多数情况下，模板高速缓存目录名为 `SunWS_cache`）的每个目录运行 `CCadmin -clean`。也可以使用 `rm -rf SunWS_cache`。

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，使用 `CC` 编译三个源文件并链接目标文件以生成名为 `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

在前面的示例中，编译器会自动生成加载器目标文件（`file1.o`、`file2.o` 和 `file3.o`），然后调用系统链接程序来为 `prgrm` 文件创建可执行程序。

编译后，目标文件（`file1.o`、`file2.o` 和 `file3.o`）仍保留不变。此约定让您易于重新链接和重新编译文件。

注 – 如果只编译了一个源文件，且在同一个操作中链接了程序，则对应的 `.o` 文件将会被自动删除。要保留所有 `.o` 文件，就不要在同一个操作中进行编译和链接，除非编译多个源文件。

如果编译失败，您将收到每个错误的对应消息。对于那些出现错误的源文件，不会生成 `.o` 文件，也不会生成可执行程序。

2.4.2 分别编译和链接

可以在不同的步骤中进行编译和链接。如果使用 `-c` 选项，将编译源文件并生成 `.o` 目标文件，但不创建可执行文件。如果不使用 `-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，仅修复一个文件就不需要完整重新编译。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          Make new object file
example% CC -o prgrm file1.o file2.o file3.o          Make executable file
```

请确保链接步骤列出了生成完整程序所需的**全部**目标文件。如果在此步骤中缺少任何目标文件，链接将会失败，并出现 "undefined external reference" 错误（缺少例程）。

2.4.3 一致编译和链接

如果在不同的步骤中进行编译和链接，则使用第 47 页中的 “3.3.3 编译时选项和链接时选项”中所列的编译器选项时，一定要在编译和链接时保持一致。

如果使用其中任何选项编译子程序，必须在链接时也使用相同的选项：

- 如果使用 `-library`、`-fast`、`-xtarget` 和 `-xarch` 选项，必须确保包括相应的链接程序选项，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。使用 `–dryrun` 查看这些选项的扩展以确定链接步骤中所需的选项。
- 如果使用 `-p`、`-xpg` 和 `-xprofile`，则在一个阶段中包括选项而在其他阶段中不包括相应选项并不影响程序的正确性，但不能进行文件配置。
- 如果使用 `-g` 和 `-g0`，则在一个阶段中包括选项而在其他阶段中不包括相应选项并不影响程序的正确性，但会影响调试程序的能力。对于没有使用其中任一选项编译但使用 `-g` 或 `-g0` 链接的模块，将无法正确调试。请注意，要对包含函数 `main` 的模块进行调试，通常必须使用 `-g` 选项或 `-g0` 选项对其进行编译。

在以下示例中，使用 `-library=stlport4` 编译器选项编译程序。

```
 example% CC -library=stlport4 sbr.cc -c
example% CC -library=stlport4 main.cc -c
example% CC -library=stlport4 sbr.o main.o -o myprogram
```

如果没有一致地使用 -library=stlport4，程序的某些部分将使用缺省的 libCstd，其他部分将使用可选的替换 STlport 库。生成的程序可能不进行链接，因此在任何情况下都不能正常运行。

如果程序使用模板，某些模板可以在链接时实例化。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 针对 64 位内存模型进行编译

使用新的 -m64 选项可指定目标编译的内存模型。生成的可执行文件仅能在运行 64 位内核的 Solaris OS 或 Linux OS 下的 64 位 UltraSPARC 或 x86 处理器上运行。64 位对象的编译链接和执行只能在支持 64 位执行的 Solaris 或 Linux OS 上进行。

2.4.5 诊断编译器

使用 -V 选项可显示 CC 调用的每个程序的名称和版本号。使用 -v 选项可显示 CC 调用的完整命令行。

使用 –verbose=%all 可显示有关编译器的其他信息。

命令行上编译器无法识别的任何参数都解释为链接程序选项、目标程序文件名或库名称。

基本区别是：

- 对于无法识别的**选项**（前面有短划线 (–) 或加号 (+)），会生成警告。
- 对于无法识别的**非选项**（即前面没有短划线或加号），不会生成警告。（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。）

在以下示例中，请注意，CC 无法识别 -bit，该选项传递给链接程序 (ld)，它会尝试解释该选项。因为一个字母的 ld 选项可以连在一起，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 is not a recognized CC option

CC: Warning: Option -bit passed to ld, if ld is invoked, ignored otherwise
```

在下一个示例中，用户本想键入 CC 选项 -fast，但遗漏了前导短划线。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                    < - The user meant to type -fast
move.CC:
ld: fatal: file fast: cannot open file; errno=2
ld: fatal: File processing errors. 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。CC 命令会自动调用其中每个组件，除非使用命令行选项进行其他指定。

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。可使用 -v 和 -dryrun 选项帮助解决此问题。

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。有关文件后缀的含义，请参阅表 2–1。

表 2–2 C++ 编译系统的组件

组件	说明	使用说明
ccfe	前端（编译器预处理程序和编译器）	
iropt	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 .il 文件
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	仅与 -instances=extern 选项一起使用
ld	链接编辑器	

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的信息。

2.5.1 Pragma

预处理程序指令 pragma 是 C++ 标准的一部分，但每个编译器中，pragma 的形式、内容和含义都不相同。有关 C++ 编译器可识别的 pragma 的详细信息，请参见附录 B，Pragma。

Sun C++ 还支持 C99 关键字 _Pragma。这两种调用

```
#pragma dumpmacros(defs)
_Pragma("dumpmacros(defs)")
```

是等效的。要使用 _Pragma 而不是 #pragma，请将 pragma 文本写成文字字符串（用括号括起来作为 _Pragma 关键字的一个参数）。

2.5.2 具有可变数目的参数的宏

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。其他参数（包括逗号）收集到一个字符串中，宏替换列表中的名称 __VA_ARGS__ 可以引用该字符串。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–2 介绍了预定义的宏。可以在 #ifdef 这样的预处理程序条件中使用这些值。+p 选项可防止自动定义 sun、unix、sparc 和 i386 预定义宏。

2.5.4 #error

发出警告后，#error 指令不再继续编译。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。然后优化器将以在命令行上通过 `-xO`*level* 选项指定的原始级别恢复后续例程。

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。可以尝试降低优化级别。或者，将最大的过程分为其自身的单独文件。

2.6.1 交换空间大小

`swap -s` 命令用于显示可用的交换空间。有关更多信息，请参见 `swap`(1M) 手册页。

以下示例演示了 `swap` 命令的用法：

```
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) 可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` 用于命令创建特定大小的文件，而 `swap -a` 用于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` 或更高级别编译非常大型的例程（一个过程中有数几千行代码）需要大量内存。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。

要在 `sh` shell 中限制虚拟内存，请使用 `ulimit` 命令。有关更多信息，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 4GB：

```
example$ ulimit -d 4000000
```

在 csh shell 中，可使用 limit 命令限制虚拟内存。有关更多信息，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 4GB：

```
example% limit datasize 4G
```

这些示例都会使优化器在数据空间达到 4GB 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。

有 8GB 的交换空间时，请使用以下命令：

在 sh shell 中：

```
example$ ulimit -d 4000000
```

在 csh shell 中：

```
example% limit datasize 4G
```

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

工作站至少应有 1 GB 内存，推荐使用 2 GB。有关详细的要求，请参见产品发行版自述文件 (http://developers.sun.com/sunstudio/documentation/)。

2.7 将 strip 命令用于 C++ 目标

不应将 Unix strip 命令用于 C++ 目标文件，因为这会导致这些目标文件不可用。

2.8 简化命令

可以通过定义特殊的 shell 别名、使用 CCFLAGS 环境变量或使用 make 来简化复杂的编译器命令。

2.8.1 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"
```

以下示例使用了别名 CCfx。

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

现在命令 CCfx 等效于：

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

2.8.2 使用 CCFLAGS 指定编译选项

可以通过设置 CCFLAGS 变量来指定选项。

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

```
example% setenv CCFLAGS '-xO2 -m64'
```

以下示例显式使用了 CCFLAGS。

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

使用 make 时，如果像上述示例那样设置 CCFLAGS 变量，且 makefile 的编译规则是隐式的，那么调用 make 的结果就相当于下面的编译：

CC -xO2 -m64 *files...*

2.8.3 使用 make

make 实用程序是功能非常强大的程序开发工具，可以方便地与所有 Sun 编译器一起使用。有关更多信息，请参见 make(1S) 手册页。

2.8.3.1 在 make 中使用 CCFLAGS

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

◆ ◆ ◆ 第 3 章

使用 C++ 编译器选项

本章解释了如何使用命令行 C++ 编译器选项，并按功能汇总它们的使用。第 198 页中的 “A.2 选项参考”中对这些选项进行了详细说明。

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
-选项	–E
–选项值	–I*pathname*
–选项=值	–xunroll=4
–选项 值	–o *filename*

圆括号、大括号、方括号、"|" 或 "-" 字符以及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。有关用法语法的详细说明，请参见本手册前言中的印刷约定。

3.2 通用指南

C++ 编译器选项的某些通用指南：

- – l*lib* 选项用于与库 lib*lib*.a（或 lib*lib*.so）进行链接。较稳妥的方式是总是将 – l*lib* 放在源文件和目标文件后面，这样可以确保库搜索顺序。
- 通常，编译器选项的处理顺序是从左到右（但 – U 选项在所有 – D 选项之后处理这种情况除外），从而可以有选择地覆盖宏选项（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` 选项会累积，但不会覆盖。
- `-D` 选项会累积，但相同名称的多个 `-D` 选项会互相覆盖。

源文件、目标文件和库是按其在命令行上的出现顺序编译和链接。

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。有关各个选项的详细说明，请参阅附录 A，C++ 编译器选项。

这些选项适用于除了特别注明之外的所有平台；基于 SPARC 的系统上的 Solaris OS 特有的功能标识为 *SPARC*，基于 x86 的系统上的 Solaris OS 特有的功能标识为 *x86*。

3.3.1 代码生成选项

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。
`-xaddr32`	将代码限定于 32 位地址空间 (x86/x64)
`-xarch`	指定目标体系结构。
`–xcode=`*a*	(SPARC) 指定代码地址空间。
`–xMerge`	(SPARC) 将数据段和文本段合并。
`-xtarget`	指定目标系统。
`–xmodel`	针对 Solaris x86 平台修改 64 位对象形式
`+w`	标识可能产生不可预料结果的代码。
`+w2`	发出由 `+w` 发出的所有警告以及关于技术违规的警告，这些技术违规可能是无害的，但可能会降低程序的最大可移植性。

表 3–2 代码生成选项 （续）

选项	操作
`-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

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。
`-m32|-m64`	指定编译的二进制对象的内存模型。
`-xinstrument`	编译程序并为其提供程序设备以便 Thread Analyzer 对其进行分析。
`–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一组共同的 include 文件。
`–xpchstop`	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 include 文件。
`–xprofile_ircache`	(SPARC) 重新使用在执行 `-xprofile=collect` 期间保存的编译数据。
`–xprofile_pathmap`	(SPARC) 支持单个配置文件目录中有多个程序或共享库。

3.3.3 编译时选项和链接时选项

下表列出了在链接时和编译时都必须指定的选项。

表 3–4 编译时选项和链接时选项

选项	操作
`-fast`	选择编译选项的最佳组合，以加快可执行代码的编译速度。
`-m32|-m64`	指定编译的二进制对象的内存模型。

表 3–4 编译时选项和链接时选项 （续）

选项	操作
-mt	扩展为 -D_REENTRANT -lthread 的宏选项。
-xarch	指定指令集体系结构。
-xautopar	针对多处理器启用自动并行化。
-xhwcprof	(SPARC) 允许编译器支持基于硬件计数器的分析。
-xipo	通过调用过程间分析组件对整个程序执行优化。
-xlinkopt	对可重定位目标文件执行链接时优化。
-xmemalign	(SPARC) 指定最大假定内存对齐以及未对齐数据访问的行为。
-xopenmp	支持 OpenMP 接口来实现显式并行化，包括一组源代码指令、运行时库例程和环境变量
-xpagesize	设置栈和堆的首选页面大小。
-xpagesize_heap	设置堆的首选页面大小。
-xpagesize_stack	设置栈的首选页面大小。
-xpg	准备目标代码，以便收集数据使用 gprof(1) 进行文件配置。
-xprofile	为配置文件收集数据或使用配置文件进行优化。
-xvector	启用自动生成对向量库函数的调用。

3.3.4 调试选项

表 3–5 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	仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。
–g	用于与调试一起使用的编译。
–g0	编译以便进行调试，但不禁用内联。
–H	打印包含文件的路径名称。

表 3–5 调试选项 （续）

选项	操作
–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	直接将选项传递到编译阶段。
–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*	控制编译器详细内容。
-xcheck	对栈溢出增加一个运行时检查。
-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。
–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。

3.3.5 浮点选项

表 3–6 浮点选项

选项	操作
-fma	(SPARC) 启用自动生成浮点乘加指令。
–fns[={no|yes}]	(SPARC) 禁用或启用 SPARC 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。
–ftrap=*tlst*	设置启动时生效的 IEEE 陷阱操作模式。
–nofstore	*x86*： 禁用表达式的强制精度。

表 3–6 浮点选项 （续）

选项	操作
–xlibmieee	使 libm 在异常情况下对于数学例程返回 IEEE 754 值。

3.3.6 语言选项

表 3–7 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缺省链接程序范围，以创建更快更安全的共享库。
–xthreadvar	(SPARC) 更改缺省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 位字符构成的文本字符串。

3.3.7 库选项

表 3–8 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。
–h*name*	为生成的动态共享库指定内部名称。
–i	通知 ld(1) 忽略任何 LD_LIBRARY_PATH 设置。
–L*dir*	将 *dir* 添加到要在其中搜索库的目录列表。
–l*lib*	将 lib*lib*.a 或 lib*lib*.so 添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。

表 3–8 库选项 （续）

选项	操作
–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	(Solaris) 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	使 libm 在异常情况下对于数学例程返回 IEEE 754 值。
–xlibmil	内联选定的 libm 库例程以进行优化。
–xlibmopt	使用优化数学例程的库。
–xnolib	禁止链接缺省系统库。
–xnolibmil	在命令行上取消 – xlibmil。
–xnolibmopt	不使用数学例程库。

3.3.8 废弃的选项

注 – 以下选项要么当前已废弃所以编译器不再接受它们，要么将从以后的发行版中删除。

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。
-xlic_lib=sunperf	使用 –library=sunperf 可链接到 Sun 性能库。
-xlicinfo	已过时。
–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。
-sb、–sbfast、–xsb、–xsbfast	已过时，默认忽略。

表 3–9 废弃的选项 （续）

选项	操作
-vdelx	废弃的选项，在以后的发行版本中将被删除。
-x386	使用适当的 –xtarget 选项。
-x486	使用适当的 –xtarget 选项。
-xcg89	使用 -xtarget=ss2。
-xcrossfile	改用 -xipo。
-xnativeconnect	已废弃，没有替代选项。
-xprefetch=yes	改用 -xprefetch=auto,explicit。
-xprefetch=no	改用 -xprefetch=no%auto,no%explicit。
-xvector=yes	改用 -xvector=lib。
-xvector=no	改用 -xvecttor=none。

3.3.9 输出选项

表 3–10 输出选项

选项	操作
–c	仅编译；生成目标 (.o) 文件，但抑制链接。
–dryrun	显示但不编译由驱动程序传递到编译器的所有命令行。
–E	仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。
-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	如果发出指示的警告消息，cc 将以失败状态退出。
-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。

表 3–10 输出选项 （续）

选项	操作
–Qproduce *sourcetype*	使 CC 驱动程序生成类型为 *sourcetype* 的输出。
–s	从可执行文件去掉符号表。
–verbose=*vlst*	控制编译器详细内容。
+w	必要时打印附加警告。
+w2	适当时仍打印更多警告。
–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	对源文件仅执行语法和语义检查，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 依赖性信息。
–xM1	生成依赖性信息，但排除 /usr/include。
–xtime	报告每个编译阶段的执行时间。
–xwe	将所有的警告转换为错误。
-z *arg*	链接程序选项。

3.3.10 运行时性能选项

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。
-fma	(SPARC) 启用自动生成浮点乘加指令。
-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。
-m32|-m64	指定编译的二进制对象的内存模型。

表 3–11 运行时性能选项 （续）

选项	操作
-xalias_level	启用编译器执行基于类型的别名分析和优化。
–xarch=*isa*	指定目标体系结构指令集。
-xbinopt	准备二进制文件以便以后进行优化、转换和分析。
-xbuiltin[=*opt*]	启用或禁用标准库调用的更多优化。
–xcache=*c*	(SPARC) 定义优化器的目标高速缓存属性。
–xcg89	为通用 SPARC v7 体系结构编译。
–xcg92	为 SPARC V8 体系结构编译。
–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	执行过程间的优化。
–xlibmil	内联选定的 libm 库例程以进行优化。
–xlibmopt	使用优化数学例程的库。
–xlinkopt	(SPARC) 在对目标文件进行优化的基础上对生成的可执行文件或动态库执行链接时优化。
–xmemalign=*ab*	(SPARC) 指定假定的最大内存对齐以及未对齐的数据访问的行为。
–xnolibmil	在命令行上取消 – xlibmil。
–xnolibmopt	不使用数学例程库。
–xO*level*	将优化级别指定为 *level*。
–xpagesize	设置栈和堆的首选页面大小。
-xpagesize_heap	设置堆的首选页面大小。
-xpagesize_stack	设置栈的首选页面大小。
-xprefetch[=*lst*]	在支持预取的体系结构上启用预取指令。
-xprefetch_level	控制 -xprefetch=auto 设置的自动插入预取指令的主动性。
–xprofile	收集运行时文件配置数据或使用运行时文件配置数据进行优化。
–xregs=*rlst*	控制临时寄存器的使用。
–xsafe=mem	(SPARC) 不允许有基于内存的陷阱。

表 3–11 运行时性能选项 （续）

选项	操作
–xspace	(SPARC) 不允许会增大代码大小的优化。
–xtarget=*t*	指定目标指令集和优化系统。
–xthreadvar	更改缺省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) 使编译器可以识别 VIS™ 指令集中定义的汇编语言模板。

3.3.11 预处理程序选项

表 3–12 预处理程序选项

选项	操作
–D*name*[= *def*]	为预处理程序定义符号 *name*。
–E	仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout。不编译。
-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* 的初始定义。
–xM	输出 makefile 依赖性信息。
–xM1	生成依赖性信息，但排除 /usr/include。

3.3.12 文件配置选项

表 3–13 文件配置选项

选项	操作
–p	准备目标代码来收集数据以使用 prof 进行文件配置。
–xa	为文件配置生成代码。
–xpg	编译以便使用 gprof 配置程序进行文件配置。
–xprofile	收集运行时文件配置数据或使用运行时文件配置数据进行优化。

3.3.13 参考选项

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

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* 添加到 `include` 文件搜索路径。
`-I-`	更改包含文件搜索规则
`–xM`	输出 makefile 依赖性信息。
`–xM1`	生成依赖性信息，但排除 `/usr/include`。

3.3.15 模板选项

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。
`–template=`*wlst*	启用或禁用各种模板选项。

3.3.16 线程选项

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) 不允许有基于内存的陷阱。
–xthreadvar	(SPARC) 更改缺省的线程局部存储访问模式。

第 2 部分

编写 C++ 程序

◆ ◆ ◆ 第 4 章

4 语言扩展

本章介绍了与此编译器相关的语言扩展。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功能。相关编译器选项在相应章节中列出。

使用 `-features=extensions` 选项可以编译其他 C++ 编译器通常接受的非标准代码。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。

本章介绍了使用 `-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

可使用下列声明说明符来协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，静态归档或目标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。也可以通过在命令行上指定 `-xldscope` 来控制变量作用域的缺省设置。

有关更多信息，请参见第 288 页中的 “A.2.136 `-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，是限制最小的链接程序作用域。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，其限制程度高于全局链接程序作用域。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`。尽管不能将 `-Bsymbolic` 与 C++ 库一起使用，但可以使用 `__symbolic` 说明符，而不会引起问题。有关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` 是限制较多的作用域，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。

可以将链接程序作用域说明符应用于结构、类和联合声明和定义中，因为 C++ 类可能要求生成隐式信息，如虚拟表和运行时类型信息。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。

4.1.1 与 Microsoft Windows 兼容

为了在动态库方面与 Microsoft Visual C++ (MSVC++) 中的相似作用域功能兼容，也支持以下语法：

`__declspec(dllexport)` 等效于 `__symbolic`

`__declspec(dllimport)` 等效于 `__global`

在 Sun C++ 中使用此语法时，应将选项 `-xldscope=hidden` 添加到 `CC` 命令行。结果与使用 MSVC++ 得到的结果相当。在 MSVC++ 中，`__declspec(dllimport)` 应当仅用于外部符号的声明，而不用于定义。示例：

```
__declspec(dllimport) int foo(); // OK
__declspec(dllimport) int bar() { ... } // not OK
```

MSVC++ 中，对于将 `dllimport` 用于定义没有严格规定，而使用 Sun C++ 时结果则不同。尤其是，使用 Sun C++ 时将 `dllimport` 用于定义得到的是具有全局链接的符号而不是符号链接。Microsoft Windows 上的动态库不支持符号的全局链接。如果遇到此问题，可以更改源代码，对定义使用 dllexport 而不是 dllimport。这样，使用 MSVC++ 和使用 Sun C++ 得到的结果相同。

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。线程局部变量声明普通变量声明与声明说明符 `__thread` 组成。有关更多信息，请参见第 329 页中的 “A.2.182 -xthreadvar[=*o*]”。

必须将 `__thread` 说明符包括在第一个线程变量声明中。使用 `__thread` 说明符声明的变量的绑定方式与没有 `__thread` 说明符时相同。

只能使用 `__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。不能使用 `__thread` 说明符声明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，允许 `__thread int x = 4;`，但不允许 `__thread int x = f();`。线程变量不能包含具有重要构造函数和析构函数的类型。具体来说，就是线程变量的类型不能为 `std::string`。

运行时对线程变量的地址运算符 (&) 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。进程中任何线程都可以在线程变量的生命周期任意使用该变量的地址。不能在线程终止后使用线程变量的地址。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++ 标准规定，覆盖虚拟函数在异常中允许的限制不得低于它覆盖的任何函数的限制。该虚函数可能与覆盖的任何函数具有相同或更多的限制。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抛出的异常。如果覆盖函数具有限制较少的规范，则不可预料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，编译器允许覆盖异常规范限制较小的函数。

4.4 对 enum 类型和变量进行前向声明

使用 -features=extensions 时，编译器允许对 enum 类型和变量进行前向声明。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 类型的大小和范围与当前平台上的 int 类型相同。

以下是两行无效代码示例，如果使用 -features=extensions 选项，可对其进行编译。

```
enum E; // invalid: forward declaration of enum not allowed
E e;    // invalid: type E is incomplete
```

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，所以从来不必对枚举类型进行前向声明。要使代码有效，可以总是先提供 enum 的完整定义，然后再使用它。

注 – 在 64 位体系结构上，enum 要求的大小可能比 int 类型大。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 类型以前向声明处理。例如，以下是无效代码，如果使用 -features=extensions 选项，可对其进行编译。

```
typedef enum E F; // invalid, E is incomplete
```

如前所述，可以总是先包括 enum 类型的定义，然后再使用。

4.6 将 enum 名称作为作用域限定符

因为 enum 声明并不引入作用域，所以 enum 名称不能作为作用域限定符来使用。例如，以下代码是无效的。

```
enum E {e1, e2, e3};
int i = E::e1; // invalid: E is not a scope name
```

要编译该无效代码，请使用 -features=extensions 选项。-features=extensions 选项指示编译器在作用域限定符是 enum 类型的名称的情况下忽略该作用域限定符。

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

匿名结构声明是既不声明结构标记也不声明对象或 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 声明示例，如果使用 -features=extensions 选项，可对其进行编译。

```
union U {
  struct {
    int a;
    double b;
  };  // invalid: anonymous struct
  struct {
    char* c;
    unsigned d;
  };  // invalid: anonymous struct
};
```

struct 成员的名称是可视的，没有 struct 成员名称的限定。如果该代码示例中提供了 U 的定义，则可以编写：

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

匿名结构与匿名联合服从相同的限制。

请注意，可以通过为每个 struct 提供一个名称以使代码有效，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。但是，如果使用 -features=extensions 选项，编译器将接受该无效代码。

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

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。程序员应确保临时变量的地址没有留下。此外，销毁临时变量（例如 f1）时，临时变量中存储的数据会丢失。

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。但是，如果使用 -features=extensions 选项，编译器将接受该代码。

程序员处理该无效代码的方法大概是在类 A 的实现文件中提供非成员 "helper" 函数。可以通过使 foo 成为静态成员函数得到相同效果。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，则任何客户端都可以“劫取”您的类。任何客户端都可以包括类的头文件，然后定义其自身的静态函数 foo，该函数将自动成为类的友元。结果就好像是您使类的所有成员成为了公共的。

4.10 将预定义 __func__ 符号用于函数名

使用 -features=extensions 时，编译器将每个函数中的标识符 __func__ 隐式声明为静态 const char 数组。如果程序使用标识符，编译器还会提供以下定义，其中，*function-name* 是函数原始名称。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
```

4.11 __packed__ 属性

此属性附加于 struct 或 union 类型定义中，它指定结构或联合的每个成员（除了零宽度位字段）的放置，以最大限度地减小所需内存。附加于 enum 定义时，此属性指示应使用最小的整数类型。

为 struct 和 union 类型指定此属性等效于对每个结构或联合成员指定 packed 属性。

在以下示例中，struct my_packed_struct 的成员紧紧打包在一起，但其成员的内部布局并不打包。为此，还需要打包 struct my_unpacked_struct。

```
struct my_unpacked_struct
{
   char c;
   int i;
;

struct __attribute__ ((__packed__)) my_packed_struct
{
```

```
    char c;
    int  i;
    struct my_unpacked_struct s;
};
```

只能对 enum、struct 或 union 的定义指定此属性，不能对未定义枚举类型、结构或联合的 typedef 指定此属性。

◆ ◆ ◆ 第 5 章

5 程序组织

C++ 程序的文件组织需要比典型的 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。要提供模板，请确保头文件能容纳多个包含（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。但是，称为“传统 C”的 Kernighan 和 Ritchie C (K&C)、ANSI C、 Annotated Reference Manual C++ (ARM C++) 以及 ISO C++ 有时要求一个头文件中同一个程序元素有不同的声明或定义。（有关语言和版本之间的变化的其他信息，请参见《C++ 迁移指南》。）要使头文件符合所有这些标准，可能需要根据预处理程序宏 `__STDC__` 和 `__cplusplus` 的存在情况或值来使用条件编译。

在 K&R C 中没有定义宏 `__STDC__`，但在 ANSI C 和 C++ 中都对其进行了定义。可使用该宏将 K&R C 代码与 ANSI C 或 C++ 代码区分开。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 declaration
#else
int function();               // K&R C
#endif
```

在 C 中没有定义宏 `__cplusplus`，但在 C++ 中对其进行了定义。

注 – 早期版本的 C++ 定义了宏 `c_plusplus`，但没有定义 `cplusplus`。现在已不再定义宏 `c_plusplus`。

可使用 `__cplusplus` 宏的定义来区分 C 和 C++。该宏在保证为函数声明指定 `extern "C"` 接口时非常有用，如以下示例所示。为了防止出现 `extern "C"` 指定不一致，切勿将 `#include` 指令放在 `extern "C"` 链接指定的作用域中。

```
#include “header.h”
...                      // ... other include files...
#if defined(__cplusplus)
extern “C” {
#endif
  int g1();
  int g2();
  int g3()
#if defined(__cplusplus)
}
#endif
```

在 ARM C++ 中，`__cplusplus` 宏的值为 1。在 ISO C++ 中，该宏的值为 199711L（用 `long` 常量表示的标准年月）。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。

```
// template function specialization
#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
/* contents of header file */
#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

在将模板的声明和定义放在使用该模板的文件中时，组织是**包括定义的**组织。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模板定义的文件时，使用模板的该文件的组织也是包括定义的组织。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 70 页中的 “5.1.2 幂等头文件”。）

5.2.2 独立的模板定义

另一种组织模板定义的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**包括任何非幂等头文件，而且通常根本不需要包括任何头文件。（请参见第 70 页中的 “5.1.2 幂等头文件”。）请注意，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 – 尽管通常会使用模板定义文件的源文件扩展名（即 .c、.C、.cc、.cpp、.cxx 或 .c++），但模板定义文件是头文件。如果需要，编译器会自动包括它们。模板定义文件**不**应单独编译。

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。有关模板定义搜索规则的信息，请参阅第 92 页中的 “7.5 模板定义搜索”。

◆ ◆ ◆ 第 6 章

6 创建和使用模板

有了模板，就可以采用类型安全方法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、缺省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。以下示例中的**声明**提供了使用模板所需的足够信息，但没有提供实现模板所需的足够信息。

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

在此示例中，*Number* 是**模板参数**，它指定模板描述的函数范围。更具体地说，*Number* 是**模板类型参数**，在模板定义中使用它表示确定的模板使用位置处的类型。

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**提供了实现模板所需的足够信息。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。该限制称为**一次定义规则**。

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。它们的**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(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

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。此类声明是**不完整的类模板**。

以下示例是名为 `Array` 类的模板声明，该类可接受任何类型作为参数。

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

该模板用于名为 `String` 的类，该类接受 `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，类模板可以同时有类型参数（如 `class Elem`）和表达式参数（如 `unsigned Size`）。表达式参数可以是：

- 具有整型或枚举的值
- 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，该模板中指定了模板参数 `template <class Elem>`。每个函数标识符都通过模板类名称和模板参数 `Array<Elem>` 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` 类模板的函数成员的定义。

```
#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元参数来限定。

```
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。以下示例中的声明根据 `Array` 模板创建 `int_array` 变量。变量的类声明及其一组方法类似于 `Array` 模板中的声明和方法，除了 `Elem` 替换为 `int`（请参第 76 页中的 “6.3 模板实例化”）。

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

此示例中的声明使用 `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**是生成采用特定模板参数组合的具体类或函数（**实例**）。例如，编译器生成一个采用 `Array<int>` 的类，另外生成一个采用 `Array<double>` 的类。通过用模板参数替换模板类定义中的模板参数，可以定义这些新的类。在前面“类模板”一节介绍的 `Array<int>` 示例中，编译器用 `int` 替换所有 `Elem`。

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

要显式实例化模板函数，请在 template 关键字后接函数的声明（不是定义），且函数标识符后接模板参数。

```
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 关键字后接类的声明（不是定义），且在类标识符后接模板参数。

```
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 关键字后接函数的声明（不是定义），且在由模板类限定的函数标识符后接模板参数。

```
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 关键字后接成员的声明（不是定义），且在由模板类限定的成员标识符后接模板参数。

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

6.4 模板组合

可以嵌套使用模板。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;}}
```

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

```
Array<int> int_array(100);   // construct an array of ints
sort(int_array);             // sort it
```

6.5 缺省模板参数

您可以将缺省值赋予类模板（但不是函数模板）的模板参数。

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

如果模板参数具有缺省值，则该参数后的所有参数也必须具有缺省值。模板参数仅能具有一个缺省值。

6.6 模板专门化

将某些模板参数组合视为特殊的参数可以提高性能，如以下 twice 示例所示。而模板说明可能无法用于其一组可能的参数，如以下 sort 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。模板专门化覆盖了缺省实例化。

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，以下代码样本说明了主模板和该模板的完全专门化。

```
template<class T, class U> class A {...}; //primary template
template<> class A<int, double> {...};    //specialization
```

以下代码说明了主模板部分专门化的示例。

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

- 示例 1 提供了用于第一个模板参数是 `int` 类型的情况的特殊模板定义。
- 示例 2 提供了用于第一个模板参数是任何指针类型的情况的特殊模板定义。
- 示例 3 提供了用于第一个模板参数是任何类型的指针到指针而第二个模板参数是 `char` 类型的情况的特殊模板定义。

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`。在 `use1.cc` 和 `use2.cc` 中这些名称的定义不同，因此在不同的编译器下可能会生成不同的结果。为了能可靠地使用模板，所有非本地名称（该示例中为 `intermediary` 和 `temporary`）在任何地方都必须有相同的定义。

```
use_common.h
// Common template definition
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"
```

一个常见的非本地名称用法是在模板内使用 `cin` 和 `cout` 流。有时程序员要将流作为模板参数传递，这时就要引用到全局变量。但 `cin` 和 `cout` 在任何地方都必须有相同的定义。

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，等效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：

示例 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` 中记录的 `Foo` 类型与在 `file2.cc` 中记录的 `Foo` 类型不同。以这种方法使用本地类型会出现错误和意外的结果。

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目标文件时，对**未**实例化的 `operator<<` 函数，会生成未定义错误。

示例 6–2 友元声明问题的示例

```
array.h
// generates undefined error for the operator<< function
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

template<class T> class array {
    int size;
```

示例 6–2 友元声明问题的示例 （续）

```
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;
}
```

请注意，因为编译器将以下代码作为普通函数（array 类的 friend）的声明进行读取，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，所以需要在声明 template class array 之前提供模板声明。但是，由于 operator<< 有一个 array<T> 类型的参数，因此必须在声明函数之前声明 array<T>。文件 array.h 必须如下所示：

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

```
// the next two lines declare operator<< as a template function
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++ 标准要求使用具有限定名的类型，这些限定名取决于要用 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”它应该是一个类型，也是如此。以下示例中的注释说明了具有要用 typename 关键字的限定名的类型。

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;                 // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

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小心。确保相邻的 ">" 字符之间至少有一个空格。

例如，以下是形式错误的语句：

```
Array<String<10>> short_string_array(100); // >> = right-shift
```

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。如果希望每个模板函数实例化的对象不同，请使对象成为函数的本地对象。

6.7.7 在同一目录中使用模板生成多个程序

如果要通过指定 `-instances=extern` 生成多个程序或库，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。有关更多信息，请参见第 92 页中的 “7.4.4 共享模板系统信息库”。

考虑以下示例的 `make` 文件 `a.cc`、`b.cc`、`x.h` 和 `x.cc`。请注意，仅当指定了 `-instances=extern` 时，此示例才有意义：

```
........
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，请在两个生成之间添加 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

7 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目标代码。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

如果使用标志 `-verbose=template`，C++ 编译器会在编译模板期间通知重要事件。但如果使用缺省值 `-verbose=no%template`，编译器不会发出通知。`+w` 选项可以在进行模板实例化时提供其他有关潜在问题的指示信息。

7.2 系统信息库管理

`CCadmin`(1) 命令管理模板系统信息库（只能与选项 `-instances=extern` 一起使用）。例如，程序中的更改会造成某些实例化过度，这样会浪费存储空间。`CCadmin – clean` 命令（以前是 `ptclean`）清除所有实例及关联数据。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

编译器通常是分别实例化各个模板类成员，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，该选项指示编译器实例化模板类的所有非模板非内联成员（如果实例化这些相同成员中的任何一个）。

ISO C++ 标准允许开发人员编写模板类，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。但是，`-template=wholeclass` 选项会实例化所有成员，因此不能用于此类使用有问题的模板参数实例化的模板类。

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：

- 调试更加简单－错误消息在上下文中出现，从而让编译器完全回溯到引用点。
- 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

缺省情况下，实例会进入特殊地址区域，链接程序会识别并丢弃重复项。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。

- 在下列情况下，外部实例可以达到最佳的执行效果：
  - 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，已过时。请参见以下内容。
- 缺省的全局实例适用于所有开发，并且在对象引用各种实例时可以达到最佳的执行效果。
- 显式实例适用于某些需精确控制的应用程序编译环境。
- 半显式实例对编译环境的控制要求较少，但是生成的目标文件较大，并且使用有限制。

本节讨论了五种实例放置和链接方法。第 76 页中的 “6.3 模板实例化”中提供了有关生成实例的其他信息。

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，所有目标文件（包括模板缓存中的任何目标文件）在使用 `-instances=extern` 时的大小总量小于在使用 `-instances=global` 时的大小总量。

模板实例接受系统信息库中的全局链接。实例是使用外部链接从当前编译单元引用的。

注 – 如果在不同的步骤中进行编译和链接，并且在编译步骤中指定了 `-instance=extern`，则还必须在链接步骤中指定该选项。

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。高速缓存是并行编译的瓶颈，这与使用 `dmake` 时一样，因为每次只能有一个编译访问高速缓存。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目标文件中创建实例（如果需要，用完后可以丢弃）要花费更长的时间。

可使用 `-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。

如果要创建包含了使用的所有模板实例的库，请结合使用 `CC` 命令与 `–xar` 选项。而**不**要使用 `ar` 命令。例如：

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

有关更多信息，请参见表 15–3。

7.3.1 可能的缓存冲突

如果指定了 `-instance=extern`，请勿在同一目录中运行不同的编译器版本，以避免可能的高速缓存冲突。使用 `-instances=extern` 模板模型时，请注意：

- 请勿在同一目录中创建不相关的二进制文件。在同一目录中创建的所有二进制文件（.o、.a、.so、可执行程序）都应该相关，因为两个或两个以上目标文件中名称相同的所有对象、函数和类型的定义都相同。
- 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 中的依赖性不允许任何内容与链接步骤以并行方式运行。

7.3.2 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` 现在提供了**静态**的所有优点而没有其缺点。早期的编译器中提供了此选项来克服现已不存在的问题。

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，模板可以在多个目标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。

可使用 `-instances=static` 编译器选项指定静态实例链接。

7.3.3 全局实例

与早期的编译器发行版不同，现在不必预防出现一个全局实例有多个副本。

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目标文件会很大，原因是多个文件中模板实例有多个副本。如果编译目标文件以便进行调试时，有些使用了 `-g` 选项，而有些没有使用该选项，则很难预测是获得链接到程序中模板实例的调试版本还是非调试版本。

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。

可使用 `-instances=global` 选项（这是缺省值）指定全局实例。

7.3.4 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。

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。

可使用 `-instances=explicit` 选项指定显式实例。

7.3.5 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。

可使用 `-instances=semiexplicit` 选项指定半显式实例。

7.4 模板系统信息库

模板系统信息库中存储需单独进行编译的模板实例，以便仅在需要时编译模板实例。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

缺省情况下，模板系统信息库位于名为 `SunWS_cache` 的高速缓存目录中。

缓存目录包含在放置目标文件的目录中。可以通过设置环境变量 `SUNWS_CACHE_NAME` 更改高速缓存目录的名称。请注意，`SUNWS_CACHE_NAME` 变量值必须是目录名称，而不能是路径名。这是因为编译器自动将模板缓存目录放置到了目标文件目录下，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，以下命令行会将目标文件写入 `./sub/a.o` 并将模板实例写入包含在 `./sub/SunWS_cache` 中的系统信息库。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目标文件的模板系统信息库读取。即，以下命令行从 `./sub1/SunWS_cache` 和 `./sub2/SunWS_cache` 读取，必要时，向 `./SunWS_cache` 写入。

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

7.4.4 共享模板系统信息库

系统信息库中的模板不得违反 ISO C++ 标准的一次定义规则。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 通过 `-instances=extern` 实现模板实例自动一致

如果指定了 `-instances=extern`，模板系统信息库管理器可确保系统信息库中实例的状态与源文件一致且是最新的。

例如，如果使用 `–g` 选项编译源文件（调试），也会使用 `–g` 编译来自数据库中的所需文件。

此外，模板系统信息库会跟踪编译中的更改。例如，如果设置了 `–DDEBUG` 标志来定义名称 `DEBUG`，数据库中会记录下该信息。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赖性的这些模板。

注 – 如果删除模板的源代码或停止使用模板，模板的实例会保留在缓存中。如果更改函数模板的签名，使用旧签名的实例会保留在缓存中。如果因为这些问题在编译时或链接时遇到了异常行为，请清除模板缓存并重新生成程序。

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，编译器必须搜索该定义。本节描述了编译器如何找到定义。

定义搜索有些复杂，并且很容易出现错误。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 71 页中的 “5.2.1 包括的模板定义”。

注 – 如果使用 -template=no%extdef 选项，编译器将不搜索单独的源文件。

7.5.1 源文件位置约定

如果没有随选项文件一起提供的特定方向，则编译器使用 Cfront 样式的方法来定位模板定义文件。此方法要求模板定义文件包含的基名与模板声明文件包含的基名相同。此方法也要求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，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名（.C、.c、.cc、.cpp、.cxx 或 .c++）。模板定义文件必须位于常规的 include 目录之一中，或位于与其匹配的头文件所在目录中。

7.5.2 定义搜索路径

可以用另外一种方法替代用 –I 设置的常规搜索路径，即使用选项 –pti*directory* 指定模板定义文件的搜索目录。多个 -pti 标志定义多个搜索目录－即搜索路径。如果使用 -pti*directory*，则编译器在该路径查找模板定义文件并忽略 –I 标志。由于 –pti*directory* 标志会使源文件的搜索规则变得复杂，因此应使用 –I 选项而不是 –pti*directory* 选项。

7.5.3 诊断有问题的搜索

有时，编译器会生成令人费解的警告或错误消息，因为它会查找您不打算编译的文件。此问题通常是由于某个文件（如 foo.h）包含模板声明，且隐式包含了另一个文件（如 foo.cc）。

如果头文件 foo.h 有模板声明，缺省情况下，编译器会搜索名为 foo 且具有 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。有关这些搜索的更多信息，请参见第 92 页中的 “7.5 模板定义搜索”。

如果有一个不打算这样处理的文件 foo.cc，有两种解决方法：

- 更改 .h 或 .cc 文件的名称，以消除名称匹配。
- 可以通过指定 -template=no%extdef 选项来禁用模板定义文件的自动搜索。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

8 异常处理

本章讨论了 C++ 编译器的异常处理实现。第 112 页中的“11.2 在多线程程序中使用异常”中提供了附加信息。有关异常处理的更多信息，请参见由 Bjarne Stroustrup 编著的《The C++ Programming Language》第三版（Addison-Wesley 出版，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。**同步异常**这一术语意味着异常只能源于 `throw` 表达式。

C++ 标准支持具有终止模型的同步异常处理。**终止**意味着一旦抛出异常，控制永远不会返回到抛出点。

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
- 在堆栈展开时，析构函数必须处理自身的异常
- 内存不足

运行时检测到错误时，错误消息会显示当前异常的类型和这五个错误消息之一。缺省情况下，会调用预定义的函数 `terminate()`，该函数又会调用 `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

如果知道程序中未使用异常，可以使用编译器选项 `-features=no%except` 抑制支持异常处理的代码的生成。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。缺省情况下，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` 和缺省运算符 `new` 要求使用异常，所以在标准模式（缺省模式）下编译时不应禁用异常。

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++ 标准中指定的类和异常相关函数。仅在标准模式（编译器缺省模式，或使用选项 `-compat=5`）下编译时才可访问该头文件。以下摘录的部分代码显示了 `<exception>` 头文件声明。

```
// standard header <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 {...};
    // Unexpected exception handling
       typedef void (*unexpected_handler)();
       unexpected_handler
         set_unexpected(unexpected_handler) throw();
       void unexpected();
    // Termination handling
       typedef void (*terminate_handler)();
       terminate_handler set_terminate(terminate_handler) throw();
       void terminate();
       bool uncaught_exception() throw();
}
```

标准类 exception 是所选语言构造或 C++ 标准库抛出的所有异常的基类。可以构造、复制及销毁类型为 exception 的对象，而不会生成异常。虚拟成员函数 what() 返回描述异常的字符串。

为了与 C++ 4.2 版中所用的异常兼容，还提供了头文件 <exception.h> 以用于标准模式下。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。应在开发进度计划许可的情况下，更新代码以遵循 C++ 标准（使用 <exception> 而非 <exception.h>）。

```
// header <exception.h>, used for transition
#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]) 下，头文件 <exception> 不可用，头文件 <exception.h> 指随 C++ 4.2 版提供的相同头文件。此处不再重述。

8.5 将异常与信号和 Setjmp/Longjmp 混合使用

可以在会出现异常的程序中使用 setjmp/longjmp 函数，只要它们不会互相影响。

使用异常和 setjmp/longjmp 的所有规则分别适用。此外，仅当在点 A 抛出与在点 B 捕获的异常具有相同的效果时，从点 A 到点 B 的 longjmp 才有效。需特别指出的是，不得使用 longjmp 进入或跳出 try 块或 catch 块（直接或间接），或使用 longjmp 跳过自动变量或临时变量的初始化或 non-trivial 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

切勿将 -Bsymbolic 用于包含 C++ 代码的程序，而应使用链接程序映射文件或链接程序作用域选项（请参见第 61 页中的 “4.1 链接程序作用域”）。如果使用 -Bsymbolic，不同模块中的引用会绑定到应是一个全局对象内容的不同副本。

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。

◆ ◆ ◆ 第 9 章

9 强制类型转换操作

本章讨论 C++ 标准中较新的强制类型转换运算符：`const_cast`、`reinterpret_cast`、`static_cast` 和 `dynamic_cast`。强制类型转换可以将对象或值从一种类型强制转换为另一种类型。

这些强制类型转换操作比以前的强制类型转换操作更好控制。`dynamic_cast<>` 运算符提供了一种检查指向多态类的指针的实际类型的方法。可以用文本编辑器搜索所有新式强制类型转换（搜索 `_cast`），而查找旧式强制类型转换需要进行语法分析。

否则，新的强制类型转换全部执行传统强制类型转换符号允许的强制类型转换子集。例如，`const_cast<int*>(v)` 可以写为 `(int*)v`。新的强制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。

强制类型转换运算符是始终启用的。强制类型转换符不能被禁用。

9.1 const_cast

可以使用表达式 `const_cast<`*T*`>(`*v*`)` 更改指针或引用的 `const` 或 `volatile` 限定符。（在新式强制类型转换中，只有 `const_cast<>` 可以删除 `const` 限定符。）`T` 必须是指针、引用或指向成员的指针类型。

```
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();                // remove const
ip = const_cast<int*> (cvip);    // remove const and volatile
}
```

9.2 reinterpret_cast

表达式 reinterpret_cast<*T*>(*v*) 用于更改对表达式 *v* 值的解释。该表达式可用于在指针和整型之间，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。

使用 reinterpret_cast 运算符可能会得到未定义的结果或实现相关结果。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。
- 如果可以使用重新解释强制类型转换将“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“指向 *T1* 类型的 X 的成员的指针”类型右值可以显式转换为“指向 *T2* 类型的 *Y* 的成员的指针”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 运算符不能用来删除 const，可使用 const_cast 来实现。
- reinterpret_cast 运算符不能用来在指向同一类分层结构中不同类的指针之间进行转换，可使用静态或动态强制类型转换来实现。（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);      // safe
     B* bp = reinterpret_cast<B*>(&a1);    // unsafe
     const A a2;
     ap = reinterpret_cast<A*>(&a2);  // error, const removed
}
```

9.3 static_cast

表达式 static_cast<*T*>(*v*) 用于将表达式 *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，任何值的类型都可以强制转换为 void，并且，如果强制类型转换与旧式强制类型转换一样合法，则任何隐式转换都可以反向执行。

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

static_cast 运算符不能用于删除 const。可以使用 static_cast 对分层结构“向下”强制类型转换（从基到派生的指针或引用），但是不会检查转换，因此结果可能无法使用。static_cast 不能用于从虚拟基类向下强制类型转换。

9.4 动态强制类型转换

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。动态强制类型转换可以实现这些功能。

注 – 在兼容模式 (-compat[=4]) 下编译时，如果程序使用动态强制类型转换，则必须使用 -features=rtti 进行编译。

动态强制类型转换可将指向一个类 *T1* 的指针（或引用）转换到指向另一个类 *T2* 的指针（或引用）。*T1* 和 *T2* 必须属于同一分层结构，类必须是可访问的（通过公共派生），并且转换必须明确。此外，除非转换是从派生类到其一个基类，否则，包括 *T1* 和 *T2* 的分层结构的最小部分必须是多态的（至少有一个虚函数）。

在表达式 dynamic_cast<*T*>(*v*) 中，*v* 是要进行强制类型转换的表达式，*T* 是要转换成的类型。*T* 必须是指向完整类的类型（其定义可见）的指针或引用，或指向 *cv* void 的指针，其中 *cv* 是空字符串、const、volatile 或 const volatile。

9.4.1 将分层结构向上强制类型转换

对分层结构进行向上强制类型转换时，如果 *T* 指向（或引用）*v* 所指向（引用）类型的基类，则该转换等效于 static_cast<*T*>(*v*)。

9.4.2 强制类型转换到 void*

如果 *T* 是 void*，则结果是指向完整对象的指针。也就是说，*v* 可能指向某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 *v* 沿分层结构向下转换为完整对象（无论什么对象）的类型，然后转换为 void*。

强制类型转换到 void* 时，分层结构必须是多态的（有虚函数）。

9.4.3 将分层结构向下或交叉强制类型转换

向下或交叉强制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。

对分层结构向下或交叉强制类型转换时，有时不能从 *v* 转换到 *T*。例如，尝试的转换可能不明确，*T* 可能无法访问，或 *v* 可能未指向（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，则强制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，不会返回任何值（没有 C++ 中的空引用），并且会抛出标准异常 std::bad_cast。

例如，此公共派生的示例成功：

```
#include <assert.h>
#include <stddef.h> // for NULL

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;        // no casts needed
  A* ap = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // succeeds
  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);
}
```

但此示例失败，因为基类 B 不可访问。

```
#include <assert.h>
#include <stddef.h> // for NULL
#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; // C-style cast needed to break protection
  A* ap = dynamic_cast<A*>(bp); // fails, B is inaccessible
  assert(ap == NULL);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // fails, B is inaccessible
  }
  catch(const std::bad_cast&) {
    return; // failed reference cast caught here
  }
  assert(0); // should not get here
}
```

如果在一个单独的基类中存在虚拟继承和多重继承，那么实际动态强制类型转换必须能够识别出唯一的匹配。如果匹配不唯一，则强制类型转换失败。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;
                    // okay: finds unique A statically
  AB*abp = dynamic_cast<AB*>(ap);
                    // fails: ambiguous
  assert(abp == NULL);
                    // STATIC ERROR: AB_B* ab_bp = (AB_B*)ap;
                    // not a dynamic cast
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // dynamic one is okay
  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，一个用于类型确定正确时处理强制类型转换，另一个用于类型确定错误时处理强制类型转换。

```
void using_dynamic_cast(A* ap)
{
  if (AB *abp = dynamic_cast<AB*>(ap))
```

```
    {            // abp is non-null,
                 // so ap was a pointer to an AB object
                 // go ahead and use abp
      process_AB(abp);}
  else
    {          // abp is null,
               // so ap was NOT a pointer to an AB object
               // do not use abp
      process_not_AB(ap);
    }
}
```

在兼容模式 (`-compat[=4]`) 下，如果尚未使用 `-features=rtti` 编译器选项启用运行时类型信息，则编译器将 `dynamic_cast` 转换到 `static_cast` 并发出警告。

如果禁用了异常，编译器会将 `dynamic_cast<T&>` 转换为 `static_cast<T&>` 并发出警告。（对引用类型的 dynamic_cast 要求在运行时发现转换无效的情况下抛出异常。）有关异常的信息，请参见第 93 页中的 “7.5.3 诊断有问题的搜索”。

动态强制类型转换需要比对应的设计模式慢，例如虚函数的转换。请参见由 Erich Gamma 编著的《Design Patterns: Elements of Reusable Object-Oriented Software》（Addison-Wesley 出版，1994）。

◆ ◆ ◆ 第 10 章

10 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。很多书中都对软件性能做了一般性介绍并具体介绍了 C++，本章不再重复这类重要信息，而只讨论那些显著影响 C++ 编译器的性能技术。

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，而非实现和使用只包含 `+` 和 `=` 的操作。例如，下面的第一行引入了用于保存 `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。

如果预计函数定义会更改**而且**重新编译所有调用程序非常耗时，请不要使用内联函数。而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，则可以使用内联函数。

编译器不能内联所有函数调用，因此要充分利用函数内联，可能需要进行一些源码更改。可使用 +w 选项了解何时不会进行函数内联。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 使用缺省运算符

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。它们都是调用的缺省运算符。类似 C 的结构具有这些缺省运算符。编译器生成缺省运算符时，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，而程序员则不能利用该功能的优点。因此缺省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。

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

- 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“不进行任何操作”的无参数构造函数。
- 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程序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程序员没有忘记考虑缺省运算符的语义。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缺省运算符。

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**直接**进行传递。对于使用了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**间接**进行传递。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]`) 下，如果类具有以下任何一项，则间接进行传递：

- 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（缺省模式）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，就使用缺省构造函数，尤其是缺省复制构造函数。
- 尽可能使用缺省析构函数。缺省析构函数不是虚拟的，因此具有缺省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体系结构上进行不同的传递。

表 10–1 在不同体系结构上结构和联合的传递

体系结构	说明
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。通过在调用程序的帧中分配临时对象并将临时对象的地址作为隐式的第一个参数传递，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。

编译器必须经常通过 `this` 指针从内存装入成员变量。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

11 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》。

另请参见《OpenMP API 用户指南》，了解有关使用 OpenMP 共享内存并行化指令来创建多线程程序的信息。

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，必须使用 `–mt` 选项编译和链接程序。此选项会将 `–D_REENTRANT` 传递给预处理程序，并按正确的顺序将 `–lthread` 传递给 `ld`。在兼容模式 (`–compat[=4]`) 下，`–mt` 选项可确保在 `libC` 之前链接 `libthread`。在标准模式（缺省模式）下，`-mt` 选项可确保 `libthread` 在 `libCrun` 之前链接。推荐使用 `–mt`，这是指定宏和库的替代方式，它更加简单且不易出错。

11.1.1 表明多线程编译

可以通过使用 `ldd` 命令检查应用程序是否链接到 `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 是多线程安全的，但不是**异步**安全。这意味着，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中。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

对于多线程而言，当前的异常处理实现是安全的，一个线程中的异常与其他线程中的异常互不影响。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。

每个线程都可设置其自己的 terminate() 或 unexpected() 函数。在一个线程中调用 set_terminate() 或 set_unexpected() 只影响该线程中的异常。对于任何线程，terminate() 的缺省函数是 abort()（请参见第 95 页中的 “8.2 指定运行时错误”）。

11.2.1 线程取消

通过调用 pthread_cancel(3T) 取消线程会导致销毁栈上的自动（局部非静态）对象，但指定了 -noex 或 -features=no%except 时例外。

pthread_cancel(3T) 使用的机制与异常相同。取消线程时，局部析构函数与用户通过 pthread_cleanup_push() 注册的清除例程交叉执行。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 MT 安全的（有些语言环境下例外），确保了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，如果将字符串按值传递给新线程，即使两个不同的线程中的字符串应用 Rogue Wave 的“copy on write（写时复制）”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 类通过下列成员函数提供了与平台无关的锁定机制：

- void acquire()－自己获取锁定，即在获得此类锁定之前一致处于阻塞状态。
- void release()－自己释放锁定。

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

_RWSTDGuard 类是封装有 _RWSTDMutex 类的对象的公用包装器类。_RWSTDGuard 对象尝试在其构造函数中获取封装的互斥锁（抛出从 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锁（析构函数从来不会抛出异常）。

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

另外，可以使用宏 _RWSTD_MT_GUARD(mutex)（以前的 _STDGUARD）有条件地在多线程生成中创建 _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>

//
// An integer shared among multiple threads.
//
int I;
```

```
//
// A mutex used to synchronize updates to I.
//
_RWSTDMutex I_mutex;

//
// Increment I by one. Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire(); // Lock the mutex.
   I++;
   I_mutex.release(); // Unlock the mutex.
}

//
// Decrement I by one. Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex); // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

11.4 在多线程环境中使用传统 iostream

本节介绍如何将 `libC` 和 `libiostream` 库的 `iostream` 类用于多线程环境的输入输出 (input-output, I/O)。另外，还提供了如何通过从 `iostream` 类派生来扩展库的功能的示例。但本节并不是指导如何采用 C++ 编写多线程代码。

此处的讨论只适用于原来的 iostream（`libC` 和 `libiostream`），而不适用于 `libCstd`（即新的 iostream，它是 C++ 标准库的一部分）。

`iostream` 库允许多线程环境中的应用程序使用其接口，以及运行支持的 Solaris 操作系统版本时使用多线程功能的程序使用其接口。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。

如果库能在有多个线程的环境中正常运行，则该库定义为 MT 安全的。通常，此处的“正常”意味着其所有公用函数都是可重入的。`iostream` 库提供了保护，防止多个线程尝试修改由多个线程共享的对象（即 C++ 类的实例）的状态。但 `iostream` 对象的 MT 安全作用域仅限于该对象的公共成员函数正在执行的那一段时间。

注 – 应用程序并不能因为使用 libC 库中的 MT 安全对象，而被自动保证为是 MT 安全的。应用程序只有按预期那样能够在多线程环境中执行时，才被定义为 MT 安全的。

11.4.1 MT 安全的 iostream 库的组织

MT 安全的 iostream 库的组织与其他版本的 iostream 库稍有不同。库的导出接口指的是 iostream 类的受保护的公共成员函数以及可用基类集合，这一点与其他版本的库相同；但类的分层结构是不同的。有关详细信息，请参见第 120 页中的 “11.4.2 iostream 库接口更改”。

原来的核心类已重命名，即添加了前缀 unsafe_。表 11–1 列出了属于 iostream 软件包核心的类。

表 11–1 原来的 iostream 核心类

类	说明
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	该类支持从 streambuf 检索的字符序列的有格式和无格式转换。
unsafe_ostream	该类支持存储到 streambuf 中的字符序列的有格式和无格式转换。
unsafe_iostream	该类合并 unsafe_istream 类和 unsafe_ostream 类，以便进行双向操作。

每个 MT 安全的类都是从基类 stream_MT 派生而来。每个 MT 安全的类（除了 streambuf 外）也都是从现有的 unsafe_ 基类派生而来。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 {...};
```

类 stream_MT 提供了使每个 iostream 类成为 MT 安全的类所需的互斥锁，另外，还提供了动态启用和禁用这些锁的功能，以便可以动态更改 MT 安全属性。用于 I/O 转换和缓冲区管理的基本功能划入 unsafe_ 类，为库增加 MT 安全的功能则划入派生类。每个类的 MT 安全版本都包含与 unsafe_ base 类相同的受保护的公共成员函数。MT 安全版本类中的每个成员函数都可用作包装器，它可以锁定对象、调用 unsafe_ base 中的相同函数以及解锁对象。

注 – streambuf 类**不**是从非安全类派生而来的。streambuf 类的受保护的公共成员函数可以通过锁定来重入。此外还提供了带 _unlocked 后缀的已解锁版本。

11.4.1.1 公共转换例程

已向 iostream 接口添加了一组 MT 安全的重入公共函数。 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	说明
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)	返回指向包含字符 chr 的 ASCII 字符串的指针。如果宽度非零，则字符串包含后跟 chr 的 width 个空格。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，其中使用了格式字符串 format 和其余参数。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的 libC 兼容的 iostream 库的公共转换例程（oct、hex、dec、chr 和 form）不是 MT 安全的。

11.4.1.2 使用 MT 安全的 libC 库进行编译和链接

生成使用 libC 库的 iostream 类以在多线程环境中运行的应用程序时，应使用 -mt 选项编译和链接该应用程序的源代码。此选项可将 -D_REENTRANT 传递给预处理程序，并将 -lthread 传递给链接程序。

注 – 请使用 -mt（而不是 -lthread）与 libC 和 libthread 链接。该选项确保了库的正确链接顺序。错误使用 -lthread 可能会导致应用程序无法正常运行。

对于使用 iostream 类 的单线程应用程序，不需要使用特殊的编译器和链接程序选项。缺省情况下，编译器会与 libC 库链接。

11.4.1.3 MT 安全的 iostream 限制

有关 iostream 库的 MT 安全性的限制定义意味着，用于 iostream 的许多编程常用方式在使用共享 iostream 对象的多线程环境中是不安全的。

检查错误状态

要实现 MT 安全，必须在具有可能导致出现错误的 I/O 操作的关键区中进行错误检查。以下示例说明了如何检查错误：

示例 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;
}
```

在此示例中，stream_locker 对象 sl 的构造函数锁定 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

要实现 MT 安全，必须在执行上次输入操作和 gcount 调用这一期间独占使用 istream 对象的线程内调用 gcount 函数。以下示例说明了对 gcount 的调用：

示例 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(); // lock the stream istr
    istr >> line;
    linecount = istr.gcount();
    sl.unlock(); // unlock istr
    ...
}
```

在此示例中，stream_locker 类的成员函数 lock 和 unlock 定义了程序中的互斥区域。

用户定义的 I/O 操作

要实现 MT 安全，必须锁定为用户定义类型定义且涉及对各个操作进行特定排序的 I/O 操作，才能定义关键区。以下示例说明了用户定义的 I/O 操作：

示例 11–3 用户定义的 I/O 操作

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

    // other definitions...
    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);

    *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

使用此版本的 libC 库中的 MT 安全类会导致一些性能开销，即使是单线程应用程序中也是如此，但如果使用 libC 的 unsafe_ 类，则可避免此开销。

可以使用作用域解析运算符执行基类 unsafe_ 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。

可以使 cout 和 cin 对象成为**不安全**对象，然后执行正常操作，而不是使用 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：

示例 11–4 禁用多线程安全

```
#include <iostream.h>
//disable mt-safety
cout.set_safe_flag(stream_MT::unsafe_object);
//disable mt-safety
cin.set_safe_flag(stream_MT::unsafe_object);
cout.put("4');
cin.read(buf, len);
```

iostream 对象是 MT 安全对象时，有互斥锁定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。为了提高性能，可以动态地启用或禁用 iostream 对象的 MT 安全性。以下示例使 iostream 对象成为 MT 不安全的对象：

示例 11–5 切换到多线程不安全

```
fs.set_safe_flag(stream_MT::unsafe_object);// disable MT-safety
    .... do various i/o operations
```

可以在多个线程**未**共享 iostream 的情况下（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），在代码中安全地使用 MT 不安全的流。

如果显式在程序中插入同步，还可以在多个线程共享 iostream 的环境中安全地使用 MT 不安全的 iostream。以下示例说明了该技术：

示例 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);
    ... do various i/o operations
```

示例 11–6 在多线程不安全的对象中使用同步 （续）

```
    generic_unlock();
```

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

注 – libC 库提供的 stream_locker 类是实现这一目的的首选机制。

有关更多信息，请参见第 123 页中的 “11.4.5 对象锁定”。

11.4.2 iostream 库接口更改

本节介绍为使 iostream 库成为 MT 安全库而对其所做的接口更改。

11.4.2.1 新增类

下表列出了已增加到 libC 接口的新类。

示例 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

下表列出了已增加到 iostream 接口的新类的分层结构。

示例 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 {...};
```

示例 11–8 新增类的分层结构 （续）

```
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

下表列出了已增加到 `iostream` 接口的新函数。

示例 11–9 新函数

```
class streambuf {
public:
  int sgetc_unlocked();
  void sgetn_unlocked(char *, int);
  int snextc_unlocked();
  int 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();
  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 {
```

示例 11–9 新函数 （续）

```
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)
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

全局和多线程应用程序中的静态数据不能在线程间安全共享。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，在缺省情况下，标准流（`cout`、`cin`、`cerr` 和 `clog`）定义为全局共享对象。由于 `iostream` 库是 MT 安全库，它可在执行 `iostream` 对象的成员函数时保护其共享对象的状态不会被其他线程访问或更改。但对象的 MT 安全性作用域限于执行该对象的公共成员函数期间。例如，

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

获得 get 缓冲区中下一个字符，并更新 *ThreadA* 中的缓冲区指针。但如果 *ThreadA* 中的下一个指令是另一个 get 调用，则 libC 库不能保证返回序列中的下一个字符。这是因为存在一些问题，例如，*ThreadB* 可能也在 *ThreadA* 中所做的两次 get 调用之间执行了 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 123 页中的 “11.4.5 对象锁定”。

11.4.4 序列执行

通常，使用 iostream 对象时，一序列 I/O 操作必须是 MT 安全的。例如，如下所示代码：

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

涉及执行 cout 流对象的三个成员函数。由于 cout 是共享对象，因此必须独立执行序列，才能在多线程环境中正常使用关键区。要独立对 iostream 类对象执行一序列操作，必须使用某种形式的锁定。

libC 库提供了 stream_locker 类用于锁定针对 iostream 对象的操作。有关 stream_locker 类的信息，请参见第 123 页中的 “11.4.5 对象锁定”。

11.4.5 对象锁定

最简单的共享对象和多线程问题处理策略是通过确保 iostream 对象是线程局部对象来避免问题。例如，

- 在线程的入口函数内局部声明对象。
- 使用特定于线程的数据声明对象。（有关如何使用特定于线程的数据的信息，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 的。

不过在许多情况下（例如缺省共享标准流对象），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 类对象执行一序列操作，必须使用某种形式的锁定。锁定会增加一些开销，即使是在单线程应用程序中也是如此。是增加锁定还是使 iostream 对象成为线程的专用对象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都使用其自己的 iostream 对象来生成或使用数据，则这些 iostream 对象专用于各自的线程，因此不需要锁定。
- 如果多个线程协同操作（即，共享同一个 iostream 对象），则必须同步对共享对象的访问，而且必须使用某种形式的锁定使序列化操作独立化。

11.4.5.1 stream_locker 类

iostream 库提供了 stream_locker 类，用于锁定针对 iostream 对象的一系列操作。因此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。

可以使用 stream_locker 类的对象使针对流对象的一序列操作独立化。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。

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

在此示例中，stream_locker 对象的构造函数定义了每次只能执行一个线程的互斥区域的开始位置。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同时独立执行，并且在原来的 *ThreadA* 读取文件之前，*ThreadB* 不能更改文件偏移。

另一种使用 stream_locker 对象的方法是显式定义互斥区域。在以下示例中，为了使 I/O 操作和后续错误检查独立化，使用了 vbstream_locker 对象的成员函数 lock 和 unlock 调用。

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

示例 11–11 令 I/O 操作和错误检查独立化 （续）

```
}
```

有关更多信息，请参见 stream_locker(3CC4) 手册页。

11.4.6 多线程安全类

可以通过派生新类来扩展 或专用化 iostream 类的功能。如果将在多线程环境中使用从派生类实例化的对象，则这些类必须是 MT 安全类。

派生 MT 安全类时的注意事项包括：

- 通过保护对象的内部状态不会被多线程修改来使类对象成为 MT 安全对象。为此，应使用互斥锁序列化对受保护的公共成员函数中成员变量的访问。
- 通过使用 stream_locker 对象，使对 MT 安全基类的成员函数的一系列调用独立化。
- 通过在 stream_locker 对象定义的关键区中使用 streambuf 的成员函数 _unlocked 来避免锁定开销。
- 在应用程序直接调用函数情况下，锁定 streambuf 类的公共虚拟函数。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。但如果多个线程共享 iword 或 pword 函数的相同索引，将会出现问题。要使线程成为 MT 安全线程，请使用适当的锁定方案。
- 锁定返回的成员变量的值大于 char 的成员函数。

11.4.7 对象析构

在删除多个线程共享的 iostream 对象之前，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。

示例 11–12 销毁共享对象

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

void *process_rtn(void*)
{
    // body of sub-threads which uses fp...
}
```

示例 11–12 销毁共享对象 （续）

```
void multi_process(const char* filename, int numthreads)
{
    fp = new fstream(filename, ios::in); // create fstream object
                                         // before creating threads.
    // create threads
    for (int i=0; i<numthreads; i++)
            thr_create(0, STACKSIZE, process_rtn, 0, 0, 0);

        ...
    // wait for threads to finish
    for (int i=0; i<numthreads; i++)
            thr_join(0, 0, 0);

    delete fp;                          // delete fstream object after
    fp = NULL;                         // all threads have completed.
}
```

11.4.8 示例应用程序

以下代码是以 MT 安全方式使用来自 libC 的 iostream 对象的多线程应用程序示例。

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。

示例 11–13 以 MT 安全方式使用 iostream 对象

```
// create tagged thread data
// the output file is of the form:
//         <tag><string of data>\n
// where tag is an integer value in a unsigned char.
// Allows up to 255 threads to be run in this application
// <string of data> is any printable characters
// Because tag is an integer value written as char,
// you need to use od to look at the output file, suggest:
//            od -c out.file |more

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

struct thread_args {
```

示例 11–13 以 MT 安全方式使用 iostream 对象 （续）

```
  char* filename;
  int thread_tag;
};

const int thread_bufsize = 256;

// entry routine for each thread
void* ThreadDuties(void* v) {
// obtain arguments for this thread
  thread_args* tt = (thread_args*)v;
  char ibuf[thread_bufsize];
  // open thread input file
  ifstream instr(tt->filename);
  stream_locker lockout(cout, stream_locker::lock_defer);
  while(1) {
  // read a line at a time
    instr.getline(ibuf, thread_bufsize - 1, '\n');
    if(instr.eof())
      break;
  // lock cout stream so the i/o operation is atomic
    lockout.lock();
  // tag line and send to cout
    cout << (unsigned char)tt->thread_tag << ibuf << "\n";
    lockout.unlock();
  }
  return 0;
}

int main(int argc, char** argv) {
  // argv: 1+ list of filenames per thread
   if(argc < 2) {
     cout << “usage: " << argv[0] << " <files..>\n";
     exit(1);
   }
  int num_threads = argc - 1;
  int total_tags = 0;

// array of thread_ids
  thread_t created_threads[thread_bufsize];
// array of arguments to thread entry routine
  thread_args thr_args[thread_bufsize];
  int i;
  for(i = 0; i < num_threads; i++) {
    thr_args[i].filename = argv[1 + i];
// assign a tag to a thread - a value less than 256
    thr_args[i].thread_tag = total_tags++;
```

示例 11–13 以 MT 安全方式使用 iostream 对象 （续）

```
// create threads
    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 3 部分

库

◆ ◆ ◆ 第 12 章

12 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 操作系统附带了安装在 `/usr/lib` 中的几个库。这些库大多有 C 接口。缺省情况下，其中的 `libc` 和 `libm` 库通过 `CC` 驱动程序进行链接。如果使用 `–mt` 选项，则链接 `libthread` 库。如果要链接其他系统库，请在链接时使用适当的 `–l` 选项。例如，要链接 `libdemangle` 库，请在链接时在 `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 C++ 应用程序都由 `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 随 C++ 编译器提供的库

C++ 编译器附带了一些库。其中一些只能在兼容模式 (`-compat=4`) 下使用，有些只能在标准模式 (`–compat=5`) 下使用，有些可以在这两种模式下使用。`libgc` 和 `libdemangle` 库都有 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	说明	可用模式
`libstlport`	标准库的 STLport 实现。	`–compat=5`

表 12–1 C++ 编译器附带的库 （续）

库	说明	可用模式
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 iostream	–compat=5
libC	C++ 运行时，传统 iostream	–compat=4
libcsunimath	支持 -xia 选项	–compat=5
libcomplex	复数库	–compat=4
librwtool	Tools.h++ 7	–compat=4、–compat=5
librwtool_dbg	支持调试的 Tools.h++ 7	–compat=4、–compat=5
libgc	垃圾收集	C 接口
libdemangle	还原	C 接口

注 – 请勿重新定义或修改用于 STLport、Rogue Wave 或 Sun Microsystems C++ 库的任何配置宏。库是按照适用于 C++ 编译器的方式进行配置和生成的。libCstd 和 Tool.h++ 配置为可互操作，因此，修改配置宏会导致程序不能编译、不能链接或不能正常运行。

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) 下编译器所需的运行时支持，并提供了对 new/delete、异常及 RTTI 的支持。

  libCstd： 这是 C++ 标准库。具体来说，该库包含了 iostream。如果有使用传统 iostream 的现有源代码，而且要使用标准 iostream，必须修改源代码以符合新接口。有关详细信息，请参见《C++ 标准库参考》联机手册。

如果您的编译器软件没有安装在 /opt 目录中，请向系统管理员询问该软件在系统中的安装路径。

- libiostream： 这是使用 – compat=5 生成的传统 iostream 库。如果有使用传统 iostream 的源代码，且要在标准模式 (–compat=5) 下编译这些源代码，可以使用 libiostream 而不必修改源代码。可使用 – library=iostream 获取此库。

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`) 下所需的库。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。可以通过指定选项 `-library=stlport4`，使用该库而非缺省的 `libCstd`。但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++ 是来自 RogueWave 的 C++ 基础类库。提供了版本 7。该库已过时，不应在新代码中使用该库。提供它是为了支持针对使用 RW Tools.h++ 的 C++ 4.2 编写的程序。
- `libgc`：该库用于部署模式或垃圾收集模式。只是与 `libgc` 库链接就会自动且永久修复程序的内存泄漏。虽然能以其他方式正常编程，但如果将程序与 `libgc` 库链接，则无需调用 `free` 或 `delete` 就可完成编程。垃圾收集库对动态装入库具有依赖性，因此在链接程序时要指定 `-lgc` 和 `-ldl`。

  有关其他信息，请参见 `gcFixPrematureFrees`(3) 和 `gcInitialize`(3) 手册页。
- `libdemangle`：该库用于还原 C++ 损坏名称。

12.2.2 访问 C++ 库的手册页

与本节所述库关联的手册页位于第 1、3、3C++ 和 3cc4 节中。

要访问 C++ 库的手册页，请输入：

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

要访问 C++ 库版本 4.2 的手册页，请输入：

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

12.2.3 缺省 C++ 库

缺省情况下，`CC` 驱动程序会链接其中一些 C++ 库，而其他库需要显式链接。在标准模式下，`CC` 驱动程序缺省链接下列库：

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

在兼容模式 (`-compat`) 下，缺省链接下列库：

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

有关更多信息，请参见第 235 页中的 “A.2.50 -library=*l*[,*l*...]”。

12.3 相关的库选项

CC 驱动程序提供了一些选项来帮助用户使用库。

- – l 选项用于指定要链接的库。
- – L 选项用于指定要在其中搜索库的目录。
- -mt 选项用于编译和链接多线程代码。
- -xia 选项用于链接区间运算库。
- -xlang 选项用于链接 Fortran 或 C99 运行时库。
- –library 选项用于指定 Sun C++ 编译器附带的下列库：
  - 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 形式的 librwtool，请使用 -library=rwtools7_std 选项。

  使用 –library 和 –staticlib 选项指定的库将静态链接。某些示例：

  以下命令动态链接传统 iostream 形式的 Tools.h++ 版本 7 和 libiostream 库。

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

- 以下命令静态链接 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 libC。因为在兼容模式下缺省链接 libC，所以不必使用 – 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，否则缺省情况下这两个库包括在内。

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

缺省情况下，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 仅链接在命令行上使用 -l 选项显式指定的那些库。（如果使用 -xnolib 或 -nolib，会忽略 -library 选项（如果有）。）

使用 –R 选项可以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。缺省情况下，CC 驱动程序将 –R<*install_directory*>/lib 传递给 ld（如果编译器安装在标准位置中）。可以使用 -norunpath 禁止在可执行文件中生成共享库的缺省路径。

对于针对部署生成的程序，应该使用 -norunpath 或 -R 选项进行生成，这样可避免在编译器目录中查找库。（请参见第 139 页中的 “12.6 使用共享库”）。

12.4 使用类库

通常，使用类库分两个步骤：

1. 在源码中包括适当的头文件。
2. 将程序与目标库链接。

12.4.1 iostream 库

C++ 编译器提供两种 iostream 实现：

- **传统 iostream**。该术语是指 C++ 4.0、4.0.1、4.1 和 4.2 编译器以及早期基于 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。在兼容模式下，该库是 libC 的一部分，在标准模式下，libiostream 中也有该库。
- **标准 iostream**。这是 C++ 标准库 libCstd 的一部分，仅用于标准模式下。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。

```
// file 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 prog1.cc，并将其链接到名为 prog1 的可执行程序中。传统 iostream 库是 libC 的一部分，兼容模式下缺省链接该库。

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

下一个示例使用标准 iostream。

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

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

以下命令编译 `prog2.cc` 并将其链接到名为 `prog2` 的可执行程序中。该程序在标准模式下编译，且缺省链接包括标准 iostream 库的 `libCstd`。

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

有关 `libCstd` 的更多信息，请参见第 143 页中的 “忠告： ”。有关 libiostream 的更多信息，请参见第 159 页中的 “13.3.1 重新分发和支持的 STLport 库”。

有关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，必须使用 `<complex>` 而非 `<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，缺省情况下链接该库。

标准模式下，没有 `complex.h` 头文件。在 C++ 4.2 中，"complex" 是类名称，但在标准 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。

```
// file ex1.cc (compatibility mode)
#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)
```

下面将 ex1.cc 重写为 ex2.cc 以在标准模式下编译：

```
// file ex2.cc (ex1.cc rewritten for standard mode)
#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，请参见表 14–4。

12.4.3 链接 C++ 库

下表列出了链接 C++ 库的编译器选项。有关更多信息，请参见第 235 页中的 “A.2.50 -library=*l*[,*l*...]”。

表 12–2 链接 C++ 库的编译器选项

库	编译模式	选项
传统 iostream	–compat=4	不需要
	–compat=5	-library=iostream
complex	–compat=4	-library=complex
	-compat=5	不需要

表 12–2 链接 C++ 库的编译器选项 （续）

库	编译模式	选项
Tools.h++ 版本 7	–compat=4	-library=rwtools7
	–compat=5	-library=rwtools7,iostream
		-library=rwtools7_std
Tools.h++ 版本 7 调试	–compat=4	-library=rwtools7_dbg
	–compat=5	-library=rwtools7_dbg,iostream
		-library=rwtools7_std_dbg
垃圾收集	–compat=4	-library=gc
	–compat=5	-library=gc
STLport 版本 4	–compat=5	-library=stlport4
STLport 版本 4 调试	–compat=5	-library=stlport4_dbg

12.5 静态链接标准库

CC 驱动程序在缺省情况下链接几个库的共享版本（包括 libc 和 libm），这通过为每个缺省库将 -l*lib* 选项传递给链接程序来实现。（有关兼容模式和标准模式下的缺省库列表，请参见第 133 页中的 “12.2.3 缺省 C++ 库”。）

如果要静态链接其中任何缺省库，可以使用 -library 选项和 –staticlib 选项来静态链接 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
```

在此示例中，没有在命令中显式包括 -library 选项。这种情况下，无需 -library 选项，因为在标准模式（缺省模式）下，-library 的缺省设置是 Cstd,Crun。

也可以使用 -xnolib 编译器选项。使用 -xnolib 选项时，驱动程序不会将任何 -l 选项传递给 ld，所以您必须自己传递这些选项。以下示例显示了在 Solaris 8 或 Solaris 9 操作系统中如何静态链接 libCrun 以及如何动态链接 libm 和 libc：

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

-l 选项的顺序很重要。– lCstd、– lCrun 和 -lm 选项位于 -lc 之前。

注 – 建议不要静态链接 libCrun 和 libCstd。而是生成 /usr/lib 中的动态版本以与其所安装在的 Solaris 版本一起使用。

有些 CC 选项链接到其他库。也可以使用 -xnolib 抑制这些库链接。例如，使用 -mt 选项会导致 CC 驱动程序将 -lthread 传递给 ld。但如果同时使用 –mt 和 –xnolib，CC 驱动程序不会将 -lthread 传递给 ld。有关更多信息，请参见第 296 页中的 “A.2.153 –xnolib”。有关 ld 的更多信息，请参见《链接程序和库指南》。

注 – /lib 和 /usr/lib 中静态版本的 Solaris 库不再可用。例如，试图静态链接 libc 的操作将失败：

```
CC hello.cc -xnolib -lCrun -lCstd -Bstatic -lc
```

12.6 使用共享库

C++ 编译器附带下列 C++ 运行时共享库：

- libCCexcept.so.1（仅限 SPARC Solaris）
- libcomplex.so.5（仅限 Solaris）
- librwtool.so.2
- libstlport.so.1

在 Linux 上，C++ 编译器附带这些附加库：

- libCrun.so.1
- libCstd.so.1
- libdemangle.so
- libiostream.so.1

在 Solaris 上，这些附加库以及其他一些库作为 Solaris C++ 运行时库软件包 SUNWlibC 的一部分安装。

如果应用程序使用 C++ 编译器附带的任何共享库，则 CC 驱动程序会安排**运行路径**（请参阅 -R 选项），该运行路径指向将在可执行文件中生成库的位置。如果之后将可执行文件部署到另一台计算机上，而该计算机上并没有在同一位置安装同一编译器版本，将找不到所需的共享库。

在程序启动时，可能根本找不到此库，或可能使用错误版本的库，从而导致错误的程序行为。在这种情况下，应该将所需库与可执行文件一起提供，并使用指向这些库将要安装的位置的**运行路径**进行生成。

《Using and Redistributing Sun Studio Libraries in an Application》（《在应用程序中使用与重新分发 Sun Studio 库》一文中详细讨论了该主题并提供了示例，请参见 http://developers.sun.com/sunstudio/documentation/techart/stdlibdistr.html。

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。有关更多信息，请参见第 158 页中的 “13.3 STLport”。

12.7.1 可以替换的内容

可以替换大多数标准库及其关联头文件。替换的库是 `libCstd`，关联的头文件如下所示：

```
<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>
```

库的可替换部分由一般称为 "STL" 的内容和字符串类、iostream 类及其帮助类组成。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。如果要替换任何一部分，就应该替换所有头文件和所有 `libCstd`。

12.7.2 不可替换的内容

标准头文件 `<exception>`、`<new>` 和 `<typeinfo>` 与编译器本身以及 `libCrun` 紧密相关，不能可靠替换。库 `libCrun` 包含了编译器依赖且不能替换的许多“帮助”函数。

从 C 继承的 17 个标准头文件（`<stdlib.h>`、`<stdio.h>`、`<string.h>` 等）与 Solaris 操作系统和 Solaris 基本运行时库 `libc` 紧密相关，不能可靠替换。这些头文件的 C++ 版本（`<cstdlib>`、`<cstdio>`、`<cstring>` 等）与基本 C 版本紧密相关，不能可靠替换。

12.7.3 安装替换库

要安装替换库，必须先确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。假定库称为 `libmyCstd.a`。（库名最好以 "lib" 开头。）

12.7.4 使用替换库

每次编译时，都使用 -I 选项指向头文件的安装位置。此外，还使用 -library=no%Cstd 选项防止查找编译器自身版本的 libCstd 头文件。例如：

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

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

每次执行程序或库链接时，都使用 -library=no%Cstd 选项防止查找编译器自身的 libCstd，使用 -L 选项指向替换库所在的目录，以及使用 -l 选项指定替换库。示例：

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

也可以直接使用库的完整路径名，而不使用 -L 和 -l 选项。例如：

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

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

12.7.5 标准头文件实现

C 有 17 个标准头文件（<stdio.h>、<string.h>、<stdlib.h> 等）。这些头文件以 Solaris 操作系统的一部分提供，位于 /usr/include 目录中。C++ 也有这些头文件，但另外要求在全局名称空间和 std 名称空间中都有各种声明的名称。在 Solaris 操作系统版本 8 之前的版本中，C++ 编译器提供了自身版本的这些头文件而不是替换 /usr/include 目录中的头文件。

C++ 也有另一个版本的各个 C 标准头文件（<cstdio>、<cstring> 和 <cstdlib> 等），仅名称空间 std 中有各种声明的名称。最后，C++ 添加了 32 个自己的标准头文件（<string>、<utility>、<iostream> 等）。

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>）可能指某目录中名为 string（或 string.h）的文件。这种明显实现有以下缺点：

- 如果头文件没有文件名后缀，则无法仅搜索头文件或为头文件创建 makefile 规则。
- 如果具有名为 string 的目录或可执行程序，可能会错误地找到该目录或程序而不是标准头文件。
- 在 Solaris 8 操作系统之前的 Solaris 操作系统版本上，启用了 .KEEP_STATE 时，makefile 的缺省依赖性会导致尝试将标准头文件替换为可执行程序。（没有后缀的文件缺省假定为要生成的程序。）

为了解决这些问题，编译器 include 目录会包含一个与头文件同名的文件和一个指向它且具有唯一后缀 .SUNWCCh （SUNW 是所有编译器相关软件包的前缀，CC 指 C++ 编译器，h 是常用的头文件后缀）的符号链接。指定 <string> 后，编译器将其重写为 <string.SUNWCCh> 并搜索该名称。后缀名只能在编译器自己的 include 目录中找到。如果这样找到的文件是符号链接（正常情况下），编译器就对链接进行一次引用解除，并将结果（此例中是 string）用作错误消息和调试器引用的文件名。忽略文件的依赖性信息时，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

源代码	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，则编译器将重新搜索 #include 指令中提供的名称。例如，如果使用指令 #include <string>，编译器就尝试查找名为 string.SUNWCCh 的文件。如果搜索失败，编译器就查找名为 string 的文件。

12.7.5.1 替换标准 C++ 头文件

由于第 141 页中的 “12.7.5 标准头文件实现”中介绍的搜索算法，您无需提供第 140 页中的 “12.7.3 安装替换库”中介绍的 SUNWCCh 版本的替换头文件。但是会遇到某些上文所述的问题。如果这样，建议为每个无后缀的头文件添加后缀为 .SUNWCCh 的符号链接。也就是说，对于文件 utility，可以运行命令

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

编译器第一次查找 utility.SUNWCCh 时，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。
- 在该目录中创建指向每个替换头文件的 `.SUNWCCh` 符号链接。
- 在每次调用编译器时使用 `-I` 指令，搜索包含替换头文件的目录。

例如，假设有 `<stdio.h>` 和 `<cstdio>` 的替换。请将文件 `stdio.h` 和 `cstdio` 放在目录 `/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。

忠告：

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

◆ ◆ ◆ 第 13 章

13 使用 C++ 标准库

当在缺省（标准）模式下编译时，编译器可以访问 C++ 标准指定的整个库。库组件包括一般称为标准模板库 (Standard Template Library, STL) 的库和下列组件。

- 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 RogueWave™ Standard C++ Library, Version 2。该库仅在编译器的缺省模式 (`-compat=5`) 下可用，并且在使用 `-compat[=4]` 选项时不支持该库。

C++ 编译器还支持 STLport 的标准库实现版本 4.5.3。`libCstd` 仍是缺省库，STLport 的产品只是备选的。有关更多信息，请参见第 158 页中的 “13.3 STLport”。

如果需要使用自己的 C++ 标准库版本而非编译器附带的某一版本，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 140 页中的 “12.7 替换 C++ 标准库”。

有关标准库的详细信息，请参见《标准 C++ 库用户指南》和《Standard C++ Class Library Reference》。

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文件的简要介绍。

表 13–1 C++ 标准库头文件

头文件	说明
`<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` 操纵符
`<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

表 13–1 C++ 标准库头文件 （续）

头文件	说明
<numeric>	通用的数字操作
<ostream>	输出 I/O 流
<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>	比较运算符
<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 (standard template library, 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	请参见与 make_heap、pop_heap、push_heap 和 sort_heap 对应的条目
Input_Iterators	只读的前移式迭代器

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
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 ostream 和 istream 的迭代器功能
__distance_type	确定迭代程序所用距离的类型（已过时）
__iterator_category	确定迭代程序所属的类别（已过时）
__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缺省分配器对象
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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> 类的对象
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
basic_ostringstream	支持写入 basic_string<charT,traits,Allocator> 类的对象
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	支持在内存中的数组中写入和读取 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	控制向与 <cstdio> 中声明的对象 stderr 关联的无缓冲流缓冲区的输出
char_traits	具有用于 basic_string 容器和 iostream 类的类型和运算的特性类
cin	控制从与 <cstdio> 中声明的对象 stdin 关联的流缓冲区的输入
clog	控制向与 <cstdio> 中声明的对象 stderr 关联的流缓冲区的输出
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
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	控制向与 <cstdio> 中声明的对象 stdout 关联的流缓冲区的输出
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
istringstream	支持从内存中的数组读取 basic_string<charT,traits,Alocator> 类的对象
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	请参阅 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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
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返回前两个不匹配的元素
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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	重新排列集合，以使按照排序顺序，低于第 *n* 个元素的所有元素位于该元素之前，高于第 *n* 个元素的所有元素位于该元素之后
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 ostream 和 istream 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
ostringstream	支持写入 basic_string<charT,traits,Allocator> 类的对象
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	采用指向二元函数的指针的函数对象，替代 binary_function
pointer_to_unary_function	采用指向函数的指针的函数对象类，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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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
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中间减 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中间减 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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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	用于实现参数化操纵符的帮助程序类
smanip_fill	用于实现参数化操纵符的帮助程序类
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
stringbuf	将输入或者输出序列与任意字符序列关联
stringstream	支持在内存中的数组中写入和读取 basic_string<charT,traits,Alocator> 类的对象
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
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	控制向与 <cstdio> 中声明的对象 stderr 关联的无缓冲流缓冲区的输出
wcin	控制从与 <cstdio> 中声明的对象 stdin 关联的流缓冲区的输入
wclog	控制向与 <cstdio> 中声明的对象 stderr 关联的流缓冲区的输出
wcout	控制向与 <cstdio> 中声明的对象 stdout 关联的流缓冲区的输出
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

表 13–2 C++ 标准库手册页 （续）

手册页	概述
wistringstream	支持从内存中的数组读取 basic_string<charT,traits,Allocator> 类的对象
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 basic_string<charT,traits,Allocator> 类的对象
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

如果要使用替换 libCstd 的标准库，请使用标准库的 STLport 实现。可以使用以下编译器选项关闭 libCstd 并改用 STLport 库：

- -library=stlport4

有关更多信息，请参见第 235 页中的 “A.2.50 -library=*l*[,*l*...]”。

本发行版包括称为 libstlport.a 的静态归档文件和称为 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 库都提供了自己的 I/O 流实现。如果使用 -library 选项指定其中多个库，会导致出现不确定的程序行为。
- 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。在将来发行版中可能不能使用编译器选项 -library=stlport4，但可能会用引用更高 STLport 版本的选项替换该选项。
- Tools.h++ 不支持 STLport。
- STLport 与缺省的 libCstd 是二进制不兼容的。如果使用标准库的 STLport 实现，则必须使用选项 -library=stlport4 编译和链接包括第三方库在内的所有文件。这意味着存在一些限制，例如，不可同时使用 STLport 实现和 C++ 区间数学库 libCsunimath。这是因为对 libCsunimath 编译时使用的是缺省的库头文件而不是 STLport。

- 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，则不能使用 STLport 实现。

13.3.1 重新分发和支持的 STLport 库

请参见运行时库自述文件，了解可依照“最终用户目标代码许可协议”的条款随您的可执行文件或库重新分发的库和目标文件列表。此自述文件的 C++ 部分列出该编译器发行版支持的 STLport .so 版本。此自述文件作为产品文档的一部分可以从 Sun Studio SDN 门户网站 (`http://developers.sun.com/sunstudio/documentation/`) 上获取。

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。具体来说，它假定 `<vector>` 或 `<iostream>` 自动包含 `<iterator>`，这是无效假定。

```
#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;
}
```

要解决该问题，请将 `<iterator>` 包含在源代码中。

◆ ◆ ◆ 第 14 章

14 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（缺省模式）下，传统 iostream 类包含在 libiostream 中。如果源代码使用传统 iostream 类，且要在标准模式下编译源代码，请使用 libiostream。如果要在标准模式下使用传统 iostream 功能，请将 iostream.h 头文件包括进来并使用 -library=iostream 选项进行编译。
- 标准 iostream 类只能用于标准模式下，且包含在 C++ 标准库 libCstd 中。

本章介绍了传统 iostream 库并提供了其使用示例，但并未完整介绍 iostream 库。有关更多详细信息，请参见 iostream 库手册页。要访问传统 iostream 手册页，请键入：
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 都是完全缓冲的。请参见第 163 页中的 “14.3.1 使用 iostream 进行输出”和第 166 页中的 “14.3.2 使用 iostream 进行输入”。

14.2 iostream 交互的基本结构

通过将 iostream 库包括进来，程序可以使用许多输入流或输出流。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 类或从其派生的类实现。
- 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，这两个类将从 streambuf 类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 类用于处理文件的输入和输出。

分别从 istream、ostream 和 iostream 派生的 istrstream、ostrstream 和 strstream 类用于处理字符数组的输入和输出。

打开输入或输出流时，要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。通常通过流构造函数执行此关联，因此不用直接使用 streambuf。iostream 库为标准输入、标准输出和错误输出预定义了流对象，因此不必为这些流创建自己的对象。

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，以及控制插入或提取的数据的格式。

如果要插入和提取新的数据类型（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

要使用来自传统 iostream 库的例程，必须针对所需的库部分将头文件包括进来。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	说明
iostream.h	声明 iostream 库的基本功能。
fstream.h	声明文件专用的 iostream 和 streambuf。包括了 iostream.h。
strstream.h	声明字符数组专用的 iostream 和 streambuf。包括了 iostream.h。
iomanip.h	声明操纵符：在 iostream 中插入或提取的值有不同的作用。包括了 iostream.h。
stdiostream.h	（已过时）声明 stdio 文件专用的 iostream 和 streambuf。包括了 iostream.h。
stream.h	（已过时）包括了 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，它由 CC 驱动程序自动链接。在标准模式（缺省模式）下，libiostream 包含传统 iostream 库。

14.3.1 使用 iostream 进行输出

使用 iostream 进行的输出通常依赖于重载的左移运算符 (<<)（在 iostream 上下文中，称为插入运算符）。要将值输出到标准输出，应将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;
```

对于所有内置类型，都会重载插入运算符，且 someValue 表示的值转换为其适当的输出表示形式。例如，如果 someValue 是 float 值，<< 运算符会将该值转换为带小数点的适当数字序列。此处是将 float 值插入输出流中，因此 << 称为浮点插入器。通常，如果给定类型 X，<< 称为 X 插入器。ios(3CC4) 手册页中讨论了输出格式以及如何控制输出格式。

iostream 库不支持用户定义的类型。如果要定义以自己的方式输出的类型，必须定义一个插入器（即，重载 << 运算符）来正确处理它们。

<< 可以多次应用。要将两个值插入 cout 中，可以使用类似于以下示例中的语句：

```
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 << 优先作为左移运算符（其内置含义）使用。与其他运算符一样，总是可以使用圆括号来指定操作顺序。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;              // + has higher precedence than <<
cout << (a+b);
cout << (a&y);            // << has precedence higher than &
cout << a&y;            // probably an error: (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:
    // (functions not relevant here)

    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);
};
```

在此示例中，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;}
```

以下是为要用于 string 而重载的 operator<< 的定义。

```
cout << string1 << string2;
```

运算符 << 将 ostream&（也就是对 ostream 的引用）作为其第一个参数，并返回相同的 ostream，这样就可以在一个语句中合并多个插入。

14.3.1.2 处理输出错误

通常情况下，重载 operator<< 时不必检查错误，因为有 iostream 库传播错误。

出现错误时，所在的 iostream 会进入错误状态。iostream 状态中的各个位根据错误的一般类别进行设置。iostream 中定义的插入器不会尝试将数据插入处于错误状态的任何流，因此这种尝试不会更改 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。本章假定您定义了函数 error，该函数可采用字符串并中止程序。error 不是预定义的函数。有关 error 函数的示例，请参见第 168 页中的 “14.3.9 处理输入错误”。可以使用运算符 ! 检查 iostream 的状态，如果 iostream 处于错误状态，该运算符会返回非零值。例如：

```
if (!cout) error("output error");
```

还有另外一种方法来测试错误。ios 类可定义 operator void *()，它在有错误时返回空指针。您可以使用如下语句：

```
if (cout << x) return; // return if successful
```

也可以使用函数 good，它是 ios 的成员：

```
if (cout.good()) return; // return if successful
```

错误位在 enum 中声明：

```
enum io_state {goodbit=0, eofbit=1, failbit=2,
badbit=4, hardfail=0x80};
```

有关错误函数的详细信息，请参见 iostream 手册页。

14.3.1.3 刷新

与大多数 I/O 库一样，iostream 通常会累积输出并将其发送到较大且效率通常较高的块中。如果要刷新缓冲区，只要插入特殊值 flush。例如：

```
cout << "This needs to get out immediately." << flush;
```

flush 是一种称为**操纵符**的对象示例，它是一个值，可以插入 iostream 中以起到一定作用，而不是使输出其值。它实际上是一个函数，采用 ostream& 或 istream& 参数，在对其执行某些操作后返回其参数（请参见第 172 页中的 “14.7 操纵符”）。

14.3.1.4 二进制输出

要获得原始二进制形式的值输出，请使用以下示例所示的成员函数 write。该示例显示了原始二进制形式的 x 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。一般情况下，这样做无关大碍，但如果 x 的类型是具有指针、虚拟成员函数的类或是需要 nontrivial 构造函数操作的类，就无法正确读回以上示例写入的值。

14.3.2 使用 iostream 进行输入

使用 iostream 进行的输入类似于输出。需要使用提取运算符 >>，可以像插入操作那样将提取操作串接在一起。例如：

```
cin >> a >> b;
```

该语句从标准输入获得两个值。与其他重载运算符一样，所用的提取器取决于 a 和 b 的类型（如果 a 和 b 的类型不同，则使用两个不同的提取器）。ios(3CC4) 手册页中详细讨论了输入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 定义了其提取运算符，如以下代码示例所示：

示例 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，并将其存储在 holder 中，直到读取了 maxline-1 字符、遇到新行或 EOF（无论先发生哪一项）。然后，holder 中的数据以空终止。最后，holder 中的字符复制到目标字符串。

按照约定，提取器转换其第一个参数中的字符（此示例中是 istream& istr），将其存储在第二个参数（始终是引用），然后返回第一个参数。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，提取字符并将其复制到 x 中，直至遇到另一个空白字符。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; // not initialized
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，还可以使用任一形式的 get 成员函数获取一个字符。例如：

```
char c;
cin.get(c); // leaves c unchanged if input fails

int b;
b = cin.get(); // sets b to EOF if input fails
```

注 – 与其他提取器不同，char 提取器不会跳过前导空白。

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（如使用成员函数 write 写入的值），可以使用 read 成员函数。以下示例说明了如何使用 read 成员函数输入原始二进制形式的 x，这是先前使用 write 的示例的反向操作。

```
cin.read((char*)&x, sizeof(x));
```

14.3.7 查看输入

可以使用 peek 成员函数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

缺省情况下，iostream 提取器会跳过前导空白。可以关闭 **跳过**标志防止这种情况发生。以下示例先关闭了 cin 跳过空白功能，然后将其打开：

```
cin.unsetf(ios::skipws); // turn off whitespace skipping
...
cin.setf(ios::skipws); // turn it on again
```

可以使用 iostream 操纵符 ws 从 iostream 中删除前导空白，不论是否启用了跳过功能。以下示例说明了如何从 iostream istr 中删除前导空白：

```
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 具有可用于错误处理的成员函数。详细信息请参见手册页。

14.3.10 结合使用 iostream 与 stdio

可以将 stdio 用于 C++ 程序，但在程序内的同一标准流中混合使用 iostream 与 stdio 时，可能会发生问题。例如，如果同时向 stdout 和 cout 写入，会发生独立缓冲，并产生不可预料的结果。如果从 stdin 与 cin 两者进行输入，问题会更加严重，因为独立缓冲可能会破坏输入。

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 与相应的预定义 stdio 文件连接起来。

```
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，所以该连接不是缺省连接。可以在应用于不同文件的同一程序中同时使用 stdio 和 iostream。也就是说，可以使用 stdio 例程向 stdout 写入，并向连接到 iostream 的其他文件写入。可以打开 stdio 文件进行输入，也可以从 cin 读取，只要不同时尝试从 stdin 读取即可。

14.4 创建 iostream

要读取或写入不是预定义的 iostream 的流，需要创建自己的 iostream。通常，这意味着创建 iostream 库中所定义类型的对象。本节讨论了可用的各种类型：

14.4.1 使用类 fstream 处理文件

处理文件类似于处理标准输入和标准输出；类 ifstream、ofstream 和 fstream 分别从类 istream、ostream 和 iostream 派生而来。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。

可将文件 fstream.h 包括进来以使用任何 fstream。如果只要执行输入，请使用 ifstream；如果只要执行输出，请使用 ofstream；如果要对流执行输入和输出，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，如以下示例所示：

```
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：

- 使用 ios::in 的缺省模式创建名为 fromFile 的 ifstream 对象，并将其连接到 thisFile。然后打开 thisFile。
- 检查新的 ifstream 对象的错误状态，如果它处于失败状态，则调用 error 函数（必须在程序中其他地方定义）。
- 使用 ios::out 的缺省模式创建名为 toFile 的 ofstream 对象，并将其连接到 thatFile。

- 按上文所述检查 toFile 的错误状态。
- 创建 char 变量用于存放传递的数据。
- 将 fromFile 的内容复制到 toFile，每次一个字符。

注 – 当然这种方法（每次复制一个字符）不适用于复制文件。该代码只是 fstream 使用示例。应该将与输入流关联的 streambuf 插入输出流中。请参见第 176 页中的 “14.10 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 中的 or-ing 位构造，它是类 ios 的公有类型，其定义如下：

```
enum open_mode {binary=0, in=1, out=2, ate=4, app=8, trunc=0x10,
     nocreate=0x20, noreplace=0x40};
```

注 – UNIX 中不需要 binary 标志，提供该标志是为了与需要它的系统兼容。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，以下代码打开了文件 someName 用于输入和输出，同时将其连接到 fstream 变量 inoutFile。

```
fstream inoutFile("someName", ios::in|ios::out);
```

14.4.1.2 在未指定文件的情况下声明 fstream

可以在未指定文件的情况下声明 fstream，并在以后打开该文件。例如，以下代码创建了 ofstream toFile，以便进行写入。

```
ofstream toFile;
toFile.open(argv[1], ios::out);
```

14.4.1.3 打开和关闭文件

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

如果了解文件描述符（如整数 1 表示标准输出），可以使用如下代码打开它：

```
ofstream outfile;
outfile.attach(1);
```

如果通过向 fstream 构造函数之一提供文件名或使用 open 函数来打开文件，则在销毁 fstream（通过 delete 销毁或其超出作用域）时，会自动关闭该文件。将文件 attach 到 fstream 时，不会自动关闭该文件。

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 是可以记录 iostream 中的位置的类型。
- 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 类的成员函数调用。
- seekg (seekp) 是查找给定位置的 istream (ostream) 成员函数。
- seek_dir enum 指定相对位置以用于 seek。

```
enum seek_dir {beg=0, cur=1, end=2};
```

例如，给定 fstream aFile：

```
streampos original = aFile.tellp();     //save current position
aFile.seekp(0, ios::end); //reposition to end of file
aFile << x;               //write a value to file
aFile.seekp(original);    //return to original position
```

seekg (seekp) 可以采用一个或两个参数。如果有两个参数，第一个参数是相对于 seek_dir 值（也就是第二个参数）指示的位置的位置。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 – 并不能方便地在文本流中进行任意查找，但总是可以返回到以前保存的 streampos 值。

14.5 iostream 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

ios(3CC4) 手册页中详细讨论了格式控制。

14.7 操纵符

操纵符是可以在 iostream 中插入或提取以起到特殊作用的值。

参数化操纵符是具有一个或多个参数的操纵符。

因为操纵符是普通的标识符，因此会用完可能的名称，而 iostream 不会为每个可能的函数定义操纵符。本章的其他部分讨论了各种操纵符和成员函数。

有 13 个预定义的操纵符，如表 14–2 中所述。使用该表时，要进行以下假设：

- i 的类型为 long。
- n 的类型为 int。
- c 的类型为 char。
- istr 是输入流。
- ostr 是输出流。

表 14–2 iostream 的预定义操纵符

	预定义的操纵符	说明
1	ostr << dec、istr >> dec	以 10 为基数进行整数转换。
2	ostr << endl	插入一个换行符 ('\n') 并调用 ostream::flush()。
3	ostr << ends	插入一个空 (0) 字符。这在处理 strstream 时很有用。
4	ostr << flush	调用 ostream::flush()。
5	ostr << hex、istr >> hex	以 16 为基数进行整数转换。
6	ostr << oct、istr >> oct	以 8 为基数进行整数转换。

表 14–2 iostream 的预定义操纵符 （续）

	预定义的操纵符	说明
7	istr >> ws	提取空白字符（跳过空白），直至找到非空白字符（留在 istr 中）。
8	ostr << setbase(n)、istr >> setbase(n)	将转换基数设置为 n（仅限 0、8、10、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 中设置的位，清除标志位向量。
11	ostr << setiosflags(i)、istr >> setiosflags(i)	根据 i 中设置的位，设置标志位向量。
12	ostr << setfill(c)、istr >> setfill(c)	将填充字符（用来填充字段）设置为 c。
13	ostr << setprecision(n)、istr >> setprecision(n)	将浮点精度设置为 n 位数。

要使用预定义的操纵符，必须在程序中包含文件 iomanip.h。

您可以定义自己的操纵符。操纵符共有两个基本类型：

- 无格式操纵符－采用 istream&、ostream& 或 ios& 参数，对流进行操作，然后返回其参数。
- 参数化操纵符－采用 istream&、ostream& 或 ios& 参数以及一个附加参数，对流进行操作，然后返回其流参数。

14.7.1 使用无格式操纵符

无格式操纵符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
- 返回操纵符的参数

由于为 iostream 预定义了采用（指向）此类函数（的指针）的移位运算符，因此可以在输入或输出运算符序列中放入函数。移位运算符会调用函数而不是尝试读取或写入值。例如，下面是将 tab 插入 ostream 的 tab 操纵符：

```
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。假设要对输入流打开或关闭空白跳过功能。可以分别调用 `ios::setf` 和 `ios::unsetf` 来打开和关闭 `skipws` 标志，也可以定义两个操纵符。

```
#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 参数化操纵符

`iomanip.h` 中包含的其中一个参数化操纵符是 `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：

```
//file setfill.cc
#include<iostream.h>
#include<iomanip.h>

//the private manipulator
static ios& sfill(ios& i, int f) {
         i.fill(f);
         return i;
}
//the public applicator
```

```
smanip_int setfill(int f) {
       return smanip_int(sfill, f);
}
```

参数化操纵符的实现分为两部分：

- **操纵符**。它使用一个额外的参数。在上面的代码示例中，采用了额外的 int 参数。由于未给这个操纵符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*。它调用该操纵符。applicator 是全局函数，您会为它生成在头文件中可用的原型。通常操纵符是文件中的静态函数，该文件包含了 applicator 的源代码。只有 applicator 可以调用该操纵符，如果您将操纵符设置为静态，就要使操纵符名称始终位于全局地址空间之外。

头文件 iomanip.h 中定义了多个类。每个类都保存一个操纵符函数的地址和一个参数的值。manip(3CC4) 手册页中介绍了 iomanip 类。上面的示例使用了 smanip_int 类，它是与 ios 一起使用。因为该类与 ios 一起使用，所以也可以与 istream 和 ostream 一起使用。上面的示例还使用了另一个类型为 int 的参数。

applicator 创建并返回类对象。在上面的代码示例中，类对象是 smanip_int，其中包含了操纵符和 applicator 的 int 参数。iomanip.h 头文件定义了用于该类的移位运算符。如果 applicator 函数 setfill 在输入或输出操作序列中，会调用该 applicator 函数，且其返回一个类。移位运算符作用于该类，以使用其参数值（存储在类中）调用操纵符函数。

在以下示例中，操纵符 print_hex：

- 将输出流设置成十六进制模式。
- 将 long 值插入流中。
- 恢复流的转换模式。

使用类 omanip_long 的原因是该代码示例仅用于输出，而且操作对象是 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 用于数组的 strstream: iostream

请参见 strstream(3CC4) 手册页。

14.9 用于 stdio 文件的 stdiobuf: iostream

请参见 stdiobuf(3CC4) 手册页。

14.10 streambuf

iostream 是由两部分（输入或输出）构成的系统的格式化部分。系统的其他部分由处理无格式字符流的输入或输出的 streambuf 组成。

通常，可以通过 iostream 使用 streambuf，因此，不必担心 streambuf 的细节。您可以选择直接使用 streambuf，例如，如果需要提高效率或解决 iostream 内置的处理或格式化错误。

14.10.1 streambuf 工作方式

streambuf 由字符流或字符序列和一个或两个指向相应序列的指针组成。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。）有两种 streambuf 指针：

- *put* 指针，它指向下一个字符的存储位置前面
- *get* 指针，它指向要获取的下一个字符前面

streambuf 可以有其中一个指针，也可以两个全有。

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。操作两个指针时它们是否都会移动取决于使用 streambuf 种类。通常，如果使用队列式 streambuf，get 和 put 指针独立移动；如果使用文件式 streambuf，get 和 put 指针总是一起移动。例如，strstream 是队列式流，fstream 是文件式流。

14.10.2 使用 streambuf

从来不创建实际的 streambuf 对象，而是只创建从 streambuf 类派生的类的对象。例如 filebuf 和 strstreambuf，filebuf(3CC4) 手册页和 ssbuf(3) 手册页中分别对它们进行了介绍。高级用户可能想从 streambuf 派生自己的类，以便提供特定设备的接口或提供基本缓冲以外的功能。sbufpub(3CC4) 和 sbufprot(3CC4) 手册页中讨论了如何执行此操作。

除了创建自己的特殊种类的 streambuf 外，您可能还想通过访问与 iostream 关联的 streambuf 来访问公用成员函数（如上面引用的手册页中所述）。此外，每个 iostream 都有采用 streambuf 指针的已定义插入器和提取器。插入或提取 streambuf 时，会复制整个流。

下面是另一种文件复制（前面讨论过）方法，这里为了清晰起见省略了错误检查：

```
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 类都有成员函数 rdbuf，它返回指向与其关联的 streambuf 对象的指针。如果是 fstream，则 streambuf 对象是类型 filebuf。与 fromFile 关联的整个文件都复制到（插入）与 toFile 关联的文件。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++ 手册页都介绍了 iostream 库的详细信息。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	详细介绍了从 streambuf 派生并专用于文件的类 filebuf 的公用接口。有关从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 和 sbufprot(3CC4) 手册页。可通过类 fstream 使用 filebuf 类。
fstream	详细介绍了类 ifstream、ofstream 和 fstream 的专用成员函数，这些类是用于文件的 istream、ostream 和 iostream 专用版本。
ios	详细介绍了作为 iostream 的基类的类 ios 的各个部分。该类也包含了所有流公共的状态数据。
ios.intro	简要介绍了 iostream。

表 14–3 iostream 手册页概述 （续）

手册页	概述
istream	详细说明了以下内容： ■ 类 istream 的成员函数，这些函数支持对从 streambuf 获取的字符进行解释 ■ 输入格式化 ■ 归为类 ostream 的一部分的定位函数 ■ 某些相关函数 ■ 相关操纵符
manip	介绍了 iostream 库中定义的输入和输出操纵符。
ostream	详细说明了以下内容： ■ 类 ostream 的成员函数，这些函数支持对写入 streambuf 的字符进行解释 ■ 输出格式化 ■ 归为类 ostream 的一部分的定位函数 ■ 某些相关函数 ■ 相关操纵符
sbufprot	介绍了对从类 streambuf 派生的类进行编码的程序员所需的接口。有关 sbufprot(3CC4) 手册页中未讨论的一些公用函数，另请参见 sbufpub(3CC4) 手册页。
sbufpub	详细介绍了 streambuf 类的公用接口，尤其是 streambuf 的公用成员函数。该手册页包含了直接处理 streambuf 类型的对象所需的信息，或是查找从 streambuf 派生的类从其继承的函数所需的信息。如果要从 streambuf 派生类，另请参见 sbufprot(3CC4) 手册页。
ssbuf	详细介绍了从 streambuf 派生并专用于处理字符数组的类 strstreambuf 的专用公用接口。有关从类 streambuf 继承的功能的详细信息，请参见 sbufpub(3CC4) 手册页。
stdiobuf	简要介绍了从 streambuf 派生并专用于处理 stdio 文件的 stdiobuf 类。有关从 streambuf 类继承的功能的详细信息，请参见 sbufpub(3CC4) 手册页。
strstream	详细介绍了由从 iostream 派生并专用于处理字符数组的一组类实现的 strstream 的专用成员函数。

14.12 iostream 术语

iostream 库说明中常常使用一些与一般编程中的术语类似的术语，但有特殊含义。下表定义了讨论 iostream 库时使用的这些术语。

表 14–4 iostream 术语

iostream 术语	定义
缓冲区	该词有两个含义，一个特定于 iostream 软件包，另一个较常适用于输入和输出。 与 iostream 库特定相关时，缓冲区是由类 streambuf 定义的类型的对象。 通常，缓冲区是一个内存块，用于将字符高效传输到输出的输入。对于已缓冲的 I/O，缓冲区已满或被强制刷新之前，字符的实际传输会延迟。 无缓冲的缓冲区是指在其中没有上文定义的通用意义的缓冲区的 streambuf。本章避免使用缓冲区一词来指 streambuf。但是，手册页和其他 C++ 文档使用缓冲区一词来表示 streambuf。
提取	从 iostream 获取输入的过程。
Fstream	专用于文件的输入或输出流。特指以 courier 字体打印时从类 iostream 派生的类。
插入	将输出发送到 iostream 中的过程。
iostream	通常为输入或输出流。
iostream 库	通过 include 文件 iostream.h、fstream.h、strstream.h、iomanip.h 和 stdiostream.h 实现的库。因为 iostream 是面向对象的库，所以应扩展该库。因此，可以对 iostream 库执行的某些操作并未实现。
流	通常是指 iostream、fstream、strstream 或用户定义的流。
Streambuf	包含字符序列的缓冲区，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，通常是指 streambuf 类或从 streambuf 派生的类的对象。任何流对象都包含从 streambuf 派生的类型的对象或指向对象的指针。
Strstream	专用于字符数组的 iostream。它是指以 courier 字体打印时的特定类。

◆ ◆ ◆ 第 15 章

15 使用复数运算库

复数是由**实**部和**虚**部组成的数。例如：

```
3.2 + 4i
1 + 3i
1 + 2.3i
```

在特例情况下，如 `0 + 3i` 是纯虚数，通常写为 `3i`；`5 + 0i` 是纯实数，通常写为 `5`。可以使用 `complex` 数据类型来表示复数。

注 – 复数运算库 (`libcomplex`) 仅可用于兼容模式 (`-compat[=4]`)。在标准模式（缺省模式）下，C++ 标准库 `libCstd` 附带具有类似功能的复数类。

15.1 复数库

复数运算库以新的数据类型实现复数数据类型，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

复数也可表示为**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）表示形式和幅度角度（极）表示形式之间进行转换的函数。

数字**复共轭**的虚部中有相反符号。

15.1.1 使用复数库

要使用复数库，应在程序中包含头文件 `complex.h`，并使用 `-library=complex` 选项进行编译和链接。

15.2 complex 类型

复数运算库定义了一个类： complex 类。complex 类的对象可以存放一个复数。复数由两部分构成：

- 实部
- 虚部

```
class complex {
    double re, im;
};
```

complex 类的对象值是一对 double 值。第一个值表示实部，第二个值表示虚部。

15.2.1 complex 类的构造函数

有两个用于 complex 的构造函数。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;}
```

如果声明复数变量时没有指定参数，则会使用第一个构造函数并初始化变量，因此两个部分都为 0。以下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。如果只给定一个参数，该参数将作为实部的值，而虚部的值设置为 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
```

也可以使用复数运算库中提供的 polar 函数（请参见第 183 页中的 “15.3 数学函数”）创建复数。polar 函数根据给定的极坐标幅度和角度创建复数值。

没有用于 complex 类型的析构函数。

15.2.2 算术运算符

复数运算库定义了所有基本的算术运算符。具体来说，以下运算符按一般方法和普通的优先级工作：

```
+ - / * =
```

减法运算符 (-) 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) {...}; // illegal
b = a *= b; // illegal
```

另外还可以使用等号 (==) 和不等号 (!=)，它们具有常规含义。

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。如果函数无法生成具有数学意义的结果，它就调用 `complex_error` 并返回适用的某值。具体来说，这些函数会尽量避免实际的溢出，而是调用 `complex_error` 并显示消息。下表描述了复数运算库函数的提示。

注 – `sqrt` 和 `atan2` 函数的实现遵循 C99 csqrt 附录 G 规范。

表 15–1 复数运算库函数

复数运算库函数	说明
`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。比 `abs` 快，但较易产生溢出。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	说明
`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` 为自然对数的基数，`x` 是为 `exp` 提供的参数。
`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。

`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* `)`.它计算出 *b* 的 *exp* 次幂。

`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。

表 15–2 复数数学函数和三角函数 （续）

复数运算库函数	说明
`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

复数库具有以下用于错误处理的定义：

```
extern int errno;
class c_exception {...};
int complex_error(c_exception&);
```

外部变量 `errno` 是来自 C 库的全局错误状态。`errno` 可以为标准头文件 `errno.h` 中所列值（请参见 `perror`(3) 手册页）。没有任何函数会将 `errno` 设置为零，但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. 在运算前将 `errno` 设置为零。
2. 测试运算。

函数 `complex_error` 采用对类型 `c_exception` 的引用并由下列复数运算库函数调用：

- `exp`
- `log`
- `log10`
- `sinh`
- `cosh`

缺省版本的 `complex_error` 返回零。这个零值的返回意味着发生了缺省的错误处理。可以提供自己的替换函数 `complex_error`，以执行其他错误处理。`cplxerr`(3CC4) 手册页中介绍了错误处理。

`cplxtrig`(3CC4) 和 `cplxexp`(3CC4) 手册页中介绍了缺省的错误处理，下表中也进行了简要介绍。

复数运算库函数	缺省错误处理汇总
`exp`	如果产生溢出，将 `errno` 设置为 `ERANGE`，并返回一个极大的复数。
`log`、`log10`	如果参数为零，将 `errno` 设置为 `EDOM`，并返回一个极大的复数。

复数运算库函数	缺省错误处理汇总
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

复数运算库提供了用于复数的缺省**提取器**和**插入器**，如以下示例所示：

```
ostream& operator<<(ostream&, const complex&); //inserter
istream& operator>>(istream&, complex&); //extractor
```

有关提取器和插入器的基本信息，请参见第 162 页中的 “14.2 iostream 交互的基本结构”和第 163 页中的 “14.3.1 使用 iostream 进行输出”。

对于输入，复数提取器 >> 从输入流中提取一对数（用圆括号括住，并由逗号分隔开），并将其读入复数对象。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，给定声明和输入语句：

```
complex x;
cin >> x;
```

以及输入 (3.45, 5)，则 x 值等于 3.45 + 5.0i。对插入器来讲反向为真。如果给定 complex x(3.45, 5)，cout<<x 将打印 (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），那么提取器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。

插入器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。这两个值都视为 double。

15.6 混合模式运算

类型 complex 专门用于处理混合模式表达式中的内置运算类型。运算类型会缺省转换为 complex 类型，而且算术运算符和大多数数学函数都有 complex 版本。例如：

```
int i, j;
double x, y;
complex a, b;
a = sin((b+i)/y) + x/j;
```

表达式 b+i 是混合模式。整数 i 通过构造函数 complex::complex(double,double=0) 转换为类型 complex（整型数先是转换为类型 double）。所得结果除以 double 类型的

y，因此 y 也转换为 complex 且使用了复数除法运算。这样，得到的商是 complex 类型，因此调用复数正弦例程，从而生成另一个 complex 结果等。

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; // OK
a != b; // OK
a <  b; // error: operator < cannot be applied to type complex
a >= b; // error: operator >= cannot be applied to type complex
```

同样，由于未明确定义概念，因此不会自动将类型 complex 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)); // OK
f(a);      // error: no match for f(complex)
```

15.7 效率

设计 complex 类主要为了提高效率。

最简单的函数声明为 inline，以消除函数调用开销。

在函数不同时就会提供函数的多个开销版本。例如，pow 函数有多个版本，分别取类型为 double 和 int 以及 complex 的指数，而前者的运算简单得多。

在包含 complex.h 时，会自动包含标准 C 数学库头文件 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);
```

在此示例中，调用了标准数学函数 sqrt(double)，且结果转换为 complex 类型，而不是先转换为 complex 类型再调用 sqrt(complex)。该结果转向重载决策规则的外部，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 15–3 有关 complex 类型的手册页

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)	算术运算符函数
cplxtrig(3CC4)	三角函数

◆ ◆ ◆ 第 16 章

16 生成库

本章解释了如何生成您自己的库。

16.1 认识库

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，因此减轻程序员在其他部分工作的负担。

生成库只不过是创建 `.o` 文件（使用 `-c` 选项编译代码）并使用 `CC` 命令将 `.o` 文件并入库中。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，库中的对象在链接时链接到程序的可执行文件中。只有库中属于应用程序所需的那些 `.o` 文件链接到可执行文件。静态（归档）库名称通常以 `.a` 后缀结尾。

对于动态（共享）库，库中的对象并不链接到程序的可执行文件，而是链接程序在可执行文件中注明程序依赖于库。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）库名称以 `.so` 后缀结尾。

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。）
- 共享库可以在运行时通过使用 `dlopen()` 函数调用来装入。

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 可以帮助克服此问题。）

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。可以使用 CC 的 –xar 选项将目标 (.o) 文件集合并入单个库中。

可以使用 CC -xar 而非直接使用 ar 命令来生成静态（归档）库。C++ 语言通常要求编译器维护的信息比传统 .o 文件提供的信息多，尤其是模板实例。–xar 选项可确保所有必要信息（包括模板实例）都包括在库中。在通常的编程环境下，可能无法完成该操作，因为 make 无法确定实际创建和引用了哪些模板文件。如果没有 CC -xar，引用的模板实例可能没有按照需要包括在库中。例如：

```
% CC -c foo.cc # Compile main file, templates objects are created.
% CC -xar -o foo.a foo.o # Gather all objects into a library.
```

–xar 标志会使 CC 创建静态（归档）库。要为新建的库命名，需要使用 –o 指令。编译器检查命令行上的目标文件，交叉引用这些目标文件与模板系统信息库中的目标文件，并将用户的目标文件所需的模板（以及主目标文件本身）添加到归档文件中。

注 – 仅可将 -xar 标志用于创建或更新现有归档文件。不要用它来维护归档。-xar 选项与 ar -cr 等效。

最好每个 .o 文件中只有一个函数。如果要链接归档文件，则在需要该归档文件中的特定 .o 文件中的符号时，整个 .o 文件都链接到应用程序中。每个 .o 文件中有一个函数可以确保将只从归档文件链接应用程序所需的那些符号。

16.3 生成动态（共享）库

动态（共享）库的生成方式与静态（归档）库的生成方式基本相同，除了在命令行上使用 –G 而不是 –xar。

不应直接使用 ld。与静态库一样，CC 命令可以确保使用模板时，模板系统信息库中所有必要的模板实例都包括在库中。在执行 main() **之前**会调用与应用程序链接的动态库中所有静态构造函数，在 main() 退出**之后**会调用所有静态析构函数。如果使用 dlopen() 打开共享库，所有静态构造函数都在执行 dlopen() 时执行，所有静态析构函数都在执行 dlclose() 时执行。

应该使用 CC -G 来生成动态库。使用 ld（链接编辑器）或 cc（C 编译器）生成动态库时，异常可能无法生效，且库中定义的全局变量未初始化。

要生成动态（共享）库，必须使用 CC 的 –Kpic 或–KPIC 选项编译每个对象来创建可重定位的目标文件。然后您就可以生成一个具有这些可重定位目标文件的动态库。如果遇到异常的链接失败，可能是忘记了使用 –Kpic 或 –KPIC 编译某些对象。

要生成名为 libfoo.so 的 C++ 动态库（该库包含源文件 lsrc1.cc 和 lsrc2.cc 中的对象），请键入：

```
% CC -G -o libfoo.so -h libfoo.so -Kpic lsrc1.cc lsrc2.cc
```

-G 选项指定动态库的构造。-o 选项指定库的文件名。-h 选项指定共享库的内部名称。-Kpic 选项指定目标文件与位置无关。

CC -G 命令不会将任何 -l 选项传递给链接程序 ld。为了确保正确的初始化顺序，共享库对其所需的每个其他共享库必须具有显式的依赖性。要创建依赖性，请对每个此类库使用 -l 选项。典型的 C++ 共享库将使用以下几组选项之一：

```
-lCstd -lCrun -lc
-library=stlport4 -lCrun -lc
```

为了确保列出了需要的所有依赖性，请使用 -zdefs 选项生成库。对于缺少的每个符号定义，链接程序都会发出错误消息。要提供缺少的定义，请针对这些库添加 -l 选项。

要确定是否包含了不需要的依赖性，请使用以下命令

```
ldd -u -r mylib.so
ldd -U -r mylib.so
```

然后可以重新生成没有不需要的依赖性的 mylib.so。

16.4 生成包含异常的共享库

对于包含 C++ 代码的程序，切勿使用 -Bsymbolic，而应使用链接程序映射文件。如果使用 -Bsymbolic，不同模块中的引用会绑定到应是一个全局对象内容的不同副本。

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。

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。具体来说，库不需要符合系统的应用程序二进制接口 (application binary interface, ABI)。例如，可以使用 -fast 选项编译库，以提高其在某已知体系结构上的性能。同样，可以使用 -xregs=float 选项编译库以提高性能。

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程序员是否可以轻松地重新编译该库。生成公用库时应该符合系统的应用程序二进制接口(application binary interface, ABI)。通常，这意味着应该避免任何特定于处理器的选项。（例如，不使用 `–fast` 或 `–xtarget`。）

SPARC ABI 为应用程序保留了一些专用寄存器。对于 V7 和 V8，这些寄存器是 `%g2`、`%g3` 和 `%g4`。对于 V9，这些寄存器是 `%g2` 和 `%g3`。由于多数编译用于应用程序，所以在缺省情况下，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。生成公用库时，请使用 `-xregs=no%appl` 选项编译所有对象，以确保不会使用应用程序寄存器。

16.7 生成具有 C API 的库

如果要生成用 C++ 编写且可用于 C 程序的库，必须创建 C API（application programming interface，应用程序编程接口）。为此，应先使所有导出的函数为 `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 接口库需要 C++ 运行时支持，且要使用 `cc` 进行链接，则在使用 C 接口库时，还必须用 `libC`（兼容模式）或 `libCrun`（标准模式）链接应用程序。（如果 C 接口库不需要 C++ 运行时支持，就不必用 `libC` 或 `libCrun` 进行链接。）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说明。

- 如果 C 接口库是在**标准模式**（缺省模式）下使用 `CC` 生成的，那么在使用该 C 接口库时，将 `-lCrun` 添加到 `cc` 命令行。
- 如果 C 接口库是在**兼容模式** (`-compat`) 下使用 `CC` 生成的，那么在使用该 C 接口库时，将 `-lC` 添加到 `cc` 命令行。

提供**共享**的 C 接口库时，必须在生成库时创建对 `libC` 或 `libCrun` 的依赖性。如果共享库具有正确的依赖性，就不必在使用该库时将 `-lC` 或 `-lCrun` 添加到命令行。

- 如果要在**兼容模式** (`-compat`) 下生成 C 接口库，应在生成库时将 `-lC` 添加到 `CC` 命令行。
- 如果要在**标准模式**（缺省模式）下生成 C 接口库，应在生成库时将 `-lCrun` 添加到 `CC` 命令行。

如果要删除对 C++ 运行时库的任何依赖性，应该在库源文件中强制应用下列代码规则：

- 不要使用任何形式的 `new` 或 `delete`，除非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

如果要使用 `dlopen()` 从 C 程序打开 C++ 共享库，应确保共享库依赖于适当的 C++ 运行时（对于 `-compat=4`，为 `libC.so.5`；对于 `-compat=5`，为 `libCrun.so.1`）。

为此，应在生成共享库时，将 `-lC`（对于 `-compat=4`）或 `lCrun`（对于 `-compat=5`）添加到命令行。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++ 运行库的依赖性，则 C 程序可能会出现无规律的行为。

第 4 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。介绍的功能适用于除了特别注明之外的所有平台；基于 SPARC 的系统上的 Solaris OS 特有的功能标识为 *SPARC*，基于 x86 的系统上的 Solaris 和 Linux OS 特有的功能标识为 *x86*。仅限于 Solaris OS 的功能标记了 *Solaris*；仅限于 Linux OS 的功能标记了 *Linux*。请注意，提及 Solaris OS 时也意指 OpenSolaris OS。

本手册的此部分使用前言中列出的印刷约定来说明各个选项。

圆括号、大括号、方括号、"|" 或 "-" 字符以及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，就请参阅其他选项的说明以获取完整的详细信息。

表 A–1 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。
缺省值	如果选项具有主缺省值或辅助缺省值，则在此处进行声明。 如果未指定选项，则主缺省值为有效选项值。例如，如果未指定 `–compat`，则缺省值为 `–compat=5`。 如果指定了选项但不给定任何值，则辅助缺省值为有效选项值。例如，如果指定了 `–compat` 但未提供值，则缺省值为 `-compat=4`。

表 A–1 选项子节 （续）

子节	内容
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用途，则在此处引用首选项。例如，“与 `-xO` 相同”表示 `-xO` 是首选项。

A.2 选项参考

本节按字母顺序列出所有的 C++ 编译器选项，并指出所有的平台限制。

A.2.1 –386

x86： 与 `–xtarget=386` 相同。**提供该选项只是为了向后兼容。**

A.2.2 –486

x86： 与 `–xtarget=486` 相同。**提供该选项只是为了向后兼容。**

A.2.3 –a

与 `–xa` 相同。

A.2.4 –B*binding*

指定 链接的库绑定是`symbolic`、`dynamic`（共享）还是 `static`（非共享）。

可以在命令行上多次使用 `–B` 选项。该选项传递给链接程序 `ld`。

注 – 在 Solaris 64 位编译环境中，许多系统库只能用作动态库。因此，请勿在命令行上将 `-Bstatic` 用作最后一个切换开关。

A.2.4.1 值

binding 必须是下列值之一：

值	含义
`dynamic`	指示链接编辑器查找 `lib`*lib*`.so`（共享）文件，如果未找到这些文件，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`	指示链接编辑器只查找 `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。
`symbolic`	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。

（`–B` 和 *binding* 值之间不能有空格。）

缺省值

如果没有指定 `-B`，则使用 `–Bdynamic`。

交互

要静态链接 C++ 缺省库，请使用 `–staticlib` 选项。

`-Bstatic` 和 `-Bdynamic` 选项会影响缺省情况下提供的库的链接。为了确保动态链接缺省库，最后使用的 `–B` 应该是 `–Bdynamic`。

在 64 位环境中，许多系统库只能用作共享动态库。其中包括 `libm.so` 和 `libc.so`（不提供 `libm.a` 和 `libc.a`）。因此，在 64 位 Solaris 操作系统中，`-Bstatic` 和 `-dn` 可能会导致产生链接错误。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 `lib`*foo*`.a`，即使 `lib`*foo*`.so` 存在也是如此，所有其他库都是动态链接的：

```
example% CC a.o –Bstatic –lfoo –Bdynamic
```

警告

对于包含 C++ 代码的程序，切勿使用 `-Bsymbolic`，而应使用链接程序映射文件。

如果使用 `-Bsymbolic`，不同模块中的引用会绑定到应是一个全局对象内容的不同副本。

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。

如果在不同的步骤中进行编译和链接，并要使用 `-B`*binding* 选项，就必须在链接步骤中包括该选项。

另请参见

`–nolib`、`–staticlib`、`ld(1)`、第 138 页中的 “12.5 静态链接标准库”和《链接程序和库指南》

A.2.5 –c

仅编译；生成 `.o` 目标文件，但抑制链接。

该选项指示 `CC` 驱动程序抑制通过 `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` 选项显式指定目标文件。

A.2.5.1 示例

如果输入 **`CC -c x.cc`**，则会生成 `x.o` 目标文件。

如果输入 **`CC -c x.cc -o y.o`**，则会生成 `y.o` 目标文件。

警告

当编译器为输入文件 (`.c`、`.i`) 生成目标代码时，编译器总是在工作目录下生成 `.o` 文件。如果抑制链接步骤，则不会删除 `.o` 文件。

另请参见

`–o` *filename* 和 `–xe`

A.2.6 –cg{89|92}

与 `–xcg{89|92}` 相同。

A.2.7 –compat[={4|5}]

设置编译器的主要发行版兼容模式。此选项控制 __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 标准在名称损坏、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 –compat 选项进行区分，下文中介绍了相应值。

A.2.7.1 值

-compat 选项可以有下列值。

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 预处理程序宏和 __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 预处理程序宏和 __SUNPRO_CC_COMPAT 预处理程序宏分别设置为 199711L 和 5。

缺省值

如果没有指定 –compat 选项，则假定 –compat=5。

如果仅指定 –compat，则假定 –compat=4。

交互

在兼容模式 (-compat[=4]) 下，不能使用标准库。

–compat[=4] 不能与下列任何选项一起使用。

- -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

–compat=5 不能与下列任何选项一起使用。

- -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 -Bsymbolic。

另请参见

《C++ 迁移指南》

A.2.8 +d

请勿扩展 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。

- 该函数的定义中使用了关键字 inline
- 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
- 已选定 -g 选项

A.2.8.1 示例

缺省情况下，编译器可以内联以下代码示例中的函数 f() 和 memf2()。此外，该类具有编译器可以内联的由编译器生成的缺省构造函数和析构函数。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;} // may be inlined
class C {
  int mf1(); // not inlined unless inline definition comes later
  int mf2() {return 0;} // may be inlined
};
```

交互

指定了调试选项 –g 时，会自动启用该选项。

但指定调试选项 –g0 不会启用 +d。

+d 选项对使用 -xO4 或 -xO5 时执行的自动内联没有影响。

另请参见

–g0 和 –g

A.2.9 -D*name*[=*def*]

为预处理程序定义宏符号 *name*。

使用该选项与在源文件开头包含 #define 指令等效。可以使用多个 -D 选项。

A.2.9.1 值

下表显示了预定义的宏。可以在诸如 #ifdef 之类的预处理程序条件下使用这些值。

表 A–2 预定义的宏

平台	宏名称	说明
SPARC 和 x86	_ _ARRAYNEW	如果启用了“数组”形式的运算符 new 和 delete，则定义 _ _ARRAYNEW。有关更多信息，请参见 -features=[no%]arraynew。
	_BUILTIN_VA_ARG_INCR	适用于 varargs.h、stdarg.h 和 sys/varargs.h 中的 _ _builtin_alloca、_ _builtin_va_alist 和 _ _builtin_va_arg_incr 关键字。
	_ _DATE_ _	
	_ _FILE_ _	
	__LINE__	
	_ _STDC_ _	设置为 0（零）
	__SUNPRO_CC=0x510	_ _SUNPRO_CC 值代表编译器的发行版本号。
	__SUNPRO_CC_COMPAT=4 或 __SUNPRO_CC_COMPAT=5	请参见第 201 页中的 “A.2.7 –compat[={4|5}]”
	_ _TIME_ _	

表 A–2 预定义的宏 （续）

平台	宏名称	说明
	_ _cplusplus	
	__‘uname -s‘_‘uname -r | tr . _‘	其中，*uname -s* 是 uname –s 的输出，*uname -r* 是 uname -r 的输出，且无效字符（如句点 (.)）替换为下划线，如 -D__SunOS_5_9 和 -D__SunOS_5_10 所示。
	_ _unix	
	_BOOL	如果启用了 bool 类型，则定义 _BOOL。有关更多信息，请参见 -features=[no%]bool。
	_WCHAR_T	
	unix	请参见**交互**。
SPARC	__SUN_PREFETCH=1	
	__SunOS_*OSversion* _*OSversion*	
	_ _SVR4	
	__sparc	
	_ _sun	
	sparc	请参见**交互**。
	sun	请参见**交互**。
SPARC v9	_ _sparcv9	只限于 64 位编译模式
x86	i386	
	linux	
	_ _amd64	
	_ _gnu_ _linux_ _	
	_ _i386	请参见**交互**。
	_ _ linux	
	_ _ linux_ _	
	_ _x86 _ 64	

如果不使用 *=def*，则 *name* 定义为 1。

交互

如果使用 +p，则不会定义 sun、unix、sparc 和 i386。

另请参见

–U

A.2.10 –d{y|n}

允许或不允许将动态库用于整个可执行文件。

该选项传递给 ld。

该选项只能在命令行出现一次。

A.2.10.1 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。

缺省值

如果没有指定 -d 选项，则使用 –dy。

交互

在 64 位环境中，许多系统库只能用作共享动态库。其中包括 libm.so 和 libc.so（不提供 libm.a 和 libc.a）。因此，在 64 位 Solaris 操作系统中，-Bstatic 和 -dn 可能会导致产生链接错误。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可用。

另请参见

ld(1) 和《链接程序和库指南》。

A.2.11 –dalign

(SPARC) -dalign 与 -xmemalign=8s 等效。有关更多信息，请参见第 294 页中的 “A.2.151 -xmemalign=*ab*”。

在 x86 平台上，默认忽略此选项。

A.2.11.1 警告

如果使用 –dalign 编译一个程序单元，就要使用 -dalign 编译程序的所有单元，否则可能会产生意外的结果。

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 驱动程序仅对 C++ 源文件运行预处理程序，并将结果发送到 stdout（标准输出）。此时，不进行编译，且不生成 .o 文件。

此选项会导致输出中包含预处理程序类型的行号信息。

A.2.13.1 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 会生成第 206 页中的 “A.2.13.1 示例”中所示的输出。

示例 A–1 预处理程序示例 foo.cc

```
#if __cplusplus < 199711L
int power(int, int);
#else
template <> int power(int, int);
#endif

int main () {
  int x;
  x=power(2, 10);
}
.
```

示例 A–2 使用 -E 选项时 foo.cc 的预处理程序输出

```
example% CC -E foo.cc
#4 "foo.cc"
template < > int power (int, int);
```

示例 A–2 使用 -E 选项时 foo.cc 的预处理程序输出 （续）

```
int main () {
int x;
x = power (2, 10);
}
```

警告

如果代码包含采用“独立定义”模型的模板，此选项的输出可能不能用作 C++ 编译的输入。

另请参见

–P

A.2.14 +e{0|1}

控制兼容模式 (-compat[=4]) 下虚拟表的生成。在标准模式下（缺省模式）无效并被忽略。

A.2.14.1 值

+e 选项可以有下列值。

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*]

此命令会抑制 C++ 编译器警告消息，但对错误消息没有影响。此选项适用于所有警告消息，无论这些警告消息是否已被 -errwarn 指定为导致非零退出状态。

A.2.15.1 值

t 是一个逗号分隔列表，它包含以下项中的一项或多项：*tag*、no%*tag*、%all、%none。顺序很重要；例如 %all,no%*tag* 抑制除 *tag* 以外的所有警告消息。下表列出了 -erroff 值：

表 A–3 -erroff 值

值	含义
tag	抑制由该 *tag* 指定的警告消息。可通过 -errtags=yes 选项来显示消息的标记。
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（缺省）。

缺省值

缺省值为 -erroff=%none。指定 -erroff 与指定 -erroff=%all 等效。

示例

例如，-erroff=*tag* 将抑制由该标记指定的警告消息。另外，-erroff=%all,no%*tag* 抑制除由 *tag* 标识的消息以外的所有警告消息。

可以使用 -errtags=yes 选项显示警告消息的标记。

警告

使用 -erroff 选项只能抑制来自 C++ 编译器前端且在使用 -errtags 选项时显示标记的警告消息。

另请参见

-errtags 和 -errwarn。

A.2.16 -errtags[= *a*]

显示来自 C++ 编译器前端且可以使用 -erroff 选项抑制或使用 -errwarn 选项使其成为致命警告的每个警告消息的消息标记。

A.2.16.1 值和缺省

a 可以是 yes 或 no。缺省值为 -errtags=no。指定 -errtags 与指定 -errtags=yes 等效。

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，因此不能使用 -erroff 抑制，也不能使用 -errwarn 使其成为致命消息。

另请参见

-erroff 和 -errwarn

A.2.17 -errwarn[= *t*]

使用 -errwarn 会导致 C++ 编译器在出现给定的警告消息时以失败状态退出。

A.2.17.1 值

t 是一个逗号分隔列表，它包含以下项中的一项或多项：*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* 以致命状态退出。

下表详细列出了 -errwarn 值：

表 A–4 -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 以致命状态退出。如果未出现 *tag*，则没有影响。
no%*tag*	防止 cc 在由 *tag* 指定的消息只以警告消息形式出现时以致命状态退出。如果未发出 *tag* 指定的消息，则不会产生任何影响。为了避免在发出警告消息时导致 *cc* 以致命状态退出，可使用该选项来还原以前用该选项和 *tag* 或 %all 指定的警告消息。
%all	使 cc 在出现任何警告消息时以致命状态退出。%all 可以后跟 no%*tag*，以避免该行为的特定警告消息。
%none	防止 cc 在出现任何警告消息时以致命状态退出。

缺省值

缺省值为 -errwarn=%none。如果单独指定 -errwarn，它与 -errwarn=%all 等效。

警告

使用 -errwarn 选项只能对来自 C++ 编译器前端且在使用了 -errtags 选项时会显示标记的警告消息进行指定，从而使编译器以失败状态退出。

由于编译器错误检查的改善和功能的增加，C++ 编译器生成的警告消息也会因发行版本而异。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 和 -xwe

A.2.18 –fast

此选项是一个宏，可以有效地用作优化可执行文件的起点，从而获得最佳运行时性能。-fast 是一个宏，随编译器发行版本的升级而变化，并扩展为目标平台特定的多个选项。可使用 -dryrun 或 -xdryrun 选项检查 -fast 的扩展，并将 -fast 的相应选项结合到正在进行的可执行文件调优过程中。

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。

A.2.18.1 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–5 -fast 扩展

选项	SPARC	x86
–fns	X	X
–fsimple=2	X	X
–nofstore	-	X
-xarch	X	X
-xbuiltin=%all	X	X
-xcache	X	X
-xchip	X	X
–xlibmil	X	X
–xlibmopt	X	X

表 A–5 -fast 扩展 （续）

选项	SPARC	x86
–xmemalign	X	-
–xO5	X	X
-xregs=frameptr	-	X
–xtarget=native	X	X

交互

-fast 宏可扩展为可能影响其他指定选项的编译选项。例如，在以下命令中，-fast 宏的扩展包括了将 -xarch 还原为某个 32 位体系结构选项的 -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（请参阅示例）。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，即该选项禁用所有陷阱操作。

示例

执行以下编译器命令，优化级别将为 –xO3。

```
example% CC –fast –xO3
```

执行以下编译器命令，优化级别将为 –xO5。

```
example% CC -xO3 –fast
```

警告

如果在不同的步骤中进行编译和链接，则编译命令和链接命令中都必须有 -fast 选项。

使用 -fast 选项编译的目标二进制文件不可移植。例如，在 UltraSPARCIII 系统中用以下命令生成的二进制文件在 UltraSPARCII 系统中无法执行。

```
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。

在早期的 SPARC 发行版中，`-fast` 宏扩展到了 `-fsimple=1`。而现在扩展到 `-fsimple=2`。

`-fast` 的扩展包括 `-D_MATHERR_ERRNO_DONTCARE`。

要在任何平台上显示 `–fast` 的扩展，请运行命令 **`CC –dryrun –fast`**。

```
>CC -dryrun -fast
###     command line files and options (expanded):
### -dryrun -xO5 -xarch=sparcvis2 -xcache=64/32/4:1024/64/4 \
-xchip=ultra3i -xmemalign=8s -fsimple=2 -fns=yes -ftrap=%none \
-xlibmil -xlibmopt -xbuiltin=%all -D__MATHERR_ERRNO_DONTCARE
```

另请参见

`-fns`、`-fsimple`、`-ftrap=%none`、`-xlibmil`、`-nofstore`、`-xO5`、`-xlibmopt` 和 `-xtarget=native`

A.2.19 –features=*a*[,*a*...]

启用/禁用逗号分隔的列表中指定的各种 C++ 语言功能。

A.2.19.1 值

在兼容模式 (`-compat[=4]`) 和标准模式（缺省模式）下，*a* 都可以具有以下值：

表 A–6 兼容模式和标准模式下的 `-features` 值

值	含义
`%all`	在指定模式（兼容模式或标准模式）下有效的所有 `-features` 选项。
`[no%]altspell`	[不] 识别替换 标记拼写（如 “and”表示“&&”）。在兼容模式下缺省值为 `no%altspell`，在标准模式下缺省值为 `altspell`。
`[no%]anachronisms`	[不] 允许过时的构造。禁用时（即 `-features=no%anachronisms`），不允许使用任何过时构造。缺省值为 `anachronisms`。
`[no%]bool`	[不] 允许 bool 类型和文字。启用时，宏为 `_BOOL=1`。未启用时，不定义宏。在兼容模式下，缺省值为 `no%bool`，在标准模式下，缺省值为 `bool`。

表 A–6 兼容模式和标准模式下的 -features 值 （续）

值	含义
[no%]conststrings	[不] 将文字字符串放入只读存储器中。在兼容模式下，缺省值为 no%conststrings，在标准模式下，缺省值为 conststrings 。
[no%]except	[不] 允许 C++ 异常。C++ 异常处于禁用状态时（即 -features=no%except），接受但忽略函数的抛出规范，编译器不生成异常代码。请注意，通常保留关键字 try、throw 和 catch。请参见第 96 页中的 “8.3 禁用异常”。缺省值为 except。
[no%]export	[不] 识别关键字 export。在兼容模式下，缺省值为 no%export，在标准模式下，缺省值为 export。此功能尚未实现，但可以识别 export 关键字。
[no%]extensions	[不] 允许其他 C++ 编译器通常接受的非标准代码。有关使用 -features=extensions 选项时编译器接受的无效代码的说明，请参见表 3–17。缺省值为 no%extensions。
[no%]iddollar	[不] 允许 $ 符号作为非首字母标识符字符。缺省值为 no%iddollar。
[no%]localfor	[不] 对 for 语句使用新的局部作用域规则。在兼容模式下，缺省值为 no%localfor，在标准模式下，缺省值为 localfor。
[no%]mutable	[不] 识别关键字 mutable。在兼容模式下，缺省值为 no%mutable，在标准模式下，缺省值为 mutable。
[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 可最大限度减小代码大小，但编译时间可能增加。缺省值为 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，编译器会将这些情况视为错误。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，而不是错误消息。在兼容模式 (-compat[=4]) 下使用 -features=transitions 时，编译器仅在指定了 +w 或 +w2 的情况下才显示有关这些构造的警告。以下构造视为转换错误：在使用了模板后再重新定义模板，遗漏模板定义中所需的 typename 指令，以及隐式声明类型 int。在以后的发行版本中可能会更改转换错误的集合。缺省值为 transitions。
%none	关闭指定模式中可以关闭的所有功能。

在标准模式（缺省模式）下，*a* 可以有下列其他值：

表 A–7 仅适用于标准模式的 -features 值

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。缺省值为 strictdestrorder。
[no%]tmplrefstatic	[不] 允许函数模板引用相关静态函数或静态函数模板。缺省值是遵循标准的 no%tmplrefstatic。
[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，多数临时对象会在其块终结时清除。）在 compat=4 模式下，缺省值为 no%tmplife，在标准模式下，缺省值为 tmplife。

在兼容模式 (-compat[=4]) 下，*a* 可以有下列其他值。

表 A–8 仅适用于兼容模式的 -features 值

值	含义
[no%]arraynew	[不] 识别数组形式的运算符 new 和 delete（如 operator new [ ] (void*)）。启用时，宏为 __ARRAYNEW=1。未启用时，不定义宏。缺省值为 no%arraynew。
[no%]explicit	[不] 识别关键字 explicit。缺省值为 no%explicit。
[no%]namespace	[不] 识别关键字 namespace 和 using。缺省值为 no%namespace。 使用 -features=namespace 是为了协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)。必须启用 RTTI 才能使用 dynamic_cast<> 和 typeid 运算符。在 -compat=4 模式下，缺省值为 no%rtti。其他情况下，缺省值为 -features=rtti，此时不允许使用选项 -features=no%rtti。

注 – 允许使用 [no%]castop 设置，以便与针对 C++ 4.2 编译器编写的 makefile 兼容，但对更高的编译器版本没有影响。新式强制类型转换（const_cast、dynamic_cast、reinterpret_cast 和 static_cast）总是会被识别且无法禁用。

缺省值

如果未指定 –features，兼容模式 (-compat[=4]) 下的缺省值为：

```
–features=%none,anachronisms,except,split_init,transitions
```

在缺省的“标准模式”下，假定采用

```
-features=%all,no%extensions,no%iddollar,no%tmplrefstatic
```

交互

该选项会累积而不覆盖。

在标准模式（缺省）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 选项不起作用，除非指定了 +w 选项或 +w2 选项。

警告

指定 -features=%all 或 -features=%none 时务必谨慎。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，程序的行为可能会发生变化。测试在使用与不使用 -features=tmplife 选项两种情况下程序是否正常运行是一种测试程序可移植性的方法。

在兼容模式 (-compt=4) 下，编译器会在缺省情况下假定 -features=split_init。如果使用 -features=%none 选项禁用其他功能，就会发现需要改用 -features=%none,split_init 重新将初始化函数分到单独的函数中。

另请参见

表 3–17 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功能。

A.2.20.1 值

filter 必须是下列值之一。

表 A–9 `-filt` 值

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++ 损坏链接程序名称。
`[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。这使您可更容易地识别出标准库模板类型的名称。
`%all`	等效于 `-filt=errors,names,returns,stdlib`。这是缺省行为。
`%none`	等效于 `-filt=no%errors,no%names,no%returns,no%stdlib`。

缺省值

如果未指定 `-filt` 选项或指定了 `-filt` 但未提供任何值，则编译器假定 `-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(); // no definition provided
};

int main()
{
    type t;
}
```

如果编译代码时不使用 `-filt` 选项，编译器就假定 `-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
```

```
[Hint: try checking whether the first non-inlined, /
non-pure virtual function of class type is defined]

ld: fatal: Symbol referencing errors. No output written to a.out
```

以下命令禁止还原 C++ 损坏链接程序名称，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); // error here
}
```

下面是指定了 `-filt=no%stdlib` 时的输出：

```
Error: Cannot use std::list<int, std::allocator<int>> to initialize
std::basic_string<char, std::char_traits<char>,
std::allocator<char>>.
```

下面是指定了 `-filt=stdlib` 时的输出：

```
Error: Cannot use std::list<int> to initialize std::string .
```

交互

指定了 `no%names` 时， `returns` 和 `no%returns` 都无效。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 -fma[={none|fused}]

(SPARC) 启用自动生成浮点乘加指令。-fma=none 禁用这些指令的生成。-fma=fused 允许编译器通过使用浮点乘加指令，尝试寻找改进代码性能的机会。

缺省值是 -fma=none。

要使编译器生成乘加指令，最低要求是 -xarch=sparcfmaf 且优先级别至少为 -xO2。如果生成了乘加指令，编译器会标记此二进制程序，以防止该程序在不受支持的平台上执行。

乘加指令会消除乘与加之间的中间舍入步骤。因此，使用 -fma=fused 编译时程序可能会产生不同的结果，尽管精度是趋于提高而不是降低。

A.2.23 –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。

A.2.23.1 缺省值

如果未指定 –fnonstd，则 IEEE 754 浮点算术异常不会中止程序，而是出现逐渐下溢情况。

扩展

x86：-fnonstd 扩展到 -ftrap=common。

SPARC：-fnonstd 扩展到 -fns ftrap=common。

另请参见

–fns、–ftrap=common 和《**数值计算指南**》。

A.2.24 –fns[={yes|no}]

- SPARC：启用/禁用 SPARC 的非标准浮点模式。

  如果使用 -fns=yes（或 -fns），则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使用非标准浮点模式或标准浮点模式的方法，它接在包括 –fns 的其他某些宏选项（如 –fast）后面。

在某些 SPARC 设备上，非标准浮点模式会禁用“逐渐下溢”，这会导致很小的结果刷新为零而不是生成次正规数，此外，还会导致次正规操作数在无提示的情况下替换为零。

对于那些硬件中不支持逐渐下溢和次正规数的 SPARC 设备，`-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）非正规数为零模式。

  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。

  此选项对不使用 SSE 或 SSE2 指令集的传统 x86 浮点运算没有影响。

A.2.24.1 值

`-fns` 选项可以有下列值。

表 A–10 `-fns` 值

值	含义
`yes`	选择非标准浮点模式
`no`	选择标准浮点模式

缺省值

如果未指定 `-fns`，则不自动启用非标准浮点模式。进行标准 IEEE 754 浮点计算（即逐渐下溢）。

如果仅指定了 `–fns`，则假定 `–fns=yes`。

示例

在以下示例中，`-fast` 扩展为多个选项，其中一个是 `-fns=yes`，即选择非标准浮点模式。后续 `-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。

如果使用 `-fns` 选项编译一个例程，就要使用 `–fns` 选项编译该程序的所有例程，否则就会产生意外的结果。

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。

使用 –fns=yes（或 -fns）选项时，如果程序中出现通常由 IEEE 浮点陷阱处理程序管理的浮点错误，则可能会生成以下消息：

另请参见

《数值计算指南》和 ieee_sun(3M)。

A.2.25 –fprecision=*p*

x86：设置非缺省浮点精度模式。

–fprecision 选项用于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。

A.2.25.1 值

p 必须是下列值之一。

表 A–11 -fprecision 值

值	含义
single	舍入到 IEEE 单精度值。
double	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，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 extended 或未使用 –fprecision 选项，则舍入精度模式保持为 extended 精度。

在 single 精度舍入模式下，结果将舍入到 24 个有效位；在 double 精度舍入模式下，结果将舍入到 53 个有效位。在缺省的 extended 精度模式下，结果将舍入到 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 类型的标称精度为 single。long double 类型的标称精度为 extended。

缺省值

如果未指定 –fprecision 选项，舍入精度模式缺省为 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.2.26 –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

含义与 ieee_flags 子例程的含义相同，可用于更改运行时的模式。

A.2.26.1 值

r 必须是下列值之一。

表 A–12 -fround 值

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。

缺省值

如果未指定 –fround 选项，舍入模式缺省为 -fround=nearest。

警告

如果使用 –fround=*r* 编译一个例程，就要使用相同的 –fround=*r* 选项编译程序的所有例程，否则可能会产生意外的结果。

只有编译主程序时该选项才有效。

A.2.27 –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。

A.2.27.1 值

如果存在 *n*，它必须是 0、1 或 2。

表 A–13 -fsimple 值

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。 在 -fsimple=1 的情况下，优化器可假定： ■ 在进程初始化之后，IEEE 754 缺省舍入/捕获模式不发生改变。 ■ 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。 在 -fsimple=1 的情况下，不允许优化器不考虑舍入或异常进行完全优化。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	包括 -fsimple=1 的所有功能，还允许可能导致许多程序生成不同的数值结果（由于舍入的更改）的主动浮点优化。例如，允许优化器将指定循环中的所有 x/y 运算替换为 x*z，其中，要保证在循环 z=1/y 中至少对 x/y 求一次值，并且在执行该循环期间已知 y 和 z 的值为常量值。

缺省值

如果未指定 –fsimple，编译器将使用 -fsimple=0。

如果指定了 -fsimple 但未指定 *n* 值，编译器将使用 -fsimple=1。

交互

-fast 隐含了 – fsimple=2。

警告

该选项可以破坏 IEEE 754 的一致性。

另请参见

-fast

Rajat Garg 和 Ilya Sharapov 合著的《Techniques for Optimizing Applications: High Performance Computing》，该书详细介绍了优化对精度的影响。

A.2.28 –fstore

x86 :

强制浮点表达式的精度。

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。
- 表达式被强制转换为较短的浮点类型。

要禁用此选项，请使用 –nofstore 选项。

A.2.28.1 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.2.29 –ftrap=*t*[,*t*...]

设置启动时生效的IEEE陷阱操作模式，但不安装 SIGFPE 处理程序。可以使用 ieee_handler(3M) 或 fex_set_handling(3M) 启用陷阱并同时安装 SIGFPE 处理程序。如果指定多个值，则按从左到右顺序处理列表。

A.2.29.1 值

t 可以是下列值之一。

表 A–14 -ftrap 值

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。

表 A–14 -ftrap 值 （续）

值	含义
%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，且必须与其中的一个值一起使用，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陷阱。

缺省值

如果未指定 –ftrap，则编译器假定 –ftrap=%none。

示例

–ftrap=%all,no%inexact 意味着设置除 inexact 以外的所有陷阱。

警告

如果使用 –ftrap=*t* 编译一个例程，就要使用相同的 -ftrap=*t* 选项编译程序的所有例程；否则可能会产生意外的结果。

使用 -ftrap=inexact 陷阱时务必谨慎。只要浮点值不能精确表示，使用 –ftrap=inexact 便会产生自陷。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。如果希望启用 IEEE 陷阱，请使用 –ftrap=common。

另请参见

ieee_handler(3M) 和 fex_set_handling(3M) 手册页。

A.2.30 –G

生成动态共享库而不是可执行文件。

缺省情况下，在命令行上指定的所有资源文件都是使用 -xcode=pic13 进行编译的。

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。

如果要通过指定 -G 与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，请确保在编译时和与生成的共享对象链接时也指定这些选项。

创建共享对象时，所有使用 -xarch=v9 编译的目标文件也必须使用 第 272 页中的 “A.2.119 –xcode=*a*” 中推荐的显式 -xcode 值进行编译。

A.2.30.1 交互

如果未指定 – c（仅编译选项），以下选项将传递给链接程序：

- –dy
- –G
- –R

警告

请勿使用 ld -G 生成共享库，而应使用 CC - G。CC 驱动程序会自动将 C++ 所需的多个选项传递给 ld。

使用 -G 选项时，编译器不将任何缺省 -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，如果要使共享库依赖于 libCrun，必须在命令行上传递 -lCrun。

另请参见

-dy、-Kpic、-xcode=pic13、–ztext、ld(1) 手册页以及第 190 页中的 “16.3 生成动态（共享）库”。

A.2.31 –g

生成附加的符号表信息，以供使用 dbx(1) 或调试器进行调试以及使用性能分析器 analyzer(1) 进行分析。

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：

- 生成以目标文件和可执行文件的符号表形式表示的详细信息（称为 *stabs*）。
- 生成某些“帮助应用程序函数”，调试器可以调用这些函数来实现其某些功能。
- 禁用函数的内联生成。
- 禁用优化的某些级别。

A.2.31.1 交互

如果将此选项与 –x0*level*（或其等效选项，如 -O）一起使用，将会获得一些特定的调试信息。有关更多信息，请参见 第 298 页中的 “A.2.157 -x0*level*”。

如果使用该选项且优化级别为 -x04 或更高，编译器会为完全优化提供尽可能多的符号信息。

指定此选项时，除非还指定 -O 或 -xO，否则会自动指定 +d 选项。

注 – 在以前的发行版中，此选项强制编译器在缺省情况下使用增量链接程序 (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，只要使用编译器来链接目标文件，编译器的缺省行为就是自动调用 ild 而不是 ld，除非在命令行上指定了 -G 或源文件。现在已不再是这种情况。增量链接程序不再可用。

要使用性能分析器的完整功能，请使用 -g 选项进行编译。尽管一些性能分析功能不要求使用 -g，但是必须使用 -g 进行编译才能查看带注释的源代码、一些函数级信息和编译器注释性消息。有关更多信息，请参阅 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编译器在编译程序时进行的优化和变换。使用 er_src(1) 命令来显示与源代码交叉的消息。

警告

如果在不同的步骤中编译和链接程序，则在一个步骤中使用 -g 选项而在另一个步骤中不使用该选项不会影响程序的正确性，但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进行调试。请注意，通常必须使用 -g 选项（或 -g0 选项）编译包含函数 main 的模块才能对其进行调试。

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 调试程序》（介绍了有关 stabs 的详细信息）和《程序性能分析工具》。

A.2.32 –g0

编译和链接以便进行调试，但不禁用内联。

此选项与 –g 相同，但 +d 处于禁用状态，dbx 无法步入内联函数。

如果指定 -g0 且优化级别为 -xO3 或更低，编译器会为近乎完全优化提供尽可能多的符号信息。尾部调用优化和后端内联被禁用。

A.2.32.1 另请参见

+d、–g 和《使用 dbx 调试程序》

A.2.33 –H

打印所包含文件的路径名。

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（每行一个）。

A.2.34 –h[ ]*name*

为生成的动态共享库指定名称 *name*。

这是一个链接程序选项，传递给 ld。通常， -h 后面的名称应该与 –o 后面的名称完全相同。–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，则没有内部名称记录在库文件中。

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。

没有使用 -h 选项生成共享库时，运行时加载器仅查找库的文件名。可以将库替换为具有相同文件名的不同库。如果共享库有内部名称，加载器会在装入文件时检查内部名称。如果内部名称不匹配，加载器不会使用替换文件。

A.2.34.1 示例

```
example% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

A.2.35 –help

与 -xhelp=flags 相同。

A.2.36 -I*pathname*

将 *pathname* 添加到 #include 文件中的搜索路径中。

该选项用于将 *pathname* 添加到在其中搜索具有相对文件名（不以斜杠开头的文件名）的 #include 文件的目录列表。

编译器按以下顺序搜索用引号引住的文件（形式为 #include "foo.h"）。

1. 在包含源代码的目录中
2. 在使用 -I 选项指定的目录（如果有）中
3. 在编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的 include 目录中
4. 在 /usr/include 目录中

编译器按以下顺序搜索用尖括号括住的文件（形式为 #include <foo.h>）。

1. 在使用 -I 选项指定的目录（如果有）中
2. 在编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的 include 目录中
3. 在 /usr/include 目录中

注 – 如果此拼写与标准头文件的名称匹配，另请参阅第 141 页中的 “12.7.5 标准头文件实现”。

A.2.36.1 交互

-I- 选项让您可以覆盖缺省的搜索规则。

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++ 标准库关联的头文件。请参见第 140 页中的 “12.7 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* 而不是 –pti*path*。

该选项会累积而不覆盖。

警告

任何时候都不要将编译器安装区域 /usr/include、/lib 或 /usr/lib 指定为搜索目录。

另请参见

-I-

A.2.37 -I-

将头文件的搜索规则更改为：

对于形式为 #include "foo.h" 的 include 文件，按以下顺序搜索目录：

1. 使用 -I 选项指定的目录（在 -I- 前后）。

2. 编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的目录。

3. /usr/include 目录。

对于 #include <foo.h> 形式的 include 文件，按以下顺序搜索目录：

1. 使用 -I 选项指定的目录（在 -I- 后面）。

2. 编译器提供的 C++ 头文件、ANSI C 头文件和专用文件的目录。

3. /usr/include 目录。

注 – 如果包括文件的名称与标准头的名称相匹配，另请参阅第 141 页中的“12.7.5 标准头文件实现”。

A.2.37.1 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
```

以下命令显示了针对形式为 #include "foo.h" 的包含语句搜索当前目录（包含文件的目录）的缺省行为。在 inc/a.h 中处理 #include "c.h" 语句时，编译器将 inc 子目录中的 c.h 头文件包含进来。在 prog.cc 中处理 #include "c.h" 语句时，编译器将包含 prog.cc 的目录中的 c.h 文件包含进来。请注意，-H 选项指示编译器打印所包含文件的路径。

```
example% CC -c -Iinc -H prog.cc
inc/a.h
        inc/c.h
inc/b.h
        inc/c.h
c.h
```

以下命令显示了 -I- 选项的效果。编译器处理形式为 #include "foo.h" 的语句时，并不先在包含目录中查找，而是按照通过 -I 选项指定的目录在命令行上的显示顺序搜索这些目录。在 inc/a.h 中处理 #include "c.h" 语句时，编译器包含 ./c.h 头文件而不是 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

命令行上有 -I- 时，编译器从不搜索当前目录，除非在 -I 指令中显式列出了该目录。甚至是形式为 #include "foo.h" 的包含语句，也是这种情况。

警告

只有命令行上的第一个 -I- 会导致出现所述行为。

任何时候都不要将编译器安装区域 /usr/include、/lib 或 /usr/lib 指定为搜索目录。

A.2.38 –i

通知链接程序 ld 忽略任何 LD_LIBRARY_PATH 设置。

A.2.39 -include *filename*

此选项使编译器处理 *filename* 的方式就相当于其是位于主源文件首行的 #include 预处理程序指令。例如以下源文件 t.c：

```
main()
{
   ...
}
```

如果使用 **cc -include t.h t.c** 命令编译 t.c，如同源文件包含以下内容来编译：

```
#include "t.h"
main()
{
   ...
}
```

编译器搜索 *filename* 的第一个目录是当前的工作目录，不是包含主源文件的目录，这是显式包含文件时的情况。例如，以下目录结构包含两个具有相同名称但在不同位置的头文件：

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

如果您的工作目录是 `foo/bar` 且使用 **`cc ../t.c -include t.h`** 命令进行编译，则编译器包括 `foo/bar` 中而不是 `foo/` 中的 `t.h`，这是使用源文件 `t.c` 中的 `#include` 指令时的情况。

如果编译器在当前工作目录中找不到使用 `-include` 指定的文件，会在常规目录路径中搜索该文件。如果指定了多个 `-include` 选项，则按照文件在命令行中出现的顺序将其包含进来。

A.2.40 -inline

与 `-xinline` 相同。

A.2.41 –instances=*a*

控制模板实例的放置和链接。

A.2.41.1 值

a 必须是下列值之一。

表 A–15 `-instances`值

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。） **注意**：如果在不同的步骤中进行编译和链接，并且在编译步骤中指定了 `-instance=extern`，则还必须在链接步骤中指定该选项。
`explicit`	将显式实例化的实例放置到当前目标文件中并赋予全局链接。不生成其他任何所需实例。
`global`	将全部所需的实例放置到当前目标文件中并赋予全局链接。

表 A–15 -instances值 （续）

值	含义
semiexplicit	将显式实例化的实例放置到当前目标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目标文件中并赋予全局链接。不生成其他任何所需实例。
static	**注意：** -instances=static 已过时。没有任何理由再使用 -instances=static，因为 -instances=global 现在提供了静态的所有优点而没有其缺点。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目标文件中并赋予静态链接。

缺省值

如果未指定 –instances，则假定 –instances=global。

另请参见

第 88 页中的 “7.2.4 模板实例的放置和链接”。

A.2.42 –instlib=*filename*

使用该选项可避免在库（共享或静态）和当前对象中生成重复的模板实例。一般来说，如果程序与库共享大量实例，可以尝试使用 -instlib=*filename*，看看编译时间是否会减少。

A.2.42.1 值：

可使用 *filename* 参数指定已知包含现有模块实例的库。filename 参数必须包含正斜杠 "/" 字符。对于相对于当前目录的路径，请使用点斜杠 "./"。

缺省：

-instlib=*filename* 选项没有缺省值，只有在指定后才能使用。该选项可被多次指定和累积。

示例：

假定 libfoo.a 和 libbar.so 库可对与源文件 a.cc 共享的大量模板实例进行实例化。添加 -instlib=*filename* 并指定库可通过避免冗余有利于减少编译时间。

```
example% CC -c -instlib=./libfoo.a -instlib=./libbar.so a.cc
```

交互：

使用 -g 进行编译时，如果使用 -instlib=*file* 指定的库没有使用 -g 编译，那么这些模板实例不可调试。解决方法是避免在使用 -g 时使用 -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 和 -pti

A.2.43 –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.2.44 –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.2.45 –keeptmp

保留编译时创建的临时文件。

该选项与 –verbose=diags 一起使用，对调试很有用。

A.2.45.1 另请参见

–v, –verbose

A.2.46 –L*path*

将 *path* 添加到要在其中搜索库的目录列表。

该选项传递给 ld。搜索顺序是先搜索通过 *path* 指定的目录，再搜索编译器提供的目录。

A.2.46.1 交互

该选项会累积而不覆盖。

警告

任何时候都不要将编译器安装区域 /usr/include、/lib 或 /usr/lib 指定为搜索目录。

A.2.47 –l*lib*

将库 lib*lib*.a 或 lib*lib*.so 添加到链接程序的搜索库列表。

该选项传递给 ld。常规库的名称如 lib*lib*.a 或 lib*lib*.so，其中，lib 和 .a 或 .so 这些部分是必需的。应该使用此选项指定 *lib* 部分。应在一个命令行上输入尽可能多的库，对这些库搜索时按照使用 –L*dir* 指定的顺序进行。

请在目标文件名之后使用该选项。

A.2.47.1 交互

该选项会累积而不覆盖。

较稳妥的方式是总是将 –l*x* 放在源文件和目标文件列表**后面**，这样可以确保按正确顺序搜索库。

警告

为了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 126 页中的 “11.4.8 示例应用程序”和《Tools.h++ 类库参考》

A.2.48 –libmieee

与 –xlibmieee 相同。

A.2.49 –libmil

与 -xlibmil 相同。

A.2.50 -library=*l*[,*l*...]

将指定的 cc 提供的库加入编译和链接操作中。

A.2.50.1 值

对于兼容模式 (–compat[-4])，*l* 必须是下列值之一。

表 A–16 适用于兼容模式的 -library 值

值	含义
[no%]f77	已过时。改用 -xlang=f77。
[no%]f90	已过时。改用 -xlang=f90。
[no%]f95	已过时。改用 -xlang=f95。
[no%]rwtools7	[不] 使用传统 iostream Tools.h++ 版本 7。
[no%]rwtools7_dbg	[不] 使用支持调试的 Tools.h++ 版本 7。
[no%]complex	[不] 将 libcomplex 用于复数算术运算。
[no%]interval	已过时。不要使用。使用 -xia。
[no%]libC	[不] 使用C++ 支持库 libC。
[no%]gc	[不] 使用垃圾收集 libgc。
[no%]sunperf	[不] 使用 Sun Performance Library™。
%none	仅使用 C++ 库 libC。

在标准模式（缺省模式）下，*l* 必须是下列值之一：

表 A–17 适用于标准模式的 -library 值

值	含义
[no%]f77	已过时。改用 -xlang=f77。
[no%]f90	已过时。改用 -xlang=f90。
[no%]f95	已过时。改用 -xlang=f95。
[no%]rwtools7	[不] 使用传统 iostream Tools.h++ 版本 7。
[no%]rwtools7_dbg	[不] 使用支持调试的 Tools.h++ 版本 7。
[no%]rwtools7_std	[不] 使用标准 iostream Tools.h++ 版本 7。
[no%]rwtools7_std_dbg	[不] 使用支持调试的标准 iostream Tools.h++ 版本 7。

表 A–17 适用于标准模式的 -library 值 （续）

值	含义
[no%]interval	已过时。不要使用。使用 -xia。
[no%]iostream	[不] 使用传统 iostream 库 libiostream。
[no%]Cstd	[不] 使用 C++ 标准库 libCstd。[不] 包括编译器提供的 C++ 标准库头文件。
[no%]Crun	[不] 使用 C++ 运行时库 libCrun。
[no%]gc	[不] 使用垃圾收集 libgc。
[no%]stlport4	[不] 使用 STLport 的标准库实现版本 4.5.3，而不是缺省的 libCstd。关于使用 STLport 实现的更多信息，请参阅 第 158 页中的 “13.3 STLport”。
[no%]stlport4_dbg	[不] 使用 STLport 的支持调试的库。
[no%]sunperf	[不] 使用 Sun Performance Library。
%none	仅使用 C++ 库 libCrun。

缺省值

- **兼容模式 (–compat[=4])**
  - 如果未指定 –library，则假定 -library=libC。
  - libC 库总是包括在内，除非使用 -library=no%libC 明确将其排除。

  标准模式（缺省模式）
  - libCstd 库总是包括在内，除非使用 -library=%none、-library=no%Cstd 或 -library=stlport4 明确将其排除。
  - libCrun 库总是包括在内，除非使用 -library=no%Crun 明确将其排除。

  无论是标准模式还是兼容模式，libm 库和 libc 库总是包括在内，即使指定了 -library=%none 也是如此。

示例

要在标准模式下没有任何 C++ 库（除 libCrun 之外）时进行链接，请使用：

```
example% CC -library=%none
```

要在标准模式下将传统 iostream Rogue tools.h++ 库包含进来，请使用：

```
example% CC –library=rwtools7,iostream
```

要在标准模式下将标准 iostream Rogue Wave tools.h++ 库包含进来，请使用：

```
example% CC -library=rwtools7_std
```

要在兼容模式下将传统 iostream Rogue Wave `tools.h++` 库包含进来，请使用：

```
example% CC -compat -library=rwtools7
```

交互

如果使用 `-library` 指定了库，则在编译期间会设置适当的 `–I` 路径。在链接期间会设置适当的 `–L`、`–Y P`、`–R` 路径和 `–l` 选项。

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`。

使用 `-library` 选项可确保针对指定库的 `-l` 选项按正确顺序传送。例如，对于 `-library=rwtools7,iostream` 和 `-library=iostream,rwtools7`，`-l` 选项都是按照 `-lrwtool -liostream` 顺序传递给 `ld`。

指定的库在系统支持库链接之前链接。

不能在同一个命令行上使用 `-library=sunperf` 和 `-xlic_lib=sunperf`。

在任何命令行中，最多只能使用 `-library=stlport4` 或 `-library=Cstd` 选项之一。

每次只能使用一个 Rogue Wave 工具库，而且不能将任何 Rogue Wave 工具库与 `-library=stlport4` 一起使用。

在标准模式（缺省模式）下包含传统 iostream Rogue Wave 工具库时，必须也要包含 `libiostream`（有关其他信息，请参见《C++ 迁移指南》。只能在标准模式下使用标准 iostream Rogue Wave 工具库。以下命令示例显示了有效使用和无效使用 Rogue Wave `tools.h++` 库选项的情况。

```
% CC -compat -library=rwtools7 foo.cc         <-- valid
% CC -compat -library=rwtools7_std foo.cc     <-- invalid

% CC -library=rwtools7,iostream foo.cc        <-- valid, classic iostreams
% CC -library=rwtools7 foo.cc                 <-- invalid

% CC -library=rwtools7_std foo.cc             <-- valid, standard iostreams
% CC -library=rwtools7_std,iostream foo.cc    <-- invalid
```

如果同时包含 `libCstd` 和 `libiostream`，必须小心，不要在程序中同时使用新旧格式的 iostream（例如，`cout` 和 `std::cout`）访问同一个文件。如果从传统和标准 iostream 代码访问同一文件，那么在相同的程序中混合标准 iostream 和传统 iostream 可能会出现问题。

不链接 `libC` 库的兼容模式程序无法使用 C++ 语言的所有功能。与之类似，不链接 `Crun` 或任何 `Cstd` 库或 `stlport4` 库的标准模式程序无法使用 C++ 语言的所有功能。

如果指定了 -xnolib，则忽略 -library。

警告

如果在不同的步骤中进行编译和链接，那么必须在链接命令中使用在编译命令中使用的那组 -library 选项。

stlport4、Cstd 和 iostream 库都提供了自己的 I/O 流实现。如果使用 -library 选项指定其中多个库，会导致出现不确定的程序行为。关于使用 STLport 实现的更多信息，请参阅 第 158 页中的 “13.3 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 126 页中的 “11.4.8 示例应用程序”、第 143 页中的 “忠告：”、第 159 页中的 “13.3.1 重新分发和支持的 STLport 库”、《Tools.h++ 用户指南》、《Tools.h++ 类库参考》、《Standard C++ Class Library Reference》和《C++ Interval Arithmetic Programming Reference》。

有关使用 -library=no%cstd 选项以便能够使用自己的 C++ 标准库的信息，请参见 第 140 页中的 “12.7 替换 C++ 标准库”。

A.2.51 -m32|-m64

指定编译的二进制对象的内存模型。

使用 -m32 来创建 32 位可执行文件和共享库。使用 -m64 来创建 64 位可执行文件和共享库。

在所有 Solaris 平台和不支持 64 位的 Linux 平台上，ILP32 内存模型（32 位 int、long、pointer 数据类型）是缺省值。在启用了 64 位的 Linux 平台上缺省为 LP64 内存模型（64 位 long 和指针数据类型）。-m64 仅允许在支持 LP64 模型的平台上使用。

使用 -m32 编译的对象文件或库无法与使用 -m64 编译的对象文件或库链接。

使用 -m32|-m64 编译的模块还必须使用 -m32|-m64 进行链接。有关在编译时和链接时都必须指定的完整编译器选项列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

当使用 -m64 在 x64 平台上编译具有大量静态数据的应用程序时，可能还需要使用 -xmodel=medium。请注意，部分 Linux 平台不支持中等模型。

请注意，在早期的编译器发行版中，由 -xarch 选项中选择的指令集来指定内存模型（ILP32 或 LP64）。从 Sun Studio 12 编译器开始，就不再是这样了。在大多数平台上，仅需将 -m64 添加到命令行便可以创建 64 位对象。

在 Solaris 上，-m32 是缺省选项。在支持 64 位程序的 Linux 系统上，-m64 -xarch=sse2 是缺省选项。

另请参见 -xarch。

A.2.52 -mc

从目标文件的 .comment 部分中删除重复的字符串。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.2.53 –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.2.54 –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。

A.2.54.1 交互

如果在 SPARC 平台上使用 #pragma pack 封装效果比类型的缺省对齐紧密，必须为应用程序的编译和链接指定 -misalign 选项。

未对齐数据由 ld 提供的陷阱机制在运行时处理。如果优化标志（-xO{1|2|3|4|5} 或等价的标志）与 -misalign 选项一起使用，则用于对齐未对齐数据的附加指令会插入生成的目标文件中，且不会生成运行时未对齐陷阱。

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进行编译和链接，那么编译命令和链接命令中都必须有 –misalign 选项。

A.2.55 -mr[, *string*]

从目标文件的 .comment 部分中删除所有字符串，如果提供了 *string*，则将 *string* 放入该部分。如果字符串包含空格，就必须使用引号将该字符串括入。如果使用此选项，会调用命令 mcs -d [-a *string*]。

A.2.56 –mt

可使用此选项在使用 Solaris 线程或 POSIX 线程 API 时编译和链接多线程代码。-mt 选项可确保以适当的顺序链接库。

此选项将 -D_REENTRANT 传递给预处理程序。

要使用 Solaris 线程，应将 thread.h 头文件包含进来并使用 –mt 选项进行编译。要在 Solaris 平台上使用 POSIX 线程，应将 pthread.h 头文件包含进来并使用 –mt –lpthread 选项进行编译。

在 Linux 平台上，只有 POSIX 线程 API 可用。（在 Linux 平台上没有 libthread。）因此，Linux 平台上的 –mt 添加 –lpthread 而不是 –lthread。要在 Linux 平台上使用 POSIX 线程，应使用 –mt 进行编译。

请注意，使用 –G 进行编译时，–mt 不会自动包含 –lthread 和 –lpthread。需要在生成共享库时显式列出这些库。

–xopenmp 选项（用于使用 OpenMP 共享内存并行化 API）自动包含 –mt。

如果使用 -mt 进行编译并在单独的步骤中进行链接，则除了在编译步骤中外，还必须在链接步骤中使用 -mt 选项。如果使用 -mt 编译和链接一个转换单元，则必须使用 -mt 编译和链接该程序的所有单元。

A.2.56.1 另请参见

–xnolib、《Solaris 多线程编程指南》和《链接程序和库指南》

A.2.57 –native

与 –xtarget=native 相同。

A.2.58 –noex

与 –features=no%except 相同。

A.2.59 –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

- 表达式或函数被强制转换为较短的浮点类型

A.2.59.1 另请参见

`–fstore`

A.2.60 –nolib

与 `–xnolib` 相同。

A.2.61 –nolibmil

与 `–xnolibmil` 相同。

A.2.62 –noqueue

（**已过时**）禁用许可证排队。

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。**该选项已过时并被忽略。**

A.2.63 –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。为此，编译器会将 `–R` 选项传递给 `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。请参阅第 139 页中的 “12.6 使用共享库”。

A.2.63.1 交互

如果使用编译器安装区域下的任何共享库，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.2.64 –O

现在，-O 宏扩展到 -xO3 而不是 -xO2。

这种特殊变化会提高运行时性能。但是，对于依赖于被自动视为 volatile 的所有变量的程序，-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。解决方法是用 -xO2 而不是 -O 进行编译。

A.2.65 –O*level*

与 –x*level* 相同。

A.2.66 –o *filename*

将输出文件或可执行文件的名称设置为 *filename*。

A.2.66.1 交互

编译器必须存储模板实例时，它将模板实例存储在输出文件目录中的模板系统信息库中。例如，以下命令将目标文件写入 ./sub/a.o 并将模板实例写入包含在 ./sub/SunWS_cache 中的系统信息库。

```
example% CC -instances=extern -o sub/a.o a.cc
```

编译器从对应于编译器读取的目标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，必要时，向 ./SunWS_cache 写入。

```
example% CC -instances=extern sub1/a.o sub2/b.o
```

有关更多信息，请参见第 91 页中的 “7.4 模板系统信息库”。

警告

filename 必须有与编译所生成文件的类型对应的适当后缀。它与源文件不是相同的文件，因此 CC 驱动程序不会覆盖源文件。

A.2.67 +p

忽略非标准预处理程序声明。

A.2.67.1 缺省值

如果没有 +p，则编译器识别非标准预处理程序声明。

交互

如果使用了 +p，就不定义下列宏：

- sun
- unix
- sparc
- i386

A.2.68 –P

仅预处理源代码，但不编译。（输出文件的后缀为 .i 。）

该选项不在输出中包括预处理程序类型的行号信息。

A.2.68.1 另请参见

–E

A.2.69 –p

已废弃，请参见第 307 页中的 “A.2.165 –xpg”。

A.2.70 –pentium

x86： 替换为 –xtarget=pentium。

A.2.71 –pg

与 –xpg 相同。

A.2.72 -PIC

SPARC： 与 –xcode=pic32 相同。

x86： 与 –Kpic 相同。

A.2.73 –pic

SPARC： 与 –xcode=pic13 相同。

x86： 与 -Kpic 相同。

A.2.74 –pta

与 –template=wholeclass 相同。

A.2.75 –pti*path*

为模板源文件指定附加搜索目录。

该选项可替换 –I*pathname* 设置的正常搜索路径。如果使用了 -pti*path* 选项，编译器将在该路径上查找模板定义文件，并忽略 –I*pathname* 选项。

如果使用 –I*pathname* 选项而非 –pti*path*，可以减少混淆情况。

A.2.75.1 交互

该选项会累积而不覆盖。

另请参见

–I*pathname* 和第 93 页中的 “7.5.2 定义搜索路径”

A.2.76 –pto

与 –instances=static 相同。

A.2.77 –ptr

该选项已废弃并被编译器忽略。

A.2.77.1 警告

即使忽略 -ptr 选项，也应该从所有编译命令中删除 -ptr，因为在后续发行版中重用该选项时可能会产生不同的行为。

另请参见

有关系统信息库目录的信息，请参见第 91 页中的 “7.4 模板系统信息库”。

A.2.78 –ptv

与 –verbose=template 相同。

A.2.79 –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。可以对使用 -Q 传递给组件的选项进行重新排序。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，请勿使用 -Q。例如，C++ 编译器识别用于链接程序 (ld) 的 -z 选项。如果发出如下命令：

```
CC -G -zallextract mylib.a -zdefaultextract ... // correct
```

则 -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 ... // error
```

则会重新排序 -z 选项，从而得到错误的结果。

A.2.79.1 值

phase 必须是下列值之一。

表 A–18 -Qoption 值

SPARC	x86
ccfe	ccfe
iropt	iropt
cg	ube
CClink	CClink
ld	ld
—	ir2hf
fbe	fbe

示例

在下列命令行上，当 CC 驱动程序调用 ld 时，–Qoption 会将 –i 和 –m 选项传递给 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.2.80 –qoption *phase option*

与 –Qoption 相同。

A.2.81 –qp

与 –p 相同。

A.2.82 –Qproduce *sourcetype*

使 cc 驱动程序生成类型为 *sourcetype* 的输出。

Sourcetype 后缀定义如下。

表 A–19 -Qproduce 值

后缀	含义
.i	来自 ccfe 的预处理 C++ 源代码
.o	目标文件代码生成器
.s	来自 cg 的汇编程序源代码

A.2.83 –qproduce *sourcetype*

与 –Qproduce 相同。

A.2.84 –R*pathname*[:*pathname*...]

将动态库搜索路径生成到可执行文件中。

该选项传递给 ld。

A.2.84.1 缺省值

如果没有 -R 选项，则在输出对象中记录且传递给运行时链接程序的库搜索路径取决于 -xarch 选项（没有 -xarch 时，假定 -xarch=generic）指定的目标体系结构指令。

检查 –dryrun 的输出和传递给链接程序 ld 的 –R 选项，以查看编译器假定的缺省路径。

交互

该选项会累积而不覆盖。

如果定义了环境变量 LD_RUN_PATH 且指定了 –R 选项，则扫描 –R 指定的路径而忽略 LD_RUN_PATH 指定的路径。

另请参见

–norunpath 和《链接程序和库指南》。

A.2.85 –readme

与 -xhelp=readme 相同。

A.2.86 –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，但不汇编程序。汇编源文件名称的后缀为 .s。

A.2.87 –s

从可执行文件中删除符号表。

该选项从输出可执行文件中删除所有符号信息。该选项传递给 ld。

A.2.88 –sb

已过时，默认忽略。

A.2.89 –sbfast

已过时，默认忽略。

A.2.90 -staticlib=*l*[,*l*...]

指示要静态链接由 -library 选项（包括其缺省值）、-xlang 选项和 -xia 选项指定的哪些 C++ 库。

A.2.90.1 值

l 必须是下列值之一。

表 A–20 -staticlib 值

值	含义
[no%]*library*	[不] 静态链接 *library*。*library* 的有效值包括了 -library 的全部有效值（除 %all 和 %none 之外）、-xlang 的全部有效值以及 interval（要与 -xia 结合使用）。
%all	静态链接 -library 选项中指定的所有库、-xlang 选项中指定的所有库以及（如果在命令行上指定了 -xia）区间库。
%none	不静态链接在 -library 选项和 -xlang 选项中指定的库。如果在命令行上指定了 -xia，则不静态链接区间库。

缺省值

如果没有指定 –staticlib，则假定 –staticlib=%none。

示例

以下命令行静态链接 libCrun，因为 Crun 是 –library 的缺省值：

```
example% CC –staticlib=Crun (correct)
```

但以下命令行并不链接 libgc，因为只有使用 -library 选项显式指定才链接 libgc：

```
example% CC –staticlib=gc (incorrect)
```

要静态链接 libgc，请使用以下命令：

```
example% CC -library=gc -staticlib=gc (correct)
```

以下命令会动态链接 librwtool 库。因为 librwtool 不是缺省库且未使用 -library 选项选择它，因此 -staticlib 不起作用：

```
example% CC -lrwtool -library=iostream \
-staticlib=rwtools7 (incorrect)
```

该命令静态链接 librwtool 库：

```
example% CC -library=rwtools7,iostream -staticlib=rwtools7 (correct)
```

该命令将动态链接 Sun 性能库，因为 -library=sunperf 必须与 -staticlib=sunperf 结合使用，-staticlib 选项才能对这些库的链接有效：

```
example% CC -xlic_lib=sunperf -staticlib=sunperf (incorrect)
```

该命令将静态链接 Sun 性能库：

```
example% CC -library=sunperf -staticlib=sunperf (correct)
```

交互

该选项会累积而不覆盖。

除缺省情况下隐式选择的 C++ 库之外，-staticlib 选项仅对使用 -xia 选项、-xlang 选项和 -library 选项显式选择的 C++ 库有效。在兼容模式 (-compat=[4]) 下，缺省选择 libC。在标准模式（缺省模式）下，缺省选择 Cstd 和 Crun。

使用 -xarch=v9、-xarch=v9a 或 -xarch=v9b （或等价的 64 位体系结构选项）时，某些 C++ 库不能用作静态库。

警告

library 的允许值集合不确定，会随发行版的不同而异。

使用 -xarch=v9、-xarch=v9a 或 -xarch=v9b（或等价的 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 和第 138 页中的 “12.5 静态链接标准库”

A.2.91 -sync_stdio=[yes| no]

可在运行时性能因 C++ iostream 和 C stdio 之间的同步而降低时使用此选项。仅当您在相同的程序中使用 iostream 写入 cout 以及使用 stdio 写入 stdout 时，才需要同步。C++

标准要求同步，因此缺省情况下 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。

A.2.91.1 缺省：

如果未指定 `-sync_stdio`，编译器会将其设置为 `-sync_stdio=yes`。

示例：

请看以下示例：

```
#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.2.92 –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。对于 `-temp` 设置的值和 `TMPDIR` 值，编译器优先采用前者。

A.2.92.1 另请参见

`–keeptmp`

A.2.93 –template=*opt*[,*opt*...]

启用/禁用各种模板选项。

A.2.93.1 值

opt 必须是下列值之一。

表 A–21 -template 值

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。

缺省值

如果未指定 -template 选项，则假定 -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%
```

指定了 -template=geninlinefuncs 时，即使在程序中没有调用 S 的两个成员函数，也会在目标文件中生成它们。

```
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 87 页中的 “7.2.2 整个类实例化”和第 92 页中的 “7.5 模板定义搜索”

A.2.94 –time

与 –xtime 相同。

A.2.95 –U*name*

删除预处理程序符号 *name* 的初始定义。

该选项会删除在命令行上通过 -D（包括 CC 驱动程序隐式放在命令行上的选项）创建的宏符号 *name* 的所有初始定义。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 驱动程序放在命令行上的 -D 选项，请将 -dryrun 选项添加到命令行上。

A.2.95.1 示例

以下命令取消预定义符号 __sun 的定义。foo.cc 中的预处理程序语句（例如 #ifdef(__sun)）会知道该符号已取消定义。

```
example% CC -U__sun foo.cc
```

交互

可以在命令行上指定多个 -U 选项。

所有 -U 选项都在出现的任何 -D 选项之后处理。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，而不管这些选项出现的顺序如何。

另请参见

-D

A.2.96 –unroll=*n*

与 –xunroll=*n* 相同。

A.2.97 –V

与 –verbose=version 相同。

A.2.98 –v

与 –verbose=diags 相同。

A.2.99 –vdelx

已过时，不使用。

仅限于兼容模式 (–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，而不是生成对 _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。

A.2.99.1 缺省

对于使用 delete[] 的表达式，编译器生成对 _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.2.100 –verbose=*v*[,*v*...]

控制编译器详细程度。

A.2.100.1 值

v 必须是下列值之一。

表 A–22 -verbose 值

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 verbose 模式（有时称为“检验”模式）。verbose 模式显示编译过程中出现的每个实例阶段。
[no%]version	[不] 指示 CC 驱动程序打印所调用程序的名称和版本号。
%all	调用以上所有内容。
%none	-verbose=%none 与 -verbose=no%template,no%diags,no%version 相同。

缺省值

如果未指定 –verbose，则假定 –verbose=%none。

交互

该选项会累积而不覆盖。

A.2.101 +w

识别出可能会产生意外后果的代码。使用 +w 选项时，如果函数过大而无法内联或未使用声明的程序元素，就不再生成警告。这些警告不指定源代码中的真正问题，因此不适合某些开发环境。从 +w 中删除这些警告就可以在这些环境下更主动地使用 +w。在 +w2 选项中仍可以使用这些警告。

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

A.2.101.1 缺省值

如果未指定 +w，则编译器发出有关极可能是问题的构造的警告。

另请参见

–w 和 +w2。

A.2.102 +w2

发出 +w 发出的所有警告以及可能无害但可能降低程序最大可移植性的技术违规的警告。

+w2 选项不再发出关于在系统头文件中使用与实现相关的构造方面的警告。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。

A.2.102.1 另请参见

+w

A.2.103 –w

抑制大部分警告消息。

该选项使编译器**不**打印警告消息。但不能抑制某些警告（尤其是有关严重记时错误的警告）。

A.2.103.1 另请参见

+w

A.2.104 –Xm

与 –features=iddollar 相同。

A.2.105 –xa

为文件配置生成代码。

如果在编译时设置，则由环境变量 TCOVDIR 指定覆盖 (.d) 文件所在的目录。如果未设置该变量，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。

A.2.105.1 交互

-xprofile=tcov 选项和 -xa 选项在单个可执行程序中兼容。也就是说，在您链接的同一个程序中，既可包含用 –xprofile=tcov 编译的一些文件，也可包含用 –xa 编译的另外一些文件。您不能同时用这两个选项来编译同一个文件。

–xa 选项与 –g 不兼容。

警告

如果在不同的步骤中进行编译和链接，且使用 -xa 进行编译，应确保使用 –xa 进行链接，否则会产生意外的结果。

另请参见

–xprofile=tcov、tcov(1) 手册页和《程序性能分析工具》。

A.2.106 -xaddr32

（**仅限** *Solaris x86/x64*）-xaddr32=yes 编译标志将生成的可执行文件或共享对象限定于 32 位地址空间。

以此方式编译的可执行文件导致创建的进程限定于 32 位地址空间。

指定 -xaddr32=no 时，会生成常见的 64 位二进制文件。

如果未指定 -xaddr32 选项，则假定 **-xaddr32=no**。

如果仅指定了 -xaddr32，则假定 -xaddr32=yes。

此选项仅适用于 -m64 编译，以及仅适用于支持 SF1_SUNW_ADDR32 软件功能的 Solaris 平台。由于 Linux 内核不支持地址空间限制，因此在 Linux 上此选项不可用。

链接时，如果使用 -xaddr32=yes 编译了单个目标文件，则假定使用 -xaddr32=yes 编译整个输出文件。

限定于 32 位地址空间的共享对象必须由受限的 32 位模式地址空间中执行的进程装入。

有关更多信息，请参阅《链接程序和库指南》中所述的“SF1_SUNW_ADDR32 软件功能定义”。

A.2.107 -xalias_level[= *n*]

(SPARC) 指定了以下命令时，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	float
signed char	unsigned short int	unsigned long int	double
unsigned char	int	long long int	long double
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
  - 动态类型对象的 `constant` 或 `volatile` 限定版本，与动态类型对象对应的带符号或无符号类型
  - 与动态类型对象的 `constant` 或 `volatile` 限定版本对应的带符号或无符号类型
  - 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 类型

  `-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
  - 与动态类型对象的 `constant` 或 `volatile` 限定版本对应的带符号或无符号类型
  - 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
  - 动态类型对象的（可能是 `constant` 或 `volatile` 限定）基类类型
  - 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。
  - 每个存在的带符号整型对应（但是不相同）一个无符号整型。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。
  - 如果两个简单旧数据 (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。

A.2.107.1 缺省值

如果未指定 `-xalias_level`，则编译器将该选项设置为 `-xalias_level=any`。如果指定了 `-xalias_level` 但未提供值，则编译器将该选项设置为 `-xalias_level=compatible`。

交互

编译器在 `-xO2` 和更低的优化级别不执行基于类型的别名分析。

警告

如果要使用 `reinterpret_cast` 或等价的旧式强制类型转换，程序可能会违反分析假定。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.2.108 - xannotate[=yes|no]

(*Solaris*) 指示编译器创建二进制文件，之后这些文件可以由 binopt(1) 之类的二进制文件修改工具进行转换。此外，将来的二进制文件分析、代码覆盖和内存错误检测工具也可以处理使用此选项生成的二进制文件。

使用 `-xannotate=no` 选项可阻止这些工具对二进制文件进行修改。

-xannotate=yes 选项必须与优化级别 -xO1 或更高级别一起使用才有效，且仅在具有新的链接程序支持库接口 -ld_open() 的系统上有效。如果在没有此链接程序接口的操作系统（如 Solaris 9 OS）上使用编译器，编译器会默认恢复为 -xannotate=no。Solaris 10 修补程序 127111-07、Solaris 10 Update 5 和 OpenSolaris 中提供了新的链接程序接口。

缺省值为 -xannotate=yes，但如果不符合以上任一条件，则缺省值恢复为 -xannotate=no。

此选项在 Linux 平台上不可用。

A.2.109 –xar

创建归档库。

生成使用模板的 C++ 归档文件时，必须将模板系统信息库中实例化的那些模板函数包括在该归档文件中。仅在使用 -instances=extern 选项编译了至少一个目标文件时才使用模板系统信息库。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。

A.2.109.1 值

对调用 ar -c -r 指定 -xar 并重新创建归档文件。

示例

以下命令行归档包含在库和目标文件中的模板函数。

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

警告

请勿在命令行上添加来自模板数据库中的 .o 文件。

请勿直接使用 ar 命令生成归档文件。应使用 CC–xar 以确保模板实例自动包括在归档文件中。

另请参见

ar(1) 和表 15–3

A.2.110 –xarch=*isa*

指定目标指令集体系结构 *(instruction set architecture, ISA)*。

该选项将编译器生成的代码限制为特定指令集体系结构的指令。此选项不保证使用任何特定于目标的指令。不过，使用该选项会影响二进制程序的可移植性。

注 – 分别使用 -m64 或 -m32 选项来指定打算使用的内存模型 LP64（64 位）或 ILP32（32 位）。-xarch 选项不再指示内存模型，除非是为了与早期的发行版兼容，如下所示。

如果在不同的步骤中编译和链接，请确保在两个步骤中为 -xarch 指定了相同的值。有关在编译时和链接时都必须指定的所有编译器选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

A.2.110.1 用于 SPARC 的 -xarch 标志

下表提供了 SPARC 平台上每个 -xarch 关键字的详细信息。

表 A–23 用于 SPARC 平台的 -xarch 标志

标志	含义
generic	使用大多数处理器通用的指令集。这是缺省值，等效于使用 –m32 编译时的 v8plus，以及使用 –m64 编译时的 sparc。
generic64	为了在大多数 64 位 平台上获得良好性能而进行编译。（仅限 Solaris）。 该选项等效于 -m64 -xarch=generic，用于与早期的发行版兼容。应使用 -m64 指定 64 位编译，而不是 -xarch=generic64。
native	为了在此系统上获得良好性能而进行编译。编译器为运行它的当前系统处理器选择适当的设置。
native64	编译以在此系统中取得良好的性能（仅限 Solaris）。该选项等效于 -m64 -xarch=native，用于与早期的发行版兼容。
sparc	针对 SPARC-V9 ISA （但不带有可视化指令集 (Visual Instruction Set, VIS)，也不带有其他特定于实现的 ISA 扩展）进行编译。该选项在 V9 ISA 上使编译器生成高性能代码。
sparcvis	针对 SPARC-V9 加可视指令集 (Visual Instruction Set, VIS) 版本 1.0 进行编译，并具有 UltraSPARC 扩展。该选项在 UltraSPARC 体系结构上使编译器生成高性能代码。
sparcvis2	此选项允许编译器在具有 UltraSPARC III 扩展的 UltraSPARC 体系结构以及可视化指令集 (VIS) 2.0 版上生成目标代码。
sparcfmaf	允许编译器使用 SPARC-V9 指令集，加 UltraSPARC 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 1.0）、UltraSPARC-III 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 2.0）以及面向浮点乘加的 SPARC64 VI 扩展中的指令。 必须将 -xarch=sparcfmaf 与 fma=fused 结合使用，并具有某个优化级别，以使编译器尝试查找机会来自动使用乘加指令。

表 A–23 用于 SPARC 平台的 -xarch 标志 （续）

标志	含义
v7	（已废弃） 针对 SPARC-V7 ISA 进行编译。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。 缺省值为 -xarch=v8plus。 示例： SPARCstation 1, SPARCstation 2.
v8a	（已过时） 针对 SPARC-V8 ISA 的 V8a 版本进行编译。按照定义，V8a 是指不包含 fsmuld 指令的 V8 ISA。 该选项使编译器能够生成可在 V8a ISA 上获得良好性能的代码。 示例： 基于 microSPARC I 芯片架构的任何系统
v8	（已过时） 针对 SPARC-V8 ISA 进行编译。使编译器能够生成用于在 V8 架构上获得良好性能的代码。示例： SPARCstation 10
v8plus	针对 SPARC-V9 ISA 的 V8plus 版本进行编译。根据定义，V8plus 是指 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 (Visual Instruction Set, VIS) 和其他特定实现的 ISA 扩展。 ■ 该选项使编译器能够生成可在 V8plus ISA 上获得良好性能的代码。 ■ 生成的目标代码采用 SPARC-V8+ ELF32 格式，只能在 Solaris UltraSPARC 环境下执行（不能在 V7 和 V8 处理器上运行）。 示例： 基于 UltraSPARC 芯片体系结构的任何系统
v8plusa	针对 SPARC-V9 ISA 的 V8plusa 版本进行编译。根据定义，V8plusa 是指 V8plus 体系结构加可视化指令集 (Visual Instruction Set,VIS) 版本 1.0 和 UltraSPARC 扩展。 ■ 该选项使编译器能够生成可在 UltraSPARC 体系结构上获得良好性能的代码，但只限于 V8plus 规范定义的 32 位子集。 ■ 生成的目标代码采用 SPARC-V8+ ELF32 格式，只能在 Solaris UltraSPARC 环境下执行（不能在 V8 处理器上运行）。 示例： 基于 UltraSPARC 芯片体系结构的任何系统

表 A–23 用于 SPARC 平台的 -xarch 标志 （续）

标志	含义
v8plusb	针对具有 UltraSPARC III 扩展的 SPARC-V8plus ISA 的 V8plusb 版本进行编译。 此选项允许编译器在具有 UltraSPARC III 扩展的 UltraSPARC 体系结构以及可视化指令集 (VIS) 2.0 版上生成目标代码。 ■ 生成的目标代码采用 SPARC-V8+ ELF32 格式，只能在 Solaris UltraSPARC-III 环境中执行。 ■ 使用此选项进行编译将使用最佳指令集，以便在 UltraSPARC III 体系结构上获得良好性能。
v9	等效于 -m64 -xarch=sparc。使用 -xarch=v9 来获取 64 位内存模型的传统 makefile 和脚本仅需使用 -m64。
v9a	等效于 -m64 -xarch=sparcvis，用于与早期发行版兼容。
v9b	等效于 -m64 -xarch=sparcvis2，用于与早期发行版兼容。

另请注意：

- SPARC 指令集体系结构 V7、V8 和 V8a 已过时，不应使用。
- 可以一起链接和执行使用 v8plus 和 v8plusa 编译的二进制目标文件 (.o)，但只能在 SPARC V8plusa 兼容平台上运行。
- 可以一起链接和执行使用 v8plus、v8plusa 和 v8plusb 编译的二进制目标文件 (.o)，但只能在 SPARC V8plusb 兼容平台上运行。
- -xarch 值 v9、v9a 和 v9b 只能在 UltraSPARC 64 位 Solaris 操作系统中使用。
- 可以一起链接和执行使用 generic64、native64、v9 和 v9a 编译的二进制目标文件 (.o)，但只能在 SPARC V9a 兼容平台上运行。
- 可以一起链接和执行使用generic64、native64、v9、v9a 和 v9b 编译的二进制目标文件 (.o)，但只能在 SPARC V9b 兼容平台上运行。

对于任何特定选择，生成的可执行文件在早期体系结构中运行时都会慢得多。此外，虽然其中多数指令集体系结构中都有四精度 (long double) 浮点指令，但编译器并不在其生成的代码中使用这些指令。

A.2.110.2 用于 x86 的 -xarch 标志

下表列出了 x86 平台上的 -xarch 标志。

表 A–24 x86 上的 -xarch 标志

标志	含义
amd64	等效于 -m64 -xarch=sse2（仅限 Solaris）。使用 -xarch=amd64 来获取 64 位内存模型的传统 makefile 和脚本仅需要使用 -m64。
amd64a	等效于 -m64 -xarch=sse2a（仅限 Solaris）。
generic	使用大多数处理器通用的指令集。这是缺省值，等效于使用 –m32 编译时的 pentium_pro，以及使用 –m64 编译时的 sse2。
generic64	为了在大多数 64 位平台上获得良好性能而进行编译。（仅限 Solaris）。该选项等效于 -m64 -xarch=generic，用于与早期的发行版兼容。应使用 -m64 指定 64 位编译，而不是 -xarch=generic64。
native	为了在此系统上获得良好性能而进行编译。编译器为运行它的当前系统处理器选择适当的设置。
native64	编译以在此系统中取得良好的性能（仅限 Solaris）。该选项等效于 -m64 -xarch=native，用于与早期的发行版兼容。
pentium_pro	使指令集限于 32 位 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 SSE 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 SSE2 体系结构中。
sse3	将 SSE3 指令集添加到 SSE2 指令集中。
ssse3	为 pentium_pro、SSE、SSE2 和 SSE3 指令集补充 SSSE3 指令集。
sse4_1	为 pentium_pro、SSE、SSE2、SSE3 和 SSSE3 指令集补充 SSE4.1 指令集。
sse4_2	为 pentium_pro、SSE、SSE2、SSE3、SSSE3 和 SSE4.1 指令集补充 SSE4.2 指令集。

A.2.110.3 x86 特殊注意事项

针对 x86 Solaris 平台进行编译时，有一些重要问题值得注意。

传统的 Sun 样式的并行 pragma 在 x86 上不可用。而改用 OpenMP。有关将传统并行化指令转换为 OpenMP 的信息，请参见《Sun Studio OpenMP API 用户指南》。

xarch 设置为 -sse、sse2、sse2a、sse3 或更高时编译的程序只能在提供这些扩展和功能的平台上运行。

在 Pentium 4 兼容的平台上 Solaris OS 发行版支持 SSE/SSE2。早期版本的 Solaris OS 不支持 SSE/SSE2。如果所运行的 Solaris OS 不支持由 -xarch 选定的指令集，则编译器无法为该指令集生成链接代码。

如果在单独的步骤中编译和链接，请始终使用编译器以及相同的 -xarch 设置进行链接，以确保链接正确的启动例程。

x86 上的数值结果可能与 SPARC 上的结果不同，这是由 x86 80 位浮点寄存器造成的。为了最大限度减少这些差异，请使用 -fstore 选项或使用 -xarch=sse2 进行编译（如果硬件支持 SSE2）。

Solaris 和 Linux 之间的数值结果也可能不同，因为内部数学库（例如，sin(x)）并不相同。

A.2.110.4 二进制兼容验证

从 Sun Studio 11 和 Solaris 10 OS 开始，会对使用这些专用的 -xarch 硬件标志编译和生成的程序二进制文件进行验证，看其是否在适当的平台上运行。

在 Solaris 10 之前的系统中，不执行任何验证，用户负责确保使用这些标志生成的对象部署在合适的硬件上。

如果在没有相应功能或指令集扩展的平台上运行使用这些 -xarch 选项编译的程序，则可能会导致段故障或不正确的结果，并且不显示任何显式警告消息。

这一警告也扩展到使用 .il 内联汇编语言功能或使用 SSE、SSE2、SSE2a 和 SSE3 指令和扩展的 __asm() 汇编程序代码。

A.2.110.5 交互

尽管可以单独使用该选项，但它是 -xtarget 选项的扩展的一部分，并且可用于覆盖由特定的 -xtarget 选项设置的 -xarch 值。例如，-xtarget=ultra2 可扩展为 -xarch=v8plusa -xchip=ultra2 -xcache=16/32/1:512/64/1。在以下命令中，-xarch=v8plusb 覆盖了由 -xtarget=ultra2 的扩展设置的 -xarch=v8plusa。

```
example% CC -xtarget=ultra2 -xarch=v8plusb foo.cc
```

不支持 –compat[=4] 与 -xarch=generic64、-xarch=native64、-xarch=v9、-xarch=v9a 或 -xarch=v9b 结合使用。

A.2.110.6 警告

如果在进行优化时使用该选项，那么在指定体系结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。

如果在不同的步骤中编译和链接，请确保在两个步骤中为 -xarch 指定了相同的值。

A.2.111 -xautopar

注 – 此选项不接受 OpenMP 并行化指令。Sun 特定的 MP pragma 已过时，并且不再受支持。有关标准的指令的迁移信息，请参见《Sun Studio OpenMP API 用户指南》。

(SPARC) 为多个处理器启用自动并行化，执行依赖性分析（对循环进行迭代间数据依赖性分析）和循环重构。如果优化级别不是 -xO3 或更高，则将优化级别提高到 -xO3 并发出警告。

如果要进行自己的线程管理，请勿使用 -xautopar。

为了使执行速度更快，该选项要求使用多处理器系统。在单处理器系统中，生成的二进制文件的运行速度通常较慢。

要在多线程环境中运行已并行化的程序，必须在执行之前设置 OMP_NUM_THREADS环境变量。有关更多信息，请参见《Sun Studio OpenMP API 用户指南》。

如果使用 -xautopar 且在一个步骤中进行编译和链接，则链接会自动将微任务化库和线程安全的 C 运行时库包含进来。如果使用 -xautopar 并在不同的步骤中进行编译和链接，则还必须使用 -xautopar 进行链接。

A.2.111.1 另请参见

第 301 页中的 “A.2.158 -xopenmp[= *i*]”

A.2.112 -xbinopt={prepare| off}

(SPARC) 指示 编译器准备二进制文件，以便以后进行优化、转换和分析，请参见 binopt(1)。此选项可用于生成可执行文件或共享对象。如果在不同的步骤中进行编译，则在编译步骤和链接步骤中都必须有 -xbinopt：

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

如果有些源代码不可用于编译，仍可使用此选项来编译其余代码。然后，应将其用于可创建最终库的链接步骤中。在此情况下，只有用此选项编译的代码才能进行优化、转换或分析。

A.2.112.1 缺省值

缺省值为 -xbinopt=off。

交互

此选项必须与 -xO1 或更高的优化级别一起使用时才有效。使用此选项生成二进制文件时，文件大小会有所增加。

使用 -xbinopt=prepare 和 -g 编译会将调试信息包括在内，从而增加可执行文件的大小。

A.2.113 -xbuiltin[={ %all|%none}]

启用或禁用对标准库调用进行优化改进。

缺省情况下，编译器会将标准库头文件中声明的函数视为普通函数。但编译器会将其中一些函数识别为“内部函数”或“内置函数”。视为内置函数时，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。有关如何读取目标文件中的编译器注解来确定编译器实际对哪些函数进行替换的说明，请参见 er_src(1) 手册页。

-xbuiltin=%all 选项表示要求编译器识别尽可能多的内置标准函数。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 选项表示采用缺省编译器行为，编译器对内置函数不进行任何特殊优化。

A.2.113.1 缺省值

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。请注意，宏 -fast 的扩展包括了 -xbuiltin=%all。

```
example% CC -fast -xbuiltin=%none -c foo.cc
```

A.2.114 –xcache=*c*

定义要由优化器使用的高速缓存属性。此选项不保证使用每个特定的缓存属性。

注 – 尽管可单独使用该选项，但它是 `-xtarget` 选项的扩展的一部分，其主要用途是覆盖 `-xtarget` 选项提供的值。

此发行版引入一个可选属性 `[/ti]`，该属性用来设置可以共享缓存的线程数。

A.2.114.1 值

c 必须是下列值之一。

表 A–25 `-xcache` 值

值	含义
`generic`	这是缺省值，该值指示编译器使用能达到以下效果的缓存属性：多数 x86 和 SPARC 处理器上都能获得良好性能，同时任何处理器性能都不会明显下降。 如果需要，在每个新的发行版本中都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*[/*t*1]	定义级别 1 缓存属性
s1/*l1*/*a1*[/*t*1]:*s2*/*l2*/*a2*[/*t*2]	定义级别 1 和 2 缓存属性
s1/*l1*/*a1*[/*t*1]:*s2*/*l2*/*a2*[/*t*2]:*s3*/*l3*/*a3*[/*t*3]	定义级别 1、2 和 3 缓存属性

高速缓存属性 `s`*i*`/l`*i*`/a`*i*`/t`*i* 的定义如下：

属性	定义
`si`	级别为 *i* 时的数据高速缓存的**大小** (KB)
`li`	级别为 *i* 时的数据高速缓存的**行大小**（字节）
`ai`	级别为 *i* 时的数据高速缓存的**关联性**

例如，*i*=1 指定 1 级高速缓存属性 *s1*/*l1*/*a1*。

缺省值

如果未指定 –xcache，则假定为缺省值 –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。

如果没有为 t 指定值，则缺省值为 1。

示例

–xcache=16/32/4:1024/32/1 指定以下内容：

级别 1 高速缓存具有	级别 2 高速缓存具有
16 KB	1024 KB
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.2.115 -xcg[89|92]

(SPARC) 已废弃，请勿使用此选项。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。应改用 -O，并利用 -xarch、-xchip 和 -xcache 编译器缺省值。

A.2.116 -xchar[=*o*]

提供此选项只是为了方便从 char 类型定义为 unsigned 的系统中迁移代码。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带符号或者无符号。

A.2.116.1 值

o 可以是下列值之一：

表 A–26 -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。这会影响已编译代码的行为，而不影响库例程的行为。
s	与 signed 等效。
unsigned	将声明为字符的字符常量和变量视为无符号的。这会影响已编译代码的行为，而不影响库例程的行为。
u	与 unsigned 等效。

缺省值

如果未指定 -xchar，编译器将假定 -xchar=s。

如果指定了 -xchar 但未指定值，编译器将假定 -xchar=s。

交互

-xchar 选项仅会更改使用 -xchar 编译的代码中 char 类型的值范围。该选项不会更改任何系统例程或头文件中 char 类型的值范围。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 char 中可编码的值的范围。

警告

如果使用 -xchar=unsigned，则在将 char 与预定义的系统宏进行比较时要特别小心，因为宏中的值可能带符号。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 char 与此类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 为通过库导出的任何接口编译例程。Solaris ABI 将 char 类型指定为带符号，并且系统库的行为也与此相适应。目前还未对系统库针对将 char 指定为无符号的效果进行广泛测试。可以不使用该选项，而是修改代码使其与 char 类型是否带符号没有关联。char 类型的符号随编译器和操作系统的不同而不同。

A.2.117 -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以及多线程程序中的从属线程栈进行栈溢出运行时检查。如果检测到栈溢出，则生成 SIGSEGV。如果您的应用程序需要以不同于处理其他地址空间违规的方式处理栈溢出导致的 SIGSEGV，请参见 sigaltstack(2)。

A.2.117.1 值

i 必须是下列值之一：

表 A–27 -xcheck 值

值	含义
%all	执行全部检查。
%none	不执行检查。
stkovf	打开栈溢出检查。
no%stkovf	关闭栈溢出检查。

缺省值

如果未指定 -xcheck，则编译器缺省使用 -xcheck=%none。

如果指定了没有任何参数的 -xcheck，则编译器缺省使用 -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.2.118 -xchip=*c*

指定要由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
- 语义上等价的其他指令可用时使用的指令

> **注 –** 尽管可单独使用该选项，但它是 -xtarget 选项的扩展的一部分，其主要用途是覆盖 -xtarget 选项提供的值。

A.2.118.1 值

c 必须是下列值之一。

表 A–28 -xchip 值

平台	值	使用定时属性优化
SPARC	generic	可在大多数 SPARC 处理器上获得高性能
	native	可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
	sparc64vi	SPARC64 VI 处理器
	sparc64vii	SPARC64 VII 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
	ultra4	UltraSPARC IV 处理器
	ultra4plus	UltraSPARC IVplus 处理器
	ultraT1	UltraSPARC T1 处理器
	ultraT2	UltraSPARC T2 处理器
	ultraT2plus	UltraSPARC T2+ 处理器
x86	generic	大多数 x86 处理器
	core2	Intel Core2 处理器

表 A–28 -xchip 值 （续）

平台	值	使用定时属性优化
	nehalem	Intel Nehalem 处理器
	opteron	AMD Opteron 处理器
	penryn	Intel Penryn 处理器
	pentium	Intel Pentium 处理器
	pentium_pro	Intel Pentium Pro 处理器
	pentium3	Intel Pentium 3 式处理器
	pentium4	Intel Pentium 4 式处理器
	amdfam10	AMD AMDFAM10 处理器

缺省值

在大多数处理器中，generic 为缺省值，即指示编译器使用最佳定时属性以获得高性能，而不会显著降低任何处理器的性能。

A.2.119 –xcode=*a*

SPARC： 指定代码地址空间。

注 – 应通过指定 -xcode=pic13 或 -xcode=pic32 生成共享对象。未使用 pic13 或 pic32 生成的共享对象不能正常工作，也可能根本无法生成。

A.2.119.1 值

a 必须是下列值之一。

表 A–29 -xcode 值

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 的大小不应超过 2**44 字节。只适用于 64 位架构。请勿将该值与动态（共享）库一起使用。
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。

表 A–29 -xcode 值 （续）

值	含义
pic13	生成快速但有范围限制的位置无关代码（小模型）。等效于 –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	生成与位置无关的代码（大模型），这可能没有 pic13 快，但有完整范围。等效于 –KPIC。允许在 32 位体系结构上最多引用 2**30 个唯一的外部符号，而在 64 位体系结构上最多可以引用 2**29 个。

要确定是使用 –xcode=pic13 还是使用 –xcode=pic32，应使用 elfdump -c（有关更多信息，请参见 elfdump(1) 手册页）检查全局偏移表 (Global Offset Table, GOT) 的大小以及 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，应根据以下准则来确定如何使用 -xcode：

- 如果要生成可执行文件，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。
- 如果要生成共享库，先使用 –xcode=pic13，一旦 GOT 大小超过 8,192 字节，就使用 -xcode=pic32。
- 如果要生成用于链接到共享库的归档库，则应该使用 -xcode=pic32。

缺省值

对于 32 位体系结构，缺省值是 -xcode=abs32。64 位体系结构的缺省值是 -xcode=abs44。

生成共享动态库时，缺省 -xcode 值 abs44 和 abs32 将与 64 位体系结构一起使用。但指定 -xcode=pic13 或 -xcode=pic32。在 SPARC 上使用 –xcode=pic13 和 –xcode=pic32 时存在两项名义性能开销：

- 用 –xcode=pic13 或 –xcode=pic32 编译的例程会在入口点执行一些附加指令，以将寄存器设置为指向用于访问共享库的全局变量或静态变量的表 (_GLOBAL_OFFSET_TABLE_)。
- 对全局或静态变量的每次访问都会涉及通过 _GLOBAL_OFFSET_TABLE_ 的额外间接内存引用。如果是使用 -xcode=pic32 进行编译，则对于每个全局和静态内存引用还要执行另外两个指令。

在考虑上述成本时，请记住：由于受到库代码共享的影响，使用 -xcode=pic13 和 -xcode=pic32 会大大减少系统内存需求。共享库中使用 -xcode=pic13 或 – xcode=pic32 编译的每页代码都可以供使用该库的每个进程共享。如果共享库中的代码页包含非 pic（即绝对）内存引用，即使仅包含单个非 pic 内存引用，该页也将变为不可共享，而且每次执行使用该库的程序时都必须创建该页的副本。

确定是否已经使用 -xcode=pic13 或–xcode=pic32 编译了 .o 文件的最简单方法是使用 nm 命令：

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

包含与位置无关的代码的 .o 文件将包含对 _GLOBAL_OFFSET_TABLE_ 无法解析的外部引用（用字母 U 标记）。

要确定是使用 –xcode=pic13 还是使用 –xcode=pic32，应使用 nm 确定库中使用或定义的不同全局变量和静态变量的数量。如果 _GLOBAL_OFFSET_TABLE_ 的大小小于 8,192 字节，就可以使用 -Kpic。否则，就必须使用 -xcode=pic32。

A.2.120 -xcrossfile[=*n*]

已废弃，不使用。改用 -xipo。

A.2.121 -xdebugformat=[stabs|dwarf]

编译器将调试器信息格式从 stabs 格式迁移为“DWARF 调试信息格式”中指定的 dwarf 格式。在此发行版中，缺省设置为 -xdebugformat=stabs。

如果要维护读取调试信息的软件，您现在可以选择将工具从 stabs 格式转换为 dwarf 格式。

出于移植工具的目的，可以通过此选项来使用新的格式。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，否则不需要使用此选项。

表 A–30 -xdebugformat 标志

值	含义
stabs	-xdebugformat=stabs 生成使用 stabs 标准格式的调试信息。
dwarf	-xdebugformat=dwarf 生成的调试信息采用 dwarf 标准格式。

如果未指定 -xdebugformat，编译器将假定 -xdebugformat=stabs。此选项需要一个参数。

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 和性能分析器软件可识别 stabs 和 dwarf 格式，因此使用此选项对任何工具的功能都没有影响。

注 – 这是过渡性接口，因此会在发行版之间发生更改而不兼容，即使在发行版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。

有关更多信息，另请参见 `dumpstabs`(1) 和 `dwarfdump`(1) 手册页。

A.2.122 -xdepend=[yes| no]

(SPARC) 对循环进行迭代间数据依赖性分析，并执行循环重构，包括循环交换、循环合并、标量替换和“死数组”赋值消除。

在 SPARC 上，对于所有优化级别 **-xO3** 以及更高级别，**-xdepend** 缺省为 **–xdepend=on**。否则，**–xdepend** 缺省为 **–xdepend=off**。指定 **-xdepend** 的显式设置会覆盖任何缺省设置。

在 x86 上，—xdepend 缺省为 **–xdepend=off**。指定 **–xdepend** 且优化级别不是 **–xO3** 或更高级别时，编译器会将优化级别提高到 **–xO3** 并发出警告。

指定不带参数的 **-xdepend** 等效于 **–xdepend=yes**。

-xautopar 中包含了依赖性分析。依赖性分析会在编译时进行。

依赖性分析在单处理器系统中可能很有用。但是，如果在单处理器系统上使用 **-xdepend**，不应该同时指定 **-xautopar**，因为将针对多处理器系统进行 **-xdepend** 优化。

另请参见：**-xprefetch_auto_type**

A.2.123 -xdumpmacros[= *value*[, *value*...]]

要查看宏在程序中的行为方式时使用此选项。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` 选项在整个文件中或在 `dumpmacros` 或 `end_dumpmacros` pragma 覆盖它之前都是有效的。请参见第 340 页中的 “B.2.5 #pragma dumpmacro s”。

A.2.123.1 值

value 可以是下列参数：

表 A–31 -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	设置该选项即表示 -xdumpmacros=defs,undefs,use,loc,conds,sys。该参数最好与 [no%] 形式的其他参数配合使用。例如，-xdumpmacros=%all,no%sys 表示输出中不包含系统头文件宏，但仍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 **效果相同**。

注 – 子选项 loc、conds 和 sys 是 defs、undefs 和 use 选项的限定符。使用 loc、conds 和 sys 本身并不会生成任何结果。例如，使用 -xdumpmacros=loc,conds,sys 不会生成什么结果。

缺省值

如果指定了没有任何参数的 -xdumpmacros，则表示 -xdumpmacros=defs,undefs,sys。如果未指定 -xdumpmacros，则缺省为 -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。但是，如果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会打印位置和宏名称。

例如以下文件 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
```

以下示例显示了使用 defs、undefs、sys 和 loc 参数时文件 t.c 的输出。

```
example% CC -c -xdumpmacros -DFOO t.c
#define __SunOS_5_9 1
#define __SUNPRO_CC 0x590
#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
command line: #define __SunOS_5_9 1
command line: #define __SUNPRO_CC 0x590
command line: #define unix 1
command line: #define sun 1
command line: #define sparc 1
command line: #define __sparc 1
command line: #define __unix 1
command line: #define __sun 1
command line: #define __BUILTIN_VA_ARG_INCR 1
command line: #define __SVR4 1
command line: #define __SUNPRO_CC_COMPAT 5
command line: #define __SUN_PREFETCH 1
command line: #define FOO 1
command line: #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
```

例如文件 `y.c`：

```
example% cat y.c
#define X 1
#define Y X
#define Z Y
int a = Z;
```

以下是使用 `-xdumpmacros=use,loc` 时有关 `y.c` 中宏的输出：

```
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.2.124 -xe

仅检查语法和语义错误。指定 `-xe` 时，编译器不生成任何目标代码。 `-xe` 的输出定向到 `stderr`。

如果不需要通过编译生成目标文件，可使用 `-xe` 选项。例如，如果要尝试通过删除代码段找出导致出现错误消息的原因，可使用 `-xe` 加速编辑和编译周期。

A.2.124.1 另请参见

–c

A.2.125 -xF[=*v*[,*v*...]]

启用通过链接程序对函数和变量进行最佳重新排列。

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指示）将这些段重新排序以优化程序性能。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。

对变量重新排序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不必要的过大工作集。
- 未用到的弱变量副本会造成不必要的过大工作集，从而降低有效数据密度。

为优化性能而对变量和函数进行重新排序时，需要执行以下操作：

1. 使用 -xF 进行编译和链接。
2. 按照《程序性能分析工具》手册中有关如何生成用于函数的映射文件的说明进行操作，或按照《链接程序和库指南》中有关如何生成用于数据的映射文件的说明进行操作。
3. 使用通过链接程序的 -M 选项生成的新映射文件重新链接。
4. 在分析器下重新执行以验证是否增强。

A.2.125.1 值

v 可以是下列其中一个或多个值：

表 A–32 -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。

缺省值

如果未指定 -xF，则缺省值为 -xF=%none。如果指定了没有任何参数的 -xF，则缺省值为 -xF=%none,func。

交互

使用 -xF=lcldata 会限制某些地址计算优化，因此，只应在必要时才使用该标志。

另请参见

analyzer(1)、debugger(1) 和 ld(1) 手册页

A.2.126 -xhelp=flags

显示了对每个编译器选项的简要描述。

A.2.127 -xhelp=readme

显示联机 readme 文件的内容。

readme 文件按环境变量 PAGER 中指定的命令分页。如果未设置 PAGER，则缺省的分页命令为 more。

A.2.128 -xhwcprof

(SPARC) 使编译器支持基于硬件计数器的文件配置。

如果启用了 -xhwcprof，编译器将生成信息，这些信息可帮助工具将文件配置的加载和存储指令与其所引用的数据类型和结构成员相关联（与由 -g 生成的符号信息一起）。它将配置文件数据同目标文件的数据空间（而不是指令空间）相关联，并对行为进行洞察，而这仅从指令配置中是无法轻易获得的。

可使用 -xhwcprof 编译一组指定的目标文件。但是，-xhwcprof 在应用于应用程序中的所有目标文件时，作用最大。它能全面识别并关联分布在应用程序目标文件中的所有内存引用。

如果分别在单独的步骤中进行编译和链接，最好在链接时使用 -xhwcprof。如果将来扩展为 -xhwcprof，则在链接时可能需要使用它。

-xhwcprof=enable 或 -xhwcprof=disable 的实例将会覆盖同一命令行中 -xhwcprof 的所有以前的实例。

在缺省情况下，禁用 -xhwcprof。指定不带任何参数的 -xhwcprof 与 -xhwcprof=enable 等效。

-xhwcprof 要求启用优化并将调试数据的格式设置为 DWARF (-xdebugformat=dwarf)。

组合使用 -xhwcprof 和 -g 会增加编译器临时文件的存储需求，而且高于单独指定 -xhwcprof 和 -g 所引起的增加总量。

下列命令可编译 example.cc，并可为硬件计数器文件配置以及针对使用 DWARF 符号的数据类型和结构成员的符号分析指定支持：

```
example% CC -c -O -xhwcprof -g -xdebugformat=dwarf example.cc
```

有关基于硬件计数器的文件配置的更多信息，请参见《程序性能分析工具》手册。

A.2.129 -xia

链接适当的区间运算库，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。

在 x86 平台上，需要支持 SSE2 指令集。

A.2.129.1 扩展

-xia 选项是一个扩展到 -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 设置的内容，则可能会导致编译器出现不正确的行为。

交互

要使用区间运算库，请将 <suninterval.h> 包含进来。

在使用区间运算库时，必须包括以下库之一： libC、Cstd 或 iostream。有关包括这些库的信息，请参见 -library。

警告

如果您使用区间并为 -fsimple、-ftrap 或 -fns 指定了不同的值，则您的程序可能有不正确的行为。

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

-library

A.2.130 -xinline[=*func_spec*[,*func_spec*...]]

指定在 -xO3 或更高的优化级别优化器可以内联用户编写的哪些例程。

A.2.130.1 值

func_spec 必须是下列值之一。

表 A–33 `-xinline` 值

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。请注意，如果没有指定 `%auto`，则在命令行上使用 `-xinline=[no%]`*func_name*... 指定显式内联后，通常会禁用自动内联。
func_name	强烈请求优化器内联函数。如果函数未声明为 extern "C"，则必须损坏 *func_name* 的值。可以对可执行文件使用 `nm` 命令来查找损坏的函数名。对于已声明为 extern "C" 的函数，编译器不损坏名称。
`no%`*func_name*	如果为列表上的例程添加名称前缀 `no%`，则会禁止内联该例程。关于 *func_name* 的损坏名称的规则也适用于 `no%`*func_name*。

只有使用了 `-xipo[=1|2]` 时，才会内联要编译的文件中的例程。优化器决定适合内联的例程。

缺省值

如果未指定 `-xinline` 选项，则编译器假定 `-xinline=%auto`。

如果指定了没有任何参数的 `-xinline=`，则不内联函数，而不管优化级别是什么。

示例

要启用自动内联同时禁用内联声明为 `int foo()` 的函数，请使用

```
example% CC -xO5 -xinline=%auto,no%__1cDfoo6F_i_ -c a.cc
```

要强烈要求内联声明为 `int foo()` 的函数，并使所有其他函数作为要内联的候选函数，请使用

```
example% CC -xO5 -xinline=%auto,__1cDfoo6F_i_ -c a.cc
```

要强烈要求内联声明为 int foo() 的函数，且不允许内联任何其他函数，请使用

```
example% CC -xO5 -xinline=__1cDfoo6F_i_ -c a.cc
```

交互

优化级别低于 `-xO3` 时，`-xinline` 选项不起作用。在 `-xO4` 或更高的优化级别上，优化器会决定应该内联哪些函数，无需指定 `-xinline` 选项即可完成。另外，在 `-xO4` 或更高的优化级别上，编译器会尝试确定内联哪些函数可以提高性能。

如果出现以下任一情况，则会内联例程。

- 优化级别为 -xO3 或更高
- 认为内联有益且安全
- 函数在要编译的文件中，或函数在使用 -xipo[=1|2] 编译了的文件中

警告

如果使用 -xinline 强制内联函数，实际上可能会降低性能。

另请参见

第 288 页中的 “A.2.136 -xldscope={*v*}”

A.2.131 -xinstrument=[no%]datarace

指定此选项编译您的程序并为其提供程序设备，以供线程分析器进行分析。有关线程分析器的更多详细信息，请参见 tha(1)。

然后可使用性能分析器以 collect -r races 来运行此检测的程序，从而创建数据竞争检测实验。可以单独运行已提供了程序设备的代码，但其运行速度将非常缓慢。

可指定 -xinstrument=no%datarace 来关闭线程分析器的源代码准备。这是缺省值。

指定 -xinstrument 而不带参数是非法操作。

如果在不同的步骤中进行编译和链接，则在编译和链接步骤都必须指定 -xinstrument=datarace。

此选项定义了预处理程序令牌 __THA_NOTIFY。可指定 #ifdef __THA_NOTIFY 来保护对 libtha(3) 例程的调用。

该选项也设置 -g。

A.2.132 -xipo[={0|1|2}]

执行过程间优化。

-xipo 选项通过调用过程间分析传递来执行部分程序优化。它会在链接步骤中对所有目标文件执行优化，且优化不限于只是编译命令中的那些源文件。但是，使用 -xipo 执行的整个程序优化不包括汇编 (.s) 源文件。

编译和链接大型多文件应用程序时，-xipo 选项特别有用。用该标志编译的对象目标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代码和预编译的程序文件中的过程间分析。但分析和优化只限于使用 -xipo 编译的目标文件，并不扩展到目标文件或库。

A.2.132.1 值

-xipo 选项可以是下列值。

表 A–34 -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

缺省值

如果未指定 -xipo，则假定 -xipo=0。

如果仅指定了 -xipo，则假定 -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。这在链接的最后一步完成，因此不必在一次编译所有源文件，可以分多次单独进行编译，每次编译时都指定 -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目标文件具有在文件内部编译的附加分析信息，这样就可以在链接步骤中执行跨文件优化。

交互

-xipo 选项要求优化级别至少为 -xO4。

不能在同一编译器命令行上同时使用 -xipo 选项和 -xcrossfile 选项。

警告

在不同的步骤中进行编译和链接时，必须在这两个步骤中都指定 -xipo 才有效。

没有使用 -xipo 编译的对象可以自由地与使用 -xipo 编译的对象链接。

即使使用 -xipo 对库进行了编译，这些库也不参与交叉文件的过程间分析，如以下示例中所示。

```
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
```

本示例中，在 one.cc、two.cc 和 three.cc 之间以及 main.cc 和 four.cc 之间执行过程间优化，但不在 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。）

由于执行跨文件优化时需要附加信息，因此 -xipo 选项会生成更大的目标文件。不过，该附加信息不会成为最终的二进制可执行文件的一部分。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.2.132.2 何时不使用 -xipo 过程间分析

在链接步骤中使用目标文件集合时，编译器试图执行整个程序分析和优化。对于该目标文件集合中定义的任何函数或子例程 foo()，编译器做出以下两个假定：

- 运行时在该目标文件集合之外定义的其他例程不显式调用 foo()。
- 目标文件集合中的任何例程调用 foo() 时，不会插入该目标文件集合之外定义的不同版本的 foo()。

如果假定 1 对于给定的应用程序不成立，请勿使用 -xipo=2 进行编译。

如果假定 2 不成立，请不要使用 -xipo=1 也不要使用 -xipo=2 进行编译。

例如，如果对函数 malloc() 创建了您自己的版本，并使用 -xipo=2 进行编译。这样，对于任何库中引用 malloc() 且与您的代码链接的所有函数，都必须使用 -xipo=2 进行编译，并且需要在链接步骤中对其目标文件进行操作。由于这对于系统库不大可能，因此不要使用 -xipo=2 编译您的 malloc 版本。

另举一例，如果生成了一个共享库，有两个外部调用（foo() 和 bar()）分别在两个不同的源文件中。并假设 bar() 调用 foo()。如果可能会在运行时插入 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，从而导致出现错误的结果。

另请参见

-xjobs

A.2.133 -xipo_archive=[*a*]

-xipo_archive 选项使编译器可在生成可执行文件之前将传递给链接程序的目标文件与使用 -xipo 编译的目标文件以及归档库 (.a) 中的目标文件一起优化。库中包含的在编译期间优化的任何目标文件都会替换为其优化后的版本。

a 是以下项之一：

表 A–35 -xipo_archive 标志

值	含义
writeback	在生成可执行文件之前，编译器将传递给链接程序的目标文件与归档库 (.a) 中已使用 -xipo 编译的目标文件一起优化。库中包含的在编译期间优化的任何目标文件都会替换为优化后的版本。 对于使用归档库通用集的并行链接，每个链接都应创建自己的归档库备份，从而在链接前进行优化。
readonly	在生成可执行文件之前，编译器将传递给链接程序的目标文件与归档库 (.a) 中已使用 -xipo 编译的目标文件一起优化。 通过 -xipo_archive=readonly 选项，可在链接时指定的归档库中进行对象文件的跨模块内联和程序间数据流分析。但是，它不启用对归档库代码的跨模块优化，除非代码已经通过跨模块内联插入到其他模块中。 要对归档库内的代码应用跨模块优化，要求 -xipo_archive=writeback。注意，这样做将修改从中提取代码的归档库的内容。
none	这是缺省值。没有对归档文件的处理。编译器不对使用 -xipo 编译和在链接时从归档库中提取的对象文件应用跨模块内联或其他跨模块优化。要执行此操作，链接时必须指定 -xipo，以及 -xipo_archive=readonly 或 -xipo_archive=writeback 中的任一个。

如果不为 -xipo_archive 指定设置，编译器会将其设置为 -xipo_archive=none。

指定不带标志的 -xipo_archive 是非法的。

A.2.134 -xjobs=*n*

指定 -xjobs 选项设置编译器为完成其工作创建的进程数。在多 CPU 计算机上，该选项可以缩短生成时间。目前，-xjobs 只能与 -xipo 选项一起使用。如果指定 -xjobs=*n*，过程间优化器将使用 *n* 作为在编译不同的文件时它能调用的最大代码生成器实例数。

A.2.134.1 值

指定 `-xjobs` 时务必要指定值。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 乘以可用处理器数。如果使用的值是可用处理器数的数倍，则会降低性能，因为有在产生的作业间进行的上下文切换开销。此外，如果使用很大的数值会耗尽系统资源（如交换空间）。

缺省值

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的编译，速度比使用相同命令但没有 `-xjobs` 选项时进行的编译快。

```
example% CC -xipo -xO4 -xjobs=3 t1.cc t2.cc t3.cc
```

A.2.135 -xlang=*language*[,*language*]

包含适当的运行时库，并确保指定语言的适当运行时环境。

A.2.135.1 值

language 必须是 `f77`、`f90`、`f95` 或 `c99`。

`f90` 和 `f95` 参数等价。`c99` 参数表示为已使用 `-xc99=%all` 编译并要使用 CC 链接的对象调用 ISO 9899:1999 C 编程语言行为。

交互

`-xlang=f90` 和 `-xlang=f95` 选项隐含了 `-library=f90`，而 `-xlang=f77` 选项隐含了 `-library=f77`。但要进行混合语言链接，只使用 `-library=f77` 和 `-library=f90` 选项是不够的，因为只有 `-xlang` 选项才能确保适当的运行时环境。

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++ 目标文件链接在一起时，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 目标文件。

```
example% CC -xlang=f95...
```

要链接 Fortran 95 和 Fortran 77 目标文件，请使用如下所示的 Fortran 95 驱动程序。

```
example% f95 -xlang=f77...
```

不能在同一编译器命令中同时使用 `-xlang` 选项和 `-xlic_lib` 选项。如果要使用 `-xlang` 且需要在 Sun 性能库中进行链接，应改用 `-library=sunperf`。

警告

请勿将 `-xnolib` 与 `-xlang` 一起使用。

如果要将并行的 Fortran 对象与 C++ 对象混合，链接行必须指定 `-mt` 标志。

另请参见

`-library` 和 `-staticlib`

A.2.136 -xldscope={*v*}

可指定 `-xldscope` 选项来更改外部符号定义的缺省链接程序作用域。由于更好的隐藏了实现，所以对缺省的更改会产生更快速更安全的共享库和可执行文件。

A.2.136.1 值

v 必须是下列值之一：

表 A–36 `-xldscope` 值

值	含义
`global`	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。
`symbolic`	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`。尽管不能将 `-Bsymbolic` 与 C++ 库一起使用，但可以使用 `-xldscope=symbolic`，而不会引起问题。有关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。

缺省值

如果未指定 -xldscope，编译器将假定 -xldscope=global。如果指定了没有任何值的 -xldscope，则编译器就会发出错误。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未以内联方式生成该函数。如果使用 -xinline 指定函数名、在可以自动进行内联的 -xO4 或更高优化级别进行编译、使用内联说明符或者要使用跨文件优化，编译器会内联函数。

例如，假定库 ABC 具有缺省的分配器函数，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，则编译器将内联库组件中出现的对 ABC_allocator 的调用。如果库的客户端要用定制的版本替换 ABC_allocator，在调用 ABC_allocator 的库组件中并不进行该替换。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 61 页中的 “4.1 链接程序作用域”。

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 和 -xcrossfile。

A.2.137 -xlibmieee

使 libm 在异常情况下对于数学例程返回 IEEE 754 值。

libm 的缺省行为是兼容 XPG。

A.2.137.1 另请参见

《数值计算指南》

A.2.138 -xlibmil

内联选定的 libm 库例程以进行优化。

注 – 该选项不影响 C++ 内联函数。

有一些可用于部分 `libm` 库例程的内联模板。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。

A.2.138.1 交互

`–fast` 选项隐含了该选项。

另请参见

`-fast` 和《数值计算指南》。

A.2.139 –xlibmopt

使用优化的数学例程库。使用此选项时，必须通过指定 `-fround=nearest` 来使用缺省的舍入模式。

此选项使用经过了性能优化的数学例程库，通常情况下，生成的代码运行速度较快。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。

A.2.139.1 交互

`–fast` 选项隐含了该选项。

另请参见

`–fast`、`–xnolibmopt` 和 `-fround`

A.2.140 –xlic_lib=sunperf

已过时，不使用。改为指定 `-library=sunperf`。有关更多信息，请参见第 235 页中的 “A.2.50 -library=*l*[,*l*...]”。

A.2.141 –xlicinfo

编译器忽略此选项且不显示任何提示。

A.2.142 -xlinkopt[=*level*]

指示编译器在对目标文件进行优化的基础上对生成的可执行文件或动态库执行链接时优化。这些优化在链接时通过分析二进制目标代码来执行。虽然未重写目标文件，但生成的可执行代码可能与初始目标代码不同。

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。优化器仍可以对未使用 -xlinkopt 进行编译的二进制目标文件执行部分受限的优化。

-xlinkopt 优化出现在编译器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。生成共享库时（使用 -G 编译），也可以使用 -xlinkopt。

A.2.142.1 值

level 设置执行的优化级别，必须为 0、1 或 2。优化级别如下：

表 A–37 -xlinkopt 值

值	含义
0	禁用链接优化器。（这是缺省情况。）
1	在链接时根据控制流分析执行优化，包括指令高速缓存着色和分支优化。
2	在链接时执行附加的数据流分析，包括无用代码删除和地址计算简化。

如果在不同的步骤中编译，-xbinopt 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，即使编译二进制目标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。

缺省值

指定 -xlinkopt 时若不带级别参数，则表示 -xlinkopt=1。

交互

当编译整个程序并且使用配置文件反馈时，该选项才最有效。文件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。这对大型应用程序尤为重要，因为在链接时执行代码优化放置可降低指令高速缓存未命中数。通常，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
```

有关使用配置文件反馈的详细信息，请参见第 314 页中的 “A.2.170 -xprofile=*p*”。

警告

使用 -xlinkopt 编译时，请不要使用 -zcombreloc 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，目标文件的大小也会增加，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.2.143 -xloopinfo

此选项用于显示并行化的循环和没有并行化的循环。它通常与 -xautopar 选项结合使用。

A.2.144 –xM

对指定的 C++ 程序只运行 C++ 预处理程序，同时请求预处理程序生成 makefile 依赖性并将结果发送到标准输出（有关 make 文件和依赖性的详细信息，请参见 make(1)）。

但是，-xM 只报告包含的头文件的依赖性，而不报告关联的模板定义文件的依赖性。可以使用 makefile 中的 .KEEP_STATE 功能在 make 实用程序创建的 .make.state 文件中生成所有依赖性。

A.2.144.1 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
```

交互

如果指定 -xM 和 -xMF，编译器会将所有 makefile 依赖性信息写入使用 -xMF 指定的文件。每次预处理程序向此文件写入时即将其覆写。

另请参见

make(1S)（了解有关 makefile 和依赖性的详细信息）。

A.2.145 -xM1

生成与 –xM 类似的 makefile 依赖性，只是它不报告 /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。

如果指定 -xM1 和 -xMF，编译器会将所有 makefile 依赖性信息写入使用 -xMF 指定的文件。每次预处理程序向此文件写入时即将其覆写。

A.2.146 -xMD

生成与 -xM 类似的 makefile 依赖性，但包括编译。-xMD 基于输入文件名但添加 .d 后缀生成 makefile 依赖性信息的输出文件。如果指定 -xMD 和 -xMF，预处理程序会将所有 makefile 依赖性信息附加到使用 -xMF 指定的文件。

A.2.147 -xMF

此选项用于指定 makefile 依赖性输出的文件。目前，没有方法可以在命令行上使用 -xMF 为多个输入文件指定单独的文件名。

A.2.148 -xMMD

此选项用于生成不包含系统头文件的 makefile 依赖性。它与 -xM1 的功能相同，只是还包括编译。-xMMD 基于输入文件名但添加 .d 后缀生成 makefile 依赖性信息的输出文件。如果同时指定了 -xMMD 和 -xMF，编译器将改用提供的文件名。

A.2.149 –xMerge

SPARC：将数据段和文本段合并。

目标文件中的数据是只读数据，并在进程之间共享，除非使用 ld-N 进行链接。

-xMerge -ztext -xprofile=collect 这三个选项不应该一起使用。**-xMerge** 强制将静态初始化的数据写入只读存储，而 **-ztext** 禁止只读存储中有与位置有关的符号重定位，**-xprofile=collect** 在可写存储中生成静态初始化的、与位置有关的符号重定位。

A.2.149.1 另请参见

ld(1) 手册页。

A.2.150 -xmaxopt[=*v*]

此命令将 pragma opt 的级别限制为指定的级别。*v* 可以为 off、1、2、3、4 或 5。缺省值为 -xmaxopt=off，表示忽略 pragma opt。如果指定 -xmaxopt 但未提供参数，相当于指定 - xmaxopt=5。

如果同时指定了 -xO 和 -xmaxopt，则使用 -xO 设置的优化级别不得超过 -xmaxopt 值。

A.2.151 -xmemalign=*ab*

(SPARC) 使用 -xmemalign 选项控制编译器对数据对齐所做的假定。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。

指定最大假定内存对齐和非对齐数据访问的行为。必须有一个同时用于 *a*（对齐）和 *b*（行为）的值。*a* 指定最大假定内存对齐，*b* 指定未对齐内存访问行为。

对于可在编译时确定对齐的内存访问，编译器会为该数据对齐生成适当的装入/存储指令序列。

对于不能在编译时确定对齐的内存访问，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。

A.2.151.1 值

下表列出了 -xmemalign 的对齐值和行为值。

表 A–38 -xmemalign 对齐值和行为值

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	仅限于 -xarch=v9 变体： 为小于或等于 4 的对齐产生信号 SIGBUS，否则解释访问并继续执行。对于其他所有 -xarch 值，f 标志与 i 等效。
8	假定最多 8 字节对齐。		
16	假定最多 16 字节对齐。		

如果要链接到某个已编译的目标文件，并且编译该目标文件时 *b* 的值设置为 i 或 f，就必须指定 -xmemalign。有关在编译时和链接时都必须指定的所有编译器选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

缺省值

以下缺省值仅适用于未使用 -xmemalign 选项时：

- -xmemalign=8i（适于所有 v8 体系结构）。
- -xmemalign=8s（适于所有 v9 体系结构）。

在有 -xmemalign 选项但未提供值时，缺省值为：

- -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–39 -xmemalign 示例

命令	情况
-xmemalign=1s	所有内存访问均未对齐，从而导致陷阱处理非常慢。
-xmemalign=8i	在发生错误的代码中存在偶然的、有目的的、未对齐访问。
-xmemalign=8s	程序中应该没有任何未对齐访问。
-xmemalign=2s	要检查可能存在的奇字节访问。
-xmemalign=2i	要检查可能存在的奇字节访问并要使程序工作。

A.2.152 -xmodel=[*a*]

(x86) -xmodel 选项使编译器可以针对 Solaris x86 平台修改 64 位对象形式，只应在要编译此类对象时指定该选项。

仅当启用了 64 位的 x64 处理器上还指定了 -m64 时，该选项才有效。

a 必须是以下值之一：

表 A–40 -xmodel 标志

值	含义
small	此选项可为小模型生成代码，其中执行代码的虚拟地址在链接时已知，并且已知在 0 到 2^31 - 2^24 - 1 的虚拟地址范围内可以找到所有符号。
kernel	按内核模型生成代码，在该模型中，所有符号都定义在 2^64 - 2^31 到 2^64 - 2^24 范围内。
medium	按中等模型生成代码，在该模型中，不对数据段的符号引用范围进行假定。文本段的大小和地址的限制与小型代码模型的限制相同。使用 -m64 编译时，具有大量静态数据的应用程序可能会要求 -xmodel=medium。

此选项不累积，因此编译器根据命令行最右侧的 -xmodel 实例设置模型值。

如果未指定 -xmodel，编译器将假定 -xmodel=small。如果指定没有参数的 -xmodel，将出现错误。

不必使用此选项编译所有转换单元。只有可以确保访问的对象在可访问范围之内，才可编译选择的文件。

您应了解，不是所有的 Linux 系统都支持中等模型。

A.2.153 –xnolib

禁用与缺省系统库链接。

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。使用该选项时，用于链接缺省系统支持库的 -l*lib* 不会传递给 ld。

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（缺省模式）下：

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

- 兼容模式 (-compat)：

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

- l 选项的顺序非常重要。-lm 选项必须位于 -lc 之前。

注 – 如果指定了 -mt 编译器选项，编译器通常先与 -lthread 链接，然后再与 -lm 链接。

要确定在缺省情况下将链接哪些系统支持库，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
```

A.2.153.1 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体系结构指令集的单线程应用程序中，请使用：

交互

如果指定了 – xnolib，就必须按给定顺序手动链接所有必需的系统支持库。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++ 语言功能要求使用 libC（兼容模式）或 libCrun（标准模式）。

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 和 –l

A.2.154 –xnolibmil

在命令行上取消 –xlibmil。

将该选项与 –fast 一起使用会忽略与优化数学库链接。

A.2.155 –xnolibmopt

不使用数学例程库。

A.2.155.1 示例

在命令行上 –fast 选项后面使用该选项，如下例中所示：

```
example% CC –fast –xnolibmopt
```

A.2.156 -xnorunpath

与 第 241 页中的 “A.2.63 –norunpath” 相同

A.2.157 -x0*level*

指定优化级别，请注意是大写字母 O 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。

在少数情况下，级别为 –x02 时的性能可能优于其他级别，–x03 也可能胜过 –x04。尝试用每个级别进行编译，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。优化器会以在 –x0*level* 选项中指定的初始级别恢复执行后续过程。

–x0 有五个级别。以下几节描述了在 SPARC 平台和 x86 平台上如何操作这些级别。

A.2.157.1 值

在 SPARC 平台上：

- –x01 只执行最小量的优化 (peephole)，也称为 postpass，即汇编级优化。除非使用 -x02 或 -x03 导致编译时间过长，或交换空间不足，否则请勿使用 -x01。
- –x02 执行基本的局部和全局优化，包括：
  - 归纳变量消除
  - 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。

–x03 除了执行在 –x02 级别所执行的优化外，还会对外部变量的引用和定义进行优化。该级别不跟踪指针赋值的结果。编译未通过 `volatile` 适当保护的设备驱动程序或修改来自信号处理程序中的外部变量的程序时，请使用 – x02。通常，使用此级别时会增加代码大小，除非将其与 `-xspace` 选项结合使用。

- –x04 除了执行 –x03 优化外，还自动内联同一文件中的多个函数。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，使用此级别时会增加代码大小，除非将其与 `-xspace` 选项结合使用。
- –x05 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。如果使用配置文件反馈执行该级别上的优化，则更容易提高性能。请参见第 314 页中的 “A.2.170 -xprofile=*p*”。

在 x86 平台上：

- –x01 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。
- –x02 执行局部通用子表达式消除、局部复制和常量传播、尾部递归消除以及级别 1 执行的优化。
- –x03 执行全局通用子表达式的消除、全局复制和常量传播、循环长度约简、归纳变量消除、循环变体优化以及级别 2 执行的优化。
- –x04 会自动内联同一文件中的多个函数，并执行级别为 3 时执行的优化。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。
- –x05 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

如果使用 `-g` 或 `-g0` 且优化级别是 `-x03` 或更低，编译器会为近乎完全优化提供尽可能多的符号信息。尾部调用优化和后端内联被禁用。

如果使用 `-g` 或 `-g0` 且优化级别是 `-x04` 或更高，编译器会为完全优化提高尽可能多的符号信息。

使用 -g 进行调试不会抑制 –x0*level*，但 –x0*level* 会对 –g 造成一些限制。例如，–x0*level* 选项会降低调试的作用，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。有关更多信息，请参见《使用 dbx 调试程序》。

-xipo 选项只有与 -x04 或 -x05 一起使用时才有效。

优化级别低于 -x03 时，-xinline 选项不起作用。优化级别为 -x04 时，优化器会决定应该内联哪些函数，而不管是否指定了 -xinline 选项。优化级别为 -x04 时，编译器还会尝试确定内联哪些函数可以提高性能。如果使用 -xinline 强制内联函数，实际上可能会降低性能。

缺省值

缺省为不优化。不过，只有不指定优化级别时才可能使用缺省设置。如果指定了优化级别，则没有任何选项可用来关闭优化。

如果尝试不设置优化级别，请不要指定任何隐含优化级别的选项。例如，-fast 是将优化级别设置为 -x05 的宏选项。隐含优化级别的所有其他选项都会给出优化已设置的警告消息。不使用任何优化来编译的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 –x03 或 –x04 级别上优化多个非常大的过程（一个过程有数千行代码），优化器会需要过多内存。在这些情况下，机器的性能就会降低。

为了防止性能降低，请使用 limit 命令限制单一进程可用的虚拟内存大小（请参见 csh(1) 手册页）。例如，将虚拟内存限制为 4 GB：

```
example% limit datasize 4G
```

如果它达到 4 GB 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，请输入： **swap– l**

要查找实际的真实内存，请输入： **dmesg | grep mem**

另请参见

-xldscope –fast、-xcrossfile=*n*、–xprofile=*p* 和 csh(1) 手册页

A.2.158 -xopenmp[= *i*]

使用 -xopenmp 选项可通过 OpenMP 指令进行显式并行化。要在多线程环境中运行已并行化的程序，必须在执行之前设置 **OMP_NUM_THREADS**环境变量。

要启用嵌套并行操作，必须将 **OMP_NESTED** 环境变量设置为 TRUE。缺省情况下，禁用嵌套并行操作。

A.2.158.1 值

下表列出了 *i* 值：

表 A–41 -xopenmp 值

值	含义
parallel	启用 OpenMP pragma 的识别。在 -xopenmp=parallel 时，最低优化级别是 -x03。编译器会在必要时将优化级别提高到 -x03 并发出警告。 此标志还定义处理器标记 **_OPENMP**。
noopt	启用 OpenMP pragma 的识别。如果优化级别低于 -O3，编译器将不会提高优化级别。 如果显式将优化级别设置为低于 -O3（如 CC -O2 -xopenmp=noopt），编译器会发出错误。如果没有使用 -xopenmp=noopt 指定优化级别，则会识别 OpenMP pragma，并相应地对程序进行并行处理，但不进行优化。 此标志还定义处理器标记 **_OPENMP**。
none	此标记是缺省的，它禁用对 OpenMP pragma 的识别，而且不更改程序的优化级别且不预定义任何预处理程序标记。

缺省值

如果未指定 -xopenmp，则编译器将该选项设置为 -xopenmp=none。

如果指定了没有参数的 -xopenmp，则编译器将该选项设置为 -xopenmp=parallel。

交互

如果使用 dbx 调试 OpenMP 程序，那么编译时选用 -g 和 -xopenmp=noopt 可以在并行区设置断点并显示变量内容。

警告

请不要将 -xopenmp 和 -xexplicitpar 或 -xparallel 一起指定。

在以后的发行版中，-xopenmp 的缺省值可能会更改。可以通过显式指定适当的优化来避免警告消息。

如果在不同的步骤中进行编译和链接，请在编译步骤和链接步骤中都指定 `-xopenmp`。如果要生成共享对象，这很重要。用于编译可执行文件的编译器的版本不得比使用 `-xopenmp` 生成 .so 的编译器低。这在编译包含 OpenMP 指令的库时尤其重要。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”

为了取得最佳的性能，请确保在系统上安装了最新的 OpenMP 运行时库 `libmtsk.so`。

另请参见

有关用于生成多重处理应用程序的 OpenMP Fortran 95、C 和 C++ 应用程序接口 (application program interface, API) 的完整摘要，请参见《Sun Studio OpenMP API 用户指南》。

A.2.159 `-xpagesize=`*n*

为栈和堆设置首选页面大小。

A.2.159.1 值

在 SPARC 上有效值包括：`4K`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、`256M`、`2G`、`16G` 或 `default`。

在 x86/x64 上有效值包括：`4K`、`2M`、`4M`、`1G` 或 `default`。

必须指定适于目标平台的有效页面大小。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)` 命令确定页面中的字节数。Solaris 操作系统不保证支持页面大小请求。可以使用 `pmap(1)` 或 `meminfo(2)` 来确定目标平台的页面大小。

缺省值

如果指定 `-xpagesize=default`，Solaris 操作系统将设置页面大小。

扩展

此选项是用于 `-xpagesize_heap` 和 `-xpagesize_stack` 的宏。这两个选项与 `-xpagesize` 接受相同的参数： `4k`、`8K`、`64K`、`512K`、`2M`、`4M`、`32M`、 `256M`、`2G`、`16G` 或 `default`。可以通过指定 `-xpagesize` 为它们设置相同值，也可以分别为它们指定不同的值。

警告

除非在编译和链接时使用，否则 `-xpagesize` 选项不会生效。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

另请参见

使用该选项进行编译与使用等效的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等效的选项运行 Solaris 9 命令 ppgsz(1) 具有相同的效果。有关详细信息，请参见 Solaris 手册页。

A.2.160 -xpagesize_heap=*n*

为堆设置内存页面大小。

A.2.160.1 值

n 可以是 4k、8K、 64K、512K、2M、4M、 32M、256M、2G、16G 或 default。必须指定适于目标平台的有效页面大小。如果不指定有效的页面大小，运行时将忽略该请求，且不显示任何提示。

在 Solaris 操作系统中使用 getpagesize(3C) 命令可以确定页面中的字节数。Solaris 操作系统不保证支持页面大小请求。可以使用 pmap(1) 或 meminfo(2) 来确定目标平台的页面大小。

缺省值

如果指定 -xpagesize_heap=default，Solaris 操作系统将设置页面大小。

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效的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等效的选项运行 Solaris 9 命令 ppgsz(1) 具有相同的效果。有关详细信息，请参见 Solaris 手册页。

A.2.161 -xpagesize_stack=*n*

为栈设置内存页面大小。

A.2.161.1 值

n 可以是 4k、8K、 64K、512K、2M、4M、 32M、256M、2G、16G 或 default。必须指定适于目标平台的有效页面大小。如果不指定有效的页面大小，运行时将忽略该请求，且不显示任何提示。

在 Solaris 操作系统中使用 getpagesize(3C) 命令可以确定页面中的字节数。Solaris 操作系统不保证支持页面大小请求。可以使用 pmap(1) 或 meminfo(2) 来确定目标平台的页面大小。

缺省值

如果指定 -xpagesize_stack=default，Solaris 操作系统将设置页面大小。

警告

除非在编译和链接时使用，否则 -xpagesize_stack 选项不会生效。

另请参见

使用该选项进行编译与使用等效的选项将 LD_PRELOAD 环境变量设置为 mpss.so.1 或在运行程序之前使用等效的选项运行 Solaris 9 命令 ppgsz(1) 具有相同的效果。有关详细信息，请参见 Solaris 手册页。

A.2.162 -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 include 文件的应用程序的编译时间，而且这些 include 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。要使用该功能，需要指定 -xpch 和 -xpchstop 选项，并使用 #pragma hdrstop 指令。

另请参见：

- 第 307 页中的 “A.2.163 -xpchstop=*file*”
- 第 341 页中的 “B.2.8 #pragma hdrstop”

A.2.162.1 创建预编译头文件

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
```

有效的预编译头文件总是有后缀 .Cpch。在指定 *pch_filename* 时，后缀可以由您自己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 include 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。include 文件的共同序列在这些源文件之间必须是一样的。请记住，在 `collect` 模式中只能使用一个源文件名值。例如，`CC -xpch=collect:foo bar.cc` 有效，而 `CC -xpch=collect:foo bar.cc foobar.cc` 无效，因为它指定了两个源文件。

使用预编译头文件

可指定 `-xpch=use:`*pch_filename* 以使用预编译头文件。您可以将 include 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` 模式中命令类似于： `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。如果以下任意条件不成立，则应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` 指定的包含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` 指定的包含头文件的内容与创建预编译头文件时指定的包含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`）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。

要在多个源文件间共享预编译头文件，这些源文件必须共享一组共同的 include 文件（按其初始标记序列）。该初始标记序列称为活前缀。活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。

源文件的活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 (if identical, passed through as is)
#pragma (if identical)
```

以上任何指令都可以引用宏。`#else`、`#elif` 和 `#endif` 指令必须在活前缀内匹配。

在共享预编译头文件的每个文件的活前缀中，每个相应的 #define 和 #undef 指令都必须引用相同的符号（例如每个 #define 必须引用同一个值）。这些指令在每个活前缀中出现的顺序也必须相同。每个相应 pragma 也必须相同，并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。

并入预编译头文件的头文件一定不得违反以下约束。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。
- 头文件不得使用 __DATE__ 和 __TIME__。使用预处理宏会产生无法预料的结果。
- 头文件不得包含 #pragma hdrstop。
- 头文件的活前缀中不得使用 __LINE__ 和 __FILE__。但可以在包含头文件中使用 __LINE__ 和 __FILE__。

如何修改 make 文件

下面是为了将 -xpch 合并到生成的程序中而对 make 文件进行修改的几种可能方法。

- 可以通过使用辅助变量 CCFLAGS 以及 make 和 dmake 的 KEEP_STATE 功能使用隐式 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，而无不是尝试使用辅助变量 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
```

- 可以通过常规编译顺便生成预编译头文件，而无需使用 KEEP_STATE，但该方法要求使用显式编译命令。

```
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.2.163 -xpchstop=*file*

可使用 -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 include 文件。在命令行上使用 -xpchstop 与将 hdrstop pragma 置于第一个（引用使用 cc 命令指定的每个源文件中的 file 的）包含指令之后等效。

在以下示例中，-xpchstop 选项指定了预编译头文件的活前缀以包含 projectheader.h 结束。因此，privateheader.h 不是活前缀的一部分。

```
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
```

A.2.163.1 另请参见

-xpch 和 pragma hdrstop。

A.2.164 -xpec[={yes|no}]

生成可移植的可执行文件代码 (Portable Executable Code, PEC) 二进制文件。PEC 二进制文件可以用于自动调优系统 (Automatic Tuning System, ATS)，该系统的工作方式是重新生成编译的 PEC 二进制文件以进行调优和故障排除－不需要原始的源代码。有关 ATS 的更多信息，请访问 http://cooltools.sunsource.net/ats/index.html。

使用 -xpec 生成的二进制文件通常会比未使用 -xpec 生成的二进制文件大五到十倍。

如果未指定 -xpec，编译器会将其设置为 -xpec=no。如果指定 -xpec 但未提供标志，编译器会将其设置为 -xpec=yes。

A.2.165 –xpg

编译以便使用 gprof 配置程序进行文件配置。

–xpg 选项用于编译文件自配置代码来收集数据，以便使用 gprof 进行文件配置。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 – 如果指定 -xpg，-xprofile 将没有用处。两者不能准备或使用对方提供的数据。

在 64 位 Solaris 平台上，使用 prof(1) 或 gprof(1) 生成配置文件，在 32 位 Solaris 平台上，则只使用 gprof 生成配置文件，配置文件中包括大概的用户 CPU 时间。这些时间来自主可执行文件中的例程以及共享库中例程（链接可执行文件时将共享库指定为链接程序参数）的 PC 示例数据（请参见 pcsample(2)）。其他共享库（在进程启动后使用 dlopen(3DL) 打开的库）不进行文件配置。

在 32 位 Solaris 系统中，使用 prof(1) 生成的配置文件仅限于可执行文件中的例程。32 位共享库通过用 -xpg 和 gprof(1) 链接可执行程序可以进行文件配置。

Solaris 10 软件不包括使用 -p 编译的系统库。因此，在 Solaris 10 平台上收集的配置文件不包含系统库例程的调用计数。

A.2.165.1 警告

如果分别进行编译和链接，且使用 –xpg 进行编译，应确保使用 –xpg 进行链接。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

另请参见

–xprofile=*p*、analyzer(1) 手册页和性能分析器手册。

A.2.166 -xport64[=(*v*)]

使用此选项可以帮助调试要移植到 64 位环境的代码。具体来说，该选项会针对遇到的问题发出警告，例如：类型（包括指针）的截断，符号扩展以及位包装更改，将代码从诸如 V8 的 32 位体系结构移植到诸如 V9 的 64 位体系结构时经常会遇到这些问题。

A.2.166.1 值

下表列出了*v* 的有效值：

表 A–42 -xport64 值

值	含义
no	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。显式强制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。

缺省值

如果未指定 -xport64，则缺省值为 -xport64=no。如果指定了 -xport64 但未指定标志，则缺省值为 -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）或显式强制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 位整型，而在 64 位模式下是 64 位整型。将较长的整型截断为较小的整型会生成警告，如下示例所示。

```
example% cat test1.c
int x[10];

int diff = &x[10] - &x[5]; //warn

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%
```

可使用 -xport64=implicit 禁用 64 位编译模式下显式强制类型转换导致数据截断时出现截断警告。

```
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。如果指定 -xport64，导致此类截断的操作（如将指针强制转换为整型）可能会导致在 V9 中出现错误诊断。

```
example% cat test2.c
char* p;
int main() {
  p =(char*) (((unsigned int)p) & 0xFF); // -xarch=v9 error
  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

还可以使用 -xport64 选项来检查这种情况：标准 ISO C 值保留规则允许在无符号整型的表达式中进行带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;  // warn
    promo(i);       // warn
}
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

可使用 -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
```

警告

请注意，仅当通过指定 `-m64` 等选项以 64 位模式编译时，才会生成警告。

另请参见

第 238 页中的 “A.2.51 -m32|-m64”

A.2.167 -xprefetch[=*a*[,*a*...]]

在支持预取的体系结构上启用预取指令。

显式预取只应在度量支持的特殊环境下使用。

a 必须是下列值之一。

表 A–43 `-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
`explicit`	(SPARC) 启用显式预取宏
`no%explicit`	(SPARC) 禁用显式预取宏

表 A–43 -xprefetch 值 （续）

值	含义
latx:*factor*	根据指定的因子，调整编译器假定的“预取到装入”和“预取到存储”延迟。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。
yes	已废弃，不使用。改用 -xprefetch=auto,explicit。
no	已废弃，不使用。改用 -xprefetch=no%auto,no%explicit。

使用 -xprefetch 和 -xprefetch=auto 时，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体系结构的性能。

如果要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缺省延迟时间。

预取延迟是从执行预取指令到所预取的数据在高速缓存中可用那一刻之间的硬件延迟。在确定发出预取指令到发出使用所预取数据的装入或存储指令之间的间隔时，编译器就采用预取延迟值。

注 – 在预取和装入之间采用的延迟可能与在预取和存储之间采用的延迟不同。

编译器可以在众多计算机与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。对于占用大量内存的应用程序，尤其是要在大型多处理器上运行的应用程序，可以通过增加预取延迟值来提高性能。要增加值，请使用大于 1 的因子。介于 .5 和 2.0 之间的值最有可能提供最佳性能。

对于数据集完全位于外部高速缓存中的应用程序，可以通过减小预取延迟值来提高性能。要减小值，请使用小于 1 的因子。

要使用 -xprefetch=latx:*factor* 选项，请首先使用接近 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。继续调整因子并运行性能测试，直到获得最佳性能。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，之后会突然出现差异，然后再趋于稳定。

A.2.167.1 缺省值

缺省值为 -xprefetch=auto,explicit。此缺省值会对实质上具有非线性内存访问模式的应用程序造成负面影响。要覆盖该缺省值，请指定 -xprefetch=no%auto,no%explicit。

除非使用参数 no%auto 或参数 no 进行显式覆盖，否则使用缺省值 auto。例如，-xprefetch=explicit 与 -xprefetch=explicit,auto 相同。

除非使用参数 no%explicit 或 no 进行显式覆盖，否则使用缺省值 explicit。例如，-xprefetch=auto 与 -xprefetch=auto,explicit 相同。

如果仅指定了 -xprefetch，则假定为 -xprefetch=auto,explicit。

如果启用了自动预取，但未指定延迟因子，则假定 -xprefetch=latx:1.0。

交互

该选项会累积而不覆盖。

sun_prefetch.h 头文件提供了用于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，必须在适当的体系结构上，将 sun_prefetch.h 包含进来，并且从编译器命令中排除 -xprefetch 或者使用 -xprefetch、-xprefetch=auto,explicit 或 -xprefetch=explicit。

如果调用宏并包含 sun_prefetch.h 头文件，但指定 -xprefetch=no%explicit，那么显式预取将不会出现在可执行文件中。

只有启用了自动预取时，才可以使用 latx:*factor*。即，除非将 latx:*factor* 与 -xprefetch=auto,latx:*factor* 结合使用，否则它会被忽略。

警告

显式预取只应在度量支持的特殊环境下使用。

因为编译器可以在众多计算机与应用程序间调整预取机制以获得最佳性能，所以当性能测试指示性能明显提高时，应当只使用 -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.2.168 -xprefetch_auto_type=*a*

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，编译器会将其设置为 -xprefetch_auto_type=no%indirect_array_access。

-xdepend、-xrestrict 和 -xalias_level 等选项会影响计算候选间接预取的主动性，进而影响因更好的内存别名歧义消除信息而自动插入间接预取的主动性。

A.2.169 -xprefetch_level[= *i*]

可使用 -xprefetch_level=*i* 选项控制由 -xprefetch=auto 确定的自动插入预取指令的主动性。编译器变得更加主动，也就是说，引入了更多预取级别 -xprefetch_level（依次增高）。

-xprefetch_level 取何适当值取决于应用程序的高速缓存未命中次数。-xprefetch_level 值越高，越有可能提高那些高速缓存未命中次数较多的应用程序的性能。

A.2.169.1 值

i 必须是 1、2 或 3。

表 A–44 -xprefecth_level 值

值	含义
1	启用预取指令的自动生成。
2	针对其他循环（超过在 -xprefetch_level=1 针对的循环）来插入预取。插入的其他预取可超过在 -xprefetch_level=1 插入的预取。
3	针对其他循环（超过在 -xprefetch_level=2 针对的循环）来插入预取。插入的其他预取可超过在 -xprefetch_level=2 插入的预取。

缺省值

指定了 -xprefetch=auto 时，缺省值为 -xprefetch_level=1。

交互

仅当使用 -xprefetch=auto 对该选项进行编译，优化级别为 3 或更高 (-xO3)，而且是在支持预取的平台（v9、v9a、v9b、generic64 和 native64）上时，该选项才有效。

A.2.170 -xprofile=*p*

可使用该选项先收集并保存执行频率数据，这样便可以在后续运行中使用这些数据以提高性能。只有将优化级别指定为 -xO2 或更高时，该选项才有效。

为编译器提供运行时性能反馈增强了在较高优化级别（如 -xO5）进行编译的功能。为了生成运行时的性能反馈，必须使用 -xprofile=collect 编译，然后针对典型数据集运行可执行文件，最后在最高的优化级别使用 -xprofile=use 重新编译。

对多线程应用程序来讲，配置文件集合是安全的。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。只有将优化级别指定为 -xO2 或更高时，该选项才有效。

A.2.170.1 值

p 必须是下列值之一。

- `collect`[:*name*]

  在 `–xprofile=use` 时优化器收集并保存执行频率，以供将来使用。编译器生成可测量语句执行频率的代码。

  name 是执行程序时存储配置文件数据的目录的可选名称。如果已指定，*name* 应该是 UNIX 绝对路径名。如果未指定 *name*，名为 *program* 的经过文件配置的程序的配置文件数据将存储在名为 *program*`.profile` 的目录中，该目录位于执行**程序**时的当前工作目录中。

  在运行时，使用 `–xprofile=collect:`*name* 编译的程序会创建子目录 *name*`.profile` 来保存运行时反馈信息。数据将写入该子目录下的文件 `feedback` 中。可以使用环境变量 `$SUN_PROFDATA` 和 `$SUN_PROFDATA_DIR` 更改反馈信息的位置。有关更多信息，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。

  如果在不同的步骤中进行编译和链接，应确保使用 `-xprofile=collect` 编译的任何目标文件也要使用 `-xprofile=collect` 进行链接。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

  `-xMerge -ztext -xprofile=collect` 这三个选项不应该一起使用。**`-xMerge`** 强制将静态初始化的数据写入只读存储，而 **`-ztext`** 禁止只读存储中有与位置有关的符号重定位，**`-xprofile=collect`** 在可写存储中生成静态初始化的、与位置有关的符号重定位。

- `use`[:*name*]

  使用先前执行使用 `–xprofile=collect` 编译的程序时生成并保存在 `feedback` 文件中的执行频率数据来优化程序。

  name 是要分析的可执行文件的名称。*name* 是可选的，如果没有指定，则假定为 `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（创建了之后生成 `feedback` 文件的编译程序的）编译的源文件和编译器选项完全相同。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中：`-xprofile=use:`*name*。

  目标文件与其配置文件数据之间的关联依据的是使用 `-xprofile=collect` 编译目标文件时该文件的 UNIX 路径名。在某些情况下，编译器不会将目标文件与其配置文件数据相关联：由于以前没有使用 `-xprofile=collect` 编译目标文件，因此目标文件没有配置文件数据；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 -xprofile=collect 在不同目录中编译了目标文件，而该目标文件与使用 -xprofile=collect 编译的其他目标文件共享一个公共基名，但却无法通过它们的包含目录名称进行唯一标识，编译器也会产生混淆。在这种情况下，即使目标文件有配置文件数据，使用 -xprofile=use 重新编译目标文件时，编译器也不能在 feedback 目录中找到该目标文件。

所有这些情况都能使编译器释放目标文件及其文件配置数据之间的关联。因此，如果目标文件有配置文件数据，但在指定了 -xprofile=use 时，编译器无法将其与目标文件的路径名关联，请使用 -xprofile_pathmap 选项标识正确的目录。请参见第 317 页中的 “A.2.172 -xprofile_pathmap”

- tcov

  使用新式 tcov 进行基本块覆盖分析。

  此选项是 tcov 的新式基本块文件配置。它的功能与 –xa 选项类似，但可以正确收集在头文件中有源代码或利用 C++ 模板的程序的数据。代码重构与 -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 之外运行，那么数据文件将存储在 /foo/bar/myprog.profile/myprog.tcovd 中。

  运行 tcov 时，必须将其传递给 – x 选项，以强制其使用新式数据。如果未传递 -x，则缺省情况下 tcov 使用原来的 .d 文件，并生成意外的输出。

  与 –xa 选项不同，环境变量 TCOVDIR 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 选项在单个可执行文件中兼容。也就是说，可以链接同时包含了用 -xprofile=tcov 编译的某些文件和用 -xa 编译的另外一些文件的程序。您不能同时用这两个选项来编译同一个文件。

如果因使用 -xinline 或 -xO4 而内联函数，那么 -xprofile=tcov 生成的代码覆盖报告不可靠。

可以设置环境变量 $SUN_PROFDATA 和 $SUN_PROFDATA_DIR 控制使用 -xprofile=collect 编译的程序放置配置文件数据的位置。如果未设置这些变量，配置文件数据就写入当前目录中的 *name*.profile/feedback（*name* 是可执行文件的名称或在 -xprofile=collect:*name* 标志中指定的名称）。如果设置了这些变量，-xprofile=collect 数据就写入 $SUN_PROFDATA_DIR/$SUN_PROFDATA。

环境变量 $SUN_PROFDATA 和 $SUN_PROFDATA_DIR 同样控制 tcov 生成的配置文件数据文件的路径和名称。有关更多信息，请参见 tcov(1) 手册页。

警告

如果在不同的步骤中进行编译和链接，则编译命令和链接命令中都必须有相同的 -xprofile 选项。虽然只在一个步骤中包括 -xprofile 而在其他步骤中不包括该项不会影响程序的正确性，但这样将无法进行文件配置。

在 Linux 平台上，**-xprofile=collect** 或 **-xprofile=tcov** 不应该与 **-G** 一起使用来生成共享库。

另请参见

-xa、tcov(1) 手册页和《程序性能分析工具》。

A.2.171 -xprofile_ircache[=*path*]

(SPARC) 可以将 -xprofile_ircache[=*path*] 与 -xprofile=collect|use 一起使用，并重新使用 collect 阶段保存的编译数据，以改善 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 use 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。缺省情况下，这些文件会作为目标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，指定路径很有用。以下是典型的命令序列：

```
example% CC -xO5 -xprofile=collect -xprofile_ircache t1.cc t2.cc
example% a.out    // run collects feedback data
example% CC -xO5 -xprofile=use -xprofile_ircache t1.cc t2.cc
```

A.2.172 -xprofile_pathmap

(SPARC) 使用 -xprofile_pathmap=*collect_prefix*:*use_prefix* 选项（同时还指定 -xprofile=use 命令时）。以下两个条件都成立且编译器无法找到使用 -xprofile=use 编译的目标文件的配置文件数据时，使用 -xprofile_pathmap。

- 使用 -xprofile=use 编译目标文件所在的目录与先前使用 -xprofile=collect 编译目标文件所在的目录不同。
- 目标文件会共享配置文件中的公共基名，但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，该目录树中的目标文件是使用 -xprofile=collect 编译的。

use-prefix 是目录树的 UNIX 路径名的前缀，该目录树中的目标文件是使用 -xprofile=use 编译的。

如果指定了 -xprofile_pathmap 的多个实例，编译器将按照这些实例的出现顺序对其进行处理。*-xprofile_pathmap* 实例指定的每个 use-prefix 与目标文件路径名进行比较，直至找到匹配的 *use-prefix* 或发现最后一个指定的 *use-prefix* 与目标文件路径名也不匹配。

A.2.173 -xreduction

对自动并行化中的约简进行循环分析。只有在同时指定了 -xautopar 时，此选项才有效。否则，编译器会发出警告。

当启用了约简识别时，编译器会并行化约简，例如点积、最大与最小查找。这些约简产生的舍入与通过非并行化代码获得的舍入不同。

A.2.174 –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。

A.2.174.1 值

r 必须是下列值之一。每个值的含义都与 -m32|-m64 的设置相关。

表 A–45 -xregs 值

值	含义
[no%]appl	(SPARC) [不] 允许编译器将应用程序寄存器用作临时寄存器来生成代码。应用程序寄存器是： g2、g3 或 g4（在 32 位平台上） g2 或 g3（在 64 位平台上） 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。 在 SPARC ABI 中，这些寄存器表示为**应用程序**寄存器。使用这些寄存器时需要的 load 和 store 指令较少，因此可以提高性能。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] 允许编译器通过将浮点寄存器用作整数值的临时寄存器来生成代码。使用浮点值可能会用到与该选项无关的这些寄存器。如果希望您的代码没有任何对浮点寄存器的引用，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。

表 A–45 -xregs 值 （续）

值	含义
[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（adb、mdb 和 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 –** 如果也指定了 -xpg，编译器将忽略 -xregs=framptr，并发出警告。另外，对于 32 位 x86 编译，编译器会忽略此选项，除非使用 -noex 选项禁用了异常。

缺省值

SPARC 缺省值为 -xregs=appl,float。

x86 缺省值是包括在 -fast 扩展中的 -xregs=no%frameptr -xregs=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 V8 和 SPARC V9 ABI

A.2.175 -xrestrict[=*f*]

(SPARC) 将返回赋值指针函数参数视为限定指针。*f* 必须是下列值之一：

表 A–46 -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。如果指定一个函数列表，编译器会将指定函数中的指针参数视为限定指针；有关更多信息，请参阅下一节第 320 页中的 “A.2.175.1 限定指针”。

此命令行选项可以单独使用，但最好将其用于优化。例如，命令：

```
%CC -xO3 -xrestrict=%all prog.cc
```

将文件 prog.c 中的所有指针参数都视为限定指针。命令：

```
%CC -xO3 -xrestrict=agc prog.cc
```

将文件 prog.c 中函数 agc 中的所有指针参数都视为限定指针。

缺省值为 %none；指定 -xrestrict 与指定 -xrestrict=%source 等效。

A.2.175.1 限定指针

为使编译器高效并行执行循环，需要确定某些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。例如下面的函数 vsq()：

示例 A–3 带两个指针的循环

```
extern "C"
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，可以并行化循环的不同迭代的执行。如果通过指针 a 和 b 访问的对象存在重叠，编译器以并行方式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函数 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。可以使用以下命令行选项指定将返回

赋值指针函数参数视为限定指针：-xrestrict[=*func1,...,funcn*]。如果指定了函数列表，则指定函数中的指针参数被视为是限定的，否则，整个源文件中的所有指针参数都被视为是限定的（不推荐）。例如，-xrestrict=vsq 限定 vsq() 函数示例中给定的指针 a 和 b。

将指针参数声明为限定的表示指针指定不同的对象。编译器可以假定 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。

正确使用 -xrestrict 至关重要。如果指针被限定为指向并非不同的对象的限定指针，编译器可能会错误地并行化循环，从而导致不确定的行为。例如，假定函数 vsq() 的指针 a 和 b 指向的对象重叠，如 b[i] 和 a[i+1] 是同一对象。如果 a 和 b 未声明为限定指针，循环将以串行方式执行。如果 a 和 b 被错误地限定为限定指针，编译器可能会并行化循环的执行，这是不安全的，因为 b[i+1] 仅应在计算 b[i] 之后进行计算。

A.2.176 –xs

允许 dbx 在没有目标 (.o) 文件的情况下进行调试。

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性能几乎没有什么影响，但需要更多磁盘空间。

该选项只有在 -xdebugformat=stabs 时才有效，缺省情况下不将调试数据复制到可执行文件。使用缺省调试格式 -xdebugformat=dwarf 时，总是将调试数据复制到可执行文件，没有选项可以阻止复制。

A.2.177 –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。

A.2.177.1 交互

仅当与优化级别 -xO5 及以下 -xarch 值中的一个一起使用时，该选项才能生效：sparc、 sparcvis 或 sparcvis2（用于 -m32 和 -m64）

A.2.177.2 警告

由于在发生诸如地址未对齐或段违规的故障时，无故障装入不会导致陷阱，因此您应该只对不会发生此类故障的程序使用该选项。因为只有很少的程序会导致基于内存的陷阱，所以您可以安全地将该选项用于大多数程序。对于显式依赖基于内存的自陷来处理异常情况的程序，请勿使用该选项。

A.2.178 –xsb

已过时，请勿使用。源代码浏览器功能已过时。默认忽略该选项。

A.2.179 –xsbfast

已过时，请勿使用。源代码浏览器功能已过时。默认忽略该选项。

A.2.180 –xspace

SPARC：不允许进行增加代码大小的优化。

A.2.181 –xtarget=*t*

为指令集和优化指定目标平台。

通过为编译器提供目标计算机硬件的精确描述，某些程序的性能可得到提高。当程序性能很重要时，目标硬件的正确指定是非常重要的。在较新的 SPARC 处理器上运行时，尤其是这样。不过，对大多数程序和较旧的 SPARC 处理器来讲，性能的提高微不足道，因此指定 generic 就足够了。

t 的值必须是下列值之一：`native`、`generic`、`native64`、`generic64` 或 *system-name*。

`-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` 选项可在运行的系统上确定 `-xtarget=native` 的扩展。

例如，`-xtarget=ultraT2` 等效于 `-xarch=sparcvis2 -xchip=ultraT2-xcache=8/16/4:4096/64/16`。

注 – `-xtarget` 在特定主机平台上的扩展在该平台上编译时扩展到的 `-xarch`、`-xchip` 或 `-xcache` 设置可能与 `-xtarget=native` 不同。

A.2.181.1 -xtarget 值（按平台）

表 A–47 `-xtarget` 值（所有平台）

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。

表 A–47 -xtarget 值（所有平台） （续）

值	含义
native64	在主机系统上获取 64 位二进制目标文件的最佳性能。编译器生成为主机系统优化的 64 位二进制目标文件。它决定了运行编译器的计算机的可用 64 位体系结构、芯片和缓存属性。
generic	这是缺省值。获取通用体系结构、芯片和高速缓存的最佳性能。
generic64	为了在大多数 64 位平台体系结构上获得 64 位二进制目标文件的最佳性能而设置参数。
system-name	获取指定平台的最佳性能。 从以下代表您所面向的实际系统的列表中选择系统名称：

-xtarget 值（SPARC 平台）

在 SPARC 还是 UltraSPARC V9 上针对 64 位 Solaris 软件进行编译，是由 -m64 选项指示。如果指定带有 native64 或 generic64 之外的标志的 -xtarget，还必须指定 -m64 选项，如下所示：-xtarget=ultra... -m64，否则编译器会使用 32 位内存模型。

表 A–48 SPARC 体系结构上的 -xtarget 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

表 A–48 SPARC 体系结构上的 -xtarget 扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
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
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

表 A–48 SPARC体系结构上的 -xtarget扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
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
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

表 A–48 SPARC 体系结构上的 -xtarget 扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
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
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

表 A–48 SPARC体系结构上的 -xtarget扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
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
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
ultra4	v8plusa	ultra4	64/32/4:8192/128/2
ultra4plus	v8plusa	ultra4plus	64/32/4:2048/64/4:32768/64/4
ultraT1	v8plusa	ultraT1	8/16/4/4:3072/64/12/32
ultraT2	sparcvis2	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10

表 A–48 SPARC 体系结构上的 -xtarget 扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
有关 UltraSPARC IVplus、UltraSPARC T1 和 UltraSPARC T2 芯片的缓存属性的更多信息，请参见 第 267 页中的 “A.2.114 –xcache=c”。			

-xtarget 值（x86 平台）

在 64 位 x86 平台上针对 64 位 Solaris 软件进行编译是由 -m64 选项指示的。如果指定带有 native64 或 generic64 之外的标志的 -xtarget，还必须指定 -m64 选项，如下所示：-xtarget=opteron... -m64，否则编译器会使用 32 位内存模型。

表 A–49 -xtarget 值（x86 平台）

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
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
nehalem	sse4_2	nehalem	32/64/8:256/64/8:8192/64/16
penryn	sse4_1	penryn	2/64/8:4096/64/16
woodcrest	ssse3	core2	32/64/8:4096/64/16
barcelona	amdsse4a	amdfam10	64/64/2:512/64/16

缺省值

在 SPARC 和 x86 设备上，如果未指定 –xtarget，则假定 –xtarget=generic。

扩展

–xtarget 选项是一个宏，使用它可以快捷地指定可在从市场上购买的平台上使用的 -xarch、–xchip 和 –xcache 组合。–xtarget 的唯一含义在其扩展中。

示例

-xtarget=sun4/15 表示 -xarch=v8a -xchip=micro -xcache=2/16/1。

交互

针对 SPARC V9 体系结构的编译用 -xarch=v9|v9a|v9b 选项指示。不必设置 –xtarget=ultra 或 ultra2，而且这也不够。如果指定了 -xtarget，则必须在 -xtarget 后面使用 –xarch=v9、v9a 或 v9b 选项。例如：

```
-xarch=v9 -xtarget=ultra
```

扩展到以下选项，并将 -xarch 值恢复为 v8。

```
-xarch=v9 -xarch=v8 -xchip=ultra -xcache=16/32/1:512/64/1
```

正确的方法是在 -xtarget 后面指定 -xarch。例如：

```
–xtarget=ultra -xarch=v9
```

警告

在不同的步骤中进行编译和链接时，必须在编译步骤和链接步骤中使用相同的 -xtarget 设置。

A.2.182 -xthreadvar[=*o*]

指定 -xthreadvar 来控制线程局部变量的实现。将此选项与 __thread 声明说明符结合使用，可利用编译器的线程局部存储功能。使用 __thread 说明符声明线程变量后，请指定 -xthreadvar，以便能够将线程局部存储用于动态（共享）库中的位置相关的代码（非 PIC 代码）。有关如何使用 __thread 的更多信息，请参见第 63 页中的 “4.2 线程局部存储”。

A.2.182.1 值

o 必须是下列值之一：

表 A–50 -xthreadvar 值

值	含义
[no%]dynamic	[不] 编译动态装入的变量。使用 -xthreadvar=no%dynamic 时对线程变量的访问明显加快，但是不能在动态库中使用目标文件。也就是说，只能在可执行文件中使用目标文件。

缺省值

如果未指定 -xthreadvar，编译器所用的缺省设置取决于是否启用与位置无关的代码。如果启用了与位置无关的代码，则该选项设置为 -xthreadvar=dynamic。如果禁用了与位置无关的代码，则该选项设置为 -xthreadvar=no%dynamic。

如果指定了 -xthreadvar 但未指定任何参数，则该选项设置为 -xthreadvar=dynamic。

交互

编译和链接使用 __thread 的文件时，必须使用 -mt 选项。

警告

如果动态库中存在与位置有关的代码，那么就必须指定 -xthreadvar。

链接程序不支持在动态库中与非 PIC 代码等效的线程变量。由于非 PIC 线程变量要快很多，所以应将其用作可执行文件的缺省设置。

另请参见

-xcode、-KPIC 和 -Kpic

A.2.183 –xtime

使 cc 驱动程序报告各种编译传递的执行时间。

A.2.184 -xtrigraphs[={ yes|no}]

启用或禁用对 ISO/ANSI C 标准定义的三字母序列的识别。

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 子选项禁用对三字符序列的识别。

A.2.184.1 值

-xtrigraphs 可以是下列值之一：

表 A–51 -xtrigraphs 值

值	含义
yes	启用整个编译单元三字母序列的识别
no	禁用整个编译单元三字母序列的识别

缺省值

如果没有在命令行上指定 -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

请考虑以下名为 trigraphs_demo.cc 的示例源文件。

```
#include <stdio.h>

int main ()
{
    (void) printf("(\?\?) in a string appears as (??)\n");
    return 0;
}
```

下面是使用 -xtrigraphs=yes 编译该代码后的输出。

```
example% CC -xtrigraphs=yes trigraphs_demo.cc
example% a.out
(??) in a string appears as (]
```

下面是使用 -xtrigraphs=no 编译该代码后的输出。

```
example% CC -xtrigraphs=no trigraphs_demo.cc
example% a.out
(??) in a string appears as (??)
```

另请参见

有关三字母的信息，请参见《C 用户指南》中关于转换到 ANSI/ISO C 的章节。

A.2.185 –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。

A.2.185.1 值

n 为 1 时，建议编译器不要解开循环。

n 为大于 1 的整数（-unroll=*n*）时，编译器会将循环解开 *n* 次。

A.2.186 -xustr={ascii_utf16_ushort|no}

如果代码中包含要编译器转换成目标文件中 UTF-16 字符串的字符串或字符文字，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 位的文本字符串。使用该选项时，U"*ASCII_string*" 文本字符串会识别为无符号短整型数组。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。

A.2.186.1 值

如果需要支持使用 ISO10646 UTF-16 文本字符串的国际化应用程序，可指定 `-xustr=ascii_utf16_ushort`。可以通过指定 *-xustr=no* 来禁用编译器对 U"`ASCII_string`" 字符串或字符文字的识别。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*" 文本字符串。这样执行时不会出现错误。

缺省值

缺省值为 `-xustr=no`。如果指定了没有参数的 `-xustr`，编译器将不接受该选项，而是发出一个警告。如果 C 或 C++ 标准定义了语法的含义，那么缺省设置是可以更改的。

示例

以下示例显示了前面带有 U 的带引号文本字符串。另外，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 foo;}
example% CC -xustr=ascii_utf16_ushort file.cc -c
```

8 位字符文字前面可以带有 `U`，以形成类型为 `unsigned short` 的 16 位 UTF-16 字符。示例：

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

A.2.187 `-xvector[=`*a*`]`

启用向量库函数调用的自动生成和/或 SIMD（Single Instruction Multiple Data，单指令多数据）指令的生成。使用此选项时，必须通过指定 `-fround=nearest` 来使用缺省的舍入模式。

a 可以为下列值：

表 A–52 -xvector 标志

值	含义
[no%]lib	（**仅限** *Solaris*）[不] 允许编译器将循环内的数学库调用转换为对等效向量数学例程的单个调用（如果能够进行此类转换）。此类转换可提高那些循环计数较大的循环的性能。
[no%]simd	[不] 指示编译器使用本机 x86 SSE SIMD 指令来提高某些循环的性能。如果目标体系结构支持 SIMD 指令，则编译器只能接受此转换。例如，必须指定 -xarch=amd64、-xarch=amd64a 或 -xarch=generic64。指定 -xvector=simd 时，还必须将优化级别指定为 -xO3 或更高，以及指定 -xdepend。
yes	此选项已过时，改为指定 -xvector=lib。
no	此选项已过时，改为指定 -xvector=none。

A.2.187.1 缺省值

缺省值为 -xvector=%none。如果指定了 -xvector 但未提供标志，编译器将假定 -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，-xvector 会触发 -xdepend。如果未指定优化级别或优化级别低于 -x03，-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。

如果使用单独的命令进行编译和链接，应确保在链接 CC 命令中使用 -xvector。有关在编译时和链接时都必须指定的选项的完整列表，请参见第 47 页中的 “3.3.3 编译时选项和链接时选项”。

A.2.188 -xvis[={yes|no}]

(SPARC) 要使用 VIS™ 指令集软件开发者工具包 (VIS instruction-set Software Developers Kit, VSDK) 中定义的汇编语言模板时，可使用 -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。

A.2.188.1 缺省值

缺省值为 -xvis=no。指定 -xvis 与指定 -xvis=yes 等效。

另请参见

有关 VSDK 的更多信息，请访问 `http://www.sun.com/processors/vis/`。

A.2.189 -xvpara

发出关于并行编程相关的潜在问题的警告，在使用 OpenMPI 时这些问题可能会导致不正确的结果。与 `-xopenmp` 和 OpenMP API 指令一起使用。

编译器检测到以下情况时会发出警告：

- 不同循环迭代之间存在数据依赖性的情况下，使用 MP 指令并行化循环
- OpenMP 数据共享属性子句有问题。例如，声明一个变量 "shared"，在 OpenMP 并行区域中其访问可能导致数据争用；或声明一个变量 "private"，在并行区域中的其值在并行区域之后使用。

如果所有并行化指令在处理期间均未出现问题，则不显示警告。

示例：

```
CC -xopenmp -xvpara any.cc
```

注 – Sun Studio 编译器支持 OpenMP 2.5 API 并行化。因此，已废弃 MP pragma 指令，不再支持此类指令。有关迁移到 OpenMP API 的信息，请参见《*OpenMP API* **用户指南**》。

A.2.190 –xwe

通过返回非零的退出状态，将所有警告转换成错误。

A.2.190.1 另请参见

第 209 页中的 “A.2.17 `-errwarn[=t]`”

A.2.191 -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。该选项传递给 ld。

A.2.191.1 值

c 必须是下列值之一：

表 A–53 -Y 标志

值	含义
P	更改 cpp 的缺省目录。
0	更改 ccfe 的缺省目录。
a	更改 fbe 的缺省目录。
2 (SPARC)	更改 iropt 的缺省目录。
c (SPARC)	更改 cg 的缺省目录。
O (SPARC)	更改 ipo 的缺省目录。
k	更改 CClink 的缺省目录。
l	更改 ld 的缺省目录。
f	更改 c++filt 的缺省目录。
m	更改 mcs 的缺省目录。
u *(x86)*	更改 ube 的缺省目录。
h *(x86)*	更改 ir2hf 的缺省目录。
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	将路径添加到缺省库搜索路径。将在缺省库搜索路径之前搜索此路径。
S	更改启动目标文件的缺省目录

交互

可以在命令行上指定多个 -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.2.192 -z[ ]*arg*

链接编辑器选项。有关更多信息，请参见 ld(1) 手册页和《Solaris 链接程序和库指南》。

◆ ◆ ◆ 附录 B

Pragma

本附录介绍了 C++ 编译器 pragma。*pragma* 是一个编译器指令，使用它可以向编译器提供其他信息。该信息可以更改您所控制的编译详细信息。例如，`pack` pragma 会影响结构内的数据布局。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，但每个编译器中，pragma 的形式、内容和含义都是不相同。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* 指特定指令；*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。请看以下示例：

```
int bar(int);
int foo(int);
int foo(double);
#pragma does_not_read_global_data(foo, bar)
```

在此示例中，foo 指 foo(double)，即 pragma 紧前面的 foo 声明；而 bar 指 bar(int)，即唯一声明的 bar。现在，请看以下示例，在此示例中再次重载了 foo：

```
int foo(int);
int foo(double);
int bar(int);
#pragma does_not_read_global_data(foo, bar)
```

在此示例中，bar 指 bar(int)，即唯一声明的 bar。但 pragma 并不知道要使用哪个版本的 foo。要更正此问题，必须将 pragma 放在希望 pragma 使用的 foo 定义的紧后面。

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 参考

本节介绍了 C++ 编译器可识别的 pragma 关键字。

B.2.1 #pragma align

```
#pragma align integer(variable [,variable...])
```

可使用 align使所列变量与 *integer* 字节内存对齐，并覆盖缺省值。请遵循以下限制：

- *integer* 必须是介于 1 和 128 之间的 2 的幂，有效值包括 1、2、4、8、16、32、64 和 128。
- *variable* 是全局或静态变量，但不能是局部变量或类成员变量。
- 如果指定的对齐比缺省值小，就使用缺省值。
- 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，必须使用损坏名称。例如，以下代码中的 #pragma align 语句就是无效的。要更正此问题，应将 #pragma align 语句中的 a、b 和 c 替换为其损坏名称。

```
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])
```

此 pragma 断言，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视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 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])
```

此 pragma 向编译器断言，将不会返回对指定例程的调用。这样编译器可以执行与假定一致的优化。例如，寄存器生命周期在调用点终止，这样可以进行进一步的优化。

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视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.4 #pragma does_not_write_global_data

```
#pragma does_not_write_global_data(funcname [, funcname])
```

此 pragma 断言，指定的一组例程不会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视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.5 #pragma dumpmacro s

```
#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 达到文件结尾或遇到 #pragma end_dumpmacro 之前一直有效。请参见第 341 页中的 “B.2.6 #pragma end_dumpmacros”。*value* 可以是下列参数：

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	另外打印 defs、undefs 和 use 的位置（路径名和行号）
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。使用 loc、conds 和 sys 本身并不会生成任何结果。例如，#pragma dumpmacros=loc,conds,sys 不会生成什么结果。

dumpmacros pragma 与命令行选项作用相同，但 pragma 会覆盖命令行选项。请参见第 275 页中的 “A.2.123 -xdumpmacros[= value[,value...]]”。

dumpmacros pragma 并不嵌套，因此以下代码行中，处理 #pragma end_dumpmacros 时，将停止打印宏信息：

```
#pragma dumpmacros (defs, undefs)
#pragma dumpmacros (defs, undefs)
...
#pragma end_dumpmacros
```

dumpmacros pragma 的作用是累积的。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，则使用的每个宏的名称仅打印一次。如果使用选项 #pragma dumpmacros=use,loc，则每次使用宏时都打印位置和宏名称。

B.2.6 #pragma end_dumpmacros

```
#pragma end_dumpmacros
```

此 pragma 标记 dumpmacros pragma 的结尾，并停止打印有关宏的信息。如果没有在 dumpmacros pragma 后面使用 end_dumpmacros pragma，dumpmacros pragma 会在文件结尾一直生成输出。

B.2.7 #pragma fini

```
#pragma fini (identifier[,identifier...])
```

可使用 fini 将 *identifier* 标记为完成函数。此类函数应为 void 类型，不接受任何参数，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调用它们。与初始化函数一样，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后面执行。在 pragma 中使用标识符之前，请先声明这些标识符。

B.2.8 #pragma hdrstop

可将 hdrstop pragma 嵌入源文件的头文件中以标识源文件活前缀的结尾。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"
```

源文件活前缀以 c.h 结束，因此可在每个文件中的 c.h 后面插入 #pragma hdrstop。

#pragma hdrstop 只能位于用 CC 命令指定的源文件活前缀的结尾处。不要在任何 include 文件中指定 #pragma hdrstop。

请参见第 304 页中的 “A.2.162 -xpch=*v*”和第 307 页中的 “A.2.163 -xpchstop=*file*”。

B.2.9 #pragma ident

#pragma ident *string*

可使用 ident 将 *string* 放在可执行文件的 .comment 部分中。

B.2.10 #pragma init

#pragma init(*identifier*[,*identifier*...])

可使用 init 将 *identifier* 标记为初始化函数。此类函数应为 void 类型，不接受任何参数，在开始执行时构造程序的内存映像的情况下调用。执行将共享对象送入内存的操作时（程序启动时，或执行某些动态装入操作时，例如 dlopen()），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析构函数后面执行。在 pragma 中使用标识符之前，请先声明这些标识符。

B.2.11 #pragma must_have_frame

#pragma must_have_frame(*funcname* [,*funcname*])

该 pragma 要求总是编译指定的一组函数来获得完整的栈帧（如 System V ABI 中所定义）。必须在使用该 pragma 列出函数之前，声明该函数的原型。

```
extern void foo(int);
extern void bar(int);
#pragma must_have_frame(foo, bar)
```

只有在声明了指定函数的原型后，才允许使用该 pragma。该 pragma 必须位于函数结尾之前。

```
void foo(int) {
  .
  #pragma must_have_frame(foo)
  .
  return;
  }
```

请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.12 #pragma no_side_effect

```
#pragma no_side_effect(name[,name...])
```

可使用 no_side_effect 指示函数不更改任何持久状态。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 参数指定当前转换单元中函数的名称。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视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.13 #pragma opt

```
#pragma opt level (funcname[, funcname])
```

funcname 指定当前转换单元中定义的 函数的名称。*level* 值指定用于所指定函数的优化级别。可以指定优化级别 0、1、2、3、4、5。可以通过将 *level* 设置为 0 来禁用优化。必须在该 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 中所列任何函数的优化级别都降为 `-xmaxopt` 值。`-xmaxopt=off` 时，忽略 pragma。

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.14 #pragma pack(*n*)

```
#pragma pack([n])
```

可使用 pack 影响对结构成员的封装。

如果使用了该项，*n* 必须为 0 或 2 的幂。0 以外的值指示编译器针对数据类型使用 *n* 字节对齐和平台的自然对齐中的较小者。例如，以下指令使在指令后面（以及后续的 pack 指令前面）定义的所有结构成员对齐时依据的字节边界不严于 2 字节边界，即使正常对齐是 4 或 8 字节边界也是如此。

```
#pragma pack(2)
```

n 为 0 或省略该项时，成员对齐还原为自然对齐值。

如果 *n* 值等于或大于平台上最严格的对齐时，则采用自然对齐。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、V8、V8a、V8plus、V8plusa、V8plusb	8
SPARC V9、V9a、V9b	16

pack 指令应用于自身与下一个 pack 指令之间的所有结构定义。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。具体来说，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。建议将 pack 指令放在要封装的结构紧前面的程序代码中，并将 `#pragma pack()` 放在该结构紧后面。

如果在 SPARC 平台上使用 `#pragma pack` 封装效果比类型的缺省对齐紧密，必须为应用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缺省对齐。

表 B–2 存储大小和缺省对齐字节数

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.2.15 #pragma rarely_called

`#pragms rarely_called(`*funcname[, funcname]*`)`

此 pragma 提示编译器，很少调用指定函数。这样，编译器可以在此类例程的调用点执行配置文件反馈样式的优化，而没有配置文件收集阶段的开销。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。

只有在声明指定函数的原型之后，才能使用 `#pragma rarely_called` 预处理程序指令。以下是 `#pragma rarely_called` 示例：

```
extern void error (char *message);
#pragma rarely_called(error)
```

有关 pragma 如何将重载的函数名视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.16 #pragma returns_new_memory

`#pragma returns_new_memory(`*name*`[,`*name*`...])`

此 pragma 断言，每个指定的函数都返回新分配内存的地址，以及指针没有与任何其他指针相关的别名。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 参数指定当前转换单元中函数的名称。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视为参数的更加详细的说明，请参见第 337 页中的 “B.1.1 将函数作为 pragma 参数进行重载”。

B.2.17 #pragma unknown_control_flow

```
#pragma unknown_control_flow(name[,name...])
```

可使用 `unknown_control_flow` 指定一组违反过程调用的常规控制流属性的例程。例如，可通过任意的任何其他例程调用来访问 `setjmp()` 调用后面的语句。该语句通过调用 `longjmp()` 来访问。

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.2.18 #pragma weak

```
#pragma weak name1 [= name2]
```

可使用 `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 可以按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++ 变量或函数的损坏名称。无效损坏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损坏名称引用的错误。无论是否产生错误，使用无效损坏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会生成错误消息。

B.2.18.1 #pragma weak *name*

采用 `#pragma weak` *name* 形式时，指令使 *name* 成为弱符号。链接程序没有找到 *name* 的符号定义时，不会显示错误消息，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有函数或变量的强定义，那么 *name* 将链接到它。如果没有 *name* 的强定义，那么链接程序符号的值为 0。

以下指令将 ping 定义为弱符号。链接程序找不到名为 ping 的符号的定义时，不会生成错误消息。

```
#pragma weak ping
```

#pragma weak *name1* = *name2*

采用 #pragma weak *name1* = *name2* 形式时，符号 *name1* 成为对 *name2* 的弱引用。如果没有在其他地方定义 *name1*，那么 *name1* 的值为 *name2*。如果在其他地方定义了 *name1*，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析对 bar（如果已在程序中某处定义）的任何引用，以及解析对 foo 的引用。

```
#pragma weak bar = foo
```

采用标识符形式时，必须在当前编译单元中声明和定义 *name2*。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"，则不需要声明函数。但 bar 必须在同一对象中定义。

```
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。尝试将标识符形式 #pragma weak 用于重载函数会出现错误。例如：

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

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_" // make float bar(int) weak
```

有关更多信息，请参见 Solaris《链接程序和库指南》。

词汇表

ABI 请参见 Application Binary Interface（应用程序二进制接口）。

abstract class（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。

abstract method（抽象方法） 不包含实现的方法。

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。

application binary interface（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。

array（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。

base class（基类） 请参见 *inheritance*（**继承**）。

binary compatibility（二进制兼容性） 链接目标文件的能力，目标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。

binding（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。

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。

class template（类模板） 描述一组类或相关数据类型的模板。

class variable（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。另请参见 instance variable（实例变量）。

class（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。

compiler option（编译器选项） 更改编译器行为的指令。例如，`-g` 选项表示通知编译器为调试器生成数据。同义字：**标志**、**开关**。

constructor（构造函数） 每当创建类对象时编译器都会自动调用的特殊类成员函数，用于确保初始化相应对象的实例变量。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 destructor（析构函数）。

data member（数据成员） 是**数据**而不是函数或类型定义的类元素。

data type（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。

derived class（派生类） 请参见 *inheritance*（**继承**）。

destructor（析构函数） 每当销毁类对象或对类指针应用运算符 `delete` 时编译器都会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 *constructor*（**构造函数**）。

dynamic binding（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**迟绑定**、**运行时绑定**。

dynamic cast（动态强制类型转换） 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。

dynamic type（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。

early binding（早绑定） 请参见 static binding（静态绑定）。

ELF file（ELF 文件） 编译器生成的可执行和链接格式文件。

exception handler（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。

exception handling（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。

exception（异常） 在正常的程序流中出现的错误，阻止程序继续运行。某些错误的原因包括了内存枯竭或被零除。

flag（标志） 请参见 compiler option（编译器选项）。

function overloading（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**。

function prototype（函数原型） 描述函数与程序其他部分之间的接口的声明。

function template（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。

functional polymorphism（函数多态） 请参见 function overloading（函数重载）。

idempotent（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。

incremental linker（增量链接程序） 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。

inheritance（继承） 面向对象编程的一个功能，使得程序员可以从现有类（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。

inline function（内联函数） 用实际函数代码替换函数调用的函数。

instance variable（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。另请参见 class variable（类变量）。

instantiation（实例化） C++ 编译器从模板创建可用的函数或对象的过程。

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。

keyword（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。

late binding（迟绑定） 请参见 dynamic binding（动态绑定）。

linker（链接程序） 连接目标代码和库以形成完整的可执行程序的工具。

local variable（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。

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。

lvalue（左值） 指定变量数据值在内存中的存储位置的表达式。即显示在赋值运算符左侧的变量实例。

mangle（损坏） 请参见 name mangling（名称损坏）。

member function（成员函数） 是函数而非数据定义或类型定义的类元素。

method（方法） 在某些面向对象的语言中，成员函数的另外一个名称。

multiple inheritance（多继承） 直接源于多个基类的派生类的继承。

multithreading（多线程） 在单处理器或多处理器系统上开发并行应用程序的软件技术。

name mangling（名称损坏） 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。编译器通过名称损坏解决这个问题－为函数创建由函数名称及其参数的某些组合组成的唯一名称－从而实现类型安全的链接。也称为**名称修饰**。

namespace（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。

operator overloading（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。

optimization（优化） 改善编译器生成的目标代码执行效率的过程。

option（选项） 请参见 compiler option（编译器选项）。

overloading（重载） 将相同的名称赋予多个函数或运算符。

polymorphism（多态性） 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。

pragma 指示编译器执行特定操作的编译器预处理程序指令或特殊的注释。

runtime binding（运行时绑定） 请参见 dynamic binding（动态绑定）。

runtime type identification（运行时类型标识） 提供标准方法以让程序在运行时决定对象类型的机制。

rvalue（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。

scope（作用域） 操作或定义应用的范围。

stab 在目标代码中生成的符号表条目。在 `a.out` 文件和 ELF 文件中使用相同的格式来包含调试信息。

stack 数据存储方法，通过该方法数据可以增加到栈顶部或从栈顶部删除数据，采用的是后进先出策略。

static binding（静态绑定） 在编译期间函数调用到函数体的连接。也称为 *early binding*（**早绑定**）。

subroutine（子例程） 函数。在 Fortran 中，子例程是不返回值的函数。

switch（开关） 请参见 compiler option（编译器选项）。

symbol table（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。

symbol（符号） 表示某些程序实体的名称或标签。

template database（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。

template options file（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。

template specialization（模板专门化） 类模板成员函数的专用实例，用于缺省不能处理给出的足够类型时覆盖缺省实例。

trapping（陷阱操作） 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。

type（类型） 使用符号方法的描述。基本类型包括 `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。

variable（变量） 标识符命名的数据项。每个变量都有类型（如 `int` 或 `void`）和作用域。另请参见 class variable（类变量）、instance variable（实例变量）、local variable（局部变量）。

VTABLE（虚拟表） 编译器为包含虚函数的每个类创建的表。

索引

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Z

《

半

包

保

本

编

变

标

别

酷

扩

垃

类

联

链

流

幂

命

模

目

内

匿

配

前

强