Oracle® Solaris Studio 12.2：C 用户指南

文件号码 821–2510
2010 年 9 月

ORACLE

101019@24999

目录

前言 21

1 C编译器介绍 25
1.1 5.11 版 Solaris Studio 12 Update 2 发行版的新增功能 25
1.2 x86 特殊注意事项 26
1.3 二进制兼容验证 26
1.4 针对 64 位平台进行编译 27
1.5 标准一致性 27
1.6 C 自述文件 27
1.7 手册页 28
1.8 编译器的组织结构 28
1.9 与 C 相关的编程工具 30

2 特定于C编译器实现的信息 31
2.1 常量 31
2.1.1 整型常量 31
2.1.2 字符常量 32
2.2 链接程序作用域说明符 32
2.3 线程局部存储说明符 33
2.4 浮点，非标准模式 34
2.5 作为值的标签 34
2.6 `long long` 数据类型 36
2.6.1 输出 `long long` 数据类型 36
2.6.2 常见算术转换 36
2.7 switch 语句中的 case 范围 37
2.8 断言 38
2.9 支持的属性 39

2.10 警告和错误 40
2.11 Pragma 40
2.11.1 align 40
2.11.2 c99 40
2.11.3 does_not_read_global_data 41
2.11.4 does_not_return 41
2.11.5 does_not_write_global_data 41
2.11.6 error_messages 42
2.11.7 fini 42
2.11.8 hdrstop 42
2.11.9 ident 43
2.11.10 init 43
2.11.11 inline 44
2.11.12 int_to_unsigned 44
2.11.13 MP serial_loop 44
2.11.14 MP serial_loop_nested 45
2.11.15 MP taskloop 45
2.11.16 nomemorydepend 45
2.11.17 no_side_effect 45
2.11.18 opt 46
2.11.19 pack 46
2.11.20 pipeloop 47
2.11.21 rarely_called 47
2.11.22 redefine_extname 48
2.11.23 returns_new_memory 49
2.11.24 unknown_control_flow 49
2.11.25 unroll 49
2.11.26 warn_missing_parameter_info 50
2.11.27 weak 50
2.12 预定义的名称 51
2.13 保留 `errno` 的值 51
2.14 扩展 52
2.14.1 `_Restrict` 关键字 52
2.14.2 `_ _asm` 关键字 52
2.14.3 `__inline` 和 `__inline__` 52
2.14.4 `__builtin_constant_p()` 52

2.14.5 __FUNCTION__ 和 __PRETTY_FUNCTION__ 53
2.15 环境变量 53
2.15.1 OMP_DYNAMIC 53
2.15.2 OMP_NESTED 53
2.15.3 OMP_NUM_THREADS 53
2.15.4 OMP_SCHEDULE 53
2.15.5 PARALLEL 53
2.15.6 SUN_PROFDATA 53
2.15.7 SUN_PROFDATA_DIR 53
2.15.8 SUNW_MP_THR_IDLE 54
2.15.9 TMPDIR 54
2.16 如何指定 include 文件 54
2.16.1 使用 -I- 选项更改搜索算法 55
2.17 在独立式环境中编译 57

3 并行化 C 代码 59
3.1 概述 59
3.1.1 使用示例 59
3.2 OpenMP 并行化 60
3.2.1 处理 OpenMP 运行时警告 60
3.3 环境变量 60
3.3.1 PARALLEL 或 OMP_NUM_THREADS 60
3.3.2 SUNW_MP_THR_IDLE 61
3.3.3 SUNW_MP_WARN 61
3.3.4 STACKSIZE 61
3.3.5 在并行代码中使用 **restrict** 62
3.4 数据依赖性和干扰 62
3.4.1 并行执行模型 63
3.4.2 私有标量和私有数组 65
3.4.3 返回存储 67
3.4.4 约简变量 67
3.5 加速 68
3.5.1 Amdahl 定律 68
3.6 负载平衡和循环调度 71
3.6.1 静态调度或块调度 71

3.6.2 自我调度 72
3.6.3 引导自我调度 72
3.7 循环变换 72
3.7.1 循环分布 72
3.7.2 循环合并 73
3.7.3 循环交换 74
3.8 别名和并行化 75
3.8.1 数组引用和指针引用 75
3.8.2 限定指针 75
3.8.3 显式并行化和 Pragma 76
3.9 内存边界内部函数 83

4 lint 源代码检验器 85
4.1 基本和增强 lint 模式 85
4.2 使用 lint 86
4.3 lint 选项 87
4.3.1 -# 87
4.3.2 -### 88
4.3.3 -a 88
4.3.4 -b 88
4.3.5 -C *filename* 88
4.3.6 -c 88
4.3.7 -dirout=*dir* 88
4.3.8 -err=warn 88
4.3.9 -errchk=*l*(, *l*) 88
4.3.10 -errfmt=*f* 89
4.3.11 -errhdr=*h* 90
4.3.12 -erroff=*tag*(, *tag*) 90
4.3.13 -errsecurity=*v* 91
4.3.14 -errtags=*a* 92
4.3.15 -errwarn=*t* 92
4.3.16 -F 93
4.3.17 -fd 93
4.3.18 -flagsrc=*file* 93
4.3.19 -h 93

4.3.20 -I*dir* 93
4.3.21 -k 93
4.3.22 -L*dir* 93
4.3.23 -l*x* 93
4.3.24 -m 94
4.3.25 -m32|-m64 94
4.3.26 -Ncheck=*c* 94
4.3.27 -Nlevel=*n* 95
4.3.28 -n 96
4.3.29 -o*x* 96
4.3.30 -p 96
4.3.31 -R*file* 96
4.3.32 -s 96
4.3.33 -u 96
4.3.34 -V 96
4.3.35 -v 96
4.3.36 -W*file* 97
4.3.37 -XCC=*a* 97
4.3.38 -Xalias_level[=*l*] 97
4.3.39 -Xarch=amd64 97
4.3.40 -Xarch=v9 97
4.3.41 -Xc99[= *o*] 97
4.3.42 -Xkeeptmp=*a* 98
4.3.43 -Xtemp=*dir* 98
4.3.44 -Xtime=*a* 98
4.3.45 -Xtransition=*a* 98
4.3.46 -Xustr={ascii_utf16_ushort| no} 98
4.3.47 -x 98
4.3.48 -y 99
4.4 lint 消息 99
4.4.1 用于禁止消息的选项 99
4.4.2 lint 消息格式 100
4.5 lint 指令 102
4.5.1 预定义值 102
4.5.2 指令 102
4.6 lint 参考和示例 105

4.6.1 由 lint 执行的诊断 105
4.6.2 lint 库 108
4.6.3 lint 过滤器 109

5 基于类型的别名分析 111
5.1 介绍基于类型的分析 111
5.2 使用 Pragma 以便更好地控制 112
5.2.1 #pragma alias_level *level* (*list*) 112
5.3 使用 lint 检查 114
5.3.1 标量指针向结构指针的强制类型转换 114
5.3.2 空指针向结构指针的强制类型转换 115
5.3.3 结构字段向结构指针的强制类型转换 115
5.3.4 要求显式别名 115
5.4 内存引用约束的示例 116
5.4.1 第一个示例 116
5.4.2 第二个示例 118
5.4.3 第三个示例 119
5.4.4 第四个示例 121
5.4.5 第五个示例 123
5.4.6 第六个示例 123
5.4.7 第七个示例 124

6 转换为 ISO C 125
6.1 基本模式 125
6.1.1 -Xc 125
6.1.2 -Xa 125
6.1.3 -Xt 125
6.1.4 -Xs 126
6.2 旧式和新式函数的混合 126
6.2.1 编写新代码 126
6.2.2 更新现有代码 126
6.2.3 混合注意事项 127
6.3 带有可变参数的函数 129
6.4 提升：无符号保留与值保留 131
6.4.1 背景 131

6.4.2 编译行为 131
6.4.3 第一个示例：强制类型转换的使用 131
6.4.4 位字段 132
6.4.5 第二个示例：相同的结果 132
6.4.6 整型常量 133
6.4.7 第三个示例：整型常量 133
6.5 标记化和预处理 134
6.5.1 ISO C 转换阶段 134
6.5.2 旧 C 转换阶段 135
6.5.3 逻辑源代码行 135
6.5.4 宏替换 135
6.5.5 使用字符串 136
6.5.6 标记粘贴 136
6.6 `const` 和 `volatile` 137
6.6.1 类型（仅适用于 `lvalue`） 137
6.6.2 派生类型中的类型限定符 137
6.6.3 `const` 意味着 `readonly` 138
6.6.4 `const` 用法示例 139
6.6.5 `volatile` 意味着精确语义 139
6.6.6 `volatile` 用法示例 139
6.7 多字节字符和宽字符 140
6.7.1 亚洲语言需要多字节字符 140
6.7.2 编码变种 140
6.7.3 宽字符 140
6.7.4 转换函数 141
6.7.5 C 语言特征 141
6.8 标准头文件和保留名称 142
6.8.1 标准头文件 142
6.8.2 保留供实现使用的名称 142
6.8.3 保留供扩展使用的名称 143
6.8.4 可安全使用的名称 143
6.9 国际化 144
6.9.1 语言环境 144
6.9.2 `setlocale()` 函数 144
6.9.3 更改的函数 145
6.9.4 新函数 146

6.10 表达式中的分组和求值 146
6.10.1 定义 147
6.10.2 K&R C 重新整理许可证 147
6.10.3 ISO C 规则 147
6.10.4 圆括号 148
6.10.5 As If 规则 148
6.11 不完全类型 149
6.11.1 类型 149
6.11.2 完成不完全类型 149
6.11.3 声明 149
6.11.4 表达式 150
6.11.5 正当理由 150
6.11.6 示例 150
6.12 兼容类型和复合类型 151
6.12.1 多个声明 151
6.12.2 分别编译兼容性 151
6.12.3 单编译兼容性 151
6.12.4 兼容指针类型 151
6.12.5 兼容数组类型 152
6.12.6 兼容函数类型 152
6.12.7 特殊情况 152
6.12.8 复合类型 152

7 转换应用程序以适用于 64 位环境 153
7.1 数据模型差异概述 153
7.2 实现单一源代码 154
7.2.1 派生类型 154
7.2.2 工具 157
7.3 转换为 LP64 数据类型模型 158
7.3.1 整型和指针长度更改 158
7.3.2 整型和长型长度更改 158
7.3.3 符号扩展 159
7.3.4 指针运算而不是整数 160
7.3.5 结构 160
7.3.6 联合 161

7.3.7 类型常量 161
7.3.8 注意隐式声明 161
7.3.9 sizeof( ) 是无符号 long 162
7.3.10 使用强制类型转换显示您的意图 162
7.3.11 检查格式字符串转换操作 162
7.4 其他考虑事项 163
7.4.1 长度增长的派生类型 163
7.4.2 检查更改的副作用 164
7.4.3 检查直接使用 long 是否仍有意义 164
7.4.4 对显式 32 位与 64 位原型使用 #ifdef 164
7.4.5 调用转换更改 164
7.4.6 算法更改 164
7.5 入门指南清单 165

8 cscope：交互检查 C 程序 167
8.1 cscope 进程 167
8.2 基本用法 168
8.2.1 步骤 1：设置环境 168
8.2.2 步骤 2：调用 cscope 程序 168
8.2.3 步骤 3：查找代码 169
8.2.4 步骤 4：编辑代码 174
8.2.5 命令行选项 175
8.2.6 视图路径 176
8.2.7 cscope 和编辑器调用栈 177
8.2.8 示例 177
8.2.9 编辑器的命令行语法 180
8.3 未知终端类型错误 181

A 按功能分组的编译器选项 183
A.1 按功能汇总的选项 183
A.1.1 优化和性能选项 183
A.1.2 编译时选项和链接时选项 185
A.1.3 数据对齐选项 186
A.1.4 数值和浮点选项 186
A.1.5 并行化选项 187

A.1.6 源代码选项 187
A.1.7 编译代码选项 189
A.1.8 编译模式选项 189
A.1.9 诊断选项 190
A.1.10 调试选项 190
A.1.11 链接选项和库选项 191
A.1.12 目标平台选项 192
A.1.13 x86 特定选项 192
A.1.14 许可证选项 193
A.1.15 废弃的选项 193

B C 编译器选项参考 195
B.1 选项语法 195
B.2 cc 选项 196
B.2.1 -# 196
B.2.2 -### 196
B.2.3 -A*name*[(*tokens*)] 196
B.2.4 -B[static| dynamic] 197
B.2.5 -C 197
B.2.6 -c 197
B.2.7 -D*name*[(*arg*[,*arg*])][= *expansion*] 197
B.2.8 -d[y|n] 198
B.2.9 -dalign 198
B.2.10 -E 198
B.2.11 -errfmt[=[no%]error] 198
B.2.12 -errhdr [=h] 198
B.2.13 -erroff[= *t*] 199
B.2.14 -errshort[= *i*] 199
B.2.15 -errtags[= *a*] 200
B.2.16 -errwarn[= *t*] 200
B.2.17 -fast 201
B.2.18 -fd 203
B.2.19 -features=[*v*] 203
B.2.20 -flags 203
B.2.21 -flteval[={ any|2}] 203

B.2.22 -fma[={none| fused}] 204
B.2.23 -fnonstd 204
B.2.24 -fns[={no|yes}] 204
B.2.25 **-fPIC** 205
B.2.26 **-fpic** 205
B.2.27 -fprecision=*p* 205
B.2.28 -fround=*r* 205
B.2.29 -fsimple[= *n*] 206
B.2.30 -fsingle 207
B.2.31 -fstore 207
B.2.32 -ftrap=*t*[,*t*...] 207
B.2.33 -G 208
B.2.34 -g 208
B.2.35 -H 209
B.2.36 -h *name* 209
B.2.37 -I[- | *dir*] 210
B.2.38 -i 210
B.2.39 -include *filename* 210
B.2.40 -KPIC 211
B.2.41 -Kpic 211
B.2.42 -keeptmp 211
B.2.43 -L*dir* 211
B.2.44 -l*name* 211
B.2.45 -m32|-m64 212
B.2.46 -mc 212
B.2.47 -misalign 212
B.2.48 -misalign2 212
B.2.49 -mr[, *string*] 213
B.2.50 -mt[={yes |no}] 213
B.2.51 -native 213
B.2.52 -nofstore 214
B.2.53 -O 214
B.2.54 -o *filename* 214
B.2.55 -P 214
B.2.56 -p 214
B.2.57 -Q[y|n] 214

B.2.58 -qp 214
B.2.59 -R*dir*[:*dir*] 215
B.2.60 -S 215
B.2.61 -s 215
B.2.62 -traceback[={ %none|common|*signals_list*}] 215
B.2.63 -U*name* 216
B.2.64 -V 216
B.2.65 -v 216
B.2.66 -W*c* ,*arg* 217
B.2.67 -w 218
B.2.68 -X[c|a| t|s] 218
B.2.69 -x386 218
B.2.70 -x486 219
B.2.71 -xaddr32[=yes|no] 219
B.2.72 -xalias_level[=*l*] 219
B.2.73 - xannotate[=yes|no] 221
B.2.74 –xarch=*isa* 221
B.2.75 -xautopar 225
B.2.76 -xbinopt={prepare| off} 226
B.2.77 -xbuiltin[=(%all|%none)] 226
B.2.78 -xCC 226
B.2.79 -xc99[=*o*] 227
B.2.80 -xcache[=*c*] 227
B.2.81 –xcg[89|92] 229
B.2.82 -xchar[=*o*] 229
B.2.83 -xchar_byte_order[=*o*] 230
B.2.84 -xcheck[=*o*] 230
B.2.85 -xchip[=*c*] 232
B.2.86 -xcode[=*v*] 234
B.2.87 -xcrossfile 236
B.2.88 -xcsi 236
B.2.89 -xdebugformat=[stabs|dwarf] 236
B.2.90 -xdepend=[yes| no] 237
B.2.91 -xdryrun 237
B.2.92 -xe 237
B.2.93 -xF[=*v*[,*v*...]] 237

B.2.94 -xhelp=*f* 238
B.2.95 -xhwcprof 238
B.2.96 -xinline=*list* 239
B.2.97 -xinstrument=[no%]datarace 240
B.2.98 -xipo[= *a*] 241
B.2.99 -xipo_archive=[*a*] 242
B.2.100 -xjobs=*n* 243
B.2.101 -xldscope={*v*} 244
B.2.102 -xlibmieee 245
B.2.103 -xlibmil 245
B.2.104 -xlibmopt 245
B.2.105 -xlic_lib=sunperf 245
B.2.106 -xlicinfo 245
B.2.107 -xlinkopt[= *level*] 246
B.2.108 -xloopinfo 247
B.2.109 -xM 247
B.2.110 -xM1 247
B.2.111 -xMD 248
B.2.112 -xMF *filename* 248
B.2.113 -xMMD 248
B.2.114 -xMerge 248
B.2.115 -xmaxopt[=*v*] 249
B.2.116 -xmemalign=*ab* 249
B.2.117 -xmodel=[*a*] 250
B.2.118 -xnolib 251
B.2.119 -xnolibmil 251
B.2.120 -xnolibmopt 251
B.2.121 -xnorunpath 252
B.2.122 -xO[1|2| 3|4|5] 252
B.2.123 -xopenmp[= *i*] 254
B.2.124 -xP 255
B.2.125 -xpagesize=*n* 255
B.2.126 -xpagesize_heap=*n* 256
B.2.127 -xpagesize_stack=*n* 256
B.2.128 -xpch=*v* 257
B.2.129 -xpchstop=[*file*|<include>] 261

B.2.130 - xpec[={yes|no}] 261
B.2.131 -xpentium 262
B.2.132 -xpg 262
B.2.133 -xprefetch[= *val*[, *val*]] 262
B.2.134 -xprefetch_auto_type=*a* 263
B.2.135 -xprefetch_level=*l* 264
B.2.136 –xprofile=*p* 264
B.2.137 -xprofile_ircache[= *path*] 267
B.2.138 -xprofile_pathmap 267
B.2.139 -xreduction 268
B.2.140 -xregs=*r*[, *r*...] 268
B.2.141 -xrestrict[=*f*] 269
B.2.142 -xs 270
B.2.143 -xsafe=mem 270
B.2.144 -xsb 270
B.2.145 -xsbfast 270
B.2.146 -xsfpconst 271
B.2.147 -xspace 271
B.2.148 -xstrconst 271
B.2.149 -xtarget=*t* 271
B.2.150 -xtemp=*dir* 274
B.2.151 -xthreadvar[= *o*] 274
B.2.152 -xtime 275
B.2.153 -xtransition 275
B.2.154 -xtrigraphs 275
B.2.155 -xunroll=*n* 276
B.2.156 -xustr={ascii_utf16_ushort| no} 276
B.2.157 -xvector[= *a*] 277
B.2.158 -xvis 277
B.2.159 -xvpara 278
B.2.160 -Y*c* , *dir* 278
B.2.161 -YA, *dir* 278
B.2.162 -YI, *dir* 279
B.2.163 -YP, *dir* 279
B.2.164 -YS, *dir* 279
B.2.165 -Zll 279

B.3 传递给链接程序的选项 279

C 实现定义的 ISO/IEC C99 行为 281
C.1 实现定义的行为 (J.3) 281
C.1.1 转换 (J.3.1) 281
C.1.2 环境 (J.3.2) 282
C.1.3 标识符 (J.3.3) 284
C.1.4 字符 (J.3.4) 284
C.1.5 整数 (J.3.5) 286
C.1.6 浮点 (J.3.6) 286
C.1.7 数组和指针 (J.3.7) 287
C.1.8 提示 (J.3.8) 287
C.1.9 结构、联合、枚举和位字段 (J.3.9) 288
C.1.10 限定符 (J.3.10) 289
C.1.11 预处理指令 (J.3.11) 289
C.1.12 库函数 (J.3.12) 290
C.1.13 体系结构 (J.3.13) 295
C.1.14 语言环境特定的行为 (J.4) 298

D 支持的 C99 功能 301
D.1 讨论和示例 301
D.1.1 浮点计算器的精度 302
D.1.2 C99 关键字 303
D.1.3 `__func__` 支持 303
D.1.4 通用字符名 (UCN) 303
D.1.5 使用 // 注释代码 304
D.1.6 禁止隐式 `int` 和隐式函数声明 304
D.1.7 使用隐式 `int` 的声明 304
D.1.8 灵活的数组成员 305
D.1.9 幂等限定符 306
D.1.10 `inline` 函数 306
D.1.11 `Static` 及数组声明符中允许的其他类型限定符 307
D.1.12 可变长度数组 (VLA)： 307
D.1.13 指定的初始化函数 308
D.1.14 混合声明和代码 309

D.1.15 for 循环语句中的声明 309
D.1.16 具有可变数目的参数的宏 309
D.1.17 _Pragma 310

E 实现定义的 ISO/IEC C90 行为 313
E.1 与 ISO 标准比较的实现 313
E.1.1 转换 (G.3.1) 313
E.1.2 环境 (G.3.2) 314
E.1.3 标识符 (G.3.3) 314
E.1.4 字符 (G.3.4) 314
E.1.5 整数 (G.3.5) 316
E.1.6 浮点 (G.3.6) 317
E.1.7 数组和指针 (G.3.7) 319
E.1.8 寄存器 (G.3.8) 319
E.1.9 结构、联合、枚举和位字段 (G.3.9) 319
E.1.10 限定符 (G.3.10) 322
E.1.11 声明符 (G.3.11) 322
E.1.12 语句 (G.3.12) 322
E.1.13 预处理指令 (G.3.13) 322
E.1.14 库函数 (G.3.14) 324
E.1.15 语言环境特定的行为 (G.4) 330

F ISO C 数据表示法 333
F.1 存储分配 333
F.2 数据表示法 334
F.2.1 整数表示法 335
F.2.2 浮点表示法 336
F.2.3 异常值 338
F.2.4 选定的数的十六进制表示 339
F.2.5 指针表示 339
F.2.6 数组存储 340
F.2.7 异常值的算术运算 340
F.3 参数传递机制 342
F.3.1 32 位 SPARC 342
F.3.2 64 位 SPARC 342

F.3.3 x86/x64 342

G 性能调节 345
G.1 `libfast.a` 库 (SPARC) 345

H K&R Solaris Studio C 与 Solaris Studio ISO C 之间的差异 347
H.1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 347
H.2 关键字 352

索引 355

前言

《Oracle Solaris Studio 12 Update 2 C 用户指南》介绍 Oracle Solaris Studio C 编译器 **cc** 的环境和命令行选项。本指南是专门为具有 C 语言实际知识并希望学习如何有效使用 Solaris Studio C 编译器的程序员编写的。同时假定读者大致了解 Solaris 或 Linux 操作环境。

受支持的平台

此 Oracle Solaris Studio 发行版支持使用以下 SPARC 和 x86 系列处理器体系结构的系统：UltraSPARC、SPARC64、AMD64、Pentium 和 Xeon EM64T。可从以下网址获得硬件兼容性列表，在该列表中可以查看您正在运行的 Oracle Solaris 操作系统版本所支持的系统：`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 Solaris Studio 文档

可以访问以下位置的文档：

- 可从以下位置的文档索引页面中获取文档：`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`。
- IDE、性能分析器、dbxtool 和 DLight 的所有组件的联机帮助可以在这些工具中通过“帮助”菜单、F1 键以及许多窗口和对话框上的“帮助”按钮获取。

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，docs.sun.com 上的 Oracle Solaris Studio 12.2 Collection - Simplified Chinese 中
《Oracle Solaris Studio 12.2 发行版的新增功能》（以前的组件自述文件）	HTML，docs.sun.com 上的 Oracle Solaris Studio 12.2 Collection - Simplified Chinese 中
手册页	使用 man 命令在 Oracle Solaris 终端中显示
联机帮助	HTML，可在 IDE、dbxtool、dbxtool、DLight 和性能分析器中通过“帮助”菜单、“帮助”按钮和 F1 键获取。
发行说明	HTML，docs.sun.com 上的 Oracle Solaris Studio 12.2 Collection - Simplified Chinese 中

相关的第三方 Web 站点引用

本文档引用了第三方 URL，以用于提供其他相关信息。

注 – Oracle 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，Oracle 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，Oracle 概不负责，也不承担任何责任。

开发者资源

要找到以下经常更新的资源，请访问 `http://www.oracle.com/technetwork/server-storage/solarisstudio`：

- 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 Oracle Solaris Studio 工具逐步完成开发任务的教程
- 有关支持级别的信息
- `http://forums.sun.com/category.jspa?categoryID=113` 上的用户论坛

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	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	《书名》新词或术语以及要强调的词	阅读《用户指南》的第 6 章。 **高速缓存**是存储在本地的副本。 请**勿**保存文件。 **注意**： 有些强调的项目在联机时以粗体显示。

命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的 shell 的缺省 UNIX 系统提示符和超级用户提示符。请注意，显示在命令示例中的缺省系统提示符会根据 Oracle Solaris 发行版的不同而有所不同。

表 P–2 Shell 提示符

Shell	提示符
Bash shell、Korn shell 和 Bourne shell	`$`
Bash shell、Korn shell 和 Bourne shell 超级用户	`#`
C shell	`machine_name%`
C shell 超级用户	`machine_name#`

文档、支持和培训

有关其他资源，请参见以下 Web 站点：

- 文档 (http://docs.sun.com)
- 支持 (http://www.oracle.com/us/support/systems/index.html)
- 培训 (http://education.oracle.com) – 单击左侧导航栏中的 Sun 链接。

Oracle 欢迎您提出意见

Oracle 欢迎您针对其文档质量和实用性提出意见和建议。如果您发现任何错误，或有其他任何改进建议，请转至 http://docs.sun.com，然后单击 Feedback（反馈）。请提供文档的标题和文件号码，以及章节和页码（如果有）。如果您需要回复，请告知。

Oracle 技术网络 (http://www.oracle.com/technology/global/cn/index.html) 提供了一系列与 Oracle 软件相关的资源：

- 在讨论论坛 (http://forums.oracle.com) 上讨论技术问题和解决方案。
- 通过 Oracle By Example (http://www.oracle.com/technology/obe/start/index.html) 获得逐步骤实用教程。
- 下载样例代码 (http://www.oracle.com/technology/sample_code/index.html)。

◆ ◆ ◆ 第 1 章

1 C 编译器介绍

本章提供有关 Oracle Solaris Studio C 编译器的基本信息。

1.1 5.11 版 Solaris Studio 12 Update 2 发行版的新增功能

请注意当前的 C 编译器发行版中以下新增和更改的功能。

ABI 更改要求重新编译：
对 C 编译器的更改将纠正以 64 位模式在 SPARC 处理器上传递和返回包含复杂类型的结构的方式。以前，这些结构值有时会传递和返回到错误的寄存器中，并且创建与 gcc 所创建的二进制文件不兼容的二进制文件。因为此更改在 C 编译器中实现时将影响现有 ABI 的元素，所以如果应用程序中的任何源文件使用具有复杂字段的结构时，必须重新编译应用程序的整个源代码库才能避免出现错误应答的可能性。但 32 位 SPARC 处理器和 32/64 位 x86 处理器的编译不受此更改的影响。

- 添加了对 SPARC-V9 ISA 的 SPARC VIS3 版本的支持。如果使用 `-xarch=sparcvis3` 选项进行编译，编译器可以使用 SPARC-V9 指令集、UltraSPARC 和 UltraSPARC-III 扩展、混合乘加指令以及可视指令集 (Visual Instruction Set, VIS) 版本 3.0 中的指令。(第 221 页中的 “B.2.74 –xarch=*isa*”)
- 在基于 x86 的系统上，`-xvector` 选项的缺省值已更改为 `-xvector=simd`。(第 277 页中的 “B.2.157 -xvector[= *a*]”)
- 现在，使用 `-xarch=amdsse4a` 选项可支持 AMD SSE4a 指令集。(第 221 页中的 “B.2.74 –xarch=*isa*”)
- `-traceback` 选项使可执行文件在出现严重错误时列显栈跟踪。(第 215 页中的 “B.2.62 -traceback[={ %none|common|*signals_list*}]”)
- `-mt` 选项已更改为 `-mt=yes` 或 `-mt=no`。(第 213 页中的 “B.2.50 -mt[={yes |no}]”)
- `#warning` 编译器会在指令中发出文本作为警告并继续编译。(第 40 页中的 “2.10 警告和错误”)
- 新的 pragma `does_not_read_global_data`、`does_not_write_global_data` 以及 `no_side_effect`。(第 40 页中的 “2.11 Pragma”)

- 现已提供头文件 `mbarrier.h`。为 SPARC 和 x86 处理器中的多线程代码定义不同的内存边界内部函数。(第 83 页中的 “3.9 内存边界内部函数”)
- `-xprofile=tcov` 选项经过增强，支持可选的配置文件目录路径名，还能够生成与 `tcov` 兼容的反馈数据。(第 264 页中的 “B.2.136 –xprofile=*p*”)
- 在此发行版中，`-xMD` 和 `-xMMD` 选项 (C/C++) 写入的依赖性文件将覆写以前存在的任何文件。(第 248 页中的 “B.2.111 -xMD”)

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。

在 `-xarch` 设置为 `sse`、`sse2`、`sse2a`、`sse3` 或更高级别的情况下编译的程序必须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）。

因为内部数学库（例如，`sin(x)`）不同，所以 Solaris 和 Linux 之间的数值结果也会不同。

1.3 二进制兼容验证

在 Solaris 系统上，从 Solaris Studio 11 开始，Solaris 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）。运行时，会检查这些标记标志以验证该二进制文件是否可以在它尝试在上面执行的硬件上运行。

如果在没有相应功能或指令集扩展的平台上运行不包含这些体系结构硬件标志的程序，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。

此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（使用 SSE、SSE2、SSE2a 和 SSE3 以及更新指令和扩展）的程序。

1.4 针对 64 位平台进行编译

使用 -m32 选项针对 ILP32 32 位模型进行编译。使用 -m64 选项针对 LP64 64 位模型进行编译。

ILP32 模型指定 C 语言 int、long 和 pointer 数据类型均为 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。

1.5 标准一致性

本书中所用术语 C99 是指 ISO/IEC 9899:1999 C 编程语言。术语 C90 是指 ISO/IEC 9899:1990 C 编程语言。

当您指定 -xc99=all,lib -Xc 时，此编译器在 Solaris 平台上完全遵循 C99 标准。

此编译器还遵循 ISO/IEC 9899:1990（编程语言－C 标准）。

由于此编译器还支持传统的 K&R C（Kernighan 和 Ritchie，即 ANSI 之前的 C），因此便于迁移到 ISO C。

有关 C90 实现特定行为的信息，请参见第 310 页中的 “D.1.17 _Pragma”。

有关 C99 特性的更多信息，请参见表 C–6。

1.6 C 自述文件

C 编译器的自述文件强调了关于编译器的重要信息。它现在属于《Oracle Solaris Studio 12.2 的新增功能》指南的一部分，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

转到 http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation 上的 Oracle Solaris Studio 12.2 文档页面，便可以看到该新增功能指南。

1.7 手册页

联机手册 (man) 页提供了有关命令、函数、子例程的当前文档或此类内容的集合。

您可以通过运行以下命令来显示 C 编译器的手册页：

```
example% man cc
```

在整个 C 文档中，手册页参考以主题名称和手册节编号显示： cc(1) 通过 man cc 进行访问。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
```

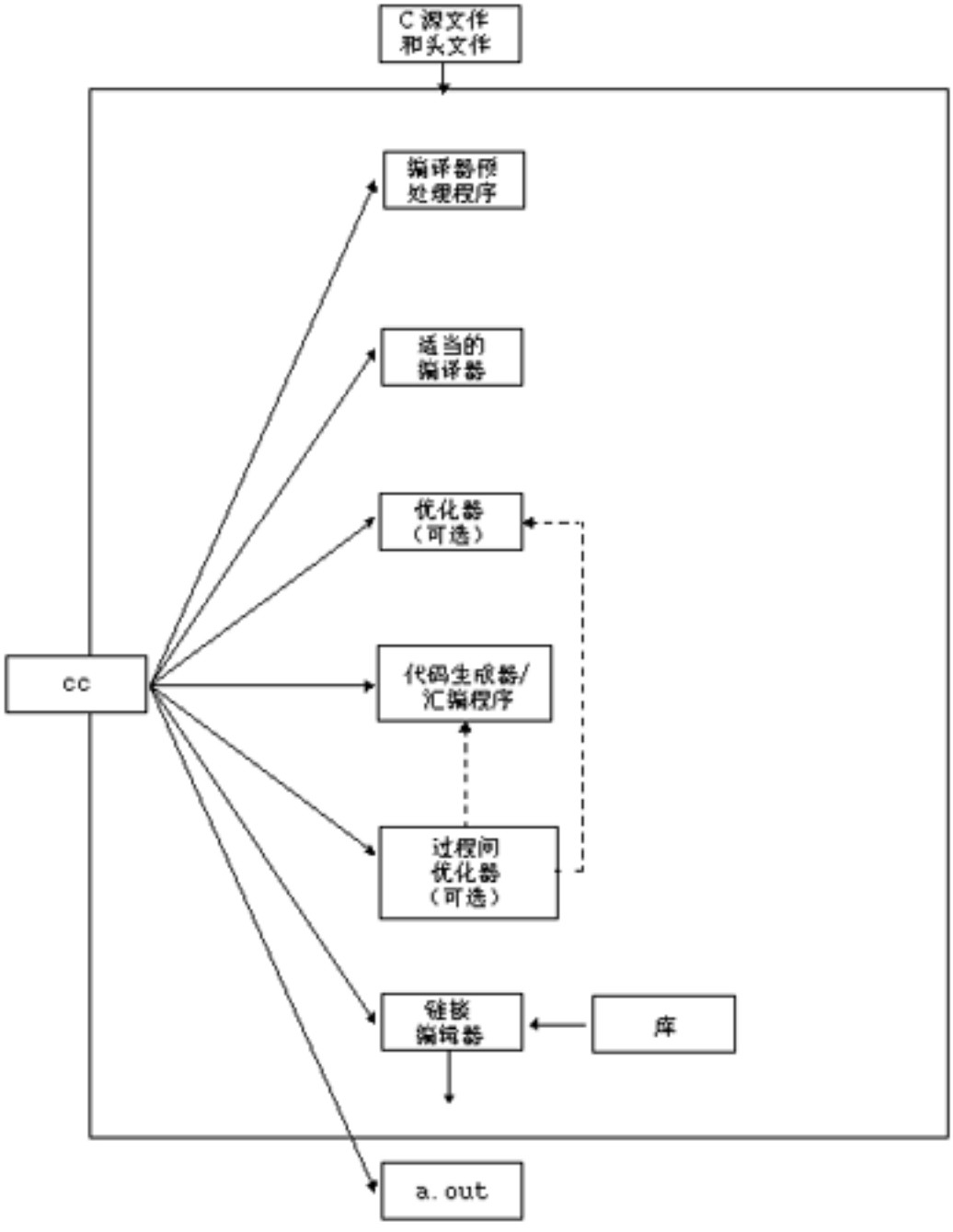
1.8 编译器的组织结构

C 编译系统由一个编译器、一个汇编程序和一个链接编辑器组成。cc 命令会自动调用这些组件中的每个组件，除非您使用命令行选项另行指定。

表 A–15 介绍了 cc 的所有可用选项。

下图显示 C 编译系统的组织结构。

图 1–1　C 编译系统的组织结构



下表汇总了编译系统的组件。

表 1–1 C 编译系统的组件

组件	说明	使用说明
cpp	预处理程序	仅适用于 -Xs
acomp	编译器（用于非 Xs 模式的内置预处理程序）	
ssbd	静态同步错误检测	*(SPARC)*
iropt	代码优化器	-O、-xO2、-xO3、-xO4、-xO5、-fast
fbe	汇编程序	
cg	代码生成器、内联函数、汇编程序	
ipo	过程间优化器	
postopt	后优化器	*(SPARC)*
ir2hf	中间代码翻译者	(x86)
ube	代码生成器	(x86)
ld	链接程序	
mcs	处理注释部分	-mr

1.9 与 C 相关的编程工具

有多种工具可用来帮助您开发、维护和改进 C 程序。本书中介绍了两个与 C 联系最紧密的工具：cscope 和 lint。此外，每个工具均有手册页。

◆ ◆ ◆ 第 2 章

2 特定于 C 编译器实现的信息

本章讨论特定于 C 编译器方面的内容。该信息已编入语言扩展和环境。

C 编译器与新的 ISO C 标准 ISO/IEC 9899-1999 中介绍的某些 C 语言功能兼容。如果您希望编译与以前的 C 标准 ISO/IEC 9889-1990 标准（及修正案 1）兼容的代码，请使用 `-xc99=none`，这样编译器将忽略 ISO/IEC 9899-1999 标准的增强功能。

2.1 常量

本节包含与特定于 Solaris Studio C 编译器的常量有关的信息。

2.1.1 整型常量

十进制、八进制和十六进制的整型常量可加后缀以指示类型，如下表所示。

表 2–1 数据类型后缀

后缀	类型
`u` 或 `U`	`unsigned`
`l` 或 `L`	`long`
`ll` 或 `LL`	`long long`（在 `-xc99=none` 和 `-Xc` 模式下不可用）
`lu`、`LU`、`Lu`、`lU`、`ul`、`uL`、`Ul` 或 `UL`	`unsigned long`
`llu`、`LLU`、`LLu`、`llU`、`ull`、`ULL`、`uLL`、`Ull`	`unsigned long long`（在 `-xc99=none` 和 `-Xc` 模式下不可用）

如果设置 `-xc99=all`，编译器将根据常量大小，使用以下列表中可以表示该值的第一项：

- int
- long int
- long long int

如果值超过 long long int 可表示的最大值，编译器会发出警告。

如果设置 -xc99=none，则为无后缀常量指定类型时，编译器将根据常量大小，使用以下列表中可以表示该值的第一项：

- int
- long int
- unsigned long int
- long long int
- unsigned long long int

2.1.2 字符常量

一个多字符常量，它不是具有从每个字符的数值派生的值的转义序列。例如，常量 '123' 的值为：

0	'3'	'2'	'1'

或 0x333231。

使用 -Xs 选项并且在 C 的其他非 ISO 版本中，该值为：

0	'1'	'2'	'3'

或者 0x313233。

2.2 链接程序作用域说明符

使用以下声明说明符有助于隐藏外部符号的声明和定义。通过使用这些说明符，您不必再使用链接程序作用域的 mapfile。此外，还可以通过在命令行中指定 -xldscope 来控制变量作用域的缺省设置。有关更多信息，请参见第 244 页中的 “B.2.101 -xldscope={*v*}”。

表 2–2 声明说明符

值	含义
`__global`	该符号具有全局链接程序作用域，并且是限制最少的链接程序作用域。该符号的所有引用都绑定到在第一个动态模块中定义该符号的定义上。该链接程序作用域是外部符号的当前链接程序作用域。
`__symbolic`	该符号具有符号链接程序作用域，该作用域的限制比全局链接程序作用域的限制更多。从所链接的动态模块内部对符号的所有引用都绑定到在模块内部定义的符号上。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`。有关链接程序的更多信息，请参见 `ld(1)`。
`__hidden`	该符号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。动态模块内部的所有引用都绑定到该模块内部的一个定义上。符号在模块外部是不可视的。

对象或函数可以用限制较多的说明符重新声明，但不能用限制较少的说明符重新声明。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。

2.3 线程局部存储说明符

通过声明线程局部变量，可以利用线程局部存储。线程局部变量声明由一个标准变量声明外加变量说明符 `__thread` 组成。有关更多信息，请参见第 274 页中的 “B.2.151 `-xthreadvar[=o]`”。

您必须在所编译的源文件中线程变量的第一个声明中包含 `__thread` 说明符。

在具有静态存储持续时间的对象的声明中，只能使用 `__thread` 说明符。您可以如初始化任何其他静态存储持续时间的对象一样静态地初始化线程变量。

使用 `__thread` 说明符声明的变量与不使用 `__thread` 说明符声明的变量具有相同的链接程序绑定。这包括临时定义，如无初始化函数的声明。

线程变量的地址不是常量。因此，线程变量的地址操作符 (&) 在运行时求值，并返回当前线程的线程变量的地址。结果，静态存储持续时间的对象被动态地初始化为线程变量的地址。

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。线程终止后，该线程的变量的所有地址都无效。

2.4 浮点，非标准模式

缺省情况下，IEEE 754 浮点运算“不停止”，并且下溢是“渐进的”。下面给出了概括性说明，有关详细信息，请参见《数值计算指南》。

不停止意味着遇到除数为零、浮点下溢或无效操作异常时执行不会停止。例如，考虑以下算式，其中 x 为零，y 为正数：

```
z = y / x;
```

缺省情况下，z 设置为值 +Inf，执行不会停止。但是，如果设置 -fnonstd 选项，此代码会导致退出，如核心转储。

下面是渐进下溢的工作方式。假设您有下列代码：

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
x = x / 10;
```

第一次执行循环时，x 设置为 1；第二次执行循环时，设置为 0.1；第三次执行循环时，设置为 0.01，依此类推。最后，x 达到比较低的值，以致机器无法表示其值。下次循环运行时将出现什么情况?

假设可表示的最小数为 1.234567e-38

下次循环运行时，将通过去掉一位尾数并且指数减去 1 来修改该数，因此新值为 1.23456e-39，然后为 1.2345e-40，依此类推。这称为“渐进下溢”，它是缺省行为。在非标准模式下，不会发生这种“去位”情况；通常，x 被简单地设置为零。

2.5 作为值的标签

C 编译器可识别称为计算转移 (computed goto) 语句的 C 扩展。使用计算转移 (computed goto) 语句能够在运行时确定分支目标。通过使用 '&&' 运算符可以获取标签的地址，并且可以将标签地址指定给 void * 类型的指针：

```
void *ptr;
...
ptr = &&label1;
```

后面的 goto 语句可以通过 ptr 转到 label1：

```
goto *ptr;
```

由于 ptr 在运行时进行计算，因此 ptr 可以接受作用域内任何标签的地址，而 goto 语句可以转到该位置。

使用计算转移 (computed goto) 语句的一种方法是用于转移表的实现：

```
static void *ptrarray[] = { &&label1, &&label2, &&label3 };
```

现在可以通过索引来选择数组元素：

```
goto *ptrarray[i];
```

标签的地址只能通过当前函数作用域计算。尝试在当前函数外部获取标签的地址会产生不可预测的结果。

转移表和开关语句的作用相似（虽然存在某些主要差异），转移表使跟踪程序流更加困难。一个显著的差异是：开关语句转移目标全都从开关保留字开始正向转移；使用计算转移 (computed goto) 语句实现转移表可进行正向和反向分支。

```
#include <stdio.h>
void foo()
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return;

  label1:
  printf("Passed!\n");
  return;
}

int main(void)
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return 0;

  label1:
  foo();
  return 0;
}
```

以下示例也使用转移表控制程序流：

```
#include <stdio.h>

int main(void)
{
  int i = 0;
  static void * ptr[3]={&&label1, &&label2, &&label3};
```

```
  goto *ptr[i];

  label1:
  printf("label1\n");
  return 0;

  label2:
  printf("label2\n");
  return 0;

  label3:
  printf("label3\n");
  return 0;
}

%example: a.out
%example: label1
```

计算转移 (computed goto) 语句的另一个应用是作为线程代码的解释程序。解释程序函数内部的标签地址可以存储在线程代码中以便快速分发。

2.6 long long 数据类型

使用 -xc99=none 进行编译时，Solaris Studio C 编译器包含数据类型 long long 和 unsigned long long，它们与数据类型 long 类似。long long 数据类型存储 64 位信息；long 使用 -m32 编译时存储 32 信息。long 数据类型使用 -m64 编译时存储 64 位信息。long long 在 -Xc 模式下不可用（发出警告）。

2.6.1 输出 long long 数据类型

要输出或扫描 long long 数据类型，请在转换说明符前面加字母 ll。例如，要以带符号十进制格式输出 llvar（long long 数据类型的变量），请使用：

```
printf("%lld\n", llvar);
```

2.6.2 常见算术转换

某些二元运算符对其操作数的类型进行转换以便两个操作数具有相同的类型，该类型也是结果的类型。下面这些转换称为常见算术转换：

- 如果两个操作数中的任何一个的类型为 long double，则另一个操作数转换成类型 long double。
- 否则，如果两个操作数中的任何一个的类型为 double，则另一个操作数转换成类型 double。
- 否则，如果两个操作数中的任何一个的类型为 float，则另一个操作数转换成类型 float。

- 否则，对两个操作数执行整型提升。然后，应用以下规则：
  - 如果两个操作数中的任何一个的类型为 `unsigned long long int`，则另一个操作数转换成类型 `unsigned long long int`。
  - 如果两个操作数中的任何一个的类型为 `long long int`，则另一个操作数转换成类型 `long long int`。
  - 如果两个操作数中的任何一个的类型为 `unsigned long int`，则另一个操作数转换成类型 `unsigned long int`。
  - 否则，仅在 SPARC V9 上进行编译并指定了 `cc -xc99=none` 时，如果一个操作数的类型为 `long int` 而另一个操作数的类型为 `unsigned int`，则两个操作数的类型均转换为 `unsigned long int`。
  - 否则，如果两个操作数中的任何一个的类型为 `long int`，则另一个操作数转换为类型 `long int`。
  - 否则，如果两个操作数中的任何一个的类型为 `unsigned int`，则另一个操作数转换为类型 `unsigned int`。
  - 否则，两个操作数的类型均为 `int`。

2.7 switch 语句中的 case 范围

在标准 C 中，switch 语句中的 case 标签只能有一个关联值。Solaris Studio C 允许使用某些编译器中使用的扩展（称为 case 范围）。

case 范围指定要与单个 case 标签关联的值范围。case 范围语法为：

`case` *low* **`...`** *high* **`:`**

case 范围的行为就好像为从 *low* 到 *high*（含）的给定范围内的每个值指定了 case 标签。（如果 *low* 和 *high* 相等，则 case 范围仅指定一个值。）较低值和较高值必须符合 C 标准的要求。也就是说，它们必须是有效的整数常量表达式（C 标准 6.8.4.2）。case 范围和 case 标签可以随意混合，一个 switch 语句中可以指定多个 case 范围。

编程示例：

```
enum kind { alpha, number, white, other };
enum kind char_class(char c);
{
     enum kind result;
     switch(c) {
         case ‘a’ ... ’z’:
         case ‘A’ ... ’Z’:
             result = alpha;
             break;
         case ’0’ ... ’9’:
             result = number;
             break;
         case ’ ’:
```

```
        case '\n':
        case '\t':
        case '\r':
        case '\v':
            result = white;
            break;
        default:
            result = other;
            break;
    }
    return result; }
```

除了 case 标签的现有要求以外的错误情形：

- 如果 low 的值大于 high 的值，则编译器会拒绝代码并显示错误消息。（其他编译器的行为不一致，因此，只有通过错误消息才能确保程序在由其他编译器编译时不会表现出不同的行为。）
- 如果 case 标签的值在已在 switch 语句中使用的 case 范围内，则编译器会拒绝代码并显示错误消息。
- 如果 case 范围重叠，则编译器会拒绝代码并显示错误消息。

请注意，如果 case 范围的一个端点是数值，则在省略号 (**...**) 两侧留空格以避免其中一个点被视为小数点。

示例：

```
case 0...4;   // error
case 5 ... 9; // ok
```

2.8 断言

以下形式的一行内容：

`#assert` *predicate (token-sequence)*

将 *token-sequence* 和断言名称空间（与用于宏定义的空间不同）中的谓词相结合。谓词必须为标识符标记。

`#assert` *predicate*

断言 *predicate* 存在，但是未与任何标记序列相结合。

缺省情况下，编译器提供以下预定义谓词（不在 `-Xc` 模式下）：

```
#assert system (unix)
#assert machine (sparc)
#assert machine (i386)(x86)
#assert cpu (sparc)
#assert cpu (i386)(x86)
```

缺省情况下，lint 提供以下预定义谓词（不在 -Xc 模式下）：

```
#assert lint (on)
```

任何断言均可使用 #unassert 进行删除，该命令的语法与 assert 的语法相同。使用不带参数的 #unassert 将删除关于谓词的所有断言；指定一个断言将只删除该断言。

可以使用以下语法在 #if 语句中测试断言：

```
#if #predicate(non-empty token-list)
```

例如，可以使用以下行测试预定义谓词 system：

```
#if #system(unix)
```

其结果为真。

2.9 支持的属性

为便于兼容性，编译器实现以下属性 (__attribute__ ((*keyword*))):

- **always_inline**－等效于 #pragma inline 和 -xinline
- **noinline**－等效于 #pragma no_inline 和 -xinline
- **pure**－等效于 #pragma does_not_write_global_data
- **const**－等效于 #pragma no_side_effect
- **malloc**－等效于 #pragma returns_new_memory
- **aligned**－大概等效于 #pragma align，但受结构和联合类型的限制。
- **constructor**－等效于 #pragma init
- **destructor**－等效于 #pragma fini
- **alias**－为已声明的功能或变量名称创建一个别名
- **weak**－等效于 #pragma weak
- **noreturn**－等效于 #pragma does_not_return
- **packed**－等效于 #pragma pack()
- **visibility**－提供链接程序作用范围，如第 32 页中的 “2.2 链接程序作用域说明符”中所述。语法是：__attribute__((visibility("*visibility_type*"))), 其中 *visibility_type* 是以下选项之一：
  - default－与 __global 连接程序作用范围相同
  - hidden－与 __hidden 连接程序作用范围相同
  - internal－与 __symbolic 连接程序作用范围相同

2.10 警告和错误

`#error` 和 `#warning` 处理器指令可用于生成编译时诊断。

`#error` *token-string*　　发送错误诊断 *token-string* 并终止编译操作

`#warning` *token-string*　　发送警告诊断 *token-string* 并继续执行编译操作。

2.11 Pragma

以下形式的预处理行：

`#pragma` *pp-tokens*

指定实现定义的操作。

编译系统可识别以下 `#pragma`。编译器忽略未识别的 pragma。如果使用 `-v` 选项，将针对无法识别的 pragma 发出警告。

2.11.1 align

`#pragma align` *integer* (*variable*[, *variable*])

align pragma 会使所有提及的变量内存与**整数**字节对齐，从而覆盖缺省值。请遵循以下限制：

- *integer* 值必须是介于 1 和 128 之间的 2 的幂，有效值包括 1、2、4、8、16、32、64 和 128。
- *variable* 是全局变量或静态变量，它不能为自动变量。
- 如果指定的对齐比缺省值小，就使用缺省值。
- pragma 行必须在它提到的变量的声明前面出现；否则，它将被忽略。
- 提到但未在 pragma 行后面的文本中声明的任何变量将被忽略。例如：

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct astruct{int a; char *b;};
```

2.11.2 c99

`#pragma c99(“implicit” | “ no%implicit”)`

该 pragma 控制隐式函数声明的诊断。如果将 c99 pragma 值设置为 "implicit"（请注意使用了引号），则编译器在找到隐式函数声明时将生成一条警告。如果将 c99 pragma 值设置为 "no%implicit"（请注意使用了引号），则编译器将无提示地接受隐式函数声明，直到该 pragma 值重置。

-xc99 选项的值会影响该 pragma。如果 -xc99=all，则该 pragma 被设置为 #pragma c99("implicit")；如果 -xc99=none，则该 pragma 被设置为 #pragma c99("no%implicit")。

缺省情况下，该 pragma 设置为 c99=("implicit")。

2.11.3 does_not_read_global_data

#pragma does_not_read_global_data (*funcname* [, *funcname*])

该 pragma 断言指定列表中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。

必须在该 pragma 之前使用原型或空参数列表声明指定的函数。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。

2.11.4 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，寄存器生命周期将在调用点终止，进而允许更多的优化。

如果指定的函数不返回，程序的行为就是未定义的。只有在使用原型或空参数列表声明指定的函数之后才允许使用该 pragma，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)
```

2.11.5 does_not_write_global_data

#pragma does_not_write_global_data (*funcname [, funcname]*)

该 pragma 断言指定列表的例程不直接或间接写全局数据。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。

必须在该 pragma 之前使用原型或空参数列表声明指定的函数。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。

2.11.6 error_messages

`#pragma error_messages (on|off| default,` *tag…* *tag*`)`

错误消息 pragma 提供源程序内部对 C 编译器和 lint 发出的消息的控制。对于 C 编译器，pragma 只对警告消息有效。C 编译器的 `-w` 选项通过禁止显示所有警告消息来覆盖该 pragma。

- `#pragma error_messages (on,` *tag…* *tag*`)`

  `on` 选项结束前面的任何 `#pragma error_messages` 选项（如 off 选项）的作用域，并覆盖 `-erroff` 选项的效果。

- `#pragma error_messages (off,` *tag…* *tag*`)`

  `off` 选项阻止 C 编译器或 lint 程序发出以 pragma 中指定的标记开头的指定消息。pragma 对任何指定的错误消息的作用域仍然有效，直到被另一个 `#pragma error_messages` 覆盖或编译结束。

- `#pragma error_messages (default,` *tag…* *tag*`)`

  `default` 选项结束前面的任何 `#pragma error_messages` 指令对指定标记的作用域。

2.11.7 fini

`#pragma fini (`*f1[, f2…,fn]*

使实现在调用 *main()* 例程之后调用函数 *f1* 至 `fn`（完成函数）。此类函数的类型应为 `void`，并且不接受任何参数，当程序正常终止或所含共享对象从内存中删除时会调用这些函数。和“初始化函数”一样，完成函数按链接编辑器的处理顺序执行。

如果完成函数影响全局程序的状态，则应当小心操作。例如，除非接口明确声明您使用系统库完成函数时会发生什么情况，否则您应捕获和恢复所有全局状态信息，如系统库完成函数可能更改的 `errno` 的值。

此类函数每出现在 **`#pragma fini`** 指令中一次，就会被调用一次。

2.11.8 hdrstop

`#pragma hdrstop`

hdrstop pragma 必须放在最后一个头文件之后，以标识要共享相同预编译头文件的每个源文件中活前缀的结束。例如，考虑以下文件：

```
example% cat a.c
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
example% cat b.h
#include "a.h"
#include "b.h"
#include "c.h"
```

活动源代码前缀在 c.h 结束，因此您需要在每个文件中在 c.h 后面插入 #pragma hdrstop。

#pragma hdrstop 必须只在使用 cc 命令指定的源文件的活前缀的末尾出现。不要在任何 include 文件中指定 #pragma hdrstop。

2.11.9 ident

`#pragma ident` *string*

将 *string* 放在可执行文件的 `.comment` 部分。

2.11.10 init

`#pragma init (`*f1[, f2…,fn]* `)`

使实现在调用 *main()* 之前调用函数 *f1* 至 *fn*（初始化函数）。此类函数的类型应为 `void`，并且不接受任何参数，在开始执行时构造程序的内存映像时会调用这些函数。如果初始化函数在共享对象中，则在执行将共享对象放入内存的操作（无论是程序启动，还是某些动态装入操作，如 `dlopen()`）时执行它们。调用初始化函数的唯一顺序是链接编辑器处理它们的顺序，静态和动态均可。

如果初始化函数影响全局程序的状态，则应当小心操作。例如，除非接口明确声明您使用系统库初始化函数时会发生什么情况，否则您应捕获和恢复所有全局状态信息，如系统库初始化函数可能更改的 `errno` 的值。

此类函数每出现在 **`#pragma init`** 指令中一次，就会被调用一次。

2.11.11 inline

`#pragma [no_]inline (`*funcname[, funcname]*`)`

该 pragma 对 pragma 的参数中列出的例程名称的内联进行控制。该 pragma 的作用域针对整个文件。该 pragma 只允许全局内联控制，不允许特定于调用点的控制。

如果您使用 `#pragma inline`，它会提示编译器内联当前文件中与 pragma 中列出的例程列表匹配的调用。在某些情况下，此建议可能被忽略。例如，当函数的主体在另一个模块并且未使用交叉文件选项时，忽略该建议。

如果您使用 `#pragma no_inline`，它会提示编译器不要内联当前文件中与 pragma 中列出的例程列表匹配的调用。

只有在使用原型或空参数列表声明函数之后，才允许使用 `#pragma inline` 和 `#pragma no_inline`，如下例所示：

```
static void foo(int);
static int bar(int, char *);
#pragma inline(foo, bar)
```

另请参见 `-xldscope`、`-xinline`、`-xO` 和 `-xcrossfile`。

2.11.12 int_to_unsigned

`#pragma int_to_unsigned (`*funcname* `)`

对于返回类型 `unsigned` 的函数，在 `-Xt` 或 `-Xs` 模式下，将函数返回值的类型更改为 `int`。

2.11.13 MP serial_loop

(SPARC) `#pragma MP serial_loop`

注 – Sun 特定的旧 MP pragma 已过时，并且不再受支持。但是，编译器改为支持 OpenMP 3.0 规范指定的 API。有关标准的指令的迁移信息，请参见《OpenMP API 用户指南》。

有关详细信息，请参阅第 77 页中的 “3.8.3.1 串行 Pragma”。

2.11.14 MP serial_loop_nested

(SPARC) `#pragma MP serial_loop_nested`

注 – Sun 特定的旧 MP pragma 已过时，并且不再受支持。但是，编译器改为支持 OpenMP 3.0 规范指定的 API。有关标准的指令的迁移信息，请参见《Solaris Studio OpenMP API 用户指南》。

有关详细信息，请参阅第 77 页中的 “3.8.3.1 串行 Pragma”。

2.11.15 MP taskloop

(SPARC) `#pragma MP taskloop`

注 – Sun 特定的旧 MP pragma 已过时，并且不再受支持。但是，编译器改为支持 OpenMP 3.0 规范指定的 API。有关标准的指令的迁移信息，请参见《OpenMP API 用户指南》。

有关详细信息，请参阅第 77 页中的 “3.8.3.2 并行 Pragma”。

2.11.16 nomemorydepend

(SPARC) `#pragma nomemorydepend`

该 pragma 指定，对于某个循环的任何迭代，不存在内存依赖性。也就是说，在某个循环的任何迭代内部，不存在对相同内存的引用。该 pragma 将允许编译器（流水线化程序）在某个循环的单次迭代内更有效地调度指令。如果某个循环的任何迭代内部存在任何内存依赖性，则程序的执行结果未定义。编译器在优化级别 3 或更高级别上使用此信息。

此 pragma 的作用域从它自身开始，在以下任何一种情况最先出现时结束：下一块的开始部分、当前块内部的下一个 for 循环、当前块的结尾。该 pragma 应用于 pragma 作用域结束前的下一个 for 循环。

2.11.17 no_side_effect

`#pragma no_side_effect(`*funcname[, funcname…]* `)`

funcname 指定当前转换单元内部某个函数的名称。必须在该 pragma 之前使用原型或空参数列表声明的函数。必须在函数的定义之前指定 pragma。对于命名的函数 *funcname*，该 pragma 声明函数无任何副作用。这意味着，*funcname* 返回一个只依赖传递参数的结果值。另外，*funcname* 和任何已调用的子函数具有以下特性：

- 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。

使用函数进行优化时，编译器使用此信息。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。编译器在优化级别 3 或更高级别上使用此信息。

2.11.18 opt

`#pragma opt` *level* (*funcname[, funcname]*)

funcname 指定当前转换单元内部定义的某个函数的名称。*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 来关闭优化。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。

2.11.19 pack

`#pragma pack(`*n*`)`

使用 `#pragma pack(`*n*`)` 将影响结构或联合的成员封装。缺省情况下，结构或联合的成员按其自然边界对齐；一个字符型 (char) 数据占一个字节，一个短整型 (short) 数据占两个字节，一个整型 (integer) 数据占四个字节，等等。如果存在 *n*，它必须为 2 的幂，并且为任何结构或联合成员指定最严格的自然对齐。不接受零。

您可以使用 `#pragma pack(`*n*`)` 为结构或联合成员指定对齐边界。例如，`#pragma pack(2)` 会使 int、long、long long、float、double、long double 和指针与双字节边界对齐，而不是与其自然边界对齐。

如果 *n* 等于或大于您使用的平台上最严格的对齐（在 x86 上使用 `-m32` 时为 4，在 SPARC 上使用 `-m32` 时为 8，但使用 `-m64` 时为 16），则指令具有自然对齐的效果。同样，如果省略 *n*，成员对齐将恢复为自然对齐边界。

`#pragma pack(`*n*`)` 指令应用于它后面的所有结构或联合定义，直到出现下一个 pack 指令。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或联合，那么程序会因某种原因而失败。特别是，不应在包含定义了预编译库的接口的头文件之前使用

`#pragma pack(`*n*`)`。`#pragma pack(`*n*`)` 的建议用法是，将它放在程序代码中紧挨在要封装的任何结构或联合的前面。在封装的结构后面紧跟 `#pragma pack( )`。

请注意，使用 `#pragma pack` 时，封装的结构或联合本身的对齐方式与其更严格对齐的成员相同。因此，该结构或联合的任何声明将使用压缩对齐。例如，只包含 char 数据的 struct 无对齐限制，而包含 double 数据的 struct 将按 8 字节边界对齐。

注 – 如果您使用 `#pragma pack` 将结构或联合成员与其自然边界以外的边界对齐，则访问这些字段通常会导致 SPARC 出现总线错误。为避免发生此类错误，请确保同时还指定了 `-xmemalign` 选项。有关编译此类程序的最佳方法，请参见第 249 页中的 “B.2.116 `-xmemalign=`*ab*”。

2.11.20 pipeloop

`#pragma pipeloop(`*n*`)`

对于参数 *n*，该 pragma 接受正整数常量值或 0。该 pragma 指定，循环可流水线化，并且循环携带的依赖性的最小依赖距离为 *n*。如果该距离为 0，则循环实际上是 Fortran 风格的 `doall` 循环，并且应在目标处理程序上流水线化。如果该距离大于 0，则编译器（流水线化程序）将只尝试流水线化 *n* 次连续迭代。编译器在优化级别 3 或更高级别上使用此信息。

此 pragma 的作用域从它自身开始，在以下任何一种情况最先出现时结束：下一块的开始部分、当前块内部的下一个 for 循环、当前块的结尾。该 pragma 应用于 pragma 作用域结束前的下一个 for 循环。

2.11.21 rarely_called

`#pragma rarely_called(`*funcname[, funcname]* `)`

该 pragma 提示编译器，指定的函数很少被调用。这样，编译器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。

必须在该 pragma 之前使用原型或空参数列表声明指定的函数。以下是 `#pragma rarely_called` 的示例：

```
extern void error (char *message);
#pragma rarely_called(error)
```

2.11.22 redefine_extname

`#pragma redefine_extname` *old_extname new_extname*

该 pragma 导致目标代码中名称 *old_extname* 的各个外部定义具体值被 *new_extname* 替换。结果，链接程序在链接时只看到名称 *new_extname*。如果在第一次使用 *old_extname* 作为函数定义、初始化函数或表达式之后遇到 `#pragma redefine_extname`，则该作用未定义。（在 `–Xs` 模式下不支持该 pragma。）

如果 `#pragma redefine_extname` 可用，编译器会提供预定义宏 `__PRAGMA_REDEFINE_EXTNAME` 的定义，这样您可以编写在有无 `#pragma redefine_extname` 的条件下均可运行的可移植代码。

`#pragma redefine_extname` 的目的是，提供一种在函数名称无法更改时重新定义函数接口的有效方法。例如，当某个库中必须保留初始函数定义时，为了与现有程序兼容，创建同一函数的新定义以供新程序使用。这可以通过用新名称将新函数定义增加到库中来完成。因此，声明函数的头文件使用 `#pragma redefine_extname`，以便对函数的所有使用均与该函数的新定义链接。

```
#if    defined(__STDC__)

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine(const long *, int *);
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(const long * arg1, int * arg2)
{
    extern int __myroutine(const long *, int*);
    return (__myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#else /* __STDC__ */

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine();
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(arg1, arg2)
    long *arg1;
    int *arg2;
{
    extern int __fixed_myroutine();
    return (__fixed_myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#endif /* __STDC__ */
```

2.11.23 returns_new_memory

`#pragma returns_new_memory (`*funcname[, funcname]*`)`

该 pragma 断言指定函数的返回值在调用点上不使用任何内存作为别名。实际上，该调用返回一个新存储单元。该信息使优化器更好地跟踪指针值并澄清存储单元。这将导致循环的调度、流水线化和并行化的改进。然而，如果断言为假，则程序的行为未定义。

只有在使用原型或空参数列表声明指定的函数之后才允许使用该 pragma，如以下示例所示：

```
void *malloc(unsigned);
#pragma returns_new_memory(malloc)
```

2.11.24 unknown_control_flow

`#pragma unknown_control_flow (`*funcname[, funcname]*`)`

为了描述改变函数调用方流程图的过程，C 编译器提供了 `#pragma unknown_control_flow` 指令。通常，该指令带有 `setjmp()` 等函数的声明。在 Solaris 系统上，include 文件 `<setjmp.h>` 包含：

```
extern int setjmp();
#pragma unknown_control_flow(setjmp)
```

同样，必须声明具有 `setjmp()` 类似属性的其他函数。

原则上，识别该属性的优化器可在控制流程图中插入相应的界限，从而在调用 `setjmp()` 的函数中安全地处理函数调用，同时保持优化流程图的未受影响部分的代码的能力。

必须在该 pragma 之前使用原型或空参数列表声明指定的函数。

2.11.25 unroll

`#pragma unroll (`*unroll_factor*`)`

对于参数 *unroll_factor*，该 pragma 接受正整数常量值。当解开因子不为 1 时，该指令作为对编译器的建议，指出指定的循环应由给定的因子解开。如果可能，编译器将使用该解开因子。如果解开因子值为 1，该指令作为一个命令，向编译器指出不解开该循环。编译器在优化级别 3 或更高级别上使用此信息。

此 pragma 的作用域从它自身开始，在以下任何一种情况最先出现时结束：下一块的开始部分、当前块内部的下一个 for 循环、当前块的结尾。该 pragma 应用于 pragma 作用域结束前的下一个 for 循环。

2.11.26 warn_missing_parameter_info

```
#pragma [no_]warn_missing_parameter_info
```

如果指定 `#pragma warn_missing_parameter_info`，编译器将针对函数声明不包含任何参数类型信息的函数调用发出一条警告。请看以下示例：

```
example% cat -n t.c
     1    #pragma warn_missing_parameter_info
     2
     3    int foo();
     4
     5    int bar () {
     6
     7       int i;
     8
     9       i = foo(i);
    10
    11       return i;
    12    }
% cc t.c -c -errtags
"t.c", line 9: warning: function foo has no prototype (E_NO_MISSED_PARAMS_ALLOWED)
example%
```

`#pragma no_warn_missing_parameter_info` 将使任何以前的 `#pragma warn_missing_parameter_info` 无效。

缺省情况下，`#pragma no_warn_missing_parameter_info` 是有效的。

2.11.27 weak

`#pragma weak` *symbol1 [= symbol2]*

定义弱全局符号。该 pragma 主要在源文件中用于生成库。如果链接程序无法解析弱符号，它并不生成错误消息。

`#pragma weak` *symbol*

将 *symbol* 定义为弱符号。如果链接程序找不到 *symbol* 的定义，它不会生成错误消息。

`#pragma weak` *symbol1* `=` *symbol2*

将 *symbol1* 定义为弱符号，它是符号 *symbol2* 的别名。只能在已定义 *symbol2*（可以在源文件中定义，也可以在其中一个包含的头文件中定义）的同一转换单元中使用这种形式的 pragma。否则，将导致编译错误。

如果程序调用但未定义 *symbol1*，并且 *symbol1* 在所链接的库中是弱符号，则链接程序使用该库中的定义。但是，如果程序定义自己的 *symbol1* 版本，则使用该程序的定义，而不使用库中 *symbol1* 的弱全局定义。如果程序直接调用 *symbol2*，则使用库中的定义；*symbol2* 的重复定义会导致出现错误。

2.12 预定义的名称

cc(1) 手册页中给出了最新的预定义列表。

下面的标识符被预定义为类似于对象的宏：

表 2–3 预定义标识符 `__STDC__`

扩展到:	编译时使用：
`1`	`-Xc`
`0`	`-Xa,-Xt`
未定义	`-Xs`

如果 `__STDC__` 未定义 (`#undef __STDC__`)，编译器将发出警告。`__STDC__` 未在 `-Xs` 模式下定义。

2.13 保留 errno 的值

使用 `-fast`，编译器可以用不设置 `errno` 变量的等效优化代码自由替换对浮点函数的调用。而且，`-fast` 还会定义宏 `__MATHERR_ERRNO_DONTCARE`，使编译器无需确保 `errno` 的有效性。因此，在浮点函数调用后依赖于 `errno` 值的用户代码可能生成不一致的结果。

解决此问题的一种方法是避免用 `-fast` 编译此类代码。但是，如果需要 `-fast` 优化并且代码依赖于在浮点库调用后正确设置的 `errno` 值，应使用以下选项进行编译

```
-xbuiltin=none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

然后，在命令行上使用 `-fast`，以禁止编译器优化此类库调用，并确保正确处理 `errno`。

2.14 扩展

C 编译器针对 C 语言实现了许多扩展。

2.14.1 _Restrict 关键字

C 编译器支持 _Restrict 关键字，该关键字与 C99 标准中的 restrict 关键字等效。_Restrict 关键字可与 -xc99=none 和 -xc99=all 一起使用，而 restrict 关键字只能与 -xc99=all 一起使用。

有关支持的 C99 特性的更多信息，请参见表 C–6。

2.14.2 __asm 关键字

__asm 关键字（注意开头的两个下划线）是 asm 关键字的同义字。如果您使用 asm 而不是 __asm，并且在 -Xc 模式下编译，则编译器会发出警告。如果您在 –Xc 模式下使用 __asm，则编译器不会发出警告。__asm 语句采用以下形式：

```
__asm("string");
```

其中 *string* 是有效的汇编语言语句。

该语句将给定的汇编程序文本直接发送到汇编文件。在文件作用域（而不是函数作用域）声明的基本 asm 语句称为**全局** *asm* **语句**。其他编译器将其称为**顶级** asm 语句。

全局 asm 语句是按指定的顺序发出的。也就是说，它们保留相对于彼此的顺序，并保持相对于前后函数的位置。

在较高优化级别中，编译器可能会删除认为不被引用的函数。由于编译器不知道从全局 asm 中引用了哪些函数，因此可能会无意中删除这些函数。

请注意，那些提供模板及操作数规范的扩展 asm 语句不允许作为全局语句。**__asm** 和 **__asm__** 是 **asm** 关键字的同义字，可以互换使用。

2.14.3 __inline 和 __inline__

__inline 和 __inline__ 是 inline 关键字的同义字（C 标准，第 6.4.1 节）

2.14.4 __builtin_constant_p()

__builtin_constant_p 是编译器内置函数。它接受一个数值参数，如果已知参数是一个编译时常量，则返回 1。返回值 0 意味着编译器无法确定参数是否是编译时常量。此内置函数的典型用法是在宏中用于手动编译时优化。

2.14.5 __FUNCTION__ 和 __PRETTY_FUNCTION__

__FUNCTION__ 和 __PRETTY_FUNCTION__ 是预定义标识符，这些标识符包含词法上封闭的函数的名称。它们在功能上等效于 c99 预定义标识符 __func__。在 Solaris 平台上，__FUNCTION__ 和 __PRETTY_FUNCTION__ 在 **-Xs** 和 **-Xc** 模式下不可用。

2.15 环境变量

本节列出用于控制编译和运行环境的环境变量。

2.15.1 OMP_DYNAMIC

启用或禁用线程数的动态调整。

2.15.2 OMP_NESTED

启用或禁用嵌套并行操作。

2.15.3 OMP_NUM_THREADS

设置执行过程中要使用的线程数。

2.15.4 OMP_SCHEDULE

设置运行时调度类型和块大小。

2.15.5 PARALLEL

指定可供程序进行多处理器执行的处理器数。如果目标机器具有多个处理器，线程可以映射到独立的处理器。运行该程序将导致创建执行程序的并行化部分的两个线程。

2.15.6 SUN_PROFDATA

控制 -xprofile=collect 命令在其中存储执行频率数据的文件的名称。

2.15.7 SUN_PROFDATA_DIR

控制 -xprofile=collect 命令在其中放置执行频率数据文件的目录。

2.15.8 SUNW_MP_THR_IDLE

控制每个辅助线程的任务结束状态，可设置为 `spin` *n*`s` 或 `sleep` *n* `ms`。缺省值为 `sleep`。有关详细信息，请参见《OpenMP API 用户指南》。

2.15.9 TMPDIR

`cc` 通常在目录 `/tmp` 中创建临时文件。可以通过将环境变量 `TMPDIR` 设置为您选定的目录，来指定其他目录。但是，如果 `TMPDIR` 不是有效目录，`cc` 将使用 `/tmp`。`-xtemp` 选项优先于 `TMPDIR` 环境变量。

如果您使用 Bourne shell，请键入：

```
$ TMPDIR=dir; export TMPDIR
```

如果您使用 C shell，请键入：

```
% setenv TMPDIR dir
```

2.16 如何指定 include 文件

要包含 C 编译系统提供的任何标准头文件，请使用以下格式：

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

尖括号 (<>) 导致预处理程序在系统上头文件的标准位置搜索头文件，此位置通常是 `/usr/include` 目录。

对于您已存储在您自己的目录中的头文件，格式不同：

```
#include "header.h"
```

对于 `#include "foo.h"` 形式的语句（其中使用了引号），编译器按以下顺序搜索 include 文件：

1. 当前目录（即放置“包含”文件的目录）
2. 以 `-I` 选项命名的目录（如果有）
3. `/usr/include` 目录

如果头文件所在的目录与包含该头文件的源文件所在的目录不同，请指定使用 `cc` 及 `-I` 选项存储头文件时所用目录的路径。例如，假设在源文件 `mycode.c` 中已包含 `stdio.h` 和 `header.h`：

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

进一步假设 header.h 存储在目录 ../defs 中。命令：

```
% cc– I../defs mycode.c
```

指示预处理程序首先在包含 mycode.c 的目录中搜索 header.h，然后在目录 ../defs 中搜索，最后在标准位置搜索。它还指示预处理程序首先在 ../defs 中搜索 stdio.h，然后在标准位置搜索。不同之处在于：仅对于其名称用引号括起的头文件，才查找当前目录。

您可以在 cc 命令行上多次指定 -I 选项。预处理程序按指定目录出现的顺序查找它们。您可以在同一命令行上对 cc 指定多个选项：

```
% cc– o prog– I../defs mycode.c
```

2.16.1 使用 -I- 选项更改搜索算法

新的 -I- 选项提供对缺省搜索规则的更多控制。只有命令行上的第一个 -I- 选项的作用如本节所述。当命令行上出现 -I- 时：

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- 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。

下例显示在编译 prog.c 时使用 -I- 的结果。

```
prog.c
#include "a.h"

#include <b.h>

#include "c.h"


c.h
#ifndef _C_H_1

#define _C_H_1
```

```
int c1;

#endif


int/a.h
#ifndef _A_H

#define _A_H

#include "c.h"

int a;

#endif


int/b.h
#ifndef _B_H

#define _B_H

#include <c.h>

int b;

#endif
int/c.h
#ifndef _C_H_2

#define _C_H_2

int c2;

#endif
```

以下命令显示了在当前目录（包含文件的目录）中搜索 `#include "foo.h"` 形式的包含语句的缺省行为。当处理 `inc/a.h` 中的 `#include "c.h"` 语句时，预处理程序包含 `inc` 子目录中的 `c.h` 头文件。当处理 `prog.c` 中的 `#include "c.h"` 语句时，预处理程序包含具有 `prog.c` 的目录中的 `c.h` 文件。请注意， `-H` 选项指示编译器输出所包含文件的路径。

```
example% cc -c -Iinc -H prog.c
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。当预处理程序处理 `#include "foo.h"` 形式的语句时，它并不首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。处理 `inc/a.h` 中的 `#include "c.h"` 语句时，预处理程序包含 `./c.h` 头文件，而不是 `inc/c.h` 头文件。

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

```
inc/b.h
            inc/c.h
./c.h
```

2.16.1.1 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。

有关更多信息，请参见第 210 页中的 “B.2.37 -I[- | *dir*]”。

2.17 在独立式环境中编译

Solaris Studio C 支持编译能与标准 C 库链接并在包含标准 C 库和其他运行时支持库的运行时环境中执行的程序。C 标准中为此类环境指明了一个**托管**环境。而为未提供标准库函数的环境指明了一个**独立式**环境。

针对于独立式环境，一般情况下 C 编译器不支持编译，因为某些可能从编译代码中调用的运行时支持函数一般仅在标准 C 库中可用。问题是：编译器转换的源代码可能将调用引入至不包含函数调用的源代码结构中的运行时支持函数中，并且这些函数一般在独立式的环境中不可用。请看以下示例：

```
% cat -n lldiv.c
     1    void
     2    lldiv(
     3        long long *x,
     4        long long *y,
     5        long long *z)
     6    {
     7        *z = *x / *y ;
     8    }
% cc -c -m32 lldiv.c
% nm lldiv.o | grep " U "
 0x00000000 U __div64
% cc -c -m64 lldiv.c
% nm lldiv.o | grep " U "
```

在该示例中，使用 **-m32** 选项编译源文件 lldiv.c 使其在 32 位平台上运行时，对第 7 行中声明的转换将导致外部引用名为 __div64 的运行时支持函数，并且此函数是 32 位版本的标准 C 库中唯一可使用的函数。

使用 **-m64** 选项编译同一源文件使其在 64 位平台上运行，编译器将使用目标计算机中的 64 位算法指令集，但其中不包括 64 位版本的标准 C 库中运行时支持函数所需要的指令。

尽管在一般情况下不支持使用 C 编译器将独立式环境视为目标，但在遵从注意事项的情况下可使用此编译器在特定的独立式环境（即 Solaris 内核和设备驱动程序）中编译代码。

必须写入在 Solaris 内核中运行的代码（包括设备驱动程序），这样外部函数调用才能引用那些只在内核中可用的函数。要使以上情况成为可能，建议遵从以下准则：

1. 对于只在用户模式下运行的库，不要包含头文件。
2. 除非确定内核中存在与标准 C 库或其他用户模式库中相同的函数，否则不要调用这些函数。
3. 不要使用浮点类型或 C99 复合类型。
4. 不要使用与运行时支持库相关联的编译选项，例如 **-xprofile** 和 **-xopenmp**。

   cc(1) 手册页的“文件”一节中介绍了与特定编译器选项关联的可重定位对象文件。相关选项的说明下面介绍了与 C 编译器选项关联的运行时支持库。

如前所述，在源代码转换后，编译器可能会生成对运行时支持函数的调用。对于 Solaris 内核，在特定情况下可能被调用的运行时支持函数集要小于一般情况下调用的相应函数集，因为内核不使用浮点/复合类型、数学库函数或与运行时支持库相关联的编译器选项。

下表列出了 C 编译器转换源代码后，可能被调入代码中的运行时支持函数，这些代码是为了能够在 Solaris 内核中运行而编译的。该表列出了其上的源代码转换生成了调用的平台、被调用函数的名称，以及导致生成函数调用的原结构或编译器功能。仅列出了 64 位平台，因为支持 C 编译器的所有版本的 Solaris 均在 64 位内核中运行。

针对于 32 位指令集进行编译时，可能会因指令集的特定限制而调用特定于其他计算机的支持函数。

功能	64 位平台	引用自
__align_cpy_*n*	SPARC	返回大结构；*n* 取值为1、2、4、8 或 16
_memcpy	x86	返回大结构
_memcpy	x86 和 SPARC	矢量化
_memmove	x86 和 SPARC	矢量化
_memset	x86 和 SPARC	矢量化

注意：一些内核的版本不提供 **_memmove()**、**_memcpy()** 或 **_memset()**，但提供与用户模式例程相似的内核模式例程，例如 **memmove()**、**memcpy()** 和 **memset()**。

可在《编写设备驱动程序》指南和《SPARC 遵从性定义》（版本 2.4）中找到其他信息。

◆ ◆ ◆ 第 3 章

3 并行化 C 代码

Oracle Solaris Studio C 编译器可以优化代码，以便在共享内存的多处理器/多内核/多线程系统上运行。编译的代码可以使用系统上的多个处理器以并行方式执行。可使用自动和显式并行化方法。本章阐述如何利用编译器的并行化功能。

3.1 概述

C 编译器为那些它确定可以安全进行并行化的循环生成并行代码。通常，这些循环具有彼此独立的迭代。对于此类循环，迭代以什么顺序执行或者是否并行执行并不重要。虽然不是全部，但是许多向量循环都属于此种类。

由于 C 中使用别名的方式，难以确定并行化的安全。为帮助编译器，Solaris Studio C 提供了 pragma 和附加指针限定，以提供程序员知道、但编译器无法确定的别名信息。有关更多信息，请参见第 5 章，基于类型的别名分析。

3.1.1 使用示例

以下示例说明了如何启用和控制并行化 C：

```
% cc -fast -xO4 -xautopar example.c -o example
```

这将生成一个称为 `example` 的可正常执行的可执行程序。如果要利用多处理器执行，请参见第 225 页中的 “B.2.75 -xautopar”。

3.2 OpenMP 并行化

C 编译器本身接受 OpenMP API，用于共享内存并行化。API 包括一组并行化 pragma。从 OpenMP Web 站点 http://www.openmp.org/ (`http://www.openmp.org`) 中可获得有关 OpenMP API 规范的信息。

要启用编译器的 OpenMP 支持以及对 OpenMP pragma 的识别，请使用 `-xopenmp` 选项进行编译。如果没有 `-xopenmp` 选项，编译器会将 OpenMP pragma 视为注释。请参见第 254 页中的 “B.2.123 -xopenmp[= *i*]”。

有关详细信息，请参见《Solaris Studio OpenMP API 用户指南》。

3.2.1 处理 OpenMP 运行时警告

OpenMP 运行时系统可针对非致命错误发出警告。使用以下函数注册一个回调函数以处理这些警告：

```
int sunw_mp_register_warn(void (*func) (void *) )
```

您可以通过对 `<sunw_mp_misc.h>` 发出 `#include` 预处理程序指令来访问该函数的原型。

如果不想注册函数，请将环境变量 `SUNW_MP_WARN` 设置为 `TRUE`，警告消息将发送给 `stderr`。有关 `SUNW_MP_WARN` 的更多信息，请参见第 61 页中的 “3.3.3 SUNW_MP_WARN”。

有关特定于此 OpenMP 实现的信息，请参见《Solaris Studio OpenMP API 用户指南》。

3.3 环境变量

与并行化 C 相关的环境变量有四种：

- `PARALLEL` 或 `OMP_NUM_THREADS`
- `SUNW_MP_THR_IDLE`
- `SUNW_MP_WARN`
- `STACKSIZE`

3.3.1 PARALLEL 或 OMP_NUM_THREADS

如果可以利用多处理器执行，请设置 `PARALLEL` 环境变量。`PARALLEL` 环境变量指定可供程序使用的处理器数。在下例中，`PARALLEL` 设置为 2：

```
% setenv PARALLEL 2
```

如果目标机器具有多个处理器，线程可以映射到独立的处理器。运行该程序将导致创建执行程序的并行化部分的两个线程。

可以使用 **PARALLEL** 或 **OMP_NUM_ HREADS**，它们是等效的

3.3.2 SUNW_MP_THR_IDLE

目前，程序的起始线程创建绑定线程。绑定线程一旦创建，将会参与执行程序的并行部分（并行循环、并行区域等），并在程序的串行部分运行时保持旋转等待状态。在程序终止之前，这些绑定线程不会休眠或停止。并行化程序在专用系统上运行时，使这些线程保持旋转等待状态通常可达到最佳性能。不过，保持旋转等待的线程会占用系统资源。

使用 SUNW_MP_THR_IDLE 环境变量控制每个线程在完成其一份并行作业后的状态。

```
% setenv SUNW_MP_THR_IDLE value
```

您可以用 spin 或 sleep[*n* s|*n* ms] 替换 *value*。缺省值为 sleep，它在旋转等待 *n* 个单位后将线程置于休眠状态。等待单位可以是秒（s，为缺省单位）或毫秒 (ms)，其中 1s 表示 1 秒；10ms 表示 10 毫秒。不带参数的 sleep 在线程完成并行任务后使线程立即进入休眠状态。sleep、sleep0、sleep0s 和 sleep0ms 均等效。如果新作业在达到 *n* 个单位时间之前到达，线程将停止旋转等待并开始执行新作业。

另一个选项 spin 表示线程在完成一个并行任务之后应保持旋转（或忙等待）状态，直到一个新的并行任务到来为止。

如果 SUNW_MP_THR_IDLE 包含非法值或未设置，那么 spin 将用作缺省值。

3.3.3 SUNW_MP_WARN

将此环境变量设置为 TRUE，可输出来自 OpenMP 和其他并行化运行时系统的警告消息。

```
% setenv SUNW_MP_WARN TRUE
```

如果通过使用 sunw_mp_register_warn() 注册某个函数来处理警告消息，那么即使将 SUNW_MP_WARN 设置为 TRUE，它也不会输出警告消息。如果未注册函数，但已将 SUNW_MP_WARN 设置为 TRUE，则 SUNW_MP_WARN 会将警告消息输出到 stderr。如果您未注册函数且未设置 SUNW_MP_WARN，则不会发出警告消息。有关 sunw_mp_register_warn() 的更多信息，请参见第 60 页中的 “3.2.1 处理 OpenMP 运行时警告”。

3.3.4 STACKSIZE

正在执行的程序会为主线程保留一个主内存栈，同时为每个从属线程保留不同的栈。栈是临时内存地址空间，用来存储子程序调用中的参数和自动变量。

主栈的缺省大小约为 8 兆字节。使用 limit 命令显示当前主栈大小并对其进行设置。

```
% limit
cputime unlimited
filesize unlimited
datasize 2097148 kbytes
stacksize 8192 kbytes <- current main stack size
coredumpsize 0 kbytes
descriptors 256
memorysize unlimited
% limit stacksize 65536 <- set main stack to 64Mb
```

多线程程序的每个从属线程均具有其自身的线程栈。该栈与主线程的主栈相似，但对该线程是唯一的。线程的私有数组和变量（对于线程是局部的）在线程栈中进行分配。

所有从属线程的栈大小都相同，缺省情况下，对于 32 位应用程序为 4MB，对于 64 位应用程序为 8MB。可以用环境变量 `STACKSIZE` 来设置该大小：

```
% setenv STACKSIZE 16483 <- Set thread stack size to 16 Mb
```

对于某些已并行的代码，可能需要将线程栈大小设置为比缺省值大的值。

有时，编译器会生成一条警告消息，指出需要更大的栈大小。然而，除了通过尝试并出错之外，不可能知道应设置多大的栈大小正合适，尤其是涉及私有/局部数组时。如果栈太小导致线程无法运行，程序将异常终止并出现段故障。

`STACKSIZE` 环境变量的设置对使用 Solaris *pthreads* API 的程序没有效果。

3.3.5 在并行代码中使用 `restrict`

关键字 `restrict` 可以与并行化 C 配合使用。正确使用关键字 restrict 有助于优化器了解所需数据的别名，从而确定代码序列是否可以并行化。有关详细信息，请参阅第 303 页中的 “D.1.2 C99 关键字”。

3.4 数据依赖性和干扰

C 编译器通过分析程序中的循环来确定并行执行循环的不同迭代是否安全。分析的目的是确定循环的两次迭代之间是否会相互干扰。通常，如果变量的一次迭代读取某个变量而另一次迭代正在写入该变量，会发生干扰。考虑以下程序片段：

示例 3–1 带依赖性的循环

```
for (i=1; i < 1000; i++) {
    sum = sum + a[i]; /* S1 */
}
```

在第 62 页中的 “3.4 数据依赖性和干扰”中，任意两次连续迭代，第 `i` 次和第 `i+1` 次，将写入和读取同一变量 `sum`。因此，为了并行执行这两次迭代，需要以某种形式锁定该变量。否则，允许并行执行这两次迭代不安全。

然而，使用锁定会产生可能降低程序运行速度的开销。C 编译器通常不会并行化第 62 页中的 “3.4 数据依赖性和干扰”中所示的循环。在第 62 页中的 “3.4 数据依赖性和干扰”中，循环的两次迭代之间存在数据依赖性。考虑另一个示例：

示例 3–2 不带依赖性的循环

```
for (i=1; i < 1000; i++) {
    a[i] = 2 * a[i]; /* S1 */
}
```

在此情况下，循环的每次迭代均引用不同的数组元素。因此，循环的不同迭代可以按任意顺序执行。由于不同迭代的两个数据元素不可能相互干扰，因此它们可以并行执行而无需任何锁定。

编译器为确定一个循环的两次不同迭代是否引用相同变量而执行的分析称为数据依赖性分析。如果其中一个引用写入变量，数据依赖性阻止循环并行化。编译器执行的数据依赖性分析有三种结果：

- 存在依赖性。在此情况下，并行执行循环不安全。第 62 页中的 “3.4 数据依赖性和干扰”说明了此情况。
- 不存在依赖性。循环可使用任意数目的进程安全地并行执行。第 62 页中的 “3.4 数据依赖性和干扰”说明了此情况。
- 无法确定依赖性。为安全起见，编译器假定存在阻止并行执行循环的依赖性，并且不会并行化循环。

在第 62 页中的 “3.4 数据依赖性和干扰”中，循环的两次迭代是否写入数组 a 的同一元素取决于数组 b 是否包含重复元素。除非编译器可以确定实际情况，否则它假定存在依赖性并且不会并行化循环。

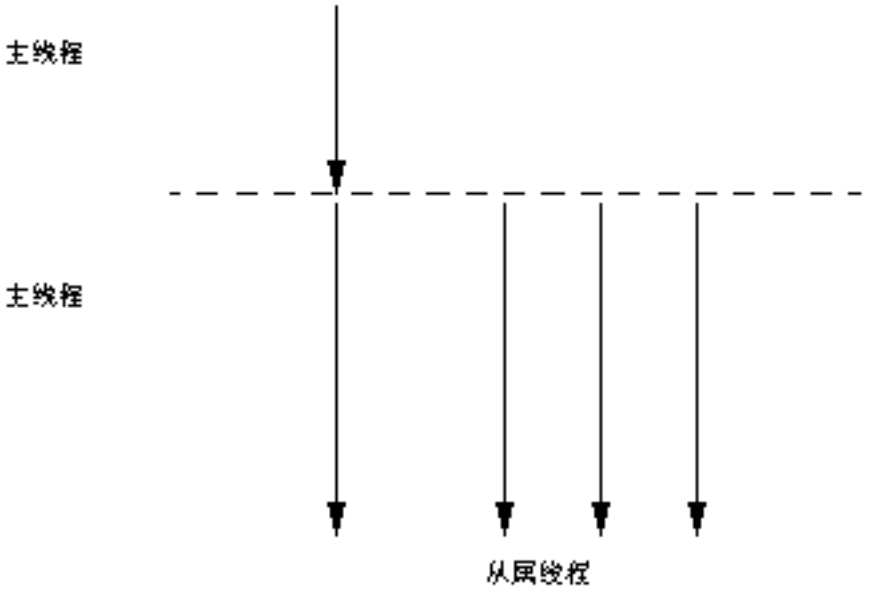
示例 3–3 可能包含也可能不包含依赖性的循环

```
for (i=1; i < 1000; i++) {
    a[b[i]] = 2 * a[i];
}
```

3.4.1 并行执行模型

循环的并行执行由 Solaris 线程完成。启动程序的初始执行的线程称为主线程。程序启动时，主线程创建多个从属线程，如下图所示。程序结束时，所有从属线程均终止。从属线程的创建只进行一次，以使开销减至最小。

图 3–1 主线程和从属线程



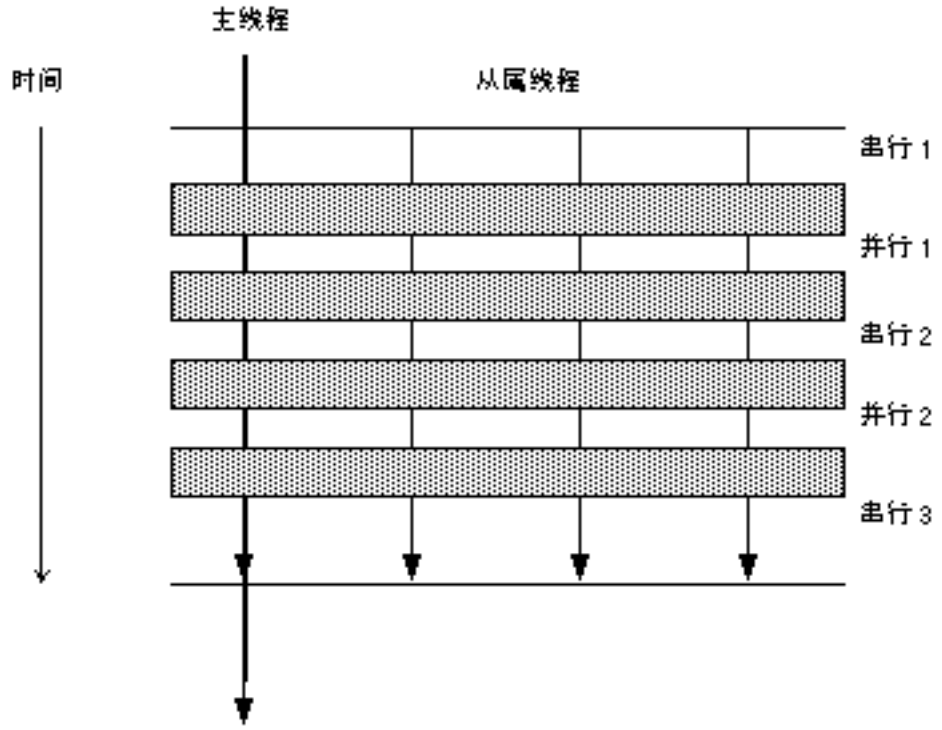
程序启动后，主线程开始执行程序，而从属线程保持空闲等待状态。当主线程遇到并行循环时，循环的不同迭代将会在启动循环执行的从属线程和主线程之间分布。在每个线程完成其块的执行之后，将与剩余线程保持同步。此同步点称为**障碍**。在所有线程完成其工作并到达障碍之前，主线程不能继续执行程序的剩余部分。从属线程在到达障碍之后进入等待状态，等待分配更多的并行工作，而主线程继续执行该程序。

在此期间，可发生多种开销：

- 同步和工作分配的开销
- 障碍同步的开销

通常存在某些特殊的并行循环，为其执行的有用工作量不足以证明开销是值得的。对于此类循环，其运行速度会明显减慢。在下图中，循环是并行化的。然而，障碍（以水平条表示）带来大量开销。如图所示，障碍之间的工作串行或并行执行。并行执行循环所需的时间比主线程和从属线程在障碍处同步所需的时间少得多。

图 3–2 循环的并行执行



3.4.2 私有标量和私有数组

对于某些数据依赖性，编译器仍能够并行化循环。考虑以下示例。

示例 3–4 带依赖性的可并行化循环

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];           /* S1 */
    b[i] = t;               /* S2 */
}
```

在本例中，假定数组 a 和 b 为非重叠数组，而由于变量 t 的存在而使任意两次迭代之间存在数据依赖性。在第一次迭代和第二次迭代时执行以下语句。

示例 3–5 第一次迭代和第二次迭代

```
t = 2*a[1];  /* 1 */
b[1] = t;    /* 2 */
t = 2*a[2];  /* 3 */
b[2] = t;    /* 4 */
```

由于第一个语句和第三个语句会修改变量 t，因此编译器无法并行执行它们。不过，t 的值始终在同一次迭代中计算并使用，因此编译器可以对每次迭代使用 t 的一个单独副本。这消除了不同迭代之间由于此类变量而产生的干扰。事实上，我们已使变量 t 成为执行迭代的每个线程的私有变量。这种情形可以说明如下：

示例 3–6 变量 t 作为每个线程的私有变量

```
for (i=1; i < 1000; i++) {
    pt[i] = 2 * a[i];       /* S1 */
    b[i] = pt[i];           /* S2 */
}
```

第 65 页中的 “3.4.2 私有标量和私有数组”中的示例与第 62 页中的 “3.4 数据依赖性和干扰”中的示例基本相同，但是每个标量变量引用 t 现在被替换为数组引用 pt。现在，每次迭代使用 pt 的不同元素，因此消除了任意两次迭代之间的所有数据依赖性。当然，本示例产生的一个问题是可能导致数组非常大。在实际运用中，编译器为参与循环执行的每个线程只分配变量的一个副本。事实上，每个此类变量是线程的私有变量。

编译器还可以私有化数组变量，以便为循环的并行执行创造机会。请看以下示例：

示例 3–7 带数组变量的可并行化循环

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            x[j] = 2 * a[i];        /* S1 */
            b[i][j] = x[j];         /* S2 */
    }
}
```

在第 65 页中的 “3.4.2 私有标量和私有数组”中，外部循环的不同迭代修改数组 x 的相同元素，因此外部循环不能并行化。不过，如果执行外部循环迭代的每个线程均具有整个数组 x 的私有副本，那么外部循环的任意两次迭代之间不存在干扰。这种情形说明如下：

示例 3–8 使用私有化数组的可并行化循环

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            px[i][j] = 2 * a[i];    /* S1 */
            b[i][j] = px[i][j];     /* S2 */
    }
}
```

如私有标量的情形一样，不必要为所有迭代展开数组，而只需要达到系统中执行的线程数。这由编译器自动完成，方式是在每个线程的私有空间中分配初始数组的一个副本。

3.4.3 返回存储

变量私有化对改进程序中的并行性十分有用。然而，如果在循环外部引用私有变量，则编译器需要确保私有变量具有正确的值。请看以下示例：

示例 3–9 使用返回存储的并行化循环

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];           /* S1 */
    b[i] = t;               /* S2 */
}
x = t;                      /* S3 */
```

在第 67 页中的 “3.4.3 返回存储”中，在语句 S3 中引用的 t 值是循环计算的最终 t 值。在变量 t 私有化并且循环完成执行之后，需要重新将 t 的正确值存储到初始变量中。这称为返回存储。此操作通过将最后一次迭代中的 t 值重新复制到变量 t 的初始位置来完成。在很多情况下，编译器自动执行此操作。但是也存在不易计算最终值的情况：

示例 3–10 不能使用返回存储的循环

```
for (i=1; i < 1000; i++) {
    if (c[i] > x[i] ) {         /* C1 */
            t = 2 * a[i];           /* S1 */
            b[i] = t;               /* S2 */
    }
}
x = t*t;                        /* S3 */
```

正确执行后，语句 S3 中的 t 值通常并不是循环最终迭代中的 t 值。事实上，它是条件 C1 为真时的最后一次迭代。通常，计算 t 的最终值十分困难。在类似情况下，编译器不会并行化循环。

3.4.4 约简变量

有时循环的迭代之间存在真正的依赖性，而导致依赖性的变量并不能简单地私有化。例如，从一次迭代到下一次迭代累计值时，会出现这种情况。

示例 3–11 可以或不可以并行化的循环

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i]; /* S1 */
}
```

在第 67 页中的 “3.4.4 约简变量”中，循环计算两个数组的向量乘积，并将结果赋给一个称为 sum 的公共变量。该循环不能以简单的方式并行化。编译器可以利用语句 S1 中计算的关联特性，并为每个线程分配一个称为 psum[i] 的私有变量。变量 psum[i] 的每个副本均初始化为 0。每个线程以自己的变量 psum[i] 副本计算自己的部分和。执行循环

之前，所有部分和均加到初始变量 sum 上。在本示例中，变量 sum 称为约简变量，因为它计算和约简。然而，将标量变量提升为约简变量存在的危险是：累计舍入值的方式会更改 sum 的最终值。只有在您专门授权这样做时，编译器才执行该变换。

3.5 加速

如果编译器不并行化所花时间量占主体的程序部分，则不会发生加速。这基本上是 Amdahl 定律的推论。例如，如果并行化一个占用程序执行时间的百分之五的循环，则总加速仅限于百分之五。然而，根据工作量和并行执行开销的大小，可能没有任何改善。

一般说来，并行化的程序执行所占的比例越大，加速的可能性就越大。

每个并行循环都会在启动和关闭期间发生少量开销。启动开销包括工作分配代价，关闭开销包括障碍同步代价。如果循环执行的工作总量不足够大，则不会发生加速。事实上，循环甚至可能减慢。因此，如果大量程序执行工作由许多短并行循环完成，则整个程序可能减慢而不是加速。

编译器执行几个试图增大循环粒度的循环变换。其中某些变换是循环交换和循环合并。因此，一般说来，如果程序中的并行量很小或者分散在小并行区域，则加速很少。

通常，按比例增大问题大小可提高程序中的并行程度。例如，考虑包含以下两部分的问题：按顺序的二次部分，以及可并行化的三次部分。对于此问题，工作量的并行部分的增长速度比顺序部分的增长速度快。因此在某些点，除非达到资源限制，否则问题可以大大加速。

尝试进行某些调节，使用指令、问题大小进行试验，重新构造程序，以便从并行 C 中获益。

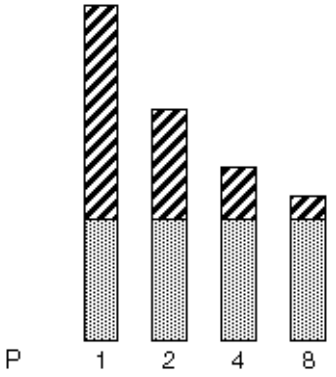
3.5.1 Amdahl 定律

固定问题大小加速通常遵守 Amdahl 定律。Amdahl 定律仅仅指出一个给定问题中的并行加速量受此问题的顺序部分限制。下面的公式介绍了某个问题的加速，其中 F 是花在顺序区域的时间部分，其余的时间部分则均匀地分布于 *P* 个处理器。如果方程式的第二项减小到零，则总加速受第一项约束，保持固定。

$$\frac{1}{S} = F + \frac{(1-F)}{P}$$

下图以图表方式说明此概念。深色阴影部分表示程序的顺序部分，并且对于 1、2、4、8 个处理器均保持不变。浅色阴影部分代表程序的并行部分，可均匀地分摊在任意多个处理器之间。

图 3–3 固定问题加速

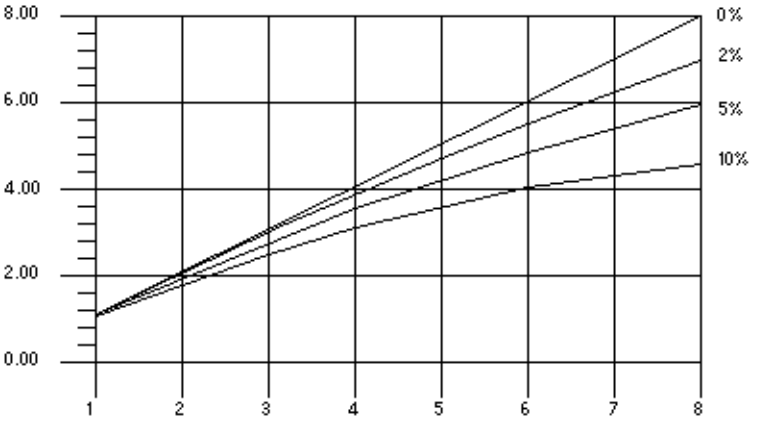


随着处理器数目的增加，每个程序并行部分所需的时间会减少，而每个程序串行部分所需的时间保持不变。

然而，实际上可能会发生由于通信以及向多个处理器分配工作而产生的开销。对于使用的任意多个处理器，这些开销可能固定，也可能不固定。

下图说明一个包含 0%、2%、5% 和 10% 顺序部分的程序的理想加速。此处假定无开销。

图 3–4 Amdahl 定律加速曲线



显示一个包含 0%、2%、5% 和 10% 顺序部分的程序的理想加速（假定无开销）的图。x 轴表示处理器的数目，y 轴表示加速。

3.5.1.1 开销

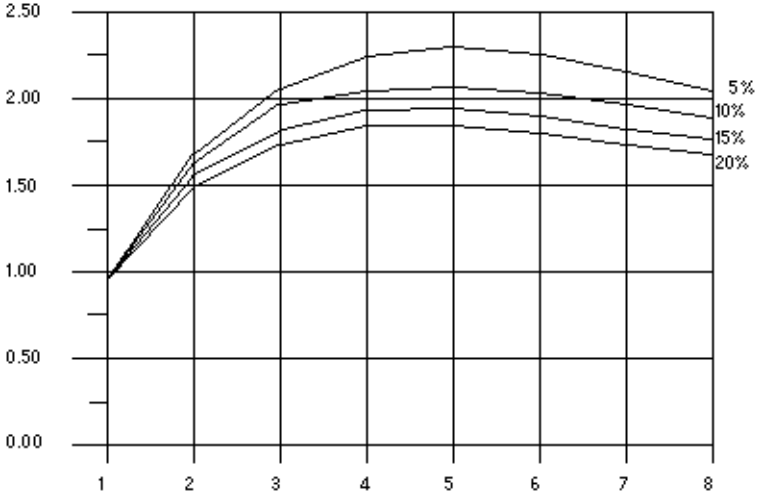
一旦将开销加入此模型中，加速曲线会发生很大的变化。为了方便说明，我们假定开销包括以下两部分：独立于处理器数的固定部分，以及随使用的处理器数呈二次增长的非固定部分：

$$\frac{1}{S} = \frac{1}{F + \left(1 - \frac{F}{P}\right) + K_1 + K_2 P^2}$$

1 除以 S 等于 1 除以下列数值之和：F 加上左括号 1 减去 F 除以 P 右括号加上 K（下标 1）加上 K（下标 2）乘以 P 的平方。

在此方程式中，K_1 和 K_2 是固定因子。在这些假定下，加速曲线如下图所示。值得注意的是，在此情况下，加速达到峰值。在某个点之后，增加更多处理器会降低性能，如下图所示。

图 3–5 带开销的加速曲线



此图显示：使用 5 个处理器时所有程序到达最高加速，然后添加到多达 8 个处理器时即会失去此加速优势。x 轴表示处理器的数目，y 轴表示加速。

3.5.1.2 Gustafson 定律

Amdahl 定律可能会在预测真实问题的并行加速时造成误导。花费在程序的顺序部分的时间有时取决于问题大小。也就是说，通过按比例缩放问题大小，您可以获得更多加速机会。以下示例说明了这一点。

示例 3–12 按比例缩放问题大小可能会获得更多加速机会

```
/*
* initialize the arrays
*/
for (i=0; i < n; i++) {
    for (j=0; j < n; j++) {
            a[i][j] = 0.0;
            b[i][j] = ...
            c[i][j] = ...
    }
}
/*
* matrix multiply
*/
for (i=0; i < n; i++) {
    for(j=0; j < n; j++) {
            for (k=0; k < n; k++) {
                a[i][j] = b[i][k]*c[k][j];
            }
    }
}
```

假定理想的开销为零，并假定只有第二个循环嵌套并行执行。不难发现，对于较小的问题大小（即 n 的值较小），程序的顺序部分和并行部分所用的时间彼此相差并不大。然而，随着 n 值的增大，花费在程序并行部分的时间比花费在顺序部分的时间增长得更快。对于此问题，随问题大小的增大而增加处理器数很有益。

3.6 负载平衡和循环调度

循环调度是将并行循环的迭代分布到多个线程的进程。为获得最大加速，在线程间均匀地分布工作而不产生太大开销十分重要。编译器针对不同情况提供多种调度类型。

3.6.1 静态调度或块调度

当循环的不同迭代执行的工作相同时，将工作均匀地分摊在系统上不同的线程之间很有益。此方法称为静态调度。

示例 3–13 静态调度的良好循环

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i];       /* S1 */
}
```

在静态调度或块调度中，每个线程将获取相同次数的迭代。如果有 4 个线程，则在以上示例中，每个线程将获取 250 次迭代。假设不存在中断，并且每个线程的进度相同，则所有线程将同时完成。

3.6.2 自我调度

一般说来，当每次迭代执行的工作不同时，静态调度不会达到良好的负载平衡。在静态调度中，每个线程获取同一迭代块。除主线程之外，每个线程在完成自己的块时，将等待参与下一个并行循环的执行。主线程将继续执行程序。在自我调度中，每个线程获取不同的小迭代块，并且在完成为其分配的块之后，尝试从同一循环中获取更多块。

3.6.3 引导自我调度

在引导自我调度 (guided self scheduling, GSS) 中，每个线程获取连续变少的块。如果每次迭代的大小不同，GSS 有助于平衡负载。

3.7 循环变换

编译器执行多个循环重构变换，帮助改进程序中循环的并行化。其中某些变换还可以提高循环的单处理器执行性能。下面描述编译器执行的变换。

3.7.1 循环分布

循环通常包含少量无法并行执行的语句以及许多可以并行执行的语句。循环分布旨在将顺序语句移到单独一个循环中，并将可并行化的语句收集到另一个循环中。这一点在以下示例中描述：

示例 3–14 循环分布举例

```
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];               /* S1 */
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;  /* S2 */
    y[i] = z[i] - x[i];                    /* S3 */
}
```

假定数组 x、y、w、a 和 z 不重叠，则语句 S1 和 S3 可以并行化，但语句 S2 不能并行化。以下是循环被分割或分布为两个不同循环后的情形：

示例 3–15 分布式循环

```
/* L1: parallel loop */
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];              /* S1 */
```

示例 3–15 分布式循环 （续）

```
    y[i] = z[i] - x[i];                    /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++) {
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

在此变换之后，循环 L1 不包含任何阻止循环并行化的语句，因此可并行执行。然而，循环 L2 仍包含来自初始循环的不可并行化的语句。

循环分布并非始终有益或安全。编译器会进行分析，以确定分布的安全性和有益性。

3.7.2 循环合并

如果循环的粒度（或循环执行的工作量）很小，则分布的效果可能并不明显。这是因为与循环工作量相比，并行循环启动的开销太大。在这种情况下，编译器使用循环合并将多个循环合并到单个并行循环中，从而增大循环的粒度。当具有相同行程计数的循环彼此相邻时，循环合并很方便且很安全。请看以下示例：

示例 3–16 工作量小的循环

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];        /* S1 */
}
/* L2: another short parallel loop */
for (i=0; i < 100; i++) {
    b[i] = a[i] * d[i];        /* S2 */
}
```

这两个短并行循环彼此相邻，可以安全地合并，如下所示：

示例 3–17 合并的两个循环

```
/* L3: a larger parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];       /* S1 */
    b[i] = a[i] * d[i];       /* S2 */
}
```

新循环产生的开销是并行循环执行产生的开销的一半。循环合并在其他方面也很有帮助。例如，如果在两个循环中引用同一数据，则合并这两个循环可以改善引用环境。

然而，循环合并并非始终安全。如果循环合并创建之前并不存在的数据依赖性，则合并可能会导致错误执行。请看以下示例：

示例 3–18 不安全合并举例

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];      /* S1 */
}
/* L2: a short loop with data dependence */
for (i=0; i < 100; i++) {
    a[i+1] = a[i] * d[i];    /* S2 */
}
```

如果合并第 73 页中的 “3.7.2 循环合并”中的循环，会产生语句 S2 至 S1 的数据依赖性。实际上，语句 S1 右边 `a[i]` 的值是在语句 S2 中计算的。如果不合并循环，此情况则不会发生。编译器执行安全性和有益性分析，以确定是否应执行循环合并。通常，编译器可以合并任意多个循环。以这种方式增大粒度有时可以大大改善循环，使其足从并行化中获益。

3.7.3 循环交换

并行化循环嵌套的最外层循环通常更有益，因为发生的开销很小。然而，由于此类循环可能携带依赖性，并行化最外层循环并非始终安全。以下对此进行说明：

示例 3–19 不能并行化的嵌套循环

```
for (i=0; i <n; i++) {
    for (j=0; j <n; j++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

在本例中，具有索引变量 `i` 的循环不能并行化，原因是循环的两次连续迭代之间存在依赖性。这两个循环可以交换，并行循环（`j` 循环）变为外部循环：

示例 3–20 交换的循环

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

交换后的循环只发生一次并行工作分配开销，而先前发生 `n` 次开销。编译器执行安全性和有益性分析，以确定是否执行循环交换。

3.8 别名和并行化

ISO C 别名通常可防止循环并行化。存在两个对同一存储单元的可能引用时，将产生别名。请看以下示例：

示例 3–21 具有对同一存储单元的两个引用的循环

```
void copy(float a[], float b[], int n) {
    int i;
    for (i=0; i < n; i++) {
            a[i] = b[i]; /* S1 */
    }
}
```

由于变量 a 和 b 为参数，因此 a 和 b 可能指向重叠的内存区域；例如，如果 copy 的调用如下：

```
copy (x[10], x[11], 20);
```

在调用的例程中，copy 循环的两次连续迭代可能读/写数组 x 的同一元素。然而，如果例程 copy 的调用如下，则循环的 20 次迭代中不可能出现重叠：

```
copy (x[10], x[40], 20);
```

通常，在不知道例程如何调用的情况下，编译器不可能正确地分析此情况。编译器提供 ISO C 的关键字扩展，使您可以传达此类别名信息。有关更多信息，请参见第 75 页中的 “3.8.2 限定指针”。

3.8.1 数组引用和指针引用

别名问题的部分原因是：C 语言可以通过指针运算来定义数组引用及定义。为使编译器有效地并行化循环（自动或显式使用 pragma），所有采用数组布局的数据均必须使用 C 数组引用语法而不是指针进行引用。如果使用指针语法，编译器将无法确定循环的不同迭代之间的关系。因此，它将保守而不会并行化循环。

3.8.2 限定指针

为使编译器有效地执行循环的并行执行任务，需要确定特定左值是否指定不同的存储区域。别名是其存储区域相同的左值。由于需要分析整个程序，因此确定对象的两个指针是否为别名是一个困难而费时的过程。例如下面的函数 `vsq()`：

示例 3–22 带两个指针的循环

```
void vsq(int n, double * a, double * b) {
    int i;
    for (i=0; i<n; i++) {
```

示例 3–22 带两个指针的循环 （续）

```
            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。

限定指针用来指定哪些指定不同对象的指针，以便编译器可以执行指针别名分析。以下是函数参数声明为限定指针的函数 vsq() 示例：

```
void vsq(int n, double * restrict a, double * restrict b)
```

指针 a 和 b 声明为限定指针，因此编译器知道 a 和 b 指向不同的存储区域。有了此别名信息，编译器就能够并行化循环。

关键字 restrict 是一个类型限定符，与 volatile 类似，但它仅限定指针类型。使用 -xc99=all（使用 -Xs 时除外）时，restrict 识别为一个关键字。在某些情况下，您可能不希望更改源代码。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*]

如果指定函数列表，则指定的函数中的指针参数将被视为限定的；否则，整个 C 文件中的所有指针参数均被视为限定的。例如，-xrestrict=vsq 限定前一个有关键字 restrict 的函数 vsq() 示例中给定的指针 a 和 b。

正确使用 restrict 至关重要。如果指针被限定为限定指针而指向不同的对象，编译器会错误地并行化循环而导致不确定的行为。例如，假定函数 vsq() 的指针 a 和 b 指向的对象重叠，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，编译器会并行化循环的执行，但这是不安全的，因为 b[i+1] 应在 b[i] 之后进行计算。

3.8.3 显式并行化和 Pragma

通常，编译器没有足够的信息来判断并行化的合法性或有益性。编译器支持 pragma，使程序员能够有效地并行化循环，否则编译器很难或根本无法处理这些循环。本节中其他地方详细介绍的 Sun 特定的旧 MP pragma 对于 OpenMP 标准来说已过时。有关该标准的指令的信息，请参见《OpenMP API 用户指南》。

3.8.3.1 串行 Pragma

注 – Sun 特定的旧 MP pragma 已过时，并且不再受支持。但是，编译器改为支持 OpenMP 3.0 标准指定的 API。有关标准的指令的迁移信息，请参见《OpenMP API 用户指南》。

有两个串行 pragma，均适用于 `for` 循环：

- `#pragma MP serial_loop`
- `#pragma MP serial_loop_nested`

`#pragma MP serial_loop` pragma 向编译器指示：不会自动并行化下一个 `for` 循环。

`#pragma MP serial_loop_nested` pragma 向编译器指示：下一个 `for` 循环以及该 `for` 循环的作用域内嵌套的任何 `for` 循环均不自动并行化。

这些 pragma 的作用域以 pragma 开始，并在下列先出现的项结束：下一个块的开头、当前块内的下一个 for 循环或当前块的结尾。

3.8.3.2 并行 Pragma

注 – Sun 特定的旧 MP pragma 已过时，并且不再受支持。但是，编译器改为支持 OpenMP 3.0 标准指定的 API。有关标准的指令的迁移信息，请参见《OpenMP API 用户指南》。

有一个并行 pragma：`#pragma MP taskloop [`*options*`]`。

`MP taskloop` pragma 可以根据需要带下列一个或多个参数。

- `maxcpus` *(number_of_processors)*
- `private` *(list_of_private_variables)*
- `shared` *(list_of_shared_variables)*
- `readonly` *(list_of_readonly_variables)*
- `storeback` *(list_of_storeback_variables)*
- `savelast`
- `reduction` *(list_of_reduction_variables)*
- `schedtype` *(scheduling_type)*

这些 pragma 的作用域以 pragma 开始，并在下列先出现的项结束：下一块的开始部分、当前块内部的下一个 for 循环、当前块的结尾。该 pragma 应用于 pragma 作用域结束前的下一个 for 循环。

每个 MP taskloop pragma 只能指定一个选项；但是，pragmas 是累积的，并应用于 pragmas 作用域中的下一个 for 循环：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop shared(a,b)
#pragma MP taskloop storeback(x)
```

这些选项在其所应用的 for 循环之前可能会多次出现。如果存在冲突选项，编译器将发出警告消息。

for 循环的嵌套

MP taskloop pragma 应用于当前块内部的下一个 for 循环。并行化 C 不存在并行化 for 循环的嵌套。

并行化的合格性

除非另有禁止，否则 MP taskloop pragma 建议编译器应并行化指定的 for 循环。

任何具有不规则控制流和未知循环迭代增量的 for 循环均不能进行并行化。例如，包含 setjmp、longjmp、exit、abort、return、goto、labels 和 break 的 for 循环均不能并行化。

特别重要的是，具有迭代间依赖性的 for 循环可以进行显式并行化。这意味着，如果为此类循环指定了一个 MP taskloop pragma，除非 for 循环被取消资格，否则编译器将完全遵循该 pragma。用户有责任确保此类显式并行化不会导致错误结果。

如果为 for 循环指定了 serial_loop（或 serial_loop_nested）pragma 和 taskloop pragma，则将使用最后指定的 pragma。

请看以下示例：

```
#pragma MP serial_loop_nested
    for (i=0; i<100; i++) {
   # pragma MP taskloop
      for (j=0; j<1000; j++) {
      ...
 }
}
```

i 循环将不并行化，但是 j 循环可能并行化。

处理器数

#pragma MP taskloop maxcpus (*number_of_processors*) 指定要用于此循环的处理器数（如果可能）。

maxcpus 的值必须为正整数。如果 maxcpus 等于 1，指定的循环将串行执行。（请注意，将 maxcpus 设置为 1 相当于指定 serial_loop pragma。）将比较 maxcpus 的值与 PARALLEL 环境变量的解释值，使用较小的值。在未指定环境变量 PARALLEL 的情况下，该 pragma 被解释为具有值 1。

如果为一个 for 循环指定了多个 maxcpus pragma，将使用最后指定的 pragma。

变量分类

循环中使用的变量可分类为 private、shared、reduction 或 readonly 变量。变量只能属于其中一种分类。通过显式 pragma 只能将变量分类为 reduction 或 readonly 变量。请参见 #pragma MP taskloop reduction 和 #pragma MP taskloop readonly。通过显式 pragma 或以下缺省作用域规则可将变量分类为 private 或 shared 变量。

private 和 shared 变量的缺省作用域规则

private 变量的值是处理 for 循环的某些迭代的每个处理器的私有值。换句话说，在 for 循环的一次迭代中赋给 private 变量的值不会传送给处理该 for 循环的其他迭代的其他处理器。而 shared 变量的当前值可处理 for 循环的迭代的所有处理器访问。处理循环迭代的一个处理器赋给 shared 变量的值可能会被处理循环的其他迭代的其他处理器识别。通过使用 #pragma MP taskloop 指令显式并行化并包含共享变量引用的循环，必须确保值的共享不会导致正确性问题（如竞争情况）。对显式并行化的循环中的共享变量的更新和访问，编译器不提供同步。

在分析显式并行化的循环时，编译器使用以下“缺省作用域规则”来确定变量是 private 变量还是 shared 变量：

- 如果变量未通过 pragma 显式分类，已声明为指针或数组，并且仅在循环内部使用数组语法进行引用，则该变量缺省分类为 shared 变量。否则，将被分类为 private 变量。
- 循环索引变量始终被视为 private 变量和返回存储变量。

强烈建议将显式并行化的 for 循环中使用的所有变量显式分类为 shared、private、reduction 或 readonly 变量，以避免使用“缺省作用域规则”。

由于编译器对共享变量的访问不执行同步，因此在对包含数组引用等的循环使用 MP taskloop pragma 之前必须格外谨慎。如果此类显式并行化的循环中存在迭代间数据依赖性，则其并行执行会导致错误结果。编译器不一定能够检测到此类潜在问题情况并发出警告消息。无论如何，编译器不会禁止具有潜在共享变量问题的循环的显式并行化。

private 变量

#pragma MP taskloop private (*list_of_private_variables*)

使用此 pragma 指定所有应视为此循环私有变量的变量。此循环中使用但未显式指定为 shared、readonly 或 reduction 变量的所有其他变量，按缺省作用域规则的定义，为 shared 变量或 private 变量。

private 变量的值是处理循环的某些迭代的每个处理器的私有值。换句话说，处理循环迭代的一个处理器赋给 private 变量的值不会传送给处理该循环的其他迭代的其他处理器。private 变量在循环的每次迭代开始时没有初始值，必须在循环的迭代内部第一次使用它之前，在该迭代内部设置为一个值。如果某个循环包含一个在设置之前使用其值的显式声明 private 变量，则执行包含该循环的程序将导致不确定的行为。

shared 变量

`#pragma MP taskloop shared (`*list_of_shared_variables*`)`

使用此 pragma 指定所有应视为此循环的 `shared` 变量的变量。循环中使用的但未显式指定为 `private`、`readonly`、`storeback` 或 `reduction` 变量的所有其他变量，按缺省作用域规则的定义，为 `shared` 变量或 `private` 变量。

`shared` 变量的当前值可被处理 `for` 循环的迭代的所有处理器访问。处理循环迭代的一个处理器赋给 `shared` 变量的值可能会被处理循环的其他迭代的其他处理器识别。

readonly 变量

`#pragma MP taskloop readonly` *(list_of_readonly_variables)*

`readonly` 变量是一类特殊的共享变量，不在循环的任何迭代中进行修改。使用此 pragma 向编译器指示，它可以对处理循环迭代的每个处理器使用该变量值的单独副本。

storeback 变量

`#pragma MP taskloop storeback` *(list_of_storeback_variables)*

使用此 pragma 指定所有应视为 `storeback` 变量的变量。

`storeback` 变量的值在循环中计算，并且计算的值在循环终止后使用。`storeback` 变量的最后一个循环迭代值在循环终止后使用。当变量为私有变量时，此类变量可以很好地通过此指令显式声明为 `storeback` 变量，通过将变量显式声明为私有变量或通过使用缺省作用域规则均可。

请注意，`storeback` 变量的返回存储操作发生在显式并行化循环的最后一次迭代中，而无论该迭代是否更新 `storeback` 变量的值。换句话说，处理循环最后一次迭代的处理器可能并不是当前包含 `storeback` 变量的最后更新值的同一处理器。请看以下示例：

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
   for (i=1; i <= n; i++) {
      if (...) {
          x=...
      }
}
   printf (“%d”, x);
```

在上例中，通过 `printf()` 调用输出的 `storeback` 变量 `x` 的值可能与通过 `i` 循环的串行版本输出的值不同，原因是，在显式并行化情况下，处理循环最后一次迭代（当 `i==n` 时）的处理器（即执行 `x` 的返回存储操作的处理器）可能不是当前包含 `x` 的最后更新值的同一处理器。编译器将尝试发出警告消息，提醒用户注意此类潜在问题。

在显式并行化的循环中，作为数组引用的变量不视为 `storeback` 变量。因此，如果需要此类返回存储操作（例如，如果作为数组引用的变量已声明为私有变量），则将作为数组引用的变量包括在 *list_of_storeback_variables* 中很重要。

savelast

`#pragma MP taskloop savelast`

使用此 pragma 指定要视为返回存储变量的所有私有变量。此 pragma 的语法如下：

`#pragma MP taskloop savelast`

通常，使用这种形式很方便，无需在将每个变量声明为返回存储变量时列出循环的每个私有变量。

reduction 变量

`#pragma MP taskloop reduction` *(list_of_reduction_variables)* 指定出现在 reduction 变量列表中的所有变量均将视为循环的 `reduction` 变量。`reduction` 变量的部分值可由处理循环迭代的每个处理器单独计算，其最终值可从其所有部分值中计算。有了 `reduction` 变量列表，便于编译器识别循环是否为约简循环，从而允许为其生成并行约简代码。请看以下示例：

```
#pragma MP taskloop reduction(x)
    for (i=0; i<n; i++) {
         x = x + a[i];
}
```

变量 *x* 是 `(sum)` 约简变量，`i` 循环是 `(sum)` 约简循环。

调度控制

Solaris Studio ISO C 编译器支持多种 pragma，这些 pragma 可与 `taskloop` pragma 配合使用，以控制给定循环的循环调度策略。此 pragma 的语法是：

`#pragma MP taskloop schedtype` *(scheduling_type)*

此 pragma 可用来指定要用来调度并行化循环的特定 *scheduling_type*。*Scheduling_type* 可为以下类型之一：

- `static`

  在 `static` 调度中，循环的所有迭代均匀地分布在参与处理的所有处理器中。请看以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(static)
    for (i=0; i<1000; i++) {
...
}
```

在以上示例中，四个处理器中的每个处理器将处理循环的 250 次迭代。

- `self` [*(chunk_size)*]

在 self 调度中，每个参与处理的处理器处理固定次数的迭代（称为“块大小”），直到循环的所有迭代均已处理完毕为止。可选的 *chunk_size* 参数指定要使用的“块大小”。*Chunk_size* 必须为正整数常量或整型变量。如果指定为变量，*chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。请看以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(self(120))
for (i=0; i<1000; i++) {
...
}
```

在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

120、120、120、120、120、120、120、120、40。

- gss [(*min_chunk_size*)]

  在 guided self 调度中，每个参与处理的处理器处理可变次数的迭代（称为“最小块大小”），直到循环的所有迭代均已处理完毕为止。可选的 *min_chunk_size* 参数指定使用的每个可变块大小至少必须为 *min_chunk_size*。*Min_chunk_size* 必须为正整数常量或整型变量。如果指定为变量，*min_chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。请看以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
for (i=0; i<1000; i++) {
...
}
```

在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

250、188、141、106、79、59、45、33、25、19、14、11、10、10、10。

- factoring [(*min_chunk_size*)]

  在 factoring 调度中，每个参与处理的处理器处理可变次数的迭代（称为“最小块大小”），直到循环的所有迭代均已处理完毕为止。可选的 *min_chunk_size* 参数指定使用的每个可变块大小至少必须为 *min_chunk_size*。*Min_chunk_size* 必须为正整数常量或整型变量。如果指定为变量，*min_chunk_size* 在循环开始时求出的值必须为正整数值。如果未指定这个可选的参数，或者其值不为正数，编译器将选择要使用的块大小。请看以下示例：

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(factoring(10))
for (i=0; i<1000; i++) {
...
}
```

在以上示例中，分配给每个参与处理的处理器的迭代次数按工作请求顺序依次为：

125, 125, 125, 125, 62、62、62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10

3.9 内存边界内部函数

编译器提供头文件 mbarrier.h，此头文件中针对 SPARC 和 x86 处理器定义了多种内存边界内部函数。这些内部函数可在开发者使用自己的同步基元编写多线程代码时使用。建议用户参阅自己的处理器文档，以确定特殊情况下需要使用这些内部函数的时间以及是否需要使用这些内部函数。

mbarrier.h 头文件支持内存排序内部函数：

- `__machine_r_barrier()`－这是**读取**边界。它确保边界前的所有装入操作能够在边界后的所有装入操作之前完成。
- `__machine_w_barrier()`－这是**写入**边界。它确保边界前的所有存储操作能够在边界后的所有存储操作之前完成。
- `__machine_rw_barrier()`－这是**读写**边界。它确保边界前的所有装入和存储操作能够在边界后的所有装入和存储操作之前完成。
- `__machine_acq_barrier()`－这是具有 *acquire* 语义的边界。它确保边界前的所有装入操作能够在边界后的所有装入和存储操作之前完成。
- `__machine_rel_barrier()`－这是具有 *release* 语义的边界。它确保边界前的所有装入和存储操作能够在边界后的所有存储操作之前完成。
- `__compiler_barrier()`－阻止编译器跨边界移动内存访问。

内部函数 `__compiler_barrier()` 出现异常的所有边界内部函数生成内存排序指令，在 x86 上这些指令是 mfence、sfence 或 lfence 指令，而在 SPARC 平台上这些指令是 membar 指令。

`__compiler_barrier()` 内部函数不会生成指令，但会通知编译器所有以前的内存操作必须在启动任何后续内存操作之前完成。以上操作将导致以下实际结果：所有的非本地变量和带有 static 存储类说明符的本地变量将会在边界前返回存储在内存中，然后在边界后重新下载，并且在此过程中编译器不会将边界前的操作与边界后的操作弄混。所有其他的边界均隐含 `__compiler_barrier()` 内部函数的行为。

例如，在以下代码中 `__compiler_barrier()` 内部函数的存在将阻止编译器合并两种循环：

```
#include "mbarrier.h"
int thread_start[16];
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
```

```
            while (thread_start[i]==1){}
        }
    }
```

◆ ◆ ◆ 第 4 章

lint 源代码检验器

本章介绍如何使用 lint 程序检查 C 代码中是否存在可能导致编译失败或在运行时出现意外结果的错误。在很多情况下，lint 会警告您存在编译器未对其作必要标志的不正确、有错误倾向或非标准的代码。

lint 程序会发出 C 编译器生成的每条错误消息和警告消息。它还发出关于潜在错误和可移植性问题的警告。由 lint 发出的许多消息有助于您提高程序的效率，其中包括减小其大小和减少必需的内存。

lint 程序使用与编译器相同的语言环境，并且 lint 的输出会定向到 stderr。有关在执行基于类型的别名歧义消除之前如何使用 lint 检查代码的更多信息和示例，请参见第 109 页中的 “4.6.3 lint 过滤器”。

4.1 基本和增强 lint 模式

lint 程序在以下两种模式下运行：

- **基本**（缺省模式）
- **增强**（包括由基本 lint 执行的一切内容以及附加的详细代码分析）

在基本模式和增强模式下，lint 通过标记文件（包括已使用的任何库）间定义和用法中的不一致，来补偿 C 中的单独编译和独立编译。特别是在大型项目环境中，同一函数可能被不同程序员用在数百个单独的代码模块中，在这种情况下，lint 有助于发现借助其他方式很难发现的错误。例如，如果调用函数时使用的参数比所需的参数少一个，该函数在栈中查找该调用从未推的值，结果在一个条件下正确，在另一个条件下不正确，具体取决于内存中该栈位置发生的情况。通过标识类似的依赖性以及对计算机体系结构的依赖性，lint 可提高运行于您的计算机或其他计算机上的代码的可靠性。

在增强模式下，lint 提供比在基本模式下更详细的报告。在基本模式下，lint 的功能包括：

- 源程序的结构和流分析

- 常量传播和常量表达式求值
- 控制流和数据流的分析
- 数据类型使用的分析

在增强模式下，lint 可以检测以下问题：

- 未使用的 #include 指令、变量和过程
- 内存释放之后内存的使用
- 未使用的赋值
- 初始化之前使用变量值
- 未分配内存的释放
- 写入常量数据段时使用指针
- 非等价宏重定义
- 未执行到的代码
- 符合联合中值类型的用法
- 实际参数的隐式强制类型转换。

4.2 使用 lint

可从命令行调用 lint 程序及其选项。要在基本模式下调用 lint，请使用以下命令：

```
% lint file1.c file2.c
```

可使用 -Nlevel 或 -Ncheck 选项调用增强 lint。例如，可以按如下所示调用增强 lint：

```
% lint -Nlevel=3 file1.c file2.c
```

lint 会检查**两遍**代码。第一遍，lint 检查 C 源文件中的错误条件；第二遍，检查 C 源文件中的不一致性。除非使用 -c 调用 lint，否则对于用户该过程不可见：

```
% lint -c file1.c file2.c
```

该命令指示 lint 仅执行第一遍检查，并在名为 *file1*.ln 和 *file2*.ln 的中间文件中收集 *file1*.c 和 *file2*.c 之间定义和用法的不一致信息（这些信息和第二遍检查相关）：

```
% ls
file1.c
file1.ln
file2.c
file2.ln
```

可见，lint 的选项 -c 类似于 cc 的选项 -c，可禁止编译的链接编辑阶段。一般说来，lint 的命令行语法严格遵循 cc 的语法。

针对 .ln 文件执行 lint 时：

```
% lint file1.ln file2.ln
```

将执行第二遍检查。lint 按文件在命令行中出现的顺序处理任意多个 .c 或 .ln 文件。因此，

```
% lint file1.ln file2.ln file3.c
```

指示 lint 检查 *file3* .c 中的内部错误以及所有三个文件中的不一致性。

lint 按与 cc 相同的顺序在目录中搜索包含的头文件。可以像使用 cc 的选项 -I 那样使用 lint 的选项 -I。请参见第 54 页中的 “2.16 如何指定 include 文件”。

可以在同一命令行中指定 lint 的多个选项。除非其中一个选项带有参数或者选项有多个字母，否则选项可以串联：

```
% lint -cp -Idir1 -Idir2 file1.c file2.c
```

该命令指示 lint 执行以下操作：

- 仅执行第一次传递
- 执行附加的可移植性检查
- 在指定的目录中搜索包含的头文件

lint 有许多选项，可用来指示 lint 执行某些任务并报告某些情形。

4.3 lint 选项

lint 程序是一个静态分析器。它不能求出它检测到的依赖性的运行时结果。例如，某些程序可能包含数百个执行不到的 break 语句，这些语句并不重要，但是 lint 仍然会标记它们。下面是一个示例，其中包含 lint 命令行选项和指令（嵌入源代码文本的特殊注释）：

- 可以使用 -b 选项调用 lint 以禁止关于执行不到的 break 语句的所有错误消息。
- 可以在任一执行不到的语句前面加上注释 /*NOTREACHED*/ 以禁止诊断该语句。

下面按字母顺序列出了 lint 选项，其中有几个 lint 选项与禁止 lint 诊断消息有关。在这些按字母顺序说明的选项之后有一个表 4–8，其中也列出了这些选项，并且列出了这些选项禁止的特定消息。用于调用增强 lint 的选项以 -N 开头。

lint 能够识别许多 cc 命令行选项，其中包括
-A、-D、-E、-g、-H、-O、-P、-U、-Xa、-Xc、-Xs、-Xt 和 -Y，尽管 -g 和 -O 会被忽略。未识别的选项被警告并忽略。

4.3.1 -#

打开详细模式，显示调用的每个组件。

4.3.2 -###

显示调用但实际上并不执行的每个组件。

4.3.3 -a

抑制某些消息。请参阅表 4–8。

4.3.4 -b

抑制某些消息。请参阅表 4–8。

4.3.5 -C *filename*

使用指定的文件名创建一个 .ln 文件。这些 .ln 文件仅是 lint 的第一遍检查产生的文件。*filename* 可以是完整路径名。

4.3.6 -c

为命令行上的每个名为 .c 的文件创建一个 .ln 文件，该文件包含与 lint 的第二遍检查相关的信息。不执行第二遍检查。

4.3.7 -dirout=*dir*

指定目录 *dir*，其中存放 lint 输出文件（.ln 文件）。此选项会影响 -c 选项。

4.3.8 -err=warn

-err=warn 是用于 -errwarn=%all 的宏。请参见第 92 页中的 “4.3.15 -errwarn=*t*”。

4.3.9 -errchk=*l*(, *l*)

执行由 *l* 指定的附加检查。缺省值为 -errchk=%none。指定 -errchk 与指定 -errchk=%all 等效。*l* 是一个以逗号分隔的检查列表，由下表中的一个或多个值组成。例如，-errchk=longptr64,structarg。

表 4–1 -errchk 标志

值	含义
%all	执行 -errchk 的所有检查。
%none	不执行 -errchk 的任何检查。这是缺省值。
[no%]locfmtchk	在 lint 的第一遍检查期间检查类似 printf 的格式字符串。无论是否使用 -errchk=locfmtchk，lint 都会在第二遍检查期间检查类似 printf 的格式字符串。
[no%]longptr64	检查是否可移植到其长整型和指针大小为 64 位、平整型大小为 32 位的环境。即使使用了显式强制类型转换，也检查指针表达式和长整型表达式是否赋值为平整型。
[no%]structarg	检查通过值传递的结构参数，并在形式参数类型未知时报告情况。
[no%]parentheses	检查代码中优先级的透明度。使用此选项可增强代码的可维护性。如果 -errchk=parentheses 返回一个警告，请考虑使用额外的括号明确地表示代码中操作的优先级。
[no%]signext	检查如下情况：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型值的符号扩展。仅当同时指定 -errchk=longptr64 时，该选项才会产生错误消息。
[no%]sizematch	检查较长整数到较短整数的赋值并发出警告。对于具有不同符号的相同长度的整数之间的赋值（无符号整型数获取带符号整型数），也会发出这些警告。

4.3.10 -errfmt=*f*

指定 lint 输出的格式。*f* 可以是下列值之一： macro、simple、src 或 tab。

表 4–2 -errfmt 标志

值	含义
macro	显示错误的源代码、行号和位置，并展开宏。
simple	对于一行（简单）诊断消息，在括号中显示错误的行号和位置号。类似于 -s 选项，但是包含错误的位置信息。
src	显示错误的源代码、行号和位置（不展开宏）。
tab	以制表格式显示。这是缺省值。

缺省值为 -errfmt=tab。指定 -errfmt 与指定 -errfmt=tab 等效。

如果指定了多种格式，则使用最后指定的格式，并且 lint 发出有关未使用格式的警告。

4.3.11 -errhdr=*h*

在同时指定了 -Ncheck 的情况下，允许 lint 报告某些头文件消息。*h* 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*dir*、no%*dir*、%all、%none、%user。

表 4–3 -errhdr 标志

值	含义
dir	报告目录 *dir* 中包含的头文件的 -Ncheck 消息。
no%*dir*	不报告目录 *dir* 中包含的头文件的 -Ncheck 消息。
%all	检查使用的所有头文件。
%none	不检查头文件。这是缺省值。
%user	检查所有已使用的用户头文件，即，除 /usr/include 及其子目录中的头文件以及由编译器提供的头文件之外的所有头文件。

缺省值为 -errhdr=%none。指定 -errhdr 与指定 -errhdr=%user 等效。

示例：

```
% lint -errhdr=inc1 -errhdr=../inc2
```

检查目录 inc1 和 ../inc2 中已使用的头文件。

```
% lint -errhdr=%all,no%../inc
```

检查除目录 ../inc 中的头文件之外的所有已使用的头文件。

4.3.12 -erroff=*tag*(,*tag*)

抑制或启用 lint 错误 消息。

t 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*tag*、no%*tag*、%all、%none。

表 4–4 -erroff 标志

值	含义
tag	禁止由该 *tag* 指定的消息。可通过 -errtags=yes 选项来显示消息的标记。
no%*tag*	启用由该 *tag* 指定的消息。
%all	禁止所有消息。
%none	启用所有消息。这是缺省值。

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

示例：

```
% lint -erroff=%all,no%E_ENUM_NEVER_DEF,no%E_STATIC_UNUSED
```

仅输出消息 "enum never defined" 和 "static unused"，并禁止其他消息。

```
% lint -erroff=E_ENUM_NEVER_DEF,E_STATIC_UNUSED
```

仅禁止消息 "enum never defined" 和 "static unused"。

4.3.13 -errsecurity=*v*

可使用 -errsecurity 选项检查代码中的安全漏洞。

v 必须是下列值之一：

表 4–5 -errsecurity 标志

值	含义
core	此级别检查的源代码构造几乎始终是不安全或难以验证的。此级别的检查包括： ■ 将可变格式字符串与 printf() 和 scanf() 系列函数一起使用 ■ 在 scanf() 函数中使用无限制字符串 (%s) 格式 ■ 使用无安全用法的函数：gets()、cftime()、ascftime() 和 creat() ■ 错误地使用 open() 和 O_CREAT 将在此级别生成警告的源代码视为错误。应更改有问题的源代码。在所有情况下，都应采用更安全简单的代码。
standard	此级别检查包括 core 级别的所有检查，以及可能安全、但有更好的可用替代代码的构造的检查。检查新近编写的代码时建议采用此级别检查。此级别的其他检查包括： ■ 使用除 strlcpy() 之外的字符串复制函数 ■ 使用弱随机数函数 ■ 使用不安全的函数生成临时文件 ■ 使用 fopen() 创建文件 ■ 使用调用 shell 的函数 使用新的或经过大幅修改的代码替换在此级别生成警告的源代码。应权衡消除传统代码中的这些警告对应用程序造成的不稳定风险。

表 4–5 -errsecurity 标志 （续）

值	含义
extended	此级别检查包含几乎所有检查，包括 Core 级别和 Standard 级别的所有检查。此外，还会生成许多有关在某些情况下可能不安全的构造的警告。此级别的检查可用作检查代码的辅助措施，但无需将这些检查用作判断源代码是否可接受的标准检查。此级别的其他检查包括： ■ 在循环中调用 getc() 或 fgetc() ■ 使用易于产生路径名争用情况的函数 ■ 使用 exec() 函数系列 ■ stat() 和其他函数之间的争用情况 检查在此级别生成警告的源代码，确定是否存在潜在安全问题。
%none	关闭 -errsecurity 检查

如果未指定 -errsecurity 的设置，lint 会将它设置为 -errsecurity=%none。如果在指定 -errsecurity 时未指定参数，lint 会将它设置为 -errsecurity=standard。

4.3.14 -errtags=*a*

显示每个错误消息的消息标志。*a* 可以是 yes 或 no。缺省值为 -errtags=no。指定 -errtags 与指定 -errtags=yes 等效。

可与所有 -errfmt 选项一起使用。

4.3.15 -errwarn=*t*

如果发出了指定的警告消息，lint 会以失败状态退出。*t* 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*tag*、no%*tag*、%all、%none。这些值的顺序很重要；例如，如果发出了除 tag 以外的任何警告，*%all,no%*tag 会导致 *lint* 以致命状态退出。下表列出了 -errwarn 值：

表 4–6 -errwarn 标志

tag	如果此 *tag* 指定的消息作为警告消息发出，将导致 lint 以致命状态退出。如果未出现 *tag*，则没有影响。
no%*tag*	如果 *tag* 指定的消息仅作为警告消息发出，将防止 lint 以致命状态退出。如果未发出 *tag*，则没有影响。使用此选项，可防止发出先前使用该选项及 *tag* 或 %all 指定的警告消息时导致 lint 以致命状态退出。
%all	如果发出了任何警告消息，将导致 lint 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	如果发出了任何警告消息，将防止任何警告消息导致 lint 以致命状态退出。

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

4.3.16 -F

引用命令行上命名的 .c 文件时，会输出命令行上提供的路径名而不仅仅是它们的基名。

4.3.17 -fd

报告有关旧样式函数的定义或声明情况。

4.3.18 -flagsrc=*file*

使用文件 file 中包含的选项执行 *lint*。可在 *file* 中指定多个选项（每行一个）。

4.3.19 -h

抑制某些消息。请参阅表 4–8。

4.3.20 -I*dir*

在目录 *dir* 中搜索包含的头文件。

4.3.21 -k

改变 /* LINTED [*message*] */ 指令或 NOTE(LINTED(*message*)) 注释的行为。通常，lint 会针对这些指令后面的代码禁止警告消息。如果不设置该选项，lint 不禁止消息，而是输出包含指令或注释内部的注释的附加消息。

4.3.22 -L*dir*

与 -l 一起使用时，在目录 *dir* 中搜索 lint 库。

4.3.23 -l*x*

访问 lint 库 llib-l*x*.ln。

4.3.24 -m

抑制某些消息。请参阅表 4–8。

4.3.25 -m32|-m64

指定所分析程序的内存模型。还搜索与所选的内存模型（32 位/64 位）相对应的 lint 库。

使用 -m32 可验证 32 位 C 程序；使用 -m64 可验证 64 位程序。

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。

请注意，在以前的编译器发行版中，内存模型 ILP32 或 LP64 是通过选择 -Xarch 选项来隐式指定的。从 Solaris Studio 12 编译器开始，就不再是这样了。在大多数平台上，只需在命令行上添加 -m64，就可以通过 lint 调用 64 位程序。

有关预定义的宏的更多信息，请参见以下 lint 选项列表后面的几节内容。

4.3.26 -Ncheck=*c*

检查头文件中的相应声明；检查宏。*c* 是一个以逗号分隔的检查列表，它包含以下项中的一项或多项： macro、extern、%all、%none、no%macro、no%extern。

表 4–7 -Ncheck 标志

值	含义
macro	检查文件之间的宏定义的一致性。
extern	检查源文件与关联的头文件（例如，file1.c 与 file1.h）之间声明的一一对应关系。确保头文件中既无多余的 extern 声明，也不缺少 extern 声明。
%all	执行 -Ncheck 的所有检查。
%none	不执行 -Ncheck 的任何检查。这是缺省值。
no%macro	不执行 -Ncheck 的任何 macro 检查。
no%extern	不执行 -Ncheck 的任何 extern 检查。

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

多个值可以用逗号分隔，例如 -Ncheck=extern,macro。

示例：

```
% lint -Ncheck=%all,no%macro
```

执行除宏检查之外的所有检查。

4.3.27 -Nlevel=*n*

通过指定用于报告问题的增强 lint 分析级别，打开增强 lint 模式。此选项允许您控制检测到的错误量。级别越高，检验时间越长。*n* 是一个数字： 1、2、3 或 4。没有缺省值。如果未指定 -Nlevel，lint 会使用其基本分析模式。如果指定不带任何参数的 -Nlevel，lint 会设置 -Nlevel=4。

有关基本和增强 lint 模式的说明，请参见第 86 页中的 “4.2 使用 lint”。

4.3.27.1 -Nlevel=1

分析单个过程。报告发生在某些程序执行路径上的无条件错误。不执行全局数据和控制流分析。

4.3.27.2 -Nlevel=2

分析整个程序，包括全局数据和控制流。报告发生在某些程序执行路径上的无条件错误。

4.3.27.3 -Nlevel=3

分析整个程序，包括常量传播、常量用作实际参数时的情况以及在 -Nlevel=2 下执行的分析。

在此分析级别检验 C 程序所用的时间比前一级别长 2 到 4 倍。需要额外的时间是因为 lint 通过为程序变量创建可能值的集合对程序进行部分解释。这些变量集合是以常量以及包含程序中的常量操作数的条件语句为基础创建的。这些集合形成创建其他集合的基础（一种常量传播形式）。作为分析结果接收的集合根据以下算法来评估正确性：

如果对象的所有可能值之中存在一个正确值，则该正确值用作进一步传播的基础；否则诊断出一个错误。

4.3.27.4 -Nlevel=4

分析整个程序，并报告使用某些程序执行路径时会发生的条件错误，以及在 -Nlevel=3 下执行的分析。

在此分析级别上，存在更多诊断消息。分析算法通常对应于 -Nlevel=3 的分析算法，所不同的是任何无效值现在会生成一条错误消息。在此级别上进行分析所需的时间增大两个数量级（大约慢 20 到 100 倍）。在这种情况下，所需的额外时间与以递归、条件语句等为特征的程序复杂性成正比。因此，对超过 100,000 行的程序使用此级别的分析可能很困难。

4.3.28 -n

抑制检查与缺省 lint 标准 C 库的兼容性。

4.3.29 -o*x*

导致 lint 创建一个名为 llib-lx.*ln* 的 lint 库。该库是从 lint 在其第二遍检查中使用的所有 .ln 文件创建的。-c 选项会使 -o 选项的任何使用无效。要生成 llib-l*x*.ln 而不产生任何无关消息，可以使用 -x 选项。如果 lint 库的源文件仅为外部接口，则 -v 选项会很有用。如果使用 -lx 调用 lint，则生成的 *lint* 库可以在以后使用。

缺省情况下，使用 lint 的基本格式创建库。如果使用 lint 的增强模式，则创建的库将为增强格式，并且只能在增强模式下使用。

4.3.30 -p

启用某些与可移植性问题相关的消息。

4.3.31 -R*file*

将 .ln 文件写入 *file*，以供 cxref(1) 使用。如果此选项打开，此选项将禁用增强模式。

4.3.32 -s

生成具有 "warning:" 或 "error:" 前缀的简单诊断。缺省情况下，lint 会缓冲某些消息以生成复合输出。

4.3.33 -u

抑制某些消息。请参阅表 4–8。此选项适用于对较大程序的文件子集运行 lint。

4.3.34 -V

将产品名称及发行版写入标准错误中。

4.3.35 -v

抑制某些消息。请参阅表 4–8。

4.3.36 -W*file*

将 .ln 文件写入 *file*，以供 cxref(1) 使用。如果此选项打开，此选项将禁用增强模式。

4.3.37 -XCC=*a*

接受 C++ 样式的注释。特别是，可使用 // 指示注释的开始。*a* 可以是 yes 或 no。缺省值为 -XCC=no。指定 -XCC 与指定 -XCC=yes 等效。

注 – 仅当使用 -xc99=none 时，才需使用此选项。使用 -xc99=all（缺省值）时，lint 接受由 // 指示的注释。

4.3.38 -Xalias_level[=*l*]

其中，*l* 是下列值之一： any、basic、weak、layout、strict、std 或 strong。有关不同的歧义消除级别的详细说明，请参见表 B–13。

如果未指定 -Xalias_level，该标志的缺省值为 -Xalias_level=any。这意味着不存在基于类型的别名分析。如果指定了 -Xalias_level，但未提供级别，则缺省值为 -Xalias_level=layout。

请确保运行 lint 时所在的歧义消除级别不如运行编译器时所在的级别严格。如果运行 lint 时所在的歧义消除级别比编译时所在的级别更严格，结果将很难解释并且可能令人误解。

有关歧义消除的详细说明以及有助于歧义消除的 pragma 的列表，请参见第 109 页中的 “4.6.3 lint 过滤器”。

4.3.39 -Xarch=amd64

（Solaris 操作系统）已过时。不要使用。请参见第 94 页中的 “4.3.25 -m32|-m64”。

4.3.40 -Xarch=v9

（Solaris 操作系统）已过时。不要使用。请参见第 94 页中的 “4.3.25 -m32|-m64”。

4.3.41 -Xc99[= *o*]

-Xc99 标志控制编译器对 C99 标准（ISO/IEC 9899:1999，编程语言－C）中已实现功能的识别。

o 可以是下列值之一： all 和 none。

-Xc99=none 会关闭对 C99 功能的识别。-Xc99=all 会打开对支持的 C99 功能的识别。

指定不带任何参数的 -Xc99 与指定 -Xc99=all 等效。

注 – 虽然编译器的支持级别缺省为表 C–6 中列出的 C99 功能，但 Solaris 软件在 /usr/include 中提供的标准头文件仍不符合 1999 ISO/IEC C 标准。如果遇到错误消息，请尝试使用 -Xc99=none 获取这些头文件的 1990 ISO/IEC C 标准行为。

4.3.42 -Xkeeptmp=*a*

保留执行 lint 期间创建的临时文件，而非自动删除它们。*a* 可以是 yes 或 no。缺省值为 -Xkeeptmp=no。指定 -Xkeeptmp 与指定 -Xkeeptmp=yes 等效。

4.3.43 -Xtemp=*dir*

将临时文件的目录设置为 *dir*。如果没有此选项，临时文件将放在 /tmp 中。

4.3.44 -Xtime=*a*

报告执行每遍 lint 检查所需的时间。*a* 可以是 yes 或 no。缺省值为 -Xtime=no。指定 -Xtime 与指定 -Xtime=yes 等效。

4.3.45 -Xtransition=*a*

针对 K&R C 和 Sun ISO C 之间的差异发出警告。*a* 可以是 yes 或 no。缺省值为 -Xtransition=no。指定 -Xtransition 与指定 -Xtransition=yes 等效。

4.3.46 -Xustr={ascii_utf16_ushort| no}

此选项允许编译器将 U"*ASCII_string*" 形式的串文字识别成无符号短整型数组。缺省值为 -Xustr=no，它禁止编译器识别 U"*ASCII_string*" 串文字。-Xustr=ascii_utf16_ushort 允许编译器识别 U"*ASCII_string*" 串文字。

4.3.47 -x

抑制某些消息。请参阅表 4–8。

4.3.48 -y

将命令行上命名的各个 .c 文件视为以指令 /* LINTLIBRARY */ 或注释 NOTE(LINTLIBRARY) 开头。lint 库通常是使用 /* LINTLIBRARY */ 指令或 NOTE(LINTLIBRARY) 注释创建的。

4.4 lint 消息

lint 的大多数消息都很简单，对于它们诊断的每个问题都输出一行语句。对于包含的文件中检测到的错误，编译器会报告多次，而 lint 仅报告一次（无论其他源文件中包含该文件多少次）。文件之间存在不一致时会发出复合消息，并且在少数情况下，文件内部存在问题时也会发出复合消息。单个消息描述所检查的文件中的各个问题。如果使用 lint 过滤器（请参见第 108 页中的 “4.6.2 lint 库”）要求对出现的每个问题都输出一条消息，可通过调用带有 -s 选项的 lint 将复合诊断转换为简单类型。

lint 的消息会写入 stderr。

4.4.1 用于禁止消息的选项

可以使用几个 lint 选项禁止 lint 诊断消息。可以使用后跟一个或多个 *tag* 的 -erroff 选项禁止消息。可以使用 -errtags=yes 选项显示这些助记符标记。

下表列出了用于禁止 lint 消息的选项。

表 4–8 用于禁止消息的 lint 选项

选项	禁止的消息
-a	assignment causes implicit narrowing conversion conversion to larger integral type may sign-extend incorrectly
-b	statement not reached (unreachable break and empty statements)
-h	assignment operator "=" found where equality operator "==" was expected constant operand to op: "!" fallthrough on case statements pointer cast may result in improper alignment precedence confusion possible; parenthesize statement has no consequent: if statement has no consequent: else
-m	declared global, could be static

表 4–8 用于禁止消息的 lint 选项 （续）

选项	禁止的消息
-erroff=*tag*	由 *tag* 指定的一条或多条 lint 消息
-u	name defined but never used name used but not defined
-v	arguments unused in function
-x	name declared but never used or defined

4.4.2 lint 消息格式

通过某些选项，lint 程序可以显示精确的源文件行以及指向发生错误的行所在位置的指针。启用此功能的选项是 -errfmt=*f*。如果设置了此选项，lint 会提供以下信息：

- 源代码行和位置
- 宏展开
- 有错误倾向的栈

例如，以下程序 Test1.c 包含一个错误。

```
1 #include <string.h>
2 static void cpv(char *s, char* v, unsigned n)
3 { int i;
4   for (i=0; i<=n; i++){
5        *v++ = *s++;}
6 }
7 void main(int argc, char* argv[])
8 {
9     if (argc != 0){
10        cpv(argv[0], argc, strlen(argv[0]));}
11}
```

如果针对 Test1.c 使用带有以下选项的 lint：

```
% lint -errfmt=src -Nlevel=2 Test1.c
```

产生以下输出：

```
      |static void cpv(char *s, char* v, unsigned n)
      |            ^  line 2, Test1.c
      |
      |         cpv(argv[0], argc, strlen(argv[0]));
      |                     ^  line 10, Test1.c
warning: improper pointer/integer combination: arg #2
      |
      |static void cpv(char *s, char* v, unsigned n)
      |                               ^  line 2, Test1.c
      |
      |cpv(argv[0], argc, strlen(argv[0]));
```

```
      |                      ^ line 10, Test1.c
      |
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
warning:use of a pointer produced in a questionable way
    v defined at Test1.c(2)    ::Test1.c(5)
      call stack:
          main()                ,    Test1.c(10)
          cpv()                 ,    Test1.c(5)
```

第一个警告指示互相矛盾的两个源代码行。第二个警告显示调用栈以及导致错误的控制流。

另一个程序 `Test2.c` 包含一个不同的错误：

```
1 #define AA(b) AR[b+l]
2 #define B(c,d) c+AA(d)
3
4 int x=0;
5
6 int AR[10]={1,2,3,4,5,6,77,88,99,0};
7
8 main()
9  {
10  int y=-5, z=5;
11  return B(y,z);
12 }
```

如果针对 `Test2.c` 使用带有以下选项的 `lint`：

```
% lint -errfmt=macro Test2.c
```

产生以下输出，并显示宏替换的步骤：

```
      | return B(y,z);
      |        ^  line 11, Test2.c
      |
      |#define B(c,d) c+AA(d)
      |                 ^  line 2, Test2.c
      |
      |#define AA(b) AR[b+l]
      |                   ^  line 1, Test2.c
error: undefined symbol: l
|
      |    return B(y,z);
      |           ^  line 11, Test2.c
      |
      |#define B(c,d) c+AA(d)
      |                 ^  line 2, Test2.c
      |
      |#define AA(b) AR[b+l]
      |                   ^  line 1, Test2.c
variable may be used before set: l
lint: errors in Test2.c; no output created
lint: pass2 not run - errors in Test2.c
```

4.5 lint 指令

4.5.1 预定义值

运行 lint 对 lint 标记进行预定义。有关预定义标记的列表，另请参见 cc(1) 手册页。

4.5.2 指令

现有注释支持 /*...*/ 形式的 lint 指令，但将来的注释不支持这些指令。建议对所有注释使用源代码注释形式 (NOTE(...)) 的指令。

通过包括 note.h 文件可指定源代码注释形式的 lint 指令，例如：

```
#include <note.h>
```

Lint 与其他多种工具共享源代码注释方案。安装 Solaris Studio C 编译器时，还会自动安装 /usr/lib/note/SUNW_SPRO-lint 文件，该文件包含 LockLint 可识别的所有注释的名称。然而，Solaris Studio C 源代码检验器 lint 也会检查 /usr/lib/note 和 Solaris Studio 缺省位置 <*install-directory*>/prod/lib/note 下的所有文件以查找所有有效的注释。

可通过设置环境变量 NOTEPATH 来指定除 /usr/lib/note 以外的位置，如下所示：

```
setenv NOTEPATH $NOTEPATH:other_location
```

下表列出了 lint 指令及其操作。

表 4–9 lint 指令

指令	操作
NOTE(ALIGNMENT(*fname,n*)) 其中，*n*=1、2、4、8、16、32、64、128	指示 lint 将其后面的函数结果设置为以 *n* 字节对齐。例如，将 malloc() 定义为实际返回字（甚至双字）对齐的指针时返回一个 char* 或 void*。 禁止以下消息： ■ improper alignment
NOTE(ARGSUSED(n)) /*ARGSUSEDn*/	该指令的作用类似于下一个函数的 -v 选项。 禁止针对其后的函数定义中除前 *n* 个参数之外的各个参数发出以下消息。缺省值为 0。对于 NOTE 格式，必须指定 *n*。 ■ argument unused in function
NOTE(ARGUNUSED (*par_name[,par_name...]*))	指示 lint 不检查提到的参数的用法（此选项仅适用于下一个函数）。 禁止对 NOTE 或指令中列出的各个参数发出以下消息。 ■ argument unused in function

表 4–9 lint 指令 （续）

指令	操作
NOTE(CONSTCOND) /*CONSTCOND*/	禁止发出关于条件表达式的常量操作数的警告消息。禁止对该指令之后的构造发出以下消息。亦作 NOTE(CONSTANTCONDITION) 或 /* CONSTANTCONDITION */。 constant in conditional context constant operands to op: "!" logical expression always false: op "&&" logical expression always true: op "||"
NOTE(EMPTY) /*EMPTY*/	禁止由于 if 语句而引起 null 语句时发出警告信息。该指令应放在测试表达式之后和分号之前。当空 if 语句后跟有效 else 语句时，提供该指令以支持空 if 语句。它禁止针对空 else 结论发出消息。 禁止在 if 的控制表达式与分号之间插入时发出以下消息。 ■ statement has no consequent: else 在 else 与分号之间插入时； ■ statement has no consequent: if
NOTE(FALLTHRU) /*FALLTHRU*/	禁止在 case 或 default 带标签语句失败时发出警告消息。该指令应放在标签之前并紧挨着标签。 禁止对该指令之后的 case 语句发出以下消息。亦作 NOTE(FALLTHROUGH) 或 /* FALLTHROUGH */。 ■ fallthrough on case statement
NOTE(LINTED (msg)) /*LINTED [msg]*/	禁止所有文件内警告（处理未使用的变量或函数时发出的警告除外）。该指令应放在紧挨着发生 lint 警告的位置之前的行上。-k 选项改变 lint 处理该指令的方式。 lint 会输出注释中包含的附加消息（如果有），而非禁止消息。该指令可与 -s 选项一起用于 post-lint 过滤。 未调用 -k 时，禁止对该指令之后的代码行发出关于文件内问题的各个警告，但以下警告除外： ■ argument unused in function ■ declarations unused in block ■ set but not used in function ■ static unused ■ variable not used in function 忽略 *msg*。

表 4–9 lint 指令 （续）

指令	操作
NOTE(LINTLIBRARY) /*LINTLIBRARY*/	调用 -o 时，仅将该指令之后的 .c 文件中的定义写入库 .ln 文件。该指令禁止发出关于此文件中存在未使用的函数和函数参数的警告消息。
NOTE(NOTREACHED) /*NOTREACHED*/	在适当的点，停止关于无法执行到的代码的注释。此注释通常放在诸如 exit(2) 的函数调用之后。 禁止在函数末尾该指令之后存在右花括号时发出以下消息。 ■ statement not reached 对于该指令之后的无法执行到的语句； ■ fallthrough on case statement 对于该指令之后无法从前面的 case 执行到的 case； ■ function falls off bottom without returning value
NOTE(PRINTFLIKE(*n*)) NOTE(PRINTFLIKE(*fun_name,n*)) /*PRINTFLIKE*n**/	将其后函数定义中的第 *n* 个参数视为 [fs]printf() 格式字符串，并在其余参数与转换定义之间不匹配时发出以下消息。缺省情况下，如果标准 C 库提供的 [fs]printf() 函数调用中存在错误，lint 会发出这些警告。 对于 NOTE 格式，必须指定 *n*。 ■ malformed format strings 对于该参数中的无效转换定义，以及函数参数类型与格式不一致； ■ too few arguments for format ■ too many arguments for format
NOTE(PROTOLIB(*n*)) /*PROTOLIB*n**/	当 *n* 为 1，并且使用了 NOTE(LINTLIBRARY) 或 /* LINTLIBRARY */ 时，仅将该指令之后的 .c 文件中的函数原型声明写入库 .ln 文件。缺省值为 0，它取消该进程。 对于 NOTE 格式，必须指定 *n*。
NOTE(SCANFLIKE(*n*)) NOTE(SCANLIKE*(fun_name,n*)) /*SCANFLIKE*n**/	与 NOTE(PRINTFLIKE(*n*)) 或 /* PRINTFLIKE*n* */ 相同，只是函数定义的第 *n* 个参数会被视为 [fs]scanf() 格式字符串。缺省情况下，如果标准 C 库提供的 [fs]scanf() 函数调用中存在错误，lint 会发出警告。 对于 NOTE 格式，必须指定 *n*。

表 4–9 lint 指令 （续）

指令	操作
NOTE(VARARGS(*n*))	禁止对以下函数声明中可变数量的参数执行常规检查。检查前 *n* 个参数的数据类型；缺少 *n* 时，会将其视为 0。建议在新代码或更新的代码中，在定义中使用省略号 (...) 作为终结符。
NOTE(VARARGS(*fun_name,n*))	对于其定义在该指令之后的函数，禁止在调用带有 *n* 个或更多参数的函数时发出以下消息。对于 NOTE 格式，必须指定 *n*。
/*VARARGS*n*/	■ **使用可变数量的参数调用的函数**

4.6 lint 参考和示例

本节提供有关 lint 的参考信息，包括由 lint 执行的检查、lint 库以及 lint 过滤器。

4.6.1 由 lint 执行的诊断

对以下三大类情况执行特定于 lint 的诊断：不一致的使用、不可移植的代码以及可疑的构造。在本节中，我们将研究在每种条件下 lint 的行为示例，并针对它们引起的问题提供可能的解决方法建议。

4.6.1.1 一致性检查

在文件内部以及各文件之间检查使用变量、参数和函数的不一致性。一般说来，对原型的使用、声明和参数执行的检查与 lint 对旧样式函数执行的检查相同。如果程序未使用函数原型，lint 将比编译器更严格地检查每个函数调用中参数的数量和类型。lint 还标识 [fs]printf() 和 [fs]scanf() 控制字符串中转换定义和参数之间的不匹配。

示例：

- 在文件内部，lint 会标记执行到底部但未向调用函数返回值的非 void 函数。过去，程序员通常通过省略返回类型指明某个函数不应返回值： fun() {}。该约定对于编译器没有任何意义，它会将 fun() 视为具有返回类型 int。可使用返回类型 void 来声明函数以消除该问题。
- 在文件之间，lint 检测非 void 函数不返回值但由于它在某个表达式中有值而仍被使用的情况以及相反的情况（即，函数返回值，但在随后调用中有时或总是被忽略）。当值**总是**被忽略时，可能表示函数定义无效率。当值**有时**被忽略时，可能是样式错误（通常不测试是否存在错误条件）。如果无需检查 strcat()、strcpy() 和 sprintf() 等字符串函数或 printf() 和 putchar() 等输出函数的返回值，可将违例调用强制转换为 void 类型。

- lint 标识已声明但未使用或定义、已使用但未定义或已定义但未使用的变量或函数。将 lint 应用于某个集合中要一起装入的某些文件而非全部文件时，它会产生关于出现以下情况的函数和变量的错误消息：
  - 在那些文件中声明，但在其他地方定义或使用
  - 在那些文件中使用，但在其他地方定义
  - 在那些文件中定义，但在其他地方使用

  可调用 -x 选项以禁止第一种错误消息，调用 -u 以禁止后两种错误消息。

4.6.1.2 可移植性检查

一些不可移植的代码在其缺省行为中带有 lint 标志，当用 -p 或 -Xc 调用 lint 时，会诊断额外的几种情况。后者导致 lint 检查不符合 ISO C 标准的构造。有关使用 -p 和 -Xc 时发出的消息，请参见第 108 页中的 “4.6.2 lint 库”。

示例：

- 在某些 C 语言实现中，未显式声明为 signed 或 unsigned 的字符变量会被视为范围通常在 -128 到 127 之间的带符号值。在其他实现中，它们会被视为范围通常在 0 到 255 之间的非负值。因此测试：

```
char c;
c = getchar();
if (c == EOF) ...
```

其中 EOF 的值为 -1，在字符变量取非负值的计算机上总是失败。使用 -p 调用的 lint 会检查暗示**无格式** char 可取负值的所有比较。然而，在以上示例中，将 c 声明为 signed char 可避免执行诊断，却无法避免出现该问题。这是因为 getchar() 必须返回所有可能的字符和一个不同的 EOF 值，因此 char 无法存储其值。此示例可能是出自于实现定义的符号扩展的最常见示例，我们之所以引用该示例，是为了说明如何通过巧妙地应用 lint 的可移植性选项来帮助您发现与可移植性无关的错误。在任何情况下，将 c 声明为 int。

- 位字段也会出现类似的问题。将常量值赋给位字段时，该字段可能太小，无法容纳该值。在将 int 类型的位字段视为无符号值的计算机上，int x:3 所允许的值在 0 到 7 的范围内；而在将其视为带符号值的计算机上，相应值的范围在 -4 到 3 之间。但是，声明为类型 int 的三位字段无法在后一种计算机上存储值 4。使用 -p 调用的 lint 会标记除 unsigned int 和 signed int 之外的所有位字段类型。它们仅是**可移植的**位字段类型。编译器支持 int、char、short 和 long 位字段类型，它们可以为 unsigned、 signed 或**无格式**。编译器还支持 enum 位字段类型。
- 在将较长的类型赋值给较短的类型时，会出现错误。如果有效位被截断，则失去准确性：

```
short s;
long l;
s = l;
```

lint 在缺省情况下会标记所有此类赋值；可通过调用 -a 选项来禁止该诊断。请记住，使用此选项或任何其他选项调用 lint 时，可能会禁止其他诊断。请查看第 108 页中的 “4.6.2 lint 库”中的列表，以了解禁止多项诊断的选项。

- 一个对象类型指针向具有更严格对齐要求的对象类型指针的强制类型转换可能不可移植。lint 标记以下代码：

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

这是因为在大多数计算机上，int 不能在一个任意字节边界上开始，而 char 则可以。可通过使用 -h 调用 lint 来禁止诊断，尽管这可能会再次禁用其他消息。但是最好通过使用通用指针 void * 来消除该问题。

- ISO C 未定义复杂表达式的求值顺序。也就是说，如果由于某个表达式的求值而更改某个变量时，函数调用、嵌套赋值语句或加减运算符会引起副作用，那么副作用发生的顺序在极大程度上取决于计算机。在缺省情况下，lint 会标记由于副作用而更改并且在同一表达式的其他地方使用的任何变量：

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

在此示例中，a[1] 的值在使用一个编译器时可能为 1，在使用另一个编译器时可能为 2。当按位逻辑运算符 & 被错误地代替逻辑运算符 && 使用时，会导致此诊断：

```
if ((c = getchar()) != EOF & c != '0')
```

4.6.1.3 可疑的构造

lint 会标记那些不能表达程序员意图的各种合法构造。示例：

- unsigned 变量总是具有非负值。因此测试：

```
unsigned x;
if (x < 0) ...
```

总是失败。测试：

```
unsigned x;
if (x > 0) ...
```

等效于：

```
if (x != 0) ...
```

这可能不是预期的操作。lint 会标记 unsigned 变量与负常量或 0 的可疑比较。要将 unsigned 变量与负数的位模式进行比较，请将该变量的类型强制转换为 unsigned：

```
if (u == (unsigned) -1) ...
```

或者使用 U 后缀：

```
if (u == -1U) ...
```

- lint 会标记用于预期出现副作用的上下文但没有副作用的表达式，即可能未表达程序员意图的表达式。它在应该出现赋值运算符的位置发现等号运算符（即预期出现副作用）时发出一个额外的警告：

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- lint 会提醒您注意给混合有逻辑运算符和按位运算符（特别是 &、|、^、<< 和 >>）的表达式加上括号，否则运算符优先级的误解会引起不正确的结果。例如，按位运算符 & 的优先级低于逻辑运算符 ==，所以表达式：

```
if (x & a == 0) ...
```

求值如下：

```
if (x & (a == 0)) ...
```

这很有可能不是您的意图。使用 -h 调用 lint 会禁用 诊断。

4.6.2 lint 库

可以使用 lint 库检查程序是否与已在其中进行调用的库函数兼容，其中包括函数返回类型的声明以及函数所预期参数的数量和类型等。标准 lint 库与 C 编译系统提供的库对应，并且通常存储在系统上的标准位置。按照约定，lint 库的名称采用 llib-lx.ln 形式。

lint 标准 C 库 lliblc.ln 在缺省情况下附加至 lint 命令行；可通过调用 -n 选项来禁止其兼容性检查。其他 lint 库作为 -l 的参数进行访问。即：

```
% lint -lx file1.c file2.c
```

指示 lint 检查 file1.c 和 file2.c 中函数和变量的用法是否与 lint 库 llib-l*x*.ln 兼容。仅由定义组成的库文件完全作为普通源文件和普通 .ln 文件处理，区别在于不会针对以下情况发出错误消息：函数和变量在库文件中的用法不一致，或者函数和变量在库文件中已定义但未在源文件中使用。

要创建您自己的 lint 库，请在 C 源文件的开头插入指令 NOTE(LINTLIBRARY)，然后使用 -o 选项以及指定给 -l 的库名称针对该文件调用 lint：

```
% lint -ox file1.c file2.c
```

导致仅将源文件中以 NOTE(LINTLIBRARY) 开头的定义写入文件 llib-l*x*.ln。（请注意，lint -o 与 cc -o 类似。）可以相同的方式从函数原型声明的文件创建库，不同的是必须在声明文件的最前面插入 NOTE(LINTLIBRARY) 和 NOTE(PROTOLIB(*n*))。如果 *n* 为 1，则原型声明写入库 .ln 文件，这与旧样式的定义相同。如果 *n* 为 0（缺省值），则取消该进程。使用 -y 调用 lint 是创建 lint 库的另一种方法。命令行：

```
% lint -y -ox file1.c file2.c
```

导致该行中命名的每个源文件被视为以 NOTE(LINTLIBRARY) 开头，并且只有其定义被写入 llib-l*x*.ln。

缺省情况下，lint 在标准位置搜索 lint 库。要指示 lint 在非标准位置的目录中搜索 lint 库，请使用 -L 选项指定目录路径：

```
% lint -Ldir -lx file1.c file2.c
```

在增强模式下，lint 生成 .ln 文件，这些文件存储的信息比在基本模式下生成的 .ln 文件存储的信息多。在增强模式下，lint 可以读取和理解由基本或增强 lint 模式生成的所有 .ln 文件。在基本模式下，lint 只能读取和理解由基本 lint 模式生成的 .ln 文件。

缺省情况下，lint 使用 /usr/lib 目录中的库。这些库采用基本 lint 格式。可以运行一次 makefile，并以新格式创建增强 lint 库，从而使增强 lint 更有效地工作。要运行 makefile 并创建新库，请输入以下命令：

```
% cd <install-directory>/prod/src/lintlib; make
```

其中 <*install-directory*> 为安装目录。运行 makefile 之后，lint 在增强模式下使用新库，而不是使用 /usr/lib 目录中的库。

系统会在搜索标准位置之前先搜索指定的目录。

4.6.3 lint 过滤器

lint 过滤器是特定于项目的后处理程序，通常使用 awk 脚本或类似程序读取 lint 的输出，并放弃项目认为**没有**标识真正问题的消息（例如，对于字符串函数，返回值有时或总是被忽略）。当 lint 选项和指令未提供对输出的足够控制时，lint 过滤器会生成定制的诊断报告。

lint 的两个选项在开发过滤器的过程中特别有用：

- 使用 -s 调用 lint 会导致将复合诊断转换为简单诊断，并对诊断的每个具体问题发出一行消息。这种易于进行语法分析的消息格式适合由 awk 脚本进行分析。

- 使用 -k 调用 lint 会导致在输出中输出您在源文件中编写的某些注释，并且有助于记录项目决策和指定后处理程序的行为。在后一个实例中，如果注释标识了预期的 lint 消息，并且报告的消息相同，则会过滤掉该消息。要使用 -k，请在要注释的代码前面的行中插入 NOTE(LINTED(*msg*)) 指令，其中 *msg* 表示在使用 -k 调用 lint 时将输出的注释。

  请参阅表 4–9 中的指令列表，以了解**未**调用 -k 时 lint 所执行操作的说明（对于包含 NOTE(LINTED(*msg*)) 的文件。

◆ ◆ ◆ 第 5 章

基于类型的别名分析

本文档说明如何使用 `-xalias_level` 选项和几个 pragma，以便编译器可以执行基于类型的别名分析和优化。您可以使用这些扩展功能表示关于 C 程序中使用指针方法的基于类型的信息。C 编译器又可以使用此信息对程序中基于指针的内存引用更好地进行别名歧义消除并且效果显著。

有关此命令语法的详细说明，请参见第 219 页中的 “B.2.72 -xalias_level[=*l*]”。此外，有关 `lint` 程序的基于类型的别名分析功能的说明，请参见第 97 页中的 “4.3.38 -Xalias_level[=*l*]”。

5.1 介绍基于类型的分析

可以使用 `-xalias_level` 选项指定七个别名级别之一。每个级别指定一组关于您在 C 程序中使用指针的方法的属性。

当您使用 `-xalias_level` 选项的较高级别进行编译时，编译器会对您的代码中的指针进行更广泛的假定。当编译器产生较少假定时，您有更大的编程自由。但是，这些狭窄假定产生的优化可能不会导致运行环境性能的显著提高。如果您依照 `-xalias_level` 选项的更高级别的编译器假定进行编码，则更有可能使产生的优化提高运行环境性能。

`-xalias_level` 选项指定应用于每个转换单元的别名级别。在越详细越有益的情况下，您可以使用新的 pragma 覆盖已生效的别名级别，以便可以明确指定转换单元中个体类型或指针变量之间的别名关系。如果转换单元中指针的使用对应于某个可用别名级别，但是一些特定指针变量的使用方法是某个可用级别不允许的不规则方法，这些 pragma 非常有用。

5.2 使用 Pragma 以便更好地控制

如果进行基于类型的分析时，更详细的信息可为您提供帮助，则可以使用以下 pragma 覆盖已生效的别名级别，并指定转换单元中个体类型或指针变量之间的别名关系。如果转换单元中指针的使用与某个可用别名级别一致，但是一些特定指针变量的使用方法是某个可用级别不允许的不规则方法，这些 pragma 非常有益。

注 – 必须在 pragma 之前声明命名的类型或变量，否则会发出警告消息并忽略 pragma。如果 pragma 出现在其含义所适用的第一个内存引用之后，则程序的结果不确定。

下列术语用于 pragma 定义。

术语	含义
level	第 219 页中的 “B.2.72 -xalias_level[=*l*]”下列出的任何别名级别。
type	以下任何类型： ■ `char`、`short`、`int`、`long`、`long long`、`float`、`double`、`long double` ■ `void`，表示所有指针类型 ■ `typedef` *name*，它是 `typedef` 声明中定义的类型的名称 ■ `struct` *name*，它是后面有 `struct` *tag* 名称的关键字 *struct* ■ `union`，它是后面有 *union tag* 名称的关键字 `union`
pointer_name	转换单元中指针类型的任何变量的名称。

5.2.1 #pragma alias_level *level* (*list*)

将 *level* 替换为以下七个别名级别之一：`any`、`basic`、`weak`、`layout`、`strict`、`std` 或 `strong`。您可以使用单一类型或逗号分隔的类型列表替换 *list*，也可以使用单一指针或逗号分隔的指针列表替换 *list*。例如，您可以按以下方式发出 `#pragma alias_level`：

- `#pragma alias_level` *level* `(type [, type] )`
- `#pragma alias_level` *level* `(pointer [, pointer] )`

此 pragma 指定，指示的别名级别应用于所列类型的转换单元的所有内存引用，或者应用于其中某个命名指针变量正在被非关联化的转换单元的所有非关联化。

如果您指定多个要应用于特定非关联化的别名级别，则指针名称（如果有）应用的级别优先于所有其他级别。类型名称（如果有）应用的级别优先于选项应用的级别。在以下示例中，如果在编译程序时将 `#pragma alias_level` 设置得比 `any` 高，则 `std` 级别应用于 *p*。

```
typedef int * int_ptr;
int_ptr p;
#pragma alias_level strong (int_ptr)
#pragma alias_level std (p)
```

5.2.1.1 #pragma alias (type, type [, type]…)

此 pragma 指定所有列出的类型互为别名。在以下示例中，编译器假定间接访问 *pt 的别名为间接访问 *pf。

```
#pragma alias (int, float)
int *pt;
float *pf;
```

5.2.1.2 #pragma alias (pointer, pointer [, pointer] …)

此 pragma 指定，在任何命名指针变量的任何非关联化点，正被非关联化的指针值可以指向与任何其他命名指针变量相同的对象。但是，指针并不仅限于命名变量中包含的对象，可以指向列表中未包含的对象。此 pragma 覆盖应用的任何别名级别的别名假定。在以下示例中，该 pragma 之后对 p 和 q 的任何间接访问无论是什么类型，均被视为别名。

```
#pragma alias(p, q)
```

5.2.1.3 #pragma may_point_to (pointer, variable [, variable] …)

此 pragma 指定，在进行命名指针变量的任何非关联化操作时，被非关联化的指针值可以指向任何命名变量中包含的对象。但是，指针并不仅限于命名变量中包含的对象，可以指向列表中未包含的对象。此 pragma 覆盖应用的任何别名级别的别名假定。在以下示例中，编译器假定对 *p 的任何间接访问的别名可以是任何直接访问 a、b 和 c。

```
#pragma alias may_point_to(p, a, b, c)
```

5.2.1.4 #pragma noalias (type, type [, type]…)

此 pragma 指定列出的类型不互为别名。在以下示例中，编译器假定对 *p 的任何间接访问不将间接访问 *ps 作为别名。

```
struct S {
   float f;
   ...} *ps;

#pragma noalias(int, struct S)
int *p;
```

5.2.1.5 #pragma noalias (pointer, pointer [, pointer] …)

此 pragma 指定，在进行任何命名指针变量的任何非关联化操作时，被非关联化的指针值不指向与任何其他命名指针变量相同的对象。此 pragma 覆盖所有其他应用的别名级别。在以下示例中，编译器假定无论两个指针是什么类型，对 *p 的任何间接访问均不将间接访问 *q 作为别名。

```
#pragma noalias(p, q)
```

5.2.1.6 #pragma may_not_point_to (pointer, variable [, variable] …)

此 pragma 指定，在命名指针变量的任何非关联化点，正被非关联化的指针值不指向任何命名变量中包含的对象。此 pragma 覆盖所有其他应用的别名级别。在以下示例中，编译器假定 *p 的任何间接访问与 a、b 或 c 的直接访问不为别名。

```
#pragma may_not_point_to(p, a, b, c)
```

5.3 使用 lint 检查

lint 程序识别与编译器的 -xalias_level 命令同级别的基于类型的别名歧义消除。lint 程序还识别与本章中说明的基于类型的别名歧义消除相关的 pragma。有关 lint -Xalias_level 命令的详细说明，请参见第 97 页中的 “4.3.38 -Xalias_level[=*l*]”。

lint 检测以下四种情况并生成警告：

- 将标量指针强制转换为结构指针
- 将空指针强制转换为结构指针
- 将结构字段强制转换为标量指针
- 将结构指针强制转换为 -Xalias_level=strict 级别上没有显式别名的结构指针。

5.3.1 标量指针向结构指针的强制类型转换

在以下示例中，整型指针 p 强制转换为 struct foo 类型的指针。如果 lint -Xalias_level=weak（或更高），这将生成错误。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
int *p;

void main()
{
    f = (struct foo *)p; /* struct pointer cast of scalar pointer error */
}
```

5.3.2 空指针向结构指针的强制类型转换

在以下示例中，空指针 vp 强制转换为结构指针。如果 lint -Xalias_level=weak（或更高），这将生成警告。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
void *vp;

void main()
{
    f = (struct foo *)vp; /* struct pointer cast of void pointer warning */
}
```

5.3.3 结构字段向结构指针的强制类型转换

在下面的示例中，结构成员 **foo.b** 的地址被强制转换为结构指针，然后指定给 **f2**。如果 lint -Xalias_level=weak（或更高），这将生成错误。

```
struct foo{
    int a;
    int b;
  };

struct foo *f1;
struct foo *f2;

void main()
{
    f2 = (struct foo *)&f1->b; /* cast of a scalar pointer to struct pointer error*/
}
```

5.3.4 要求显式别名

在以下示例中，struct fooa 类型的指针 f1 正在被强制转换为 struct foob 类型的指针。如果 lint -Xalias_level=strict（或更高），则除非结构类型相同（相同类型的相同数目的字段），否则此类强制类型转换要求显式别名。此外，在别名级别 standard 和 strong 上，假定标记必须匹配才能出现别名。在给 f1 赋值前使用 #pragma alias (struct fooa, struct foob)，lint 将停止生成警告。

```
struct fooa {
    int a;
};

struct foob {
```

```
    int b;
};

struct fooa *f1;
struct foob *f2;

void main()
{
    f1 = (struct fooa *)f2; /* explicit aliasing required warning */
}
```

5.4 内存引用约束的示例

本节提供可能出现在您的源文件中的代码示例。每个示例后跟对编译器代码假定的讨论，这些假定取决于所应用的基于类型的分析级别。

5.4.1 第一个示例

考虑以下代码。可以使用不同的别名级别对它进行编译，以说明显示类型的别名关系。

```
struct foo {
    int f1;
    short f2;
    short f3;
    int f4;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;

int *ip;
short *sp;
```

如果该示例是使用 `-xalias_level=any` 选项编译的，编译器将认为以下间接访问互为别名：

`*ip`、`*sp`、`*fp`、`*bp`、`fp->f1`、`fp->f2`、`fp->f3`、`fp->f4`、`bp->b1`、`bp->b2`、`bp->b3`

如果该示例是使用 `-xalias_level=basic` 选项编译的，编译器将认为以下间接访问互为别名：

`*ip`、`*bp`、`fp->f1`、`fp->f4`、`bp->b1`、`bp->b2`、`bp->b3`

另外，`*sp`、`fp->f2` 和 `fp->f3` 可以互为别名，`*sp` 和 `*fp` 可以互为别名。

但是，在 `-xalias_level=basic` 条件下，编译器作出以下假定：

- *ip 不将 *sp 作为别名。
- *ip 不将 fp->f2 和 fp->f3 作为别名。
- *sp 不将 fp->f1、fp->f4、bp->b1、bp->b2 和 bp->b3 作为别名。

由于两个间接访问的访问类型是不同的基本类型，因此编译器作出这些假定。

如果该示例是使用 -xalias_level=weak 选项编译的，编译器将假定以下别名信息：

- *ip 可将 *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2 和 bp->b3 作为别名。
- *sp 可将 *fp、fp->f2 和 fp->f3 作为别名。
- fp->f1 可将 bp->b1 作为别名。
- fp->f4 可将 bp->b3 作为别名。

由于 f1 是结构中偏移为 0 的字段，而 b2 是结构中偏移为 4 个字节的字段，因此编译器假定 fp->fp1 不将 bp->b2 作为别名。同样，编译器假定 fp->f1 不将 bp->b3 作为别名，fp->f4 不将 bp->b1 或 bp->b2 作为别名。

如果该示例是使用 -xalias_level=layout 选项编译的，编译器将假定以下别名信息：

- *ip 可将 *fp、*bp、fp->f1、fp->f4、bp->b1、bp->b2 和 bp->b3 作为别名。
- *sp 可将 *fp、fp->f2 和 fp->f3 作为别名。
- fp->f1 可将 bp->b1 和 *bp 作为别名。
- *fp 和 *bp 可以互为别名。

由于 f4 和 b3 不是 foo 和 bar 的公共初始序列中的对应字段，因此 fp->f4 不将 bp->b3 作为别名。

如果该示例是使用 -xalias_level=strict 选项编译的，编译器将假定以下别名信息：

- *ip 可将 *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2 和 bp->b3 作为别名。
- *sp 可将 *fp、fp->f2 和 fp->f3 作为别名。

如果 -xalias_level=strict，则编译器假定 *fp、*bp、fp->f1、fp->f2、fp->f3、fp->f4、bp->b1、bp->b2 和 bp->b3 不互为别名，因为在忽略字段名时 foo 和 bar 是不同的。但是，fp 可将 fp->f1 作为别名，bp 可将 bp->b1 作为别名。

如果该示例是使用 -xalias_level=std 选项编译的，编译器将假定以下别名信息：

- *ip 可将 *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2 和 bp->b3 作为别名。
- *sp 可将 *fp、fp->f2 和 fp->f3 作为别名。

但是，fp->f1 不将 bp->b1、bp->b2 或 bp->b3 作为别名，因为在考虑字段名时 foo 和 bar 并不相同。

如果该示例是使用 -xalias_level=strong 选项编译的，编译器将假定以下别名信息：

- *ip 不将 fp->f1、fp->f4、bp->b1、bp->b2 和 bp->b3 作为别名，因为指针（如 *ip）不应指向结构内部。

- 同样，*sp 不将 fp->f1 或 fp->f3 作为别名。
- 由于类型不同，*ip 不将 *fp、*bp 和 *sp 作为别名。
- 由于类型不同，*sp 不将 *fp、*bp 和 *ip 作为别名。

5.4.2 第二个示例

考虑以下源代码示例。当使用不同的别名级别编译时，它说明显示的类型的别名关系。

```
struct foo {
    int f1;
    int f2;
    int f3;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;
```

如果该示例是使用 -xalias_level=any 选项编译的，编译器将假定以下别名信息：

*fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2 和 bp->b3 都可以互为别名，因为任何两个内存访问在 -xalias_level=any 级别上可互为别名。

如果该示例是使用 -xalias_level=basic 选项编译的，编译器将假定以下别名信息：

*fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2 和 bp->b3 都可以互为别名。在本示例中，由于所有结构字段均为同一基本类型，因此任何两个使用指针 *fp 和 *bp 的字段访问都可以互为别名。

如果该示例是使用 -xalias_level=weak 选项编译的，编译器将假定以下别名信息：

- *fp 和 *fp 可以互为别名。
- fp->f1 可将 bp->b1、*bp 和 *fp 作为别名。
- fp->f2 可将 bp->b2、*bp 和 *fp 作为别名。
- fp->f3 可将 bp->b3、*bp 和 *fp 作为别名。

但是，-xalias_level=weak 强加以下限制：

- 由于 f1 的偏移为零，与 b2 的偏移（四字节）和 b3 的偏移（八字节）不同，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于 f2 的偏移为四字节，与 b1 的偏移（零字节）和 b3 的偏移（八字节）不同，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于 f3 的偏移为八字节，与 b1 的偏移（零字节）和 b2 的偏移（四字节）不同，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果该示例是使用 -xalias_level=layout 选项编译的，编译器将假定以下别名信息：

- *fp 和 *bp 可以互为别名。
- fp->f1 可将 bp->b1、*bp 和 *fp 作为别名。
- fp->f2 可将 bp->b2、*bp 和 *fp 作为别名。
- fp->f3 可将 bp->b3、*bp 和 *fp 作为别名。

但是，-xalias_level=layout 强加以下限制：

- 由于在 foo 和 bar 的公共初始序列中字段 f1 对应于字段 b1，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于在 foo 和 bar 的公共初始序列中字段 f2 对应于字段 b2，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于在 foo 和 bar 的公共初始序列中字段 f3 对应于字段 b3，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果该示例是使用 -xalias_level=strict 选项编译的，编译器将假定以下别名信息：

- *fp 和 *bp 可以互为别名。
- fp->f1 可将 bp->b1、*bp 和 *fp 作为别名。
- fp->f2 可将 bp->b2、*bp 和 *fp 作为别名。
- fp->f3 可将 bp->b3、*bp 和 *fp 作为别名。

但是，-xalias_level=strict 强加以下限制：

- 由于在 foo 和 bar 的公共初始序列中字段 f1 对应于字段 b1，因此 fp->f1 不将 bp->b2 或 bp->b3 作为别名。
- 由于在 foo 和 bar 的公共初始序列中字段 f2 对应于字段 b2，因此 fp->f2 不将 bp->b1 或 bp->b3 作为别名。
- 由于在 foo 和 bar 的公共初始序列中字段 f3 对应于字段 b3，因此 fp->f3 不将 bp->b1 或 bp->b2 作为别名。

如果该示例是使用 -xalias_level=std 选项编译的，编译器将假定以下别名信息：

fp->f1、fp->f2、fp->f3、bp->b1、bp->b2 和 bp->b3 不互为别名。

如果该示例是使用 -xalias_level=strong 选项编译的，编译器将假定以下别名信息：

fp->f1、fp->f2、fp->f3、bp->b1、bp->b2 和 bp->b3 不互为别名。

5.4.3 第三个示例

考虑下列源代码示例，它说明某些别名级别无法处理内部指针。有关内部指针的定义，请参见表 B–13。

```
struct foo {
        int f1;
        struct bar *f2;
```

```
        struct bar *f3;
        int f4;
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

weak、layout、strict 或 std 不支持该示例中的非关联化。在指针赋值 bp=(struct bar*)(&fp->f2) 之后，以下各对内存访问将访问相同的存储单元：

- fp->f2 和 bp->b2 访问相同的存储单元
- fp->f3 和 bp->b3 访问相同的存储单元
- fp->f4 和 bp->b4 访问相同的存储单元

但是，使用选项 weak、layout、strict 和 std 时，编译器假定 fp->f2 和 bp->b2 不设定别名。由于 b2 的偏移为零，与 f2 的偏移（四字节）不同，并且 foo 和 bar 没有公共初始序列，因此编译器作出该假定。同样，编译器还假定 bp->b3 不将 fp->f3 作为别名，bp->b4 不将 fp->f4 作为别名。

因此，指针赋值 bp=(struct bar*)(&fp->f2) 将使编译器关于别名信息的假定不正确。这可能会导致不正确的优化。

请在进行以下示例中显示的修改之后尝试编译。

```
struct foo {
        int f1;
        struct bar fb;   /* Modified line */
#define f2 fb.b2         /* Modified line */
#define f3 fb.b3         /* Modified line */
#define f4 fb.b4         /* Modified line */
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

在指针赋值 bp=(struct bar*)(&fp->f2) 之后，以下各对内存访问将访问相同的存储单元：

- fp->f2 和 bp->b2
- fp->f3 和 bp->b3

- fp->f4 和 bp->b4

通过检查前面的代码示例中显示的更改，您可以看到表达式 fp->f2 是表达式 fp->fb.b2 的另一种形式。由于 fp->fb 为 bar 类型，因此 fp->f2 访问 bar 的 b2 字段。此外，bp->b2 也访问 bar 的 b2 字段。因此，编译器假定 fp->f2 将 bp->b2 作为别名。同样，编译器假定 fp->f3 将 bp->b3 作为别名，fp->f4 将 bp->b4 作为别名。结果，编译器假定的别名与指针赋值产生的实际别名匹配。

5.4.4 第四个示例

考虑以下源代码示例。

```
struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

struct cat {
        int c1;
        struct foo cf;
        int c2;
        int c3;
} *cp;

struct dog {
        int d1;
        int d2;
        struct bar db;
        int d3;
} *dp;
```

如果该示例是使用 -xalias_level=weak 选项编译的，编译器将假定以下别名信息：

- fp->f1 可将 bp->b1、cp->c1、dp->d1、cp->cf.f1 和 df->db.b1 作为别名。
- fp->f2 可将 bp->b2、cp->cf.f1、dp->d2、cp->cf.f2、df->db.b2、cp->c2 作为别名。
- bp->b1 可将 fp->f1、cp->c1、dp->d1、cp->cf.f1 和 df->db.b1 作为别名。
- bp->b2 可将 fp->f2、cp->cf.f1、dp->d2、cp->cf.f1 和 df->db.b2 作为别名。

fp->f2 可将 cp->c2 作为别名，因为 *dp 可将 *cp 作为别名，且 *fp 可将 dp->db 作为别名。

- cp->c1 可将 fp->f1、bp->b1、dp->d1 和 dp->db.b1 作为别名。
- cp->cf.f1 可将 fp->f1、fp->f2、bp->b1、bp->b2、dp->d2 和 dp->d1 作为别名。

cp->cf.f1 不将 dp->db.b1 作为别名。

- cp->cf.f2 可将 fp->f2、bp->b2、dp->db.b1 和 dp->d2 作为别名。
- cp->c2 可将 dp->db.b2 作为别名。

cp->c2 不将 dp->db.b1 作为别名，cp->c2 不将 dp->d3 作为别名。

对于偏移，仅当 *dp 将 cp->cf 作为别名时，cp->c2 才能将 db->db.b1 作为别名。但是，如果 *dp 将 cp->cf 作为别名，则 dp->db.b1 必须在 foo cf 末尾之后设定别名，这是对象约束所禁止的。因此，编译器假定 cp->c2 不能将 db->db.b1 作为别名。

cp->c3 可将 dp->d3 作为别名。

请注意，cp->c3 不将 dp->db.b2 作为别名。由于具有非关联化所涉及的类型的字段的偏移不同并且不重叠，因此这些内存引用不设定别名。基于这种情况，编译器假定它们不能设定别名。

- dp->d1 可将 fp->f1、bp->b1 和 cp->c1 作为别名。
- dp->d2 可将 fp->f2、bp->b2 和 cp->cf.f1 作为别名。
- dp->db.b1 可将 fp->f1、bp->b1 和 cp->c1 作为别名。
- dp->db.b2 可将 fp->f2、bp->b2、cp->c2 和 cp->cf.f1 作为别名。
- dp->d3 可将 cp->c3 作为别名。

请注意，dp->d3 不将 cp->cf.f2 作为别名。由于具有非关联化所涉及的类型的字段的偏移不同并且不重叠，因此这些内存引用不设定别名。基于这种情况，编译器假定它们不能设定别名。

如果该示例是使用 -xalias_level=layout 选项编译的，编译器只假定以下别名信息：

- fp->f1、bp->b1、cp->c1 和 dp->d1 都可以互为别名。
- fp->f2、bp->b2 和 dp->d2 都可以互为别名。
- fp->f1 可将 cp->cf.f1 和 dp->db.b1 作为别名。
- bp->b1 可将 cp->cf.f1 和 dp->db.b1 作为别名。
- fp->f2 可将 cp->cf.f2 和 dp->db.b2 作为别名。
- bp->b2 可将 cp->cf.f2 和 dp->db.b2 作为别名。

如果该示例是使用 -xalias_level=strict 选项编译的，编译器只假定以下别名信息：

- fp->f1 和 bp->b1 可互为别名。
- fp->f2 和 bp->b2 可互为别名。
- fp->f1 可将 cp->cf.f1 和 dp->db.b1 作为别名。
- bp->b1 可将 cp->cf.f1 和 dp->db.b1 作为别名。
- fp->f2 可将 cp->cf.f2 和 dp->db.b2 作为别名。
- bp->b2 可将 cp->cf.f2 和 dp->db.b2 作为别名。

如果该示例是使用 -xalias_level=std 选项编译的，编译器只假定以下别名信息：

- fp->f1 可将 cp->cf.f1 作为别名。
- bp->b1 可将 dp->db.b1 作为别名。
- fp->f2 可将 cp->cf.f2 作为别名。

- bp->b2 可将 dp->db.b2 作为别名。

5.4.5 第五个示例

考虑以下源代码示例。

```
struct foo {
        short f1;
        short f2;
        int   f3;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

union moo {
        struct foo u_f;
        struct bar u_b;
} u;
```

下面是编译器根据以下别名级别作出的假定：

- 如果该示例是使用 -xalias_level=weak 选项编译的，fp->f3 和 bp->b2 可互为别名。
- 如果该示例是使用 -xalias_level=layout 选项编译的，没有字段可以互为别名。
- 如果该示例是使用 -xalias_level=strict 选项编译的，fp->f3 和 bp->b2 可互为别名。
- 如果该示例是使用 -xalias_level=std 选项编译的，没有字段可互为别名。

5.4.6 第六个示例

考虑以下源代码示例。

```
struct bar;

struct foo {
        struct foo *ffp;
        struct bar *fbp;
} *fp;

struct bar {
        struct bar *bbp;
        long        b2;
} *bp;
```

下面是编译器根据以下别名级别作出的假定：

- 如果该示例是使用 -xalias_level=weak 选项编译的，只有 fp->ffp 和 bp->bbp 可互为别名。

- 如果该示例是使用 `-xalias_level=layout` 选项编译的，只有 `fp->ffp` 和 `bp->bbp` 可互为别名。
- 如果该示例是使用 `-xalias_level=strict` 选项编译的，则没有字段可以互为别名，原因是即使删除两种结构类型的标记，两种结构类型仍不相同。
- 如果该示例是使用 `-xalias_level=std` 选项编译的，则没有字段可以互为别名，原因是两种类型和标记均不相同。

5.4.7 第七个示例

考虑以下源代码示例：

```
struct foo;
struct bar;
#pragma alias (struct foo, struct bar)

struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        short b1;
        short b2;
        int   b3;
} *bp;
```

此示例中的 pragma 告知编译器，允许 `foo` 和 `bar` 互为别名。编译器作出关于别名信息的以下假定：

- `fp->f1` 可将 `bp->b1`、`bp->b2` 和 `bp->b3` 作为别名
- `fp->f2` 可将 `bp->b1`、`bp->b2` 和 `bp->b3` 作为别名

◆ ◆ ◆ 第 6 章

6 转换为 ISO C

本章提供的信息可以帮助您移植 K&R 风格 C 语言应用程序，以符合 9899:1990 ISO/IEC C 标准。由于您不想符合更新的 9899:1999 ISO/IEC C 标准，因此本章提供的信息假定您使用 `-xc99=none`。C 编译器缺省为 `-xc99=all`，支持 9899:1999 ISO/IEC C 标准。

6.1 基本模式

ISO C 编译器允许同时使用旧式和新式 C 代码。如果您使用下列 `-X`（注意大小写）选项并且 `-xc99=none`，则编译器提供不同的 ISO C 标准一致性级别。`-Xa` 是缺省模式。请注意，编译器的缺省模式为 `-xc99=all`，因此在设置每个 `-X` 选项的情况下编译器的行为取决于 `-xc99` 的设置。

6.1.1 -Xc

（`c` = 一致性）在没有 K&R C 兼容性扩展的情况下，在最大程度上与 ISO C 一致。编译器对使用 ISO C 构造的程序发出错误和警告。

6.1.2 -Xa

ISO C 以及 K&R C 兼容性扩展，具有 ISO C 要求的语义更改。如果 K&R C 和 ISO C 为相同构造指定不同语义，则编译器发出关于冲突的警告并使用 ISO C 解释。这是缺省模式。

6.1.3 -Xt

（`t` = 转换）ISO C 以及 K&R C 兼容性扩展，**没有** ISO C 要求的语义更改。如果 K&R C 和 ISO C 为相同构造指定不同语义，则编译器发出关于冲突的警告并使用 K&R C 解释。

6.1.4 -Xs

(s = K&R C) 编译的语言包括与 ISO K&R C 兼容的所有功能。编译器对在 ISO C 和 K&R C 之间具有不同行为的所有语言构造发出警告。

6.2 旧式和新式函数的混合

1990 ISO C 标准在语言方面的最大变化是借鉴了 C++ 语言的函数原型。通过为每个函数指定其参数的数目和类型，不仅使各常规编译获得对每个函数调用的形参 (argument) 和实参 (parameter) 检查（类似于 `lint` 的参数检查）的益处，而且参数自动转换（与赋值情形相同）为函数预期的类型。由于存在许许多多可以而且应该转换为使用原型的现有 C 代码行，因此 1990 ISO C 标准包括了控制旧式和新式函数声明混合的规则。

1999 ISO C 标准使得旧式函数声明被废弃。

6.2.1 编写新代码

编写全新的程序时，在头文件中使用新式函数声明（函数原型），在其他 C 源文件中使用新式函数声明和定义。但是，如果将来可能将代码移植到使用传统的（即采纳 ISO 标准之前）C 编译器的计算机，我们建议您在头文件和源文件中使用宏 `__STDC__`（仅为 ISO C 编译系统定义）。有关示例，请参阅第 127 页中的 “6.2.3 混合注意事项”。

只要同一对象或函数的两个不兼容声明处于同一作用域中，符合 ISO C 的编译器就必须发出诊断。如果使用原型来声明和定义所有函数，并且相应的头文件包含在正确的源文件中，则所有调用应与函数的定义一致。此协议消除一种最常见的 C 编程错误。

6.2.2 更新现有代码

如果您有现有的应用程序并且要获取函数原型的益处，则可以进行很多更新，取决于您要更改的代码的数目：

1. 重新编译而不进行任何更改。

   如果使用 –v 选项进行调用，即使不更改代码，编译器也会对参数类型和数目的不匹配发出警告。

2. 仅在头文件中增加函数原型。

   包括所有全局函数调用。

3. 在头文件中增加函数原型，并使每个源文件以其局部（静态）函数的函数原型开头。

   包括所有函数调用，但是这样做需要在源文件中为每个局部函数键入两次接口。

4. 更改所有函数声明和定义以使用函数原型。

对于大多数程序员，第 2 种选择和第 3 种选择可能最具成本效益。遗憾的是，这些选项要求程序员详细了解混合新旧风格的规则。

6.2.3 混合注意事项

为了使函数原型声明适用于旧式函数定义，两者都必须使用 ISO C 术语指定功能相同的接口或必须具有**兼容类型**。

对于具有可变参数的函数，不能混合 ISO C 的省略号和旧式 `varargs()` 函数定义。对于具有固定数目参数的函数，情况相当简单：只需指定在先前实现中传递的参数的类型。

在 K&R C 中，根据缺省参数提升，就在将每个参数传递到被调用函数之前对其进行转换。这些提升规定，所有比 `int` 短的整数类型均要提升为 `int` 长度，并且任何 `float` 参数均要提升为 `double`，从而简化了编译器和库。函数原型更具有表现力－指定的参数类型即为传递给函数的类型。

因此，如果为现有的（旧式）函数定义编写函数原型，则在具有以下任一类型的函数原型中不应有参数：

`char`	`signed char`	`unsigned char`	`float`
`short`	`signed short`	`unsigned short`	

在编写原型方面仍存在两个复杂因素：`typedef` 名称以及短的无符号类型的提升规则。

如果旧式函数中的参数是使用 `typedef` 名称（如 `off_t` 和 `ino_t`）声明的，则知道 `typedef` 名称指定的类型是否受到缺省参数提升的影响是至关重要的。对于这两个名称，`off_t` 是 `long` 类型的，因此它适合于在函数原型中使用；`ino_t` 过去是 `unsigned short` 类型的，因此如果在原型中使用它，则编译器将发出诊断，因为旧式定义和原型指定的接口不同且不兼容。

考虑究竟应该使用什么来代替 `unsigned short` 致使问题最终复杂化。K&R C 和 1990 ISO C 编译器之间一个最大的不兼容性是用于将 `unsigned char` 和 `unsigned short` 展宽为 `int` 值的提升规则。（请参见第 131 页中的 “6.4 提升：无符号保留与值保留”。）与这样的旧式参数匹配的参数类型取决于编译时使用的编译模式：

- `-Xs` 和 `–Xt` 应使用 `unsigned int`
- `–Xa` 和 `–Xc` 应使用 `int`

最佳方法是将旧式定义更改为指定 `int` 或 `unsigned int` 并使用函数原型中的匹配类型。如有必要，在输入函数后，您可以始终将其值赋给具有更窄类型的局部变量。

请注意原型中 ID 的使用，它可能受预处理的影响。请看以下示例：

```
#define status 23
void my_exit(int status);   /* Normally, scope begins */
                            /* and ends with prototype */
```

不要将函数原型与包含窄类型的旧式函数声明混合在一起。

```
void foo(unsigned char, unsigned short);
void foo(i, j) unsigned char i; unsigned short j; {...}
```

正确使用 __STDC__ 可生成一个可用于新旧编译器的头文件：

```
header.h:
    struct s { /* .  .  .  */ };
    #ifdef __STDC__
       void errmsg(int, ...);
       struct s *f(const char *);
       int g(void);
    #else
      void errmsg();
      struct s *f();
      int g();
    #endif
```

以下函数使用原型，但仍可在较旧的系统中编译：

```
struct s *
#ifdef __STDC__
    f(const char *p)
#else
    f(p) char *p;
#endif
{
    /* .  .  .  */
}
```

下面是更新的源文件（与上述选项 3 相同）。局部函数仍使用旧式定义，但包括了原型供较新的编译器使用：

```
source.c:
   #include “header.h”
      typedef /* .  .  .  */ MyType;
   #ifdef __STDC__
      static void del(MyType *);
      /* .  .  .  */
      static void
      del(p)
      MyType *p;
      {
      /* .  .  .  */
      }
      /* .  .  .  */
```

6.3 带有可变参数的函数

在以前的实现中，不能指定函数预期的参数类型，但 ISO C 鼓励您使用原型执行该操作。为支持诸如 printf() 之类的函数，原型语法包括特殊的省略号(…) 终结符。由于一个实现可能需要执行一些特殊操作来处理可变数目的参数，因此 ISO C 要求此类函数的所有声明和定义均包含省略号终结符。

由于参数的 "…" 部分没有名称，因此 stdarg.h 中包含的一组特殊宏为函数提供对这些参数的访问权。此类函数的早期版本必须使用 varargs.h 中包含的类似的宏。

我们假定要编写的函数是一个称为 errmsg() 的错误处理程序，它返回 void，并且函数的唯一固定参数是指定关于错误消息的详细信息的 int。此参数后面可以跟一个文件名和/或一个行号，在它们之后是格式和参数，与指定错误消息文本的 printf() 类似。

为使示例可以使用较早的编译器进行编译，我们广泛使用了仅针对 ISO C 编译系统定义的宏 __STDC__。因此，该函数在相应头文件中的声明为：

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

在包含 errmsg() 定义的文件中，新旧风格变得复杂。首先，要包含的头文件取决于编译系统：

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

包含 stdio.h 是因为我们稍后调用 fprintf() 和 vfprintf()。

其次是函数的定义。标识符 va_alist 和 va_dcl 是旧式 varargs.h 接口的一部分。

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* Note: no semicolon! */
#endif
{
   /* more detail below */
}
```

由于旧式可变参数机制不允许指定任何固定参数，因此必须安排在可变部分之前访问它们。此外，由于参数的 "…" 部分缺少名称，新的 va_start() 宏具有第二个参数－"…" 终结符前面的参数的名称。

作为一种扩展，Solaris Studio ISO C 允许在没有固定参数的情况下声明和定义函数，如下所示：

```
int f(...);
```

对于此类函数，应在第二个参数为空的情况下调用 va_start()，如下所示：

```
va_start(ap,)
```

以下是函数的主体：

```
{
    va_list ap;
    char *fmt;
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;
    va_start(ap);
    /* extract the fixed argument */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

va_arg() 和 va_end() 宏对旧式版本和 ISO C 版本的处理方式相同。由于 va_arg() 更改 ap 的值，因此对 vfprintf() 的调用不能为：

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

FILENAME、LINENUMBER 和 WARNING 宏的定义可能包含在与 errmsg() 的声明相同的头文件中。

对 errmsg() 的调用的样例为：

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n",
argv[optind]);
```

6.4 提升：无符号保留与值保留

1990 ISO C 标准的补充材料 "Rationale" 部分出现以下信息： "QUIET CHANGE"。依赖于无符号保留算术转换的程序表现各异，可能没有错误消息。这被认为是委员会对普遍的当前实践的最重大的更改。

本节研究此更改如何影响代码。

6.4.1 背景

根据 K&R 的《The C Programming Language》（第一版），`unsigned` 准确地指定一种类型；不存在 `unsigned char`、`unsigned short` 或 `unsigned long`，但是在此之后不久，大多数 C 编译器增加了这些类型。有些编译器未实现 `unsigned long`，但是包含了其他两种类型。自然地，当这些新类型与在表达式中其他类型混合时，实现为类型提升选择不同的规则。

在大多数 C 编译器中，使用比较简单的规则“无符号保留”：当无符号类型需要展宽时，将被展宽为无符号类型；当无符号类型与带符号类型混合时，结果为无符号类型。

ISO C 指定的另一个规则称为“值保留”，其中结果类型取决于操作数类型的相对长度。当展宽 `unsigned char` 或 `unsigned short` 类型时，如果 `int` 的长度足以表示较短类型的所有值，则结果类型为 `int`。否则，结果类型为 `unsigned int`。对于大多数表达式，值保留规则产生最常见类型的算术结果。

6.4.2 编译行为

只有在转换模式或 ISO 模式（`-Xt` 或 `-Xs`）下，ISO C 编译器才使用无符号保留提升；在其他两种模式下，即符合标准模式 (`–Xc`) 和 ISO 模式 (`–Xa`)，使用值保留提升规则。

6.4.3 第一个示例：强制类型转换的使用

在以下代码中，假定 `unsigned char` 比 `int` 窄。

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

   return (i + uc) < 17;
}
```

以上代码导致编译器在您使用 `-xtransition` 选项时发出以下警告：

```
line 6: warning: semantics of "<" change in ISO C; use explicit cast
```

加法运算结果的类型为 int（值保留）或 unsigned int（无符号保留），但二者之间的位模式不会更改。在二进制补码机器上，我们获得：

```
    i:       111...110 (-2)
+   uc:      000...001 ( 1)
====================
        111...111 (-1 or UINT_MAX)
```

这种位表示对应于 -1（对于 int）或 UINT_MAX（对于 unsigned int）。因此，如果结果类型为 int，则使用带符号比较且小于测试为真；如果结果类型为 unsigned int，则使用无符号比较且小于测试为假。

强制类型转换的加法用来指定这两种行为之中所期望的行为：

```
value preserving:
    (i + (int)uc) < 17
unsigned preserving:
    (i + (unsigned int)uc) < 17
```

由于不同的编译器对相同的代码选择的不同的含义，因此该表达式存在歧义。强制类型转换的加法帮助阅读器并消除警告消息。

6.4.4 位字段

相同的位字段值的提升存在同样的情况。在 ISO C 中，如果 int 或 unsigned int 位字段中的位数小于 int 中的位数，则所提升的类型为 int；否则，所提升的类型为 unsigned int。在大多数较旧的 C 编译器中，对于显式无符号位字段，所提升的类型为 unsigned int，在其他情况下为 int。

强制类型转换的类似使用可以消除存在歧义的情况。

6.4.5 第二个示例：相同的结果

在以下代码中，假定 unsigned short 和 unsigned char 均比 int 短。

```
int f(void)
{
    unsigned short us;
    unsigned char uc;
    return uc < us;
}
```

在此示例中，两个自动变量会同时提升为 int 或 unsigned int，因此比较有时无符号，有时带符号。然而，由于两种选择的结果相同，因此 C 编译器并不向您发出警告。

6.4.6 整型常量

与表达式一样，有些整型常量的类型规则已更改。在 K&R C 中，只有在无后缀十进制常量的值用 int 足以表示时，其类型才为 int；只有在无后缀八进制或十六进制常量的值用 unsigned int 足以表示时，其类型才为 int。否则，整型常量的类型为 long。有时，值用结果类型不足以表示。在 1990 ISO/IEC C 标准中，常量类型是以下列表中与值对应的第一个类型：

- 无后缀十进制：int、long、unsigned、long
- 无后缀八进制或十六进制：int、unsigned int、long、unsigned long
- U 后缀：unsigned int、unsigned long
- L 后缀：long、unsigned long
- UL 后缀：unsigned long

当您使用 -xtransition 选项时，对于其行为可能会根据所涉及常量的类型处理规则而更改的任何表达式，ISO C 编译器会向您发出警告。旧整型常量类型处理规则仅在转换模式下使用；ISO 模式和符合标准模式使用新规则。

注 – 无后缀十进制常量的类型处理规则已按照 1999 ISO C 标准更改。请参见第 31 页中的 “2.1.1 整型常量”。

6.4.7 第三个示例：整型常量

在以下代码中，假定 int 为 16 位。

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

由于十六进制常量的类型为 int（在二进制补码机器上，值为 –1）或 unsigned int（值为 65535），因此在 –Xs 和 -Xt 模式下比较为真，在 –Xa 和 –Xc 模式下比较为假。

同样，相应的强制类型转换澄清代码并禁止警告：

```
-Xt, -Xs modes:
    i > (int)0xffff

-Xa, -Xc modes:
    i > (unsigned int)0xffff
       or
    i > 0xffffU
```

U 后缀字符是 ISO C 的新增功能，对于较旧的编译器它可能会产生错误消息。

6.5 标记化和预处理

这可能是以前的 C 版本中最少涉及的部分，涉及将每个源文件从一串字符转换为标记序列（即可进行语法分析）的操作。这些操作包括白空间（包括注释）的识别、将连续指令捆绑为标记、处理预处理指令行以及宏替换。然而，从未对其各自顺序提供保证。

6.5.1 ISO C 转换阶段

这些转换阶段的顺序由 ISO C 指定。

替换源文件中的每个三字符序列。ISO C 正好有九个为弥补不完善的字符集而单独构造的三字符序列，它们是命名 ISO 646-1983 字符集之外的字符的三字符序列：

表 6–1 三字符序列

三字符序列	转换为
??=	#
??-	~
??(	[
??)	]
??!	|
??<	{
??>	}
??/	\
??’	^

ISO C 编译器必定理解这些序列，但我们建议不要使用它们。在您使用 `-xtransition` 选项时，只要 ISO C 编译器在转换 (`–Xt`) 模式下替换三字母（甚至是在注释中），它就会向您发出警告。例如，考虑以下情形：

```
/* comment *??/
/* still comment? */
```

`??/` 变为反斜杠。该字符和后面的换行符被删除。结果字符为：

```
/* comment */* still comment? */
```

第二行的第一个 `/` 是注释的结尾。下一个标记是 `*`。

1. 删除每个反斜杠/换行符对。

2. 源文件转换为预处理标记和白空间序列。每个注释有效地替换为一个空格字符。
3. 处理各个预处理指令并替换所有宏调用。每个 `#included` 源文件在其内容替换指令行之前运行较早的阶段。
4. 解释各个换码序列（形式为字符常量和文本字符串）。
5. 串联相邻文本字符串。
6. 各个预处理标记转换为常规标记，编译器正确分析这些标记并生成代码。
7. 解析所有外部对象和函数引用，形成最终程序。

6.5.2 旧 C 转换阶段

以前的 C 编译器不执行如此简单的阶段序列，也不保证何时应用这些步骤。独立预处理程序识别标记和空白的时间基本上与它替换宏和处理指令行的时间相同。然后输出由适当的编译器完全重新标记化，接着编辑器分析语言并生成代码。

由于预处理程序中标记化进程的操作时间很短，且宏替换是作为基于字符（而不是基于标记）的操作执行的，因此在预处理过程中标记和空白可能会发生很大的变化。

这两种方法存在很多差异。本节其余部分讨论代码行为如何因宏替换过程中发生的行拼接、宏替换、字符串化以及标记粘贴而更改。

6.5.3 逻辑源代码行

在 K&R C 中，仅允许将反斜杠/换行符对作为一种将指令、文本字符串或字符常量延续到下一行的方法。ISO C 扩展了该概念以便反斜杠/换行符对可以将任何内容延续到下一行。结果为逻辑源代码行。因此，依赖于反斜杠/换行符对任一侧上单独标记识别的任何代码的行为不像预期的那样。

6.5.4 宏替换

在 ISO C 之前，从未详细描述宏替换过程。这种不明确性产生很多有分歧的实现。依赖于比明显常量替换和简单类函数宏更奇特的事情的任何代码可能并不真正可移植。本手册无法指出旧 C 宏替换实现与 ISO C 版本之间的所有差异。除标记粘贴和字符串化之外的几乎所有宏替换的使用产生的标记系列均与以前完全相同。此外，ISO C 宏替换算法可以完成在旧 C 版本中无法完成的工作。例如，

```
#define name (*name)
```

使 `name` 的任何使用均替换为通过 `name` 进行的间接引用。旧 C 预处理程序会产生大量圆括号和星号，并最终产生关于宏递归的错误。

ISO C 对宏替换方法的主要更改是：要求在替换标记列表中进行替换之前针对宏参数（而不是那些本身是宏替换操作符 `#` 和 `##` 的操作数）进行递归扩展。然而，这种更改很少在结果标记中产生实际差异。

6.5.5 使用字符串

注 – 在 ISO C 中，如果您使用 `-xtransition` 选项，则以下带有 `?` 标记的示例将生成警告。仅在转换模式（`–Xt` 和 `-Xs`）下，结果才与以前版本的 C 相同。

在 K&R C 中，以下代码生成文本字符串 `"x y!"`：

```
#define str(a) "a!"   ?
str(x y)
```

因此，预处理程序在文本字符串和字符常量中查找看起来类似宏参数的字符。ISO C 认识到此功能的重要性，但不允许对部分标记的操作。在 ISO C 中，以上宏的所有调用均生成文本字符串 `"a!"`。为在 ISO C 中实现旧效果，我们使用 `#` 宏替换操作符和文本字符串串联。

```
#define str(a) #a "!"
str(x y)
```

以上代码生成两个文本字符串 `"x y"` 和 `"!"`，这两个字符串串联后，会生成相同的 `"x y!"`。

不直接替换字符常量的类似操作。此功能的主要用法与以下类似：

```
#define CNTL(ch) (037 & 'ch')    ?
CNTL(L)
```

它生成

```
(037 & 'L')
```

求值为 ASCII control-L 字符。我们知道的最佳解决办法是将此宏的用法更改为：

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

此代码的可读性和实用性更强，因为它还可以应用于表达式。

6.5.6 标记粘贴

在 K&R C 中，将两个标记组合在一起至少有两种方法。以下代码中的两个调用均使用 `x` 和 `1` 两个标记生成单个标识符 `x1`。

```
#define self(a) a
#define glue(a,b) a/**/b ?
self(x)1
glue(x,1)
```

同样，ISO C 不认可这两种方法。在 ISO C 中，以上两个调用均生成两个独立标记 x 和 1。可以通过使用 ## 宏替换操作符针对 ISO C 重新编写以上第二种方法：

```
#define glue(a,b) a ## b
glue(x, 1)
```

只有在定义了 __STDC__ 时，才应将 # 和 ## 用作宏替换操作符。由于 ## 是实际操作符，因此对于定义和调用中的空白，调用更加自由。

编译器针对未定义的 ## 运算（C 标准，第 3.4.3 节）（其中一个 ## 结果未经定义，当进行预处理时，包含多个标记而不是一个标记（C 标准，第 6.10.3.3(3) 节））发出警告诊断。未定义的 ## 运算的结果现在定义为通过预处理连接 ## 操作数所创建的字符串而生成的第一个独立标记。

没有什么直接方式可用来实现两种旧式粘贴方案中第一种方案，但是由于它在调用时引入了粘贴的任务，因此使用它的频率比使用其他形式要低。

6.6 const 和 volatile

关键字 const 是 C++ 的特征之一，它与 ISO C 是共通的。随着 ISO C 委员会发明了关键字 volatile，“类型限定符”类别应运而生。

6.6.1 类型（仅适用于 lvalue）

const 和 volatile 属于标识符的类型，而不属于标识符的存储类。然而，当从表达式求值中获取对象的值时，确切地说是当 lvalue 变为 rvalue 时，经常会将这些类型从类型的最顶端删除。这些术语起源于原型赋值 "L=R"，其中左侧必须仍直接引用对象（一个 lvalue），右侧只需为一个值（一个 rvalue）。因此，只有本身是 lvalues 的表达式才可以由 const 和/或 volatile 限定。

6.6.2 派生类型中的类型限定符

类型限定符可修改类型名称和派生类型。派生类型是 C 声明的那些可反复应用而生成越来越复杂的类型的部分：指针、数组、函数、结构和联合。除函数之外，可使用一个或两个类型限定符更改派生类型的行为。

例如，

```
const int five = 5;
```

声明并初始化类型为 const int 并且其值未被相应的程序更改的对象。关键字的顺序对于 C 并不重要。例如，声明：

```
int const five = 5;
```

和

```
const five = 5;
```

与以上声明在效果上相同。

声明

```
const int *pci = &five;
```

声明一个类型为指向 const int 的指针的对象，该对象最初指向以前声明的对象。指针本身没有限定类型－它指向限定类型，在程序执行过程中几乎可以更改为指向任何 int。除非使用强制类型转换，否则不能使用 pci 修改它所指向的对象，如下所示：

```
*(int *)pci = 17;
```

如果 pci 实际上指向 const 对象，则此代码的行为不确定。

声明

```
extern int *const cpi;
```

表明程序中某个位置存在类型为指向 int 的 const 指针的全局对象的定义。在此情况下，cpi 的值将不会被相应的程序更改，但是可用来修改它指向的对象。请注意，在以上声明中，const 位于 * 之后。以下一对声明产生的效果相同：

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

这些声明可以合并为以下声明，其中对象的类型声明为指向 const int 的 const 指针：

```
const int *const cpci;
```

6.6.3 const 意味着 readonly

根据经验，对于关键字，readonly 优于 const。如果以此方式读取 const，则如下声明：

```
char *strcpy(char *, const char *);
```

很容易理解，即第二个参数仅用于读取字符值，而第一个参数覆写它指向的字符。此外，尽管在以上示例中，cpi 的类型是指向 const int 的指针，但您仍可以通过其他某些方法更改它指向的对象的值，除非它确实指向被声明为 const int 类型的对象。

6.6.4 const 用法示例

const 的两种主要用法是将在编译时初始化的大型信息表声明为无变化，以及指定该指针参数不修改它们所指向的对象。

第一种用法潜在允许某个程序的部分数据被同一程序的其他并行调用共享。它可能导致尝试将此不变数据修改为通过某种内存保护故障立即检测，因为该数据驻留在内存的只读部分中。

第二种用法有助于在演示期间生成内存故障之前查找潜在错误。例如，如果将指针传递给无法进行如此修改的字符串，则临时将空字符置入字符串中间的函数会在编译时被检测到.

6.6.5 volatile 意味着精确语义

到目前为止，示例均使用了 const，因为它在概念上较为简单。但是，volatile 到底意味着什么？对于编译器编写者，它只有一种含义：访问此类对象时不采用代码生成快捷方式。在 ISO C 中，声明具有相应属性及 volatile 限定类型的各个对象是程序员的责任。

6.6.6 volatile 用法示例

volatile 对象的四个常见示例为：

- 为内存映射 I/O 端口的对象
- 多个并行进程之间共享的对象
- 异步信号处理程序修改的对象
- 调用 setjmp 的函数中声明的自动存储持续时间对象，其值在 setjmp 调用和相应的 longjmp 调用之间会更改

前三个示例是具有特殊行为的对象的所有实例：在程序执行期间的任何点均可修改其值。因此，表面上的死循环：

```
flag = 1;
while (flag);
```

实际上有效，只要 flag 具有 volatile 限定类型。某些异步事件将来可能将 flag 设置为零。否则，由于 flag 的值在循环主体中保持不变，编译系统会将以上循环更改为完全忽略 flag 值的真正死循环。

第四个示例涉及调用 setjmp 的函数的局部变量，因此进一步加以讨论。关于 setjmp 和 longjmp 行为的细小字体注释表明对于符合第四种情形的对象的值没有任何保证。对于大多数所期望的行为，有必要让 longjmp 检查调用 setjmp 的函数与调用 longjmp 的函数之间的各个栈帧是否有保存的寄存器值。由于存在异步创建栈帧的可能性，使该作业更加困难。

在将自动对象声明为 volatile 限定类型时，编译系统知道生成的代码必须与程序员编写的代码完全匹配。因此，此类自动对象的最新值始终保存在内存中，而不是仅保存在寄存器中，且调用 longjmp 。

6.7 多字节字符和宽字符

最初，ISO C 的国际化仅影响库函数。但是，国际化的最终阶段（多字节字符和宽字符）还影响语言。

6.7.1 亚洲语言需要多字节字符

亚洲语言计算机环境中的根本问题是 I/O 需要大量表意文字。为了适应普通计算机体系结构，这些表意文字被编码为字节序列。相关的操作系统、应用程序和终端将这些字节序列理解为单个表意字符。此外，所有这些编码都允许将常规单字节字符与表意字符字节序列混杂在一起。识别不同表意字符的难度取决于使用的编码方案。

无论使用什么编码方案，ISO C 均将术语“多字节字符”定义为表示为表意字符编码的字节序列。所有多字节字符都是“扩展字符集”的成员。常规的单字节字符仅仅是多字节字符的特殊情形。对编码的唯一要求是多字节字符不能将空字符用作它的编码的一部分。

ISO C 指定程序注释、文本字符串、字符常量和头文件名均为多字节字符序列。

6.7.2 编码变种

编码方案分为两种。第一种方案是，每个多字节字符都是自标识的，即，可以在任何多字节字符对之间插入任何多字节字符。

第二种方案是，特殊的移位字节的存在会更改后续字节的解释。一个示例是，某些字符终端进入和退出行绘制模式所用的方法。对于使用与移位状态相关的编码以多字节字符编写的程序，ISO C 要求每个注释、文本字符串、字符常量和头文件名称都必须以未移位状态开始和结束。

6.7.3 宽字符

如果所有字符的字节数或位数都相同，则会消除处理多字节字符的一些不便之处。由于在这样的字符集中可能存在成千上万的表意字符，因此应使用 16 位或 32 位大小的整数值容纳所有成员。（整个中文字母表包含的表意字符超过 65,000 个！）ISO C 包括 typedef 名称 wchar_t，将其作为大得足以容纳扩展字符集的所有成员的实现定义整数类型。

对于每个宽字符，都存在对应的多字节字符，反之亦然；必须具有对应于常规单字节字符的宽字符，才能具有与其单字节值相同的值，包括空字符。但是，并不保证宏 EOF 的值可以存储在 wchar_t 中，因为 EOF 可能无法表示为 char。

6.7.4 转换函数

1990 ISO/IEC C 标准提供了五个管理多字节字符和宽字符的库函数，1999 ISO/IEC C 标准提供了更多此类函数。

6.7.5 C 语言特征

为了给亚洲语言环境中的程序员带来更大的灵活性，ISO C 提供了宽字符常量和宽文本字符串。它们具有与其非宽版本相同的形式，但位置是紧邻字母 L 之后：

- 'x' 常规字符常量
- '¥' 常规字符常量
- L'x' 宽字符常量
- L'¥' 宽字符常量
- "abc¥xyz" 常规文本字符串
- L"abcxyz" 宽文本字符串

在常规版本和宽版本中，多字节字符均有效。生成表意字符 ¥ 所必需的字节序列与编码有关，但是如果它由多个字节组成，则字符常量 '¥' 的值是实现定义的，正如 'ab' 的值是实现定义的一样。除了换码序列之外，常规文本字符串包含引号之间指定的字节，包括每个指定的多字节字符的字节。

当编译系统遇到宽字符常量或宽文本字符串时，每个多字节字符都将转换为宽字符，如同调用了 mbtowc() 函数一样。因此，L'¥' 的类型为 wchar_t；abc¥xyz 的类型为八位数组 wchar_t。正如常规文本字符串那样，每个宽文本字符串都附加有额外的零值元素，但是在这些情况下，它是值为零的 wchar_t。

正如常规文本字符串可用作字符数组初始化的快捷方法，宽文本字符串可用于初始化 wchar_t 数组：

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

在以上示例中，x、y 和 z 这三个数组以及 wp 指向的数组具有相同长度。所有数组均使用相同的值进行初始化。

最后，正如常规文本字符串一样，串联相邻宽文本字符串。但是，对于 1990 ISO/IEC C 标准，相邻常规文本字符串和宽文本字符串会产生不确定的行为。此外，1990 ISO/IEC C 标准还指定如果编译器不接受此类串联，也不必生成错误。

6.8 标准头文件和保留名称

在标准化过程早期，ISO 标准委员会选择包含库函数、宏和头文件作为 ISO C 的一部分。

本节介绍各种保留名称及有关其保留的某些基本原理。最后是一系列规则，遵循这些规则可使程序扫清任何保留名称。

6.8.1 标准头文件

标准头文件如下：

表 6–2 标准头文件

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

大多数实现提供更多头文件，但是严格符合 1990 ISO/IEC 标准的 C 程序只能使用这些头文件。

关于其中某些头文件的内容，其他标准稍有不同。例如，POSIX (IEEE 1003.1) 指定 `fdopen` 在 `stdio.h` 中声明。为了允许这两种标准共存，POSIX 要求在包含任何头文件之前对宏 `_POSIX_SOURCE` 进行 `#defined`，以保证这些附加名称存在。在其可移植性指南中，X/Open 对其扩展也使用这种宏方案。X/Open 的宏是 `_XOPEN_SOURCE`。

ISO C 要求标准头文件同时是自给自足和幂等的。标准头文件之前或之后不需要任何其他头文件进行 `#included`，并且每个标准头文件可多次进行 `#included` 而不会导致问题。该标准还要求它的头文件只能在安全上下文中进行 `#included`，以便保证头文件中使用的名称保持不变。

6.8.2 保留供实现使用的名称

标准进一步限制与其库相关的实现。以前，大多数程序员认为不应该在 UNIX 系统上对他们自己的函数使用 `read` 和 `write` 等名称。ISO C 要求实现中的引用仅引入该标准保留的名称。

因此，该标准保留所有可能名称的子集供实现使用。此类名称由标识符组成，标识符以下划线开头，后面是其他下划线或大写字母。此类名称包含与以下常规表达式匹配的所有名称：

_[_A-Z][0-9_a-zA-Z]*

严格地说，如果程序使用此标识符，其行为不确定。因此，使用 _POSIX_SOURCE（或 _XOPEN_SOURCE）的程序具有不确定的行为。

但是，不确定的行为具有不同的程度。如果在符合 POSIX 标准的实现中使用 _POSIX_SOURCE，则您知道程序的不确定行为包括某些头文件中的某些附加名称，并且该程序仍符合公认的标准。ISO C 标准中的预留漏洞允许实现符合表面上不兼容的规范。另一方面，当遇到 _POSIX_SOURCE 等名称时，不符合 POSIX 标准的实现按任意方式执行。

该标准还保留以下划线开头的所有其他名称，以用于作为常规文件作用域标识符以及作为结构和联合的标记的头文件，但不用于局部作用域。允许以下公共实践：让命名为 _filbuf 和 _doprnt 的函数实现库的隐藏部分。

6.8.3 保留供扩展使用的名称

除了显式保留的所有名称之外，1990 ISO/IEC C 标准还保留（供实现和将来标准使用）与某些模式匹配的名称：

表 6–3 保留供扩展使用的名称

文件	保留名称模式
errno.h	E[0-9A-Z].*
ctype.h	(to|is)[a-z].*
locale.h	LC_[A-Z].*
math.h	当前函数名称 [fl]
signal.h	(SIG|SIG_)[A-Z].*
stdlib.h	str[a-z].*
string.h	(str|mem|wcs)[a-z].*

在以上列表中，只有在包含相关头文件时，以大写字母开头的名称才是宏并被保留。其余名称可指定函数，不能用于为任何全局对象或函数命名。

6.8.4 可安全使用的名称

可以遵循以下四个简单规则以避免与任何 ISO C 保留名称冲突：

- #include 源文件顶部的所有系统头文件（除非可能在 _POSIX_SOURCE 和/或 _XOPEN_SOURCE 的 #define 之后）。

- 不要定义或声明以下划线开头的任何名称。
- 在所有文件作用域标记和常规名称的前几个字符内的某位置使用下划线或大写字母。请注意 stdarg.h 或 varargs.h 中的 va_ 前缀。
- 在所有宏名称的前几个字符内的某个位置使用数字或非大写字母。如果 errno.h 为 #included，则保留几乎所有以 E 开头的名称。

这些规则仅仅是要遵循的一般准则，缺省情况下大多数实现将继续向标准头文件增加名称。

6.9 国际化

第 140 页中的 “6.7 多字节字符和宽字符”介绍了标准库的国际化。本节讨论受影响的库函数，并提供一些关于应如何编写程序以便利用这些功能的提示。本节只讨论关于 1990 ISO/IEC C 标准的国际化。1999 ISO/IEC C 标准并未进行重大扩展以支持高于此处讨论的国际化。

6.9.1 语言环境

任何时候，C 程序都有当前语言环境－描述适合于某个民族、文化和语言的惯例的信息集合。语言环境具有字符串名称。唯一的两个标准化语言环境名称为 "C" 和 ""。每个程序在 "C" 语言环境中启动，这导致所有库函数像过去一样工作。"" 语言环境是实现在选择适合程序的调用的正确惯例集时的首选。"C" 和 "" 可导致相同的行为。实现可能提供其他语言环境。

为了实用和方便，语言环境被划分为一系列种类。程序可更改整个语言环境，或者只更改一个或多个种类。通常，每个种类影响与受其他种类影响的函数分开的函数集，因此可以临时更改一个种类。

6.9.2 setlocale() 函数

setlocale() 函数是指向程序语言环境的接口。通常，使用调用国家惯例的任何程序在程序执行路径的开头部分应发出一个调用，如：

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

。该调用导致程序的当前语言环境更改为相应的本地版本，因为 LC_ALL 是指定整个语言环境而不是某个种类的宏。以下是标准种类：

LC_COLLATE	排序信息
LC_CTYPE	字符分类信息
LC_MONETARY	货币输出信息
LC_NUMERIC	数值输出信息
LC_TIME	日期和时间输出信息

这些宏中的任何宏均可作为 setlocale() 的第一个参数传递以指定该种类。

setlocale() 函数返回给定种类的当前语言环境的名称（或 LC_ALL），当其第二个参数为空指针时，它仅用于查询。因此，如下代码可用于在有限持续时间内更改语言环境或其中一部分：

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_category, NULL);
if (setlocale(LC_category, "new") != 0)
{
        /* use temporarily changed locale */
    (void)setlocale(LC_category, oloc);
}
```

大多数程序不需要此功能。

6.9.3 更改的函数

只要可能并且适当，就会将现有库函数扩展为包括与语言环境相关的行为。这些函数分为两组：

- ctype.h 头文件声明的函数（字符分类和转换），以及
- 转换为和转换自数值的可输出形式和内部形式的函数，如 printf() 和 strtod()。

对于附加字符，如果当前语言环境的 LC_CTYPE 种类不是 "C"，则除 isdigit() 和 isxdigit() 之外的所有 ctype.h 判定函数都可返回非零（真）值。在西班牙语言环境中，isalpha('ñ') 应为真。同样，字符转换函数 tolower() 和 toupper() 应相应地处理 isalpha() 函数标识的任何额外字母字符。ctype.h 函数通常是使用由字符参数索引的查表而实现的宏。通过将表重新设置为新语言环境的值可更改这些函数的行为，因此没有性能影响。

当前语言环境的 LC_NUMERIC 种类不是 "C" 时，写入或解释可输出浮点值的那些函数可以更改为使用非句点 (.) 的小数点字符。不存在将任何数值转换为包含千位分隔符类型字符的可输出形式的规定。从可输出形式转换为内部形式时，允许实现接受此类附加形式，同样是在非 "C" 语言环境中。使用小数点字符的函数是 printf() 和 scanf() 系列、atof() 以及 strtod()。允许实现定义的扩展的函数是 atof()、atoi()、atol()、strtod()、strtol()、strtoul() 和 scanf() 系列。

6.9.4 新函数

某些语言环境相关的功能是以新标准函数的形式添加的。除了 `setlocale()`（它允许控制语言环境本身）之外，该标准还包括以下新函数：

`localeconv()`	数值/货币转换
`strcoll()`	两个字符串的整理顺序
`strxfrm()`	转换字符串以便整理
`strftime()`	设置日期和时间的格式

此外，还有多字节函数 `mblen()`、`mbtowc()`、`mbstowcs()`、`wctomb()` 和 `wcstombs()`。

`localeconv()` 函数返回一个指针，该指针指向包含对设置数值格式有用的信息以及适合当前语言环境的 `LC_NUMERIC` 和 `LC_MONETARY` 种类的货币信息的结构。这是唯一的一个其行为依赖于多个种类的函数。对于数值，结构描述小数点字符、千位分隔符和分隔符应在的位置。有十五个描述如何格式化货币值的其他结构成员。

`strcoll()` 函数类似于 `strcmp()` 函数，只是它根据当前语言环境的 `LC_COLLATE` 种类比较两个字符串。`strxfrm()` 函数也可用于将一个字符串转换为另一个字符串，以便任何两个此类转换后字符串均可以传递到 `strcmp()`，并且可获得与 `strcoll()` 传递两个预转换字符串时返回的排序类似的排序。

`strftime()` 函数提供与 `sprintf()` 对 `struct tm` 中的值使用的格式设置类似的格式设置，并提供依赖当前语言环境的 `LC_TIME` 种类的某些日期和时间表示。此函数基于 `ascftime()` 函数，后者作为 UNIX System V 发行版 3.2 的一部分发行。

6.10 表达式中的分组和求值

Dennis Ritchie 在设计 C 时所作的选择之一是为编译器提供一个许可证，以便重新整理包含算术上可交换并且关联的相邻操作符（甚至出现圆括号）的表达式。这在 Kernighan 和 Ritchie 合著的《*The C Programming Language*》的附录中已明确指出。但是，ISO C 没有给编译器同样的自由。

本节通过考虑以下代码片段中的表达式语句，讨论这两个 C 定义之间的差异，并阐明表达式的副作用、分组以及求值之间的差别。

```
int i, *p, f(void), g(void);
/*...*/
i = *++p + f() + g();
```

6.10.1 定义

表达式的副作用是修改内存并访问 volatile 限定对象。以上表达式的副作用是更新 i 和 p 以及函数 f() 和 g() 内包含的任何副作用。

表达式的分组是值与其他值和运算符相结合的一种方式。以上表达式的分组主要是加法的执行顺序。

表达式的求值包括生成结果值所必需的所有运算。要对表达式求值，所有指定的副作用必须在上下两个序列点之间发生，并且使用特定的分组执行指定的操作。对于以上表达式，必须在前一个语句之后和该表达式语句的 ; 之前更新 i 和 p；函数调用可以在前一个语句之后使用它们的返回值之前的任何时候，按任何顺序发生。特别地，在使用操作的值之前，导致内存更新的操作符不需要分配新值。

6.10.2 K&R C 重新整理许可证

由于加法在算术上可交换并且关联，因此 K&R C 重新整理许可证适用于以上表达式。为了区别常规圆括号和表达式的实际分组，左、右花括号指定分组。表达式的三种可能的分组为：

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

给定 K&R C 规则，所有这些分组均有效。此外，即使表达式是按以下任意方式编写的，所有这些分组仍有效：

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

如果在溢出导致异常的体系结构上对该表达式求值，或者加法和减法在溢出时不是互逆运算，则当一个加法运算溢出时，这三种分组表现不同。

对于这些体系结构上的此类表达式，K&R C 中唯一可用的求助措施是分割表达式以强制进行特定的分组。以下是分别强制执行以上三种分组的可能重写：

```
i = *++p; i += f(); i += g()
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

6.10.3 ISO C 规则

对于算术上可交换并且关联但实际上在目标体系结构上并非如此的操作，ISO C 不允许进行重新整理。因此，ISO C 语法的优先级和关联性完整描述了所有表达式的分组；所有表达式在进行语法分析时必须进行分组。所考虑的表达式按以下方式分组：

```
i = { {*++p + f()} + g() };
```

此代码仍不表示必须在 g() 之前调用 f()，也不表示必须在调用 g() 之前增大 p。

在 ISO C 中，不需要为了避免意外的溢出而分割表达式。

6.10.4 圆括号

由于不完全的理解或不准确的表示，ISO C 经常被错误地描述为支持圆括号或根据圆括号求值。

由于 ISO C 表达式仅仅具有语法分析指定的分组，因此圆括号仍然仅用作控制表达式语法分析方式的方法；表达式的自然优先级和关联性与圆括号同等重要。

以上表达式可写为：

```
i = (((*(++p)) + f()) + g());
```

对其分组或求值没有不同影响。

6.10.5 As If 规则

使用 K&R C 重新整理规则的几个理由：

- 重新整理为优化提供了更多的机会，如编译时常量折叠。
- 在大多数机器上，重新整理不会更改整型表达式的结果。
- 在所有机器上，一些操作在算术上和计算上可交换并且关联。

ISO C 委员会最终承认：重新整理规则应用于所描述的目标体系结构时，本来是要作为 *as if* 规则的一个实例。ISO C 的 *as if* 规则是通用许可证，它允许实现任意偏离抽象机器描述，只要偏离不更改有效 C 程序的行为。

因此，由于无法通知此类重新分组，因此允许在任何机器上重新整理所有二元按位运算符（移位除外）。在溢出回绕的典型二进制补码机器上，可以由于相同原因重新整理涉及乘法或加法的整型表达式。

因此，C 中的此更改对大多数 C 程序员没有重大影响。

6.11 不完全类型

ISO C 标准引入术语“不完全类型”使 C 的基本（但容易造成误解）部分形式化，这种类型的开头具有某种暗示。本节描述不完全类型、其允许位置以及它们有用的原因。

6.11.1 类型

ISO 将 C 的类型分为三个不同的集合：函数、对象和不完全。函数类型很明显；对象类型包含其他一切，除非不知道对象的大小。该标准使用术语“对象类型”指定指派的对象必须具有已知大小，但是除 `void` 之外的不完全类型也称为对象，知道这一点很重要。

不完全类型有三种不同形式：`void`、未指定长度的数组以及具有非指定内容的结构和联合。`void` 类型与其他两种类型不同，因为它是无法完成的不完全类型，并且它用作特殊函数返回和参数类型。

6.11.2 完成不完全类型

通过在表示相同对象的相同作用域中的后面声明中指定数组大小，可完成数组类型。当声明并在相同声明中初始化不具有大小的数组时，仅在其声明符的末尾与其初始化函数的末尾之间，数组才具有不完全类型。

通过在具有相同标记的相同作用域中的后面声明中指定内容，可完成不完全结构或联合类型。

6.11.3 声明

某些声明可使用不完全类型，但是其他声明需要完全对象类型。需要对象类型的声明是数组元素、结构或联合的成员以及函数的局部对象。所有其他声明允许不完全类型。特别地，允许下列构造：

- 指向不完全类型的指针
- 返回不完全类型的函数
- 不完全函数参数类型
- 不完全类型的 `typedef` 名称

函数返回和参数类型特殊。除 `void` 之外，在定义或调用函数之前，必须完成以这种方式使用的不完全类型。返回类型 `void` 指定不返回值的函数，单个参数类型 `void` 指定不接受参数的函数。

由于数组和函数的参数类型重写为指针类型，因此表面上不完全的数组参数类型实际上并非不完全。`main` 的 `argv` 的典型声明（即 `char *argv[]`，一个未指定长度的字符指针数组）重写为指向字符指针的指针。

6.11.4 表达式

大多数表达式运算符需要完全对象类型。仅有的三个例外是一元运算符 `&`、逗号运算符的第一个操作数以及 `?:` 运算符的第二个和第三个操作数。除非需要指针运算，否则接受指针操作数的大多数运算符也允许指向不完全类型的指针。该列表包含一元运算符 `*`。例如，给定：

```
void *p
```

`&*p` 是使用该声明的有效子表达式。

6.11.5 正当理由

为什么不完全类型是必要的？在忽略 `void` 的情况下，只有一个由不完全类型提供的功能是 C 无法以其他方式处理的，而必须利用对结构和联合的正向引用。如果两个结构需要相互指向的指针，则唯一的方法是使用不完全类型：

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

具有某种形式的指针以及异构数据类型的所有强类型编程语言提供处理这种情形的某些方法。

6.11.6 示例

为不完全结构和联合类型定义 `typedef` 名称通常很有用。如果您有一系列包含许多相互指向的指针的复杂数据结构，结构前面有一个 `typedef` 列表（可能在中央头文件中），则可以简化声明。

```
typedef struct item_tag Item;
typedef union note_tag Note;
typedef struct list_tag List;
.  .  .
struct item_tag { .  .  .  };
.  .  .
struct list_tag {
    struct list_tag {
};
```

此外，对于其内容不应该用于程序其余部分的结构和联合，头文件可以声明不带该内容的标记。程序的其他部分可以使用指向不完全结构或联合的指针而不会出现任何问题，除非它们尝试使用它的任何成员。

频繁使用的不完全类型是未指定长度的外部数组。通常，要使用某个数组的内容，无需知道该数组的范围。

6.12 兼容类型和复合类型

对于 K&R C，引用同一实体的两个声明可能是不同的；对于 ISO C 更是如此。ISO C 中使用的术语“兼容类型”表示“足够接近”的类型。本节描述兼容类型和“复合类型”（合并两种兼容类型而产生的结果）。

6.12.1 多个声明

如果只允许 C 程序声明每个对象或函数一次，则不需要兼容类型。链接（允许两个或更多声明引用相同实体）、函数原型和分别编译全部需要此功能。独立转换单元（源文件）具有与单个转换单元不同的类型兼容性规则。

6.12.2 分别编译兼容性

由于每个编译可能查看不同的源文件，因此独立编译中的大多数兼容类型规则实质上是结构化的：

- 匹配标量（整型、浮点和指针）类型必须兼容，如同它们在相同的源文件中一样。
- 匹配结构、联合和枚举必须具有相同数目的成员。每个匹配成员都必须具有兼容类型（从单独编译的意义上讲），包括位字段宽度。
- 匹配结构必须具有相同顺序的成员。联合和枚举成员的顺序并不重要。
- 匹配枚举成员必须具有相同的值。

  附加要求是，对于结构、联合和枚举，成员的名称（包括缺少未命名成员的名称）必须匹配，但是它们各自的标记不必匹配。

6.12.3 单编译兼容性

当相同作用域内的两个声明描述相同的对象或函数时，这两个声明必须指定兼容类型。然后这两种类型合并为与这两种类型兼容的单个复合类型。后面将详细讨论复合类型。

兼容类型是递归定义的。底部为类型说明符关键字。规则规定，`unsigned short` 与 `unsigned short int` 相同，不带类型说明符的类型与带有 `int` 的类型相同。所有其他类型仅当派生它们的类型兼容时才为兼容类型。例如，如果限定符 `const` 和 `volatile` 是相同的，且未限定基类型是兼容的，则两个限定类型是兼容的。

6.12.4 兼容指针类型

要使两种指针类型兼容，它们指向的类型必须兼容，并且必须对这两个指针进行相同的限定。考虑到指针的限定符在 * 之后指定，因此以下两个声明

```
int *const cpi;
int *volatile vpi;
```

声明指向相同类型 int 的两个以不同方式限定的指针。

6.12.5 兼容数组类型

要使两个数组类型兼容，它们的元素类型必须兼容。如果两个数组类型具有指定的大小，则它们必须匹配，即，不完全数组类型（请参见第 149 页中的 “6.11 不完全类型”）同时与另一不完全数组类型和一个具有指定大小的数组类型兼容。

6.12.6 兼容函数类型

要使函数兼容，请遵守以下规则：

- 要使两个函数类型兼容，它们的返回类型必须兼容。如果其中一个或两个函数类型均具有原型，则规则更加复杂。
- 为了使具有原型的两个函数类型兼容，它们还必须具有相同数目的参数，包括省略号 (...) 的使用，而且对应参数必须是参数兼容的。
- 为了使旧式函数定义与具有原型的函数类型兼容，原型参数**不得**以省略号 (...) 结尾。应用缺省参数提升后，每个原型参数与对应的旧式参数必须是参数兼容的。
- 为了使旧式函数声明（而不是定义）与具有原型的函数类型兼容，原型参数不得以省略号 (...) 结尾。所有原型参数的类型必须不受缺省参数提升的影响。
- 为了使两种类型是参数兼容的，在删除顶级限定符（如果有）后，以及将函数或数组类型转换为相应的指针类型后，这两种类型必须是兼容的。

6.12.7 特殊情况

signed int 的行为与 int 相同，不同之处可能在于位字段，其中无格式 int 可能表示无符号的值。

另一点值得注意的是，每个枚举类型必须与某些整数类型兼容。对于可移植的程序，这意味着枚举类型是独立类型。通常，ISO C 标准将枚举类型视为独立类型。

6.12.8 复合类型

由两个兼容类型构成的复合类型也是递归定义的。兼容类型可能彼此不同的原因在于不完全数组或旧式函数类型。因此，复合类型最简单的描述是，它是与两个原始类型均兼容的类型，包括原始类型中的各个可用数组大小和各个可用参数列表。

◆ ◆ ◆ 第 7 章

7 转换应用程序以适用于 64 位环境

本章提供为 32 位或 64 位编译环境编写代码所需的信息。

尝试为 32 位和 64 位编译环境编写或修改代码时，会面临下列两个基本问题：

- 不同数据类型模型之间的数据类型一致性
- 使用不同数据类型模型的应用程序之间的交互

维护包含尽可能少的 `#ifdefs` 的单个源代码通常比维护多个源代码树更好。因此，本章提供了一些指导性信息，用于指导如何编写在 32 位和 64 位编译环境中都能正确运行的代码。在某些情况下，转换当前代码只需重新编译以及与 64 位库重新链接。但是，对于需要更改代码的情况，本章讨论使转换更容易的工具和策略。

7.1 数据模型差异概述

32 位和 64 位编译环境之间的最大差异是数据类型模型的更改。

32 位应用程序的 C 数据类型模型是 ILP32 模型，之所以这样命名是因为整型、长型和指针是 32 位数据类型。LP64 数据模型（之所以这样命名是因为长型和指针增长为 64 位）由业界各公司联合创建。其余 C 类型 `int`、`long long`、`short` 和 `char` 在这两种数据类型模型中相同。

无论数据类型模型如何，C 整型间的标准关系始终为真：

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
```

下表列出基本 C 数据类型及其对于 ILP32 和 LP64 数据模型的相应长度（位数）。

表 7–1 ILP32 和 LP64 的数据类型长度

C 数据类型	LP32	LP64
`char`	8	8

表 7–1 ILP32 和 LP64 的数据类型长度 （续）

C 数据类型	LP32	LP64
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64
enum	32	32
float	32	32
double	64	64
long double	128	128

当前 32 位应用程序通常假设整型、指针和长型的长度相同。由于长型和指针的长度在 LP64 数据模型中更改，因此您需要注意，单是这种更改就可以导致许多 ILP32 至 LP64 转换问题。

此外，检查声明和强制类型转换变得很重要；类型更改时，表达式的求值方式会受到影响。标准 C 转换规则的效果受数据类型长度更改的影响。要充分表示您的意图，您需要显式声明常量的类型。您也可以在表达式中使用强制类型转换，以确保表达式按您想要的方式求值。特别是在符号扩展的情况下，显式强制类型转换对说明意图至关重要，此时这样做更有必要。

7.2 实现单一源代码

以下各节介绍可用于编写支持 32 位和 64 位编译的单一源代码的一些可用资源。

7.2.1 派生类型

使用系统派生类型使代码对于 32 位和 64 位编译环境均安全。通常，使用派生类型以适应更改是良好的编程实践。使用派生数据类型时，只有系统派生类型由于数据模型更改或由于某个端口而需要更改。

系统 include 文件 <sys/types.h> 和 <inttypes.h> 包含有助于使应用程序对于 32 位和 64 位编译环境均安全的常量、宏和派生类型。

7.2.1.1 <sys/types.h>

在应用程序源文件中包含 <sys/types.h> 以访问 _LP64 和 _ILP32 的定义。此头文件还包含适当时应使用的多个基本派生类型。尤其是以下类型更为重要：

- clock_t 表示系统时间（以时钟周期为单位）。
- dev_t 用于设备号。
- off_t 用于文件大小和偏移量。
- ptrdiff_t 是一种带符号整型，用于对两个指针执行减法运算后所得的结果。
- size_t 反映内存中对象的大小（以字节为单位）。
- ssize_t 供返回字节计数或错误提示的函数使用。
- time_t 以秒为单位计时。

所有这些类型在 ILP32 编译环境中保持为 32 位值，并会在 LP64 编译环境中增长为 64 位值。

7.2.1.2 <inttypes.h>

include 文件 <inttypes.h> 提供有助于使代码与显式指定大小的数据项兼容（无论编译环境如何）的常量、宏和派生类型。它包含用于处理 8 位、16 位、32 位和 64 位对象的机制。该文件是新的 1999 ISO/IEC C 标准的一部分，文件内容反映了导致它包含在 1999 ISO/IEC C 标准中的建议。文件即将更新，以便完全与 1999 ISO/IEC C 标准一致。<inttypes.h> 提供的基本功能包括：

- 定宽整型
- 诸如 uintptr_t 的有用类型
- 常量宏
- 限制
- 格式字符串宏

以下各节提供有关 <inttypes.h> 基本功能的更多信息。

定宽整型

<inttypes.h> 提供的定宽整型包括带符号整型（如 int8_t、int16_t、int32_t、int64_t）和无符号整型（如 uint8_t、uint16_t、uint32_t、uint64_t）。

定义为可容纳规定位数的最短整型的派生类型包括 int_least8_t、int_least64_t、uint_least8_t、uint_least64_t 等。

对于循环计数器和文件描述符等操作，使用 int 或无符号 int 是安全的；对于数组索引，使用 long 也是安全的。但是，不应不加选择地使用这些定宽类型。可将定宽类型用于下列各项的显式二进制表示：

- 磁盘数据
- 通过数据线
- 硬件寄存器

- 二进制接口规范
- 二进制数据结构

诸如 unintptr_t 的有用类型

<inttypes.h> 文件包括大小足以容纳一个指针的带符号整型和无符号整型。这些类型以 intptr_t 和 uintptr_t 形式提供。此外，<inttypes.h> 还提供 intmax_t 和 uintmax_t，后两者是可用的最长（以位为单位）带符号整型和无符号整型。

使用 uintptr_t 类型作为指针的整型而非基本类型，如无符号 long。尽管在 ILP32 和 LP64 数据模型中，无符号 long 与指针的长度相同，但如果使用 uintptr_t，则在数据模型更改时，只有 uintptr_t 的定义受影响。这使您的代码可移植到许多其他系统中。它也是在 C 中更清楚地表达意图的方式。

需要执行地址运算时，intptr_t 和 uintptr_t 类型对于强制转换指针非常有用。因此，应使用 intptr_t 和 uintptr_t 类型，而不是 long 或无符号 long。

常量宏

使用宏 INT8_C(c)、INT64_C(c)、UINT8_C(c)、UINT64_C(c) 等指定给定常量的大小和符号。基本上，必要时这些宏会在常量的末尾添上 l、ul、ll 或 ull。例如，对于 ILP32，INT64_C(1) 会在常量 1 后面附加 ll；对于 LP64，则附加 l。

可使用 INTMAX_C(c) 和 UINTMAX_C(c) 宏使常量成为最长类型。这些宏对于指定第 158 页中的 “7.3 转换为 LP64 数据类型模型”中介绍的常量类型会非常有用。

限制

由 <inttypes.h> 定义的限制是用于指定各种整型的最小值和最大值的常量，其中包括每个定宽类型的最小值（如 INT8_MIN、INT64_MIN 等）和最大值（如 INT8_MAX、INT64_MAX 等）及其对应的无符号的最小值和最大值。

<inttypes.h> 文件还提供每个最短长度类型的最小值和最大值，其中包括 INT_LEAST8_MIN、INT_LEAST64_MIN、INT_LEAST8_MAX、INT_LEAST64_MAX 等及其对应的无符号的最小值和最大值。

最后，<inttypes.h> 还定义支持的最长整型的最小值和最大值，其中包括 INTMAX_MIN 和 INTMAX_MAX 及其对应的无符号的最小值和最大值。

格式字符串宏

<inttypes.h> 文件还包括指定 printf(3S) 和 scanf(3S) 格式说明符的宏。实质上，如果宏名称内置了参数的位数，这些宏将在格式说明符前面添加 l 或 ll，以便将参数标识为 long 或 long long。

printf(3S) 的宏以十进制、八进制、无符号和十六进制格式输出最短和最长整型，如以下示例所示：

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

同样，scanf(3S) 的宏以十进制、八进制、无符号和十六进制格式读取最短和最长整型。

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

不要不加区别地使用这些宏。最好将它们与第 155 页中的 “定宽整型”中介绍的定宽类型一起使用。

7.2.2 工具

lint 程序的 -errchk 选项检测潜在的 64 位端口问题。也可以指定 cc -v，该选项指示编译器执行更严格的附加语义检查（与不使用 -v 进行编译相比）。-v 选项还会针对指定文件启用某些类似 lint 的检查。

将代码增强到 64 位安全时，应使用 Solaris 操作系统中出现的头文件，因为这些文件具有 64 位编译环境的派生类型和数据结构的正确定义。

7.2.2.1 lint

使用 lint 检查为 32 位和 64 位编译环境编写的代码。指定 -errchk=longptr64 选项以生成 LP64 警告。同时使用 -errchk=longptr64 标志来检查是否可将代码移植到下述环境中：长整型和指针的长度为 64 位而无格式整型的长度为 32 位。即使使用了显式强制类型转换，-errchk=longptr64 标志也会检查指针表达式和长整型表达式对无格式整型的赋值。

使用 -errchk=longptr64,signext 选项查找符合以下条件的代码：其中标准 ISO C 值保留规则允许在无符号整型表达式中使用带符号整型值的符号扩展。

如果只想检查要在 Solaris 64 位编译环境中运行的代码，请使用 lint 的 -m64 选项。

当 lint 生成警告时，它将输出错误代码的行号、描述问题的消息以及是否涉及指针。警告消息还指明涉及的数据类型的长度。如果确定涉及指针并且知道数据类型的长度，便可以查找特定的 64 位问题，并避免 32 位和更短类型之间的已有问题。

但请注意，尽管 lint 会提供有关潜在 64 位问题的警告，但也无法检测所有问题。另外在许多情况下，符合应用程序意图且正确无误的代码会生成警告。

通过在上一行中放置 "NOTE(LINTED("<optional message>))" 形式的注释，可以禁止对指定代码行发出警告。希望 lint 忽略某些代码行（如强制类型转换和赋值）时，这种做法会很有用。使用 "NOTE(LINTED("<optional message>))" 注释时请务必谨慎，因为它可能会掩盖真实问题。使用 NOTE 时，请包含 #include<note.h>。有关更多信息，请参阅 lint 手册页。

7.3 转换为 LP64 数据类型模型

以下示例说明在转换代码时可能遇到的某些常见问题。适当时会显示相应的 lint 警告。

7.3.1 整型和指针长度更改

由于整型和指针在 ILP32 编译环境中长度相同，因此某些代码依赖以下假定。通常会将指针强制转换为 int 或 unsigned int 以进行地址运算。但是，由于 long 和指针在 ILP32 和 LP64 数据类型模型中长度相同，因此可以将指针强制转换为 long。请使用 uintptr_t 而不是显式使用 unsigned long，因为前者可更贴切地表达意图并使代码具有更强的可移植性，从而使其不会受到将来变化的影响。请看以下示例：

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
%
warning: conversion of pointer loses bits
```

下面是修改的版本：

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

7.3.2 整型和长型长度更改

由于整型和长型在 ILP32 数据类型模型中从未真正加以区分，因此现有代码可能会不加区分地使用它们。修改交换使用整型和长型的任何代码，使其可同时符合 ILP32 和 LP64 数据类型模型的要求。整型和长型在 ILP32 数据类型模型中均为 32 位，而长型在 LP64 数据类型模型中为 64 位。

请看以下示例：

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;

%
warning: assignment of 64-bit integer to 32-bit integer
```

此外，与 int 或 unsigned int 数组相比，大型整型数组（如 long 或 unsigned long）可能会导致 LP64 数据类型模型中的性能显著下降。大型的 long 或 unsigned long 数组还可能会导致显著增加缓存未命中的情况，并占用更多的内存。

因此，如果对于应用程序而言 int 与 long 的效果一样好，最好使用 int，而不要使用 long。

也是出于这种原因，使用 int 数组而不要使用指针数组。某些 C 应用程序在转换为 LP64 数据类型模型后出现显著的性能下降，这是因为它们依赖于很多较大的指针数组。

7.3.3 符号扩展

转换到 64 位编译环境时，经常会遇到符号扩展问题，这是因为类型转换和提升规则有些模糊。为防止出现符号扩展问题，请使用显式强制类型转换以取得预期结果。

要了解出现符号扩展的原因，了解 ISO C 的转换规则会有所帮助。可能会导致 32 位和 64 位编译环境之间大多数符号扩展问题的转换规则在以下操作过程中有效：

- 整型提升

  无论有无符号，均可在调用整型的任何表达式中使用 `char`、`short`、枚举类型或位字段。

  如果一个整型可以容纳初始类型的所有可能值，则值转换为整型；否则，值转换为无符号整型数。

- 带符号整型数和无符号整型数之间的转换

  当一个带负号的整数被提升为同一类型或更长类型的无符号整型数时，它首先被提升为更长类型的带符号等价值，然后转换为无符号值。

以下示例被编译为 64 位程序时，即使 `addr` 和 `a.base` 均为无符号类型，`addr` 变量仍可成为带符号扩展变量。

```
%cat test.c
struct foo {
unsigned int base:19, rehash:13;
};

main(int argc, char *argv[])
{
  struct foo a;
  unsigned long addr;

  a.base = 0x40000;
  addr = a.base << 13;  /* Sign extension here! */
  printf("addr 0x%lx\n", addr);

 addr = (unsigned int)(a.base << 13); /* No sign extension here! */
 printf("addr 0x%lx\n", addr);
}
```

发生此符号扩展的原因是按以下方式应用了转换规则：

- 由于整型提升规则，`a.base` 将从无符号 `int` 转换为 `int`。因此，表达式 `a.base << 13` 的类型为 `int`，但是未发生符号扩展。
- 表达式 `a.base << 13` 的类型为 `int`，但是在赋值给 `addr` 之前，由于带符号和无符号整型数提升规则，会转换为 `long`，然后转换为无符号 `long`。从 `int` 转换为 `long` 时，会发生符号扩展。

```
% cc -o test64 -xarch=v9 test.c
% ./test64
addr 0xffffffff80000000
addr 0x80000000
%
```

如果将同一示例编译为 32 位程序，则不显示任何符号扩展：

```
cc -o test test.c
%test

addr 0x80000000
addr 0x80000000
```

有关转换规则的详细讨论，请参见 ISO C 标准。此标准中还包含对普通算术转换和整型常量有用的规则。

7.3.4 指针运算而不是整数

通常，由于指针运算独立于数据模型，而整数不可以，因此使用指针运算比整数好。此外，通常可以使用指针运算简化代码。请看以下示例：

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);

%
warning: conversion of pointer loses bits
```

下面是修改的版本：

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

7.3.5 结构

检查应用程序中的内部数据结构有无漏洞。在结构中的字段之间使用额外填充，以满足对齐要求。对于 LP64 数据类型模型，当长型或指针字段增至 64 位时，会分配此额外填充。在 SPARC 平台上的 64 位编译环境中，所有类型的结构均与结构中最长成员的长度对齐。当您重组结构时，请遵循将长型和指针字段移到结构开头的简单规则。考虑以下结构定义：

```
struct bar {
   int i;
   long j;
   int k;
   char *p;
};   /* sizeof (struct bar) = 32 */
```

下面是在结构开头定义了长型和指针数据类型的相同结构：

```
struct bar {
  char *p;
  long j;
  int i;
  int k;
};   /* sizeof (struct bar) = 24 */
```

7.3.6 联合

请确保对联合进行检查，因为其字段的长度在 ILP32 和 LP64 数据类型模型之间可能会发生变化。

```
typedef union {
   double _d;
   long _l[2];
} llx_t;
```

下面是修改的版本

```
typedef union {
   double _d;
   int _l[2];
} llx_t;
```

7.3.7 类型常量

在某些常量表达式中，缺少精度会导致数据丢失。请在常量表达式中显式指定数据类型。通过增加 {u,U,l,L} 的组合指定每个整型常量的类型。您也可以使用强制类型转换来指定常量表达式的类型。请看以下示例：

```
int i = 32;
long j = 1 << i; /* j will get 0 because RHS is integer */
                                 /* expression */
```

下面是修改的版本：

```
int i = 32;
long j = 1L << i;
```

7.3.8 注意隐式声明

如果使用 `-xc99=none`，C 编译器会假定在模块中使用却未在外部定义或声明的函数或变量为整型。编译器的隐式整型声明会将以此方式使用的任何长型和指针截断。将函数或变量的相应外部声明置于头文件而非 C 模块中。在使用函数或变量的 C 模块中包含此头文件。如果它是系统头文件定义的函数或变量，您还需要在代码中包含正确的头文件。请看以下示例：

```
int
main(int argc, char *argv[])
{
  char *name = getlogin();
  printf("login = %s\n", name);
  return (0);
}

%
warning: improper pointer/integer combination: op "="
warning: cast to pointer from 32-bit integer
implicitly declared to return int
getlogin        printf
```

修改的版本中现在有正确的头文件

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

int
main(int argc, char *argv[])
{
  char *name = getlogin();
  (void) printf("login = %s\n", name);
  return (0);
}
```

7.3.9 sizeof( ) 是无符号 long

在 LP64 数据类型模型中，sizeof() 的有效类型为无符号长型。有时，sizeof() 会传递给需要使用类型为 int 参数的函数，或者赋值给整型或强制转换为整型。有些情况下，这种截断会导致数据丢失。

```
long a[50];
unsigned char size = sizeof (a);

%
warning: 64-bit constant truncated to 8 bits by assignment
warning: initializer does not fit or is out of range: 0x190
```

7.3.10 使用强制类型转换显示您的意图

关系表达式可能会因为转换规则而显得错综复杂。您应该通过在必要的地方增加强制类型转换很明确地指定表达式的求值方式。

7.3.11 检查格式字符串转换操作

确保 printf(3S)、sprintf(3S)、scanf(3S) 和 sscanf(3S) 的格式字符串可以容纳长型或指针参数。对于指针参数，格式字符串中提供的转换操作应为 %p，以便能在 32 位和 64 位编译环境中运行。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);

%
warning: function argument (number) type inconsistent with format
sprintf (arg 3)     void *: (format) int
```

下面是修改的版本

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%p", (void *)devi);
```

对于长型参数，长型长度规范l应前置于格式字符串中的转换操作字符前面。另外，还要检查以确保 buf 指向的存储器足以包含 16 个数字。

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%d%%%d from heap got%x.%x returns%x\n",
nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);

%
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
```

下面是修改的版本

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%lu%%%lu from heap got%lx.%lx returns%lx\n",
nbytes, align, raddr, raddr + alloc, addr);
```

7.4 其他考虑事项

其余指导原则将重点说明将应用程序转换为完全 64 位程序时遇到的常见问题。

7.4.1 长度增长的派生类型

许多派生类型已进行了更改，以便在 64 位应用程序环境中表示 64 位值。此更改不会影响 32 位应用程序；但是，使用或导出这些类型所描述的数据的任何 64 位应用程序均需要重新求值。关于这一点的一个示例是直接处理 utmp(4) 或 utmpx(4) 文件的应用程序。要在 64 位应用程序环境中正确操作，请勿尝试直接访问这些文件， 而应使用 getutxent(3C) 及相关的函数系列。

7.4.2 检查更改的副作用

需要注意的一个问题是，一个区域中的类型更改可能会导致另一个区域中进行意外的 64 位转换。例如，如果某个函数以前返回 `int` 而现在返回 `ssize_t`，则需要检查所有调用方。

7.4.3 检查直接使用 `long` 是否仍有意义

定义为 `long` 的变量在 ILP32 数据类型模型中为 32 位，在 LP64 数据类型模型中为 64 位。如有可能，通过重新定义变量并使用可移植性更强的派生类型避免出现问题。

与此相关的是，许多派生类型在 LP64 数据类型模型中已更改。例如，在 32 位环境中，`pid_t` 仍为 `long`，而在 64 位环境中，`pid_t` 为 `int`。

7.4.4 对显式 32 位与 64 位原型使用 `#ifdef`

在某些情况下，一个接口存在特定的 32 位和 64 位版本是不可避免的。可以通过在头文件中指定 `_LP64` 或 `_ILP32` 功能测试宏区分这些版本。同样，在 32 位和 64 位环境中运行的代码需要利用相应的 `#ifdefs`，具体取决于编译模式。

7.4.5 调用转换更改

当您通过值传递结构并针对 64 位环境编译代码时，若结构足够小，则通过寄存器传递结构而不是将结构作为副本的指针。如果您尝试在 C 代码与手写汇编代码之间传递结构，这会导致问题。

浮点参数的工作方式类似；有些通过值传递的浮点值通过浮点寄存器传递。

7.4.6 算法更改

在确认代码对 64 位环境是安全的之后，请再次检查代码，以验证算法和数据结构是否仍有意义。数据类型较长，因此数据结构可能占用更多空间。代码的性能也可能变化。出于这些利害关系考虑，您可能需要相应地修改代码。

7.5 入门指南清单

使用以下清单有助于您将代码转换为 64 位。

- 查看所有数据结构和接口，验证它们在 64 位环境中是否仍有效。
- 在代码中包含 `<inttypes.h>`，以获取 `_ILP32` 或 `_LP64` 定义以及多种基本派生类型。系统程序可能需要包含 `<sys/types.h>`（或至少包含 `<sys/isa_defs.h>`），以获取 `_ILP32` 或 `_LP64` 定义。
- 将函数原型以及具有非局部作用域的外部声明移到头文件中，并将这些头文件包含在代码中。
- 使用 `-m64` 和 `-errchk=longptr64` 及 `signext` 选项运行 lint。分别检查每个警告。请注意，并非所有警告均需要更改代码。根据所进行的更改，在 32 位和 64 位模式下再次运行 lint。
- 除非提供的应用程序仅为 64 位，否则请将代码编译为 32 位和 64 位两种形式。
- 测试应用程序，方法是：在 32 位操作系统上执行 32 位版本，在 64 位操作系统上执行 64 位版本。也可以在 64 位操作系统上测试 32 位版本。

◆ ◆ ◆ 第 8 章

8 cscope：交互检查C程序

cscope是在C、lex或yacc源文件中查找指定代码元素的交互式程序。使用cscope，可以比常规编辑器更高效地搜索和编辑源文件。原因是cscope支持函数调用（调用函数时，它执行调用时）以及C语言标识符和关键字。

本章是关于此发行版附带的cscope浏览器的教程。

注 – cscope程序尚未更新，无法理解针对1999 ISO/IEC C标准编写的代码。例如，它尚不能识别1999 ISO/IEC C标准中引入的新关键字。

8.1 cscope进程

为一组C、lex或yacc源文件调用cscope时，它会为这些文件中的函数、函数调用、宏、变量和预处理程序符号生成符号交叉引用表。然后您可以查询该表，了解您指定的符号的位置。首先，它显示一个菜单，要求您选择要执行的搜索的类型。例如，您可能想让cscope查找调用某个指定函数的所有函数。

cscope完成搜索后，将输出一个列表。每个列表条目均包含文件名、行号以及cscope在其中找到指定代码的行的文本。在我们的示例中，列表还包括调用指定函数的函数的名称。您可以选择请求执行另一次搜索或使用编辑器检查某个列出的行。如果您选择后者，cscope将为包含该行的文件调用编辑器，并使光标位于该行。您现在可以在上下文中查看代码，并在需要时将文件作为任何其他文件进行编辑。然后您可以从编辑器返回菜单，请求执行新搜索。

由于执行的过程取决于现行任务，因此没有一套固定的使用cscope时可遵循的指令集。有关其用法的详细示例，请查阅下一节中介绍的cscope会话。该会话将说明在不了解所有代码的情况下如何找到程序中的错误。

8.2 基本用法

假定让您负责维护程序 prog。有时在程序启动时，出现错误消息 "out of storage"。现在您要使用 cscope 查找生成该消息的代码部分。您可以按以下方式执行。

8.2.1 步骤 1：设置环境

cscope 是面向屏幕的工具，只能在终端信息实用程序 (terminfo) 数据库中列出的终端上使用。确保已将 TERM 环境变量设置为您的终端类型，以便 cscope 可以验证它是否列在 terminfo 数据库中。如果您尚未这样做，请为 TERM 赋值并将其输出到 shell，如下所示：

在 Bourne shell 中，键入：

```
$ TERM=term_name; export TERM
```

在 C shell 中，键入：

```
% setenv TERM term_name
```

现在，您可能希望为 EDITOR 环境变量赋值。缺省情况下，cscope 调用 vi 编辑器。（本章中的示例说明了 vi 的用法。）如果您不想使用 vi，请将 EDITOR 环境变量设置为您选择的编辑器并导出 EDITOR，如下所示：

在 Bourne shell 中，键入：

```
$ EDITOR=emacs; export EDITOR
```

在 C shell 中，键入：

```
% setenv EDITOR emacs
```

您可能必须编写 cscope 与您的编辑器之间的接口。有关详细信息，请参见第 180 页中的 “8.2.9 编辑器的命令行语法”。

如果仅希望使用 cscope 进行浏览（而不进行编辑），则可以将 VIEWER 环境变量设置为 pg 并导出 VIEWER。然后 cscope 将调用 pg 而不是 vi。

可以设置名为 VPATH 的环境变量，以指定搜索源文件的目录。请参见第 176 页中的 “8.2.6 视图路径”。

8.2.2 步骤 2：调用 cscope 程序

缺省情况下，cscope 为当前目录中的所有 C、lex 和 yacc 源文件以及当前目录或标准位置中包含的任何头文件生成符号交叉引用表。因此，如果要浏览的程序的所有源文件均位于当前目录，并且其头文件也位于此处或标准位置，请不带参数调用 cscope：

```
% cscope
```

要浏览选定的源文件，请将这些文件的名称用作参数调用 cscope：

```
% cscope file1.c file2.c file3.h
```

有关调用 cscope 的其他方式，请参见第 175 页中的 “8.2.5 命令行选项”。

首次在要浏览的程序的源文件上使用 cscope 时，它生成符号交叉引用表。缺省情况下，该表存储在当前目录中的文件 cscope.out 中。在后续调用中，仅当源文件已被修改或源文件列表不同时，cscope 才重新生成交叉引用。重新生成交叉引用时，从旧的交叉引用复制未更改的文件的数据，这样重新生成的速度比初始生成的速度要快，并且减少了后续调用的启动时间。

8.2.3 步骤 3：查找代码

现在返回到本节开头我们所承担的任务：确定导致输出错误消息 "out of storage" 的问题。已调用 cscope，并已生成交叉引用表。cscope 任务菜单显示在屏幕上。

cscope 任务菜单：

```
% cscope

cscope     Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

按回车键使光标在屏幕上向下移动（显示屏底部有环绕），按 ^p (Ctrl-p) 使光标向上移动；或者使用向上 (ua) 和向下 (da) 方向键。可使用以下单键命令操作菜单和执行其他任务：

表 8–1 cscope 菜单操作命令

Tab	移至下一个输入字段。
回车键	移至下一个输入字段。
^n	移至下一个输入字段。
^p	移至上一个输入字段。
^y	使用上次键入的文本进行搜索。

表 8–1 cscope 菜单操作命令 （续）

^b	移至上一个输入字段并搜索模式。
^f	移至下一个输入字段并搜索模式。
^c	搜索时在不区分/区分大小写之间切换。例如，不区分大小写时，搜索 FILE 将找到 file 和 File。
^r	重新生成交叉引用。
!	启动交互式 shell。键入 **^d** 以返回到 cscope。
^l	刷新屏幕。
?	显示命令列表。
^d	退出 cscope。

如果您要查找的文本的第一个字符与这些命令之一匹配，您可以通过在该字符之前输入 \（反斜杠）使该命令转义。

现在将光标移至第五个菜单项 Find this text string，输入文本 "out of storage"，然后按 Return 键。

cscope 函数： 请求查找文本字符串：

```
$ cscope

cscope     Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

注 – 按照相同过程执行菜单中列出的除第六项 Change this text string 之外的任何其他任务。由于该任务比其他任务稍微复杂一些，因此要按照另一个过程执行。有关如何更改文本字符串的说明，请参见第 177 页中的 “8.2.8 示例”。

cscope 查找指定的文本，找到包含该文本的一行，并报告其查找结果。

cscope 函数： 列出包含文本字符串的行：

```
Text string: out of storage

  File Line
```

```
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

在 cscope 显示成功搜索的结果之后，您有多个选项。您可能要更改其中一行或在编辑器中检查该行周围的代码。或者，如果 cscope 找到很多行，以致这些行的列表不能一次显示在屏幕上，您可能要查看列表的下一部分。下表显示 cscope 找到指定文本之后可用的命令：

表 8–2 初次搜索之后可使用的命令

1 -9	编辑该行引用的文件。您键入的数字对应于 cscope 输出的行列表中的一项。
空格	显示下一组匹配行。
+	显示下一组匹配行。
^v	显示下一组匹配行。
–	显示上一组匹配行。
^e	按顺序编辑显示的文件。
>	将显示的行列表附加至某个文件。
|	将所有行通过管道传送至某个 shell 命令。

同样，如果您要查找的文本的第一个字符与这些命令之一匹配，您可以通过在该字符之前输入反斜杠以避免执行命令。

现在，检查新找到的行周围的代码。输入 **1**（该行在列表中的编号）。系统通过文件 alloc.c 调用编辑器，并使光标位于 alloc.c 第 63 行的开头。

cscope 函数：检查代码行：

```
{
   return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
```

```
    if (p == NULL) {
        (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
        exit(1);
    }
    return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

您会发现变量 p 为 NULL 时将生成错误消息。要确定在什么情况下传递给 alloctest() 的变量为 NULL，首先必须识别调用 alloctest() 的函数。

使用标准退出惯例退出编辑器。您即返回至任务菜单。现在，请在第四项 Find functions calling this function 后面键入 **alloctest**。

cscope 函数： 请求调用 alloctest() 的函数的列表：

```
Text string: out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

cscope 查找并列出三个此类函数。

cscope 函数：列出调用 alloctest() 的函数：

```
Functions calling this function: alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned) nelem, (unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
```

```
Find this egrep pattern:
Find this file:
Find files #including this file:
```

现在希望知道哪些函数调用 `mymalloc()`。`cscope` 找到十个此类函数。它会在屏幕上列出其中九个，并指示您按空格键以查看列表的其余部分。

`cscope` 函数：列出调用 `mymalloc()` 的函数：

```
Functions calling this function: mymalloc

File         Function       Line
1 alloc.c    stralloc       24 return(strcpy(mymalloc
                            (strlen(s) + 1), s));
2 crossref.c crossref       47 symbol = (struct symbol *)mymalloc
                            (msymbols * sizeof(struct symbol));
3 dir.c      makevpsrcdirs  63 srcdirs = (char **) mymalloc
                            (nsrcdirs * sizeof(char*));
4 dir.c      addincdir      167 incdirs = (char **)mymalloc
                            (sizeof(char *));
5 dir.c      addincdir      168 incnames = (char **)
                            mymalloc(sizeof(char *));
6 dir.c      addsrcfile     439 p = (struct listitem *) mymalloc
                            (sizeof(struct listitem));
7 display.c  dispinit       87 displine = (int *) mymalloc
                            (mdisprefs * sizeof(int));
8 history.c  addcmd         19  h = (struct cmd *) mymalloc
                            (sizeof(struct cmd));
9 main.c     main           212 s = mymalloc((unsigned )
                            (strlen(reffile) +strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

由于您知道错误消息 "out of storage" 在程序开始时生成，因此您可以猜测问题可能发生在函数 `dispinit()`（显示初始化）中。

要查看 `dispinit()`（列表中第七个函数），请键入 **7**。

`cscope` 函数：在编辑器中查看 `dispinit()`：

```
void
dispinit()
{
        /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE -  4;
    mdisprefs = lastdispline -  REFLINE + 1;
    if (mdisprefs > 9) {
```

```
       mdisprefs = 9;
    }
       /* allocate the displayed line array */
   displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

mymalloc() 失败，原因是调用它时使用了很大的数或使用了负数。通过检查 FLDLINE 和 REFLINE 的可能值，您会发现 mdisprefs 的值为负的情况，也就是说，您尝试使用负数调用 mymalloc()。

8.2.4 步骤 4：编辑代码

在开窗口终端上，可能具有多个任意大小的窗口。可能由于在行过少的窗口中运行 prog 而出现错误消息 "out of storage"。换句话说，可能属于使用负数调用 mymalloc() 的情况之一。现在您要注意，将来程序在这种情况下异常终止时，在输出更有意义的错误消息 "screen too small" 之后，它会显示上述消息。按以下方式编辑函数 dispinit()。

cscope 函数： 纠正问题：

```
void
dispinit()
{
/* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
       mdisprefs = 9;
    }
/* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

您已纠正我们在本节开始时讨论的问题。现在，如果 prog 在行过少的窗口中运行，它不会简单地以无启示性的错误消息 "out of storage" 而失败。相反，在退出之前，它会检查窗口大小并生成更富有意义的错误消息。

8.2.5 命令行选项

如上所述，缺省情况下，cscope 为当前目录中的 C、lex 和 yacc 源文件生成符号交叉引用表。即，

```
% cscope
```

等效于：

```
% cscope *.[chly]
```

我们还发现，您可以将选定的源文件的名称作为参数调用 cscope，以便浏览这些文件：

```
% cscope file1.c file2.c file3.h
```

在指定交叉引用中包含的源文件方面，cscope 为命令行选项提供了更大的灵活性。当您使用 –s 选项和任意数目的目录名称（用逗号隔开）调用 cscope 时：

```
% cscope– s dir1,dir2,dir3
```

cscope 为指定目录以及当前目录中的所有源文件生成交叉引用。要遍历在**文件**中列出其名称的所有源文件（文件名用空格、制表符或换行符分隔），请使用 –i 选项和包含列表的文件的名称调用 cscope：

```
% cscope– i file
```

如果源文件位于目录树中，请使用以下命令进行浏览：

```
% find .–  name '*.[chly]'– print | sort > file
% cscope– i file
```

但是，如果选择此选项，cscope 将忽略在命令行上出现的任何其他文件。

cscope 使用 –I 选项的方式和 cc 使用 –I 选项的方式相同。请参见第 54 页中的 “2.16 如何指定 include 文件”。

您可以通过调用 –f 选项，指定一个除缺省 cscope.out 之外的交叉引用文件。对于在同一目录中保存单独的符号交叉引用文件，这是很有用的。如果两个程序在同一目录中，但不共享所有相同文件，您可能需要这样做：

```
% cscope– f admin.ref admin.c common.c aux.c libs.c
% cscope– f delta.ref delta.c common.c aux.c libs.c
```

在本示例中，admin 和 delta 两个程序的源文件在同一目录中，但这两个程序由不同的文件组组成。通过在调用 cscope 时为每组源文件指定不同的符号交叉引用文件，可单独保存两个程序的交叉引用信息。

您可以使用 –p*n* 选项指定让 cscope 在列出搜索结果时显示文件的路径名或路径名的一部分。您为 – p 指定的数代表您要显示的路径名的最后 *n* 个元素。缺省值为 1，即文件本身的名称。因此，如果您的当前目录为 home/common，则命令：

```
% cscope– p2
```

使 cscope 在列出搜索结果时显示 common/file1.c、common/file2.c 等等。

如果要浏览的程序包含大量的源文件，可以使用 –b 选项，以便 cscope 在生成交叉引用后停止；cscope 不显示任务菜单。在流水线中使用 cscope -b 与 batch(1) 命令时，cscope 将在后台生成交叉引用：

```
% echo 'cscope -b' | batch
```

生成交叉引用后，只要其间未更改源文件或源文件列表，就只需指定：

```
% cscope
```

用于要复制的交叉引用以及要以正常方式显示的任务菜单。如果希望无需等待 cscope 完成其初始处理即可继续工作，可以使用此命令序列。

-d 选项指示 cscope 不更新符号交叉引用。如果您确定未进行此类更改，则可以使用该选项以节省时间；cscope 不检查源文件是否更改。

注 – 使用 – d 选项时请小心。如果在错误地认为源文件尚未更改的情况下指定 -d，则在响应查询时 cscope 将引用过时的符号交叉引用。

有关其他命令行选项，请查看 cscope(1) 手册页。

8.2.6 视图路径

正如我们所看到的，缺省情况下 cscope 在当前目录中搜索源文件。如果设置环境变量 VPATH，cscope 将在包含您的视图路径的目录中查找源文件。视图路径是有序的目录列表，每个目录下面具有相同的目录结构。

例如，假设您参与某个软件项目。在 /fs1/ofc 下面的目录中有一组**正式**源文件。每个用户都有一个起始目录 (/usr/you)。如果您更改软件系统，则 /usr/you/src/cmd/prog1 中可能只有您正在更改的文件的副本。完整程序的正式版本可在目录 /fs1/ofc/src/cmd/prog1 中找到。

假设您使用 cscope 浏览组成 prog1 的三个文件，即 f1.c、f2.c 和 f3.c。您需要将 VPATH 设置为 /usr/you 和 /fs1/ofc 并将其导出，如下所示：

在 Bourne shell 中，键入：

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

在 C shell 中，键入：

```
% setenv VPATH /usr/you:/fs1/ofc
```

然后创建当前目录 /usr/you/src/cmd/prog1，并调用 cscope：

```
% cscope
```

程序将定位视图路径中的所有文件。如果找到重复文件，cscope 使用其父目录在 VPATH 中较早出现的文件。因此，如果 f2.c 位于您的目录中，并且所有三个文件位于正式目录中，cscope 将从您的目录中检查 f2.c，并从正式目录中检查 f1.c 和 f3.c。

VPATH 中的第一个目录必须为您将在其中工作的目录的前缀，通常为 $HOME。VPATH 中的每个冒号分隔目录都必须是绝对目录：它应该以 / 。

8.2.7 cscope 和编辑器调用栈

cscope 和编辑器调用可以形成栈。也就是说，当 cscope 使您进入编辑器查看某个符号的引用时，如果存在另一个相关引用，您可以从编辑器内部再次调用 cscope 以查看第二个引用，而无需退出对 cscope 或编辑器的当前调用。然后，您可以通过使用相应的 cscope 命令和编辑器命令退出，执行倒退操作。

8.2.8 示例

本节中的示例说明如何使用 cscope 执行以下三项任务：将常量更改为预处理程序符号、向函数增加参数以及更改变量的值。第一个示例说明了更改文本字符串的过程，这与 cscope 菜单上的其他任务稍有不同。也就是说，在您输入要更改的文本字符串之后，cscope 会提示您输入新文本，显示包含旧文本的行，并等待您指定要更改的行。

8.2.8.1 将常量更改为预处理程序符号

假设您要将常量 100 更改为预处理程序符号 MAXSIZE。选择第六个菜单项 Change this text string，然后输入 **\100**。必须使用反斜杠将 1 转义，原因是它对 cscope 具有特殊意义（菜单上的第 1 项）。现在按回车键。cscope 将提示您输入新文本字符串。键入 **MAXSIZE**。

cscope 函数： 更改文本字符串：

```
cscope            Press the ? key for help
```

```
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

cscope 显示包含指定文本字符串的行，并等待您选择要更改其中文本的行。

cscope 函数： 提示要更改的行：

```
cscope            Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

您知道列表中第 1、2 和 3 行（列出的源文件的第 4、26 和 8 行）中的常量 `100` 应更改为 `MAXSIZE`。您还知道 `read.c` 中的 `0100` 和 `err.c` 中的 `100.0`（列表中的第 4 行和第 5 行）不应更改。使用以下单键命令选择要更改的行：

表 8–3 用于选择要更改的行的命令

`1-9`	标记要更改的行或去标记。
`*`	标记要更改的所有显示行或去标记。
空格	显示下一组行。
`+`	显示下一组行。
`–`	显示上一组行。
`a`	标记要更改的所有行。
`^d`	更改标记的行并退出。
`Esc`	退出而不更改标记的行。

在本例中，输入 **1**、**2** 和 **3**。您键入的数字不会输出在屏幕上。相反，cscope 通过在列表中您要更改的每个列表项的行号后面输出 `>`（大于）符号来标记这些项。

cscope 函数： 标记要更改的行：

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

现在键入 **^d** 更改所选行。cscope 将显示已更改的行并提示您继续执行。

cscope 函数： 显示已更改的文本行：

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

Press the RETURN key to continue:
```

当您响应此提示而按回车键时，cscope 将刷新屏幕，从而使其恢复到您选择要更改的行之前的状态。

下一步是为新符号 MAXSIZE 添加 #define。由于将要显示 #define 的头文件不在其行得到显示的文件之列，因此您必须通过键入 **!** 退回到 shell。shell 提示符出现在屏幕底部。然后进入编辑器，并添加 #define。

cscope 函数： 退出到 Shell：

```
Text string: 100

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;                                    /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
```

```
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

要恢复 cscope 会话，请退出编辑器并键入 **^d** 以退出 shell。

8.2.8.2 向函数增加参数

向函数增加参数包括两个步骤：编辑函数本身，以及向代码中调用函数的各个位置增加新参数。

首先，使用第二个菜单项 `Find this global definition` 编辑函数。接着，找出调用函数的位置。使用第四个菜单项 `Find functions calling this function` 获取调用此函数的所有函数的列表。使用此列表，您可以通过分别输入每行在列表中的编号为该行调用编辑器，或通过键入 **^e** 自动为所有行调用编辑器。使用 cscope 进行此类更改可确保您需要编辑的任何函数均不会被忽略。

8.2.8.3 更改变量的值

有时，您可能要了解建议的更改如何影响代码。

假设您要更改变量或预处理程序符号的值。在这样做之前，请使用第一个菜单项 `Find this C symbol` 获取受到影响的引用的列表。然后使用编辑器检查每个引用。此步骤有助于预测您建议的更改的总体影响。之后，可以按相同的方式使用 cscope 来验证是否已进行更改。

8.2.9 编辑器的命令行语法

缺省情况下，cscope 调用 vi 编辑器。可以通过将首选编辑器分配给 EDITOR 环境变量并导出 EDITOR 来覆盖缺省设置，如第 168 页中的 “8.2.1 步骤 1：设置环境”中所述。但是，cscope 期望它使用的编辑器具有以下形式的命令行语法：

```
% editor +linenum filename
```

与 vi 一样。如果要使用的编辑器没有此命令行语法，则必须编写 cscope 和编辑器之间的接口。

假设您要使用 ed。由于 ed 不允许在命令行上指定行号，因此除非编写一个包含以下行的 shell 脚本，否则不能在 cscope 中使用它查看或编辑文件：

```
/usr/bin/ed $2
```

让我们将 shell 脚本命名为 myedit。现在将 EDITOR 的值设置为 shell 脚本并导出 EDITOR：

在 Bourne shell 中，键入：

```
$ EDITOR=myedit; export EDITOR
```

在 C shell 中，键入：

```
% setenv EDITOR myedit
```

当 cscope 为您指定的列表项（比如说 main.c 中的第 17 行）调用编辑器时，它将用以下命令行调用 shell 脚本：

```
% myedit +17 main.c
```

myedit 然后丢弃行号 ($1) 并使用文件名 ($2) 正确调用 ed。当然，不会自动移至文件的第 17 行，必须执行相应的 ed 命令以显示和编辑该行。

8.3 未知终端类型错误

如果您看到以下错误消息：

```
Sorry, I don't know how to deal with your "term" terminal
```

您的终端可能未在当前装入的终端信息实用程序 (terminfo) 数据库中列出。请确保将正确的值赋给 TERM。如果该消息再出现，请尝试重新装入终端信息实用程序。

如果显示以下消息：

```
Sorry, I need to know a more specific terminal type than "unknown"
```

设置并导出 TERM 变量，如第 168 页中的 “8.2.1 步骤 1：设置环境”中所述。

◆ ◆ ◆ 附 录 A

按功能分组的编译器选项

本章总结了按功能分组的 C 编译器选项。表 A–15 提供了这些选项以及编译器命令行语法的详细说明。

A.1 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，请参阅表 A–15 和附录 B，C 编译器选项参考。某些标志用于多个目的，因此出现多次。

这些选项适用于除了特别注明之外的所有平台；基于 SPARC 的系统所特有的功能标识为 (SPARC)，基于 x86/x64 的系统所特有的功能标识为 *(x86)*。仅适用于 Solaris 平台的选项标记为 (Solaris)。适用于仅 Linux 平台的选项标记为 (Linux)。

A.1.1 优化和性能选项

表 A–1 优化和性能选项表

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。
`-fma`	(SPARC) 启用自动生成浮点乘加指令。
`-p`	准备目标代码，以便收集数据进行文件配置
`-xalias_level`	使编译器可执行基于类型的别名分析和优化。
`-xannotate`	(Solaris) 指示编译器创建以后可由诸如 binopt(1) 之类的二进制修改工具进行转换。
`-xbinopt`	准备二进制文件，以便随后进行优化、转换和分析。
`-xbuiltin`	改进对调用标准库函数的代码的优化。

表 A–1 优化和性能选项表 （续）

选项	操作
-xdepend	分析循环以了解迭代间数据依赖性并执行循环重构。
-xF	允许链接程序对数据和函数重新排序。
-xhwcprof	(SPARC) 允许编译器支持基于硬件计数器的分析。
-xinline	尝试仅内联指定函数。
-xinstrument	编译程序并为其提供程序设备以便 Thread Analyzer 对其进行分析。
-xipo	通过调用过程间分析组件对整个程序执行优化。
-xipo_archive	使交叉文件优化可包含归档 (.a) 库。
-xjobs	设置编译器创建的进程数。
-xlibmil	内联一些库例程，以加快执行的速度。
-xlic_lib=sunperf	在 Sun 性能库中进行链接。
-xlinkopt	对可重定位对象文件执行链接时优化。
-xlibmopt	启用已优化数学例程的库。
-xmaxopt	此命令会将 pragma opt 级别限制为指定级别。
-xnolibmil	不内联数学库例程。
-xnolibmopt	禁用已优化数学例程的库。
-x0	优化对象代码。
-xnorunpath	禁止在可执行文件中包含共享库的运行时搜索路径。
-xpagesize	设置栈和堆的首选页面大小。
-xpagesize_stack	设置栈的首选页面大小。
-xpagesize_heap	设置堆的首选页面大小。
-xpch	缩短其源文件共享同一组 include 文件的应用程序的编译时间。
-xpec	生成一个可与自动调优系统 (Automatic Tuning System, ATS) 一起使用的可移植的可执行代码 (Portable Executable Code, PEC) 二进制文件。有关更多信息，请访问 http://cooltools.sunsource.net。
-xpchstop	可与 -xpch 结合使用以指定活前缀的最后一个 include 文件。
-xpentium	*(x86)* 针对 Pentium 处理器进行优化。
-xprefetch	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。

表 A–1 优化和性能选项表 （续）

选项	操作
-xprefetch_auto_type	控制生成间接预取指令的方式。
-xprofile	为配置文件收集数据或使用配置文件进行优化。
-xprofile_ircache	通过重用 -xprofile=collect 阶段保存的编译数据缩短 -xprofile=use 阶段的编译时间。
-xprofile_pathmap	支持单个配置文件目录中的多个程序或共享库。
-xrestrict	将返回赋值指针的函数参数视为限定指针。
-xsafe	(SPARC) 允许编译器假定不会发生基于内存的陷阱。
-xspace	不对增加代码大小的循环执行优化和并行化。
-xunroll	建议优化器解开循环 *n* 次。

A.1.2 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 A–2 编译时选项和链接时选项表

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。
-mt	扩展为 -D_REENTRANT -lthread 的宏选项。
-p	准备目标代码，以便收集数据使用 prof(1) 进行文件配置。
-xarch	指定指令集结构体系。
-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	对可重定位对象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	支持显式并行化 OpenMP 接口，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_stack	设置栈的首选页面大小。

表 A–2 编译时选项和链接时选项表 （续）

选项	操作
-xpagesize_heap	设置堆的首选页面大小。
-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=lib	启用自动生成对向量库函数的调用。

A.1.3 数据对齐选项

表 A–3 数据对齐选项表

选项	操作
-xchar_byte_order	通过按指定字节顺序排列多字符字符常量的字符来生成整型常量。
-xdepend	分析循环以了解迭代间数据依赖性并执行循环重构。
-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	支持显式并行化 OpenMP 接口，包括一组源代码指令、运行时库例程和环境变量。

A.1.4 数值和浮点选项

表 A–4 数值和浮点选项表

选项	操作
-flteval	*(x86)* 控制浮点计算。
-fma	(SPARC) 启用自动生成浮点乘加指令。
-fnonstd	导致对浮点运算硬件的非标准初始化。
-fns	启用非标准浮点模式。
-fprecision	*(x86)* 初始化浮点控制字中的舍入精度模式位。
-fround	设置程序初始化期间运行时建立的 IEEE 754 舍入模式。
-fsimple	允许优化器进行有关浮点运算的简化假定。
-fsingle	导致编译器按单精度而非双精度计算 float 表达式。
-fstore	*(x86)* 导致编译器将浮点表达式或函数的值转换为赋值左侧的类型。
-ftrap	设置 IEEE 754 捕获模式在启动时生效。

表 A–4 数值和浮点选项表 （续）

选项	操作
-nofstore	*(x86)* 不将浮点表达式或函数的值转换为赋值左侧的类型。
-xdepend	分析循环以了解迭代间数据依赖性并执行循环重构。
-xlibmieee	强制 IEEE 754 样式在异常情况下返回数学例程值。
-xsfpconst	将无后缀浮点常量表示为单精度。
-xvector	启用自动生成对向量库函数的调用。

A.1.5 并行化选项

表 A–5 并行化选项表

选项	操作
-mt	扩展为 -D_REENTRANT -lthread 的宏选项。
-xautopar	针对多处理器启用自动并行化。
-xcheck	添加针对栈溢出的运行时检查并初始化局部变量。
-xdepend	分析循环以了解迭代间数据依赖性并执行循环重构。
-xloopinfo	显示哪些循环已并行化以及哪些循环未并行化。
-xopenmp	支持显式并行化 OpenMP 接口，包括一组源代码指令、运行时库例程和环境变量。
-xreduction	自动并行化期间启用约简识别。
-xrestrict	将返回赋值指针的函数参数视为限定指针。
-xvpara	对已指定 #pragma MP 指令但可能未针对并行化进行正确指定的循环发出警告。
-xthreadvar	控制线程局部变量的实现。
-Zll	为 lock_lint 创建程序数据库，但不生成可执行代码。

A.1.6 源代码选项

表 A–6 源代码选项表

选项	操作
-A	将 *name* 作为谓词与指定的 *token* 相关联，这与使用 #assert 预处理指令类似。

表 A–6 源代码选项表 （续）

选项	操作
-C	阻止预处理程序删除注释（预处理指令行中的注释除外）。
-D	将 *name* 与指定的**标记**相关联，这与使用 #define 预处理指令类似。
-E	仅通过预处理程序运行源文件，并将输出发送到 stdout。
-fd	报告 K&R 样式的函数的定义和声明。
-H	将当前编译期间涉及的每个文件的路径名输出到标准错误中，每行一个路径名。
-I	将目录添加到用于搜索具有相对文件名的 #include 文件的列表中。
-include	使编译器将参数 *filename* 视为作为 #include 预处理程序指令出现在主源文件的第一行。
-P	仅通过 C 预处理程序运行源文件。
-U	删除预处理程序符号 *name* 的所有初始定义。
-X	-X 选项指定符合 ISO C 标准的各种级别。
-xCC	接受 C++ 样式的注释。
-xc99	控制编译器识别支持的 C99 功能。
-xchar	帮助从字符被定义为无符号类型的系统中迁移。
-xcsi	允许 C 编译器接受在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。
-xM	对指定 C 程序仅运行预处理程序，同时请求生成 makefile 依赖性并将结果发送到标准输出。
-xM1	收集类似 -xM 的依赖性，但 /usr/include 文件除外。
-xMD	像 -xM 一样生成 makefile 依赖性，但包括编译。
-xMF	指定可存储 makefile 依赖性信息的文件名。
-xMMD	生成 makefile 依赖性，但不包括系统头文件。
-xP	输出在此模块中定义的所有 K&R C 函数的原型。
-xpg	准备目标代码，为使用 *gprof*(1) 进行文件配置而收集数据。
-xtrigraphs	确定三字符序列的识别。
-xustr	启用对十六位字符构成的串文字的识别。

A.1.7 编译代码选项

表 A–7 编译代码选项表

选项	操作
-c	指示编译器禁止与 *ld*(1) 链接并在当前工作目录中为每个源文件生成一个 .o 文件。
-o	指定输出文件。
-S	指示编译器生成汇编源文件但不汇编程序。

A.1.8 编译模式选项

表 A–8 编译模式选项表

选项	操作
-#	启用详细模式，显示命令选项展开的方式以及调用的组件。
-###	显示将要调用的每个组件，但是并不真正执行。还显示命令选项扩展的过程。
-features	启用或禁用各项 C 语言功能。
-keeptmp	保留编译期间创建的临时文件，而非将它们自动删除。
-V	指示 cc 在编译器执行时输出每个组件的名称和版本 ID。
-W	向 C 编译系统组件传递参数。
-X	-X 选项指定符合 ISO C 标准的各种级别。
-xc99	控制编译器识别支持的 C99 功能。
-xchar	保留字符的符号。
-xhelp	显示联机帮助信息。
-xjobs	设置编译器创建的进程数。
-xpch	缩短其源文件共享同一组 include 文件的应用程序的编译时间。
-xpchstop	可与 -xpch 结合使用以指定活前缀的最后一个 include 文件。
-xtemp	将 cc 使用的临时文件的目录设置为 *dir*。
-xtime	报告每个编译组件所用的时间和资源。
-Y	指定一个新目录查找 C 编译系统组件。
-YA	更改搜索组件的缺省目录。

表 A–8 编译模式选项表 （续）

选项	操作
-YI	更改搜索 include 文件的缺省目录。
-YP	更改查找库文件的缺省目录。
-YS	更改启动对象文件的缺省目录。

A.1.9 诊断选项

表 A–9 诊断选项表

选项	操作
-errfmt	将字符串 "error: " 作为错误消息的前缀，以便区分于警告消息。
-errhdr	将来自头文件的警告限定到指定的组。
-erroff	禁止编译器警告消息。
-errshort	控制编译器发现类型不匹配时生成的错误消息中的详细信息量。
-errtags	显示每条警告消息的消息标记。
-errwarn	如果发出指示警告消息， cc 将以故障状态退出。
-v	指示编译器执行更严格的语义检查并启用类似 lint 的其他检查。
-w	禁止编译器警告消息。
-xe	对源文件仅执行语法和语义检查，但不生成任何目标代码或可执行代码。
-xtransition	发出关于 K&R C 与 Solaris Studio C 之间存在差异的警告。
-xvpara	对已指定 #pragma MP 指令但可能未针对并行化进行正确指定的循环发出警告。

A.1.10 调试选项

表 A–10 调试选项表

选项	操作
-xcheck	添加针对栈溢出的运行时检查并初始化局部变量。
-g	为调试器生成附加符号表信息。
-s	删除输出对象文件中的所有符号调试信息。
-xdebugformat	生成 dwarf 格式而非 stabs 格式的调试信息。

表 A–10 调试选项表 （续）

选项	操作
-xpagesize	设置栈和堆的首选页面大小。
-xpagesize_stack	设置栈的首选页面大小。
-xpagesize_heap	设置堆的首选页面大小。
-xs	禁用 dbx 的对象文件自动读取。
-xvis	(SPARC) 启用编译器对 VIS[tm] 指令集中定义的汇编语言模板的识别。

A.1.11 链接选项和库选项

表 A–11 链接选项和库选项表

选项	操作
-B	指定用于链接的库绑定是 static 还是 dynamic。
-d	指定链接编辑器中的链接类型是动态还是静态。
-G	将选项传递给链接编辑器，以生成共享对象而非动态链接的可执行内容。
-h	为共享动态库指定名称，从而获得库的不同版本。
-i	将选项传递给链接程序，以忽略所有 LD_LIBRARY_PATH 设置。
-L	向链接程序搜索库的列表中添加目录。
-l	与目标库 lib*name*.so 或 lib*name*.a 链接。
-mc	删除对象文件的 .comment 部分中重复的字符串。
-mr	删除 .comment 部分中所有的字符串。也可以向目标文件的该部分中插入一个**字符串**。
-Q	向输出文件发出或不发出标识信息。
-R	将用于指定库搜索目录的、以冒号分隔的目录列表传递给运行时链接程序。
-xMerge	将数据段合并到文本段中。
-xcode	指定代码地址空间。
-xldscope	控制变量的缺省作用域和函数定义，以创建更快、更安全的共享库。
-xnolib	缺省情况下，不链接任何库。
-xnolibmil	不内联数学库例程。

表 A–11 链接选项和库选项表 （续）

选项	操作
-xstrconst	在以后的发行版中可能会废弃此选项。改用 -features=[no%]conststrings。 将串文字插入文本段（而非缺省数据段）的只读数据部分。

A.1.12 目标平台选项

表 A–12 目标平台选项表

选项	操作
-m32|-m64	指定编译的二进制对象的内存模型。
-xarch	指定指令集结构体系。
-xcache	定义供优化器使用的高速缓存属性。
-xchip	指定供优化器使用的目标处理器。
-xregs	为生成的代码指定寄存器的用法。
-xtarget	为指令集和优化指定目标系统。

A.1.13 x86 特定选项

表 A–13 x86 特定选项表

选项	操作
-flteval	控制浮点计算。
-fprecision	初始化浮点控制字中的舍入精度模式位
-fstore	导致编译器将浮点表达式或函数的值转换为赋值左侧的类型
-nofstore	不将浮点表达式或函数的值转换为赋值左侧的类型
-xmodel	针对 Solaris x86 平台修改 64 位对象的形式
-xpentium	针对 Pentium 处理器进行优化。

A.1.14 许可证选项

表 A–14 许可证选项表

选项	操作
`-xlicinfo`	返回有关许可证系统的信息。

A.1.15 废弃的选项

下表列出了已过时的选项。请注意，编译器仍可接受这些选项，但在以后的版本中可能不接受这些选项。所以，请尽快开始使用建议的替代选项。

表 A–15 废弃选项表

选项	操作
`-dalign`	改用 `-xmemalign=8s`。
`-KPIC` (SPARC)	改用 `-xcode=pic32`。
`-Kpic` (SPARC)	改用 `-xcode=pic13`。
`-misalign`	改用 `-xmemalign=1i`。
`-misalign2`	改用 `-xmemalign=2i`。
`-x386`	改用 `-xchip=generic`。
`-x486`	改用 `-xchip=generic`。
`-xa`	改用 `-xprofile=tcov`。
`-xarch=v7,v8,v8a`	已过时。
`-xcg`	改用 `-O`，以利用 `-xarch`、`-xchip` 和 `-xcache` 的缺省值。
`-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`。
`-xsb`	已废弃，没有替代选项。
`-xsbfast`	已废弃，没有替代选项。
`-xtarget=386`	改用 `-xtarget=generic`。
`-xtarget=486`	改用 `-xtarget=generic`。

表 A–15 废弃选项表 （续）

选项	操作
`-xvector=yes`	改用 `-xvector=lib`。
`-xvector=no`	改用 `-xvector=none`。

◆ ◆ ◆ 附 录 B

C 编译器选项参考

本章按字母顺序介绍 C 编译器选项。有关按功能分组的选项，请参见附录 A，按功能分组的编译器选项。例如，表 A–1 列出了所有优化和性能选项。

请注意，缺省情况下，C 编译器识别 1999 ISO/IEC C 标准的某些构造。具体来说，附录 D，支持的 C99 功能 中详细介绍了受支持的功能。如果要用 1990 ISO/IEC C 标准限制编译器，请使用 `-xc99=none` 命令。

如果您正在将 K&R（Kernighan 和 Ritchie）C 程序移植到 ISO C，请特别留意有关兼容性标志的章节：第 218 页中的 “B.2.68 -X[c|a| t|s]”。使用它们可以简化到 ISO C 的转换。另请参阅第 116 页中的 “5.4 内存引用约束的示例”中有关转换的介绍。

B.1 选项语法

`cc` 命令的语法为：

```
% cc [options] filenames [libraries]...
```

其中：

- *options* 表示表 A–15 中介绍的一个或多个选项。
- *filenames* 表示在生成可执行程序过程中使用的一个或多个文件。

  C 编译器接受包含在由 *filenames* 指定的文件列表中的 C 源文件和目标文件的列表。除非使用 `-o` 选项，否则最终可执行代码将位于 `a.out` 中。在这种情况下，代码位于由 `-o` 选项指定的文件中。

  使用 C 编译器可编译和链接以下任何组合：

  - C 源文件，带有 `.c` 后缀
  - 内联模板文件，带有 `.il` 后缀（仅当使用 .c 文件指定时）
  - C 预处理源文件，带有 `.i` 后缀
  - 目标代码文件，带有 `.o` 后缀

- 汇编程序源文件，带有 .s 后缀

  在链接之后，C 编译器将所链接的文件（当前在可执行代码中）置于一个名称为 a.out 的文件中，或由 -o 选项指定的文件中。当编译器生成每个 .i 或 .c 输入文件的目标代码时，始终会在当前工作目录中创建一个目标 (.o) 文件。

libraries 表示许多标准库或用户提供库中的任意库，这些库包含函数、宏和常量的定义。

请参见选项 -Y*P*, *dir*，以了解如何更改用于查找库的缺省目录。*dir* 是以冒号分隔的路径列表。通过使用 **-###** 或 **-xdryrun** 选项并检查 **ld** 调用的 **-Y** 选项可查看缺省库搜索顺序。

cc 使用 getopt 来分析命令行选项。这些选项被视为单个字母或后面带一个参数的单个字母。请参见 getopt(3c)。

B.2 cc 选项

本节按字母顺序介绍 cc 选项。手册页 cc (1) 中也提供了这些说明。使用 cc -flags 选项可获得一行有关这些说明的摘要。

注明特定于一个或多个平台的选项可被接受，且不会出现错误，但在其他所有平台上会被忽略。

B.2.1 -#

打开详细模式，显示命令选项的展开方式。显示调用的每个组件。

B.2.2 -###

在调用每个组件时显示该组件，但不实际执行它。还显示命令选项扩展的过程。

B.2.3 -A*name*[(*tokens*)]

将 *name* 作为谓词与指定的 *tokens* 关联，如同使用 #assert 预处理指令一样。预断言：

- system(unix)
- machine(sparc) (SPARC)
- machine(i386) *(x86)*
- cpu(sparc) (SPARC)
- cpu(i386) *(x86)*

这些预断言在 -Xc 模式下无效。

如果 -A 后面仅跟随一个短横线 (-)，将导致所有预定义宏（除那些以 __ 开头的宏以外）和预定义断言被忽略。

B.2.4 -B[static| dynamic]

指定用于链接的库绑定是 static 类型还是 dynamic 类型，可分别指明库是非共享的还是共享的。

如果给定 -lx 选项，则 *-Bdynamic* 使链接编辑器查找名称为 libx.*so* 的文件，然后查找名称为 libx.*a* 的文件。

–Bstatic 使链接编辑器仅查找名称为 lib*x*.a 的文件。此选项可作为一个切换开关在命令行中多次指定。此选项及其参数将传递给 ld(1)。

注 – 在 Solaris 64 位编译环境中，许多系统库（例如 libc）都只能作为动态库使用。因此，请勿在命令行上将 -Bstatic 用作最后一个切换开关。

此选项及其参数传递给链接程序。

B.2.5 -C

防止 C 预处理程序删除注释，位于预处理指令行中的注释除外。

B.2.6 -c

指示 C 编译器用 ld(1) 抑制链接并为每个源文件生成一个 .o 文件。您可使用 -o 选项显式指定单个目标文件。当编译器生成每个 i 或 .c 输入文件的对象代码时，总是会在当前的工作目录中创建一个对象 (.o) 文件。如果抑制链接步骤，将会同时抑制删除目标文件。

B.2.7 -D*name*[(*arg*[,*arg*])][= *expansion*]

使用可选参数定义宏，如同使用 #define 预处理指令定义宏一样。如果未指定 =*expansion*，则编译器假定为 1。

有关编译器预定义宏的列表，请参见 **cc**(1) 手册页。

B.2.8 -d[y|n]

-dy 指定动态链接，这是链接编辑器中的缺省链接。

-dn 指定链接编辑器中的静态链接。

此选项及其参数将传递给 ld(1)。

注 – 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。

B.2.9 -dalign

(SPARC) 已废弃。您不该使用此选项。改用 -xmemalign=8s。有关更多信息，请参见第 249 页中的 “B.2.116 -xmemalign=*ab*”。有关已废弃的选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。在 x86 平台上会在无提示的情况下忽略此选项。

B.2.10 -E

仅通过预处理程序运行源文件，并将输出发送给 stdout。预处理程序直接构建到编译器中，但 -Xs 模式除外，因为该模式会调用 /usr/ccs/lib/cpp。包含预处理程序行号信息。另请参见 –P 选项。

B.2.11 -errfmt[=[no%]error]

如果要将字符串 "error:" 作为前缀添加到错误消息开头以将错误消息与警告消息相区分，可使用此选项。此前缀也可附加到通过 -errwarn 转换成错误的警告。

表 B–1 -errfmt 标志

标志	含义
error	向所有错误消息添加前缀 "error: "。
no%error	不向任何错误消息添加前缀 "error: "。

如果不指定此选项，则编译器将其设置为 -errfmt=no%error。如果您指定 -errfmt，但不提供值，则编译器将其设置为 -errfmt=error。

B.2.12 -errhdr [=h]

将来自头文件的警告限制为由以下标志所指示的头文件组：

表 B–2 -errhdr 选项

值	含义
%all	检查使用的所有头文件。
%none	不检查头文件。
%user	检查所有用户头文件，但 /usr/include 及其子目录中的头文件除外。此外，还检查编译器提供的所有头文件。这是缺省值。

B.2.13 -erroff[= *t*]

此命令抑制 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。

t 是一个以逗号分隔的列表，它包含以下项中的一项或多项：*tag*、no%*tag*、%all、%none。顺序是很重要的；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 B–3 -erroff 标志

标志	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*	启用此 *tag* 指定的警告消息
%all	抑制所有警告消息
%none	启用所有警告消息（缺省值）

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

-erroff 选项只能抑制来自 C 编译器前端并在使用 -errtags 选项时显示标记的警告消息。您可以更好地控制错误消息抑制。请参见第 42 页中的 “2.11.6 error_messages”。

B.2.14 -errshort[= *i*]

使用此选项可在编译器发现类型不匹配时控制所生成错误消息的详细程度。当编译器发现涉及到大聚集的类型不匹配时，此选项特别有用。

i 可以是以下各项之一：

表 B–4 -errshort 标志

标志	含义
short	以短形式输出错误消息，且不会出现类型扩展。不扩展聚集成员、函数参数和返回类型。
full	以完全详细形式输出错误消息，并显示不匹配类型的完全扩展。
tags	对于具有标记名称的类型，输出错误消息的标记名称。无标记名称的类型将以扩展形式显示。

如果不指定 -errshort，编译器会将该选项设置为 -errshort=full。如果指定 -errshort 但不提供值，编译器会将该选项设置为 -errshort=tags。

此选项不会累积，它接受在命令行指定的最后一个值。

B.2.15 -errtags[= *a*]

为 C 编译器前端的每条警告消息显示消息标志，可以使用 -erroff 选项抑制该消息，或使用 -errwarn 选项将其设置为致命错误。来自 C 编译器驱动程序以及 C 编译系统其他组件的消息不带错误标记，使用 -errof 选项并不能抑制这些消息，而使用 -errwarn 选项也不会产生致命错误。

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

B.2.16 -errwarn[= *t*]

使用 -errwarn 选项可以在出现指定的警告消息时使 C 编译器退出，并指示故障状态。

t 是一个以逗号分隔的列表，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。

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。

只有来自 C 编译器前端在使用 -errtags 选项时会显示标记的警告消息可以使用 -errwarn 选项进行指定，从而使 C 编译器以失败状态退出。

下表详细列出了 -errwarn 值：

表 B–5 -errwarn 标志

标志	含义
tag	在此 *tag* 指定的消息作为警告消息发出时导致 cc 以致命状态退出。如果未出现 *tag*，则没有影响。
no%*tag*	在 *tag* 指定的消息仅作为警告消息发出时防止 cc 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	在发出任何警告消息时导致编译器以致命状态退出。%all 可以后跟 no%*tag*，以避免该行为的特定警告消息。
%none	在发出任何警告消息时防止警告消息导致编译器以致命状态退出。

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

B.2.17 -fast

此选项是一个宏，可将其有效用作为尽量提高运行时性能而调整可执行程序的起点。-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 -# 选项或 -xdryrun 检查 -fast 的扩展选项，并将 -fast 的相应选项结合到正在进行的可执行文件优化过程中。

-fast 的扩展选项包括 -xlibmopt 选项，该选项使编译器可使用优化的数学例程库。有关更多信息，请参见第 245 页中的 “B.2.104 -xlibmopt”。

-fast 选项会影响 errno 的值。有关更多信息，请参见第 51 页中的 “2.13 保留 errno 的值”。

用 -fast 编译的模块必须也用 -fast 进行链接。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 185 页中的 “A.1.2 编译时选项和链接时选项”。

–fast 选项不适于计划在编译机器之外的其他目标机器上运行的程序。在这种情况下，请在 -fast 后附带相应的 -xtarget 选项。例如：

```
cc -fast -xtarget=generic ...
```

对于依赖于 SUID 指定的异常处理的 C 模块，请在 -fast 后附带 -xnolibmil：

```
% cc -fast -xnolibmil
```

使用 -xlibmil 时，通过设置 errno 或调用 matherr(3m) 无法标记异常。

–fast 选项不适于要求与 IEEE 754 标准严格一致的程序。

下表列出了 -fast 在各平台中选择的选项集。

表 B–6 -fast 扩展选项标志

选项	SPARC	x86
-fns	X	X
-fsimple=2	X	X
-fsingle	X	X
-nofstore	-	X
-xalias_level=basic	X	X
-xbuiltin=%all	X	X
-xlibmil	X	X
-xlibmopt	X	X
-xmemalign=8s	X	-
-xO5	X	X
-xprefetch=auto,explicit	X	-
-xregs=frameptr	-	X
-xtarget=native	X	X

注 – 有些优化对程序行为进行特定假定。如果程序不符合这些假定，应用程序将会崩溃或产生错误结果。请参阅各个选项的描述，以确定您的程序是否适合用 -fast 编译。

由这些选项执行的优化可能会改变由 ISO C 和 IEEE 标准定义的程序的行为。有关详细信息，请参见特定选项的描述。

–fast 的作用与命令行上的宏扩展相同。因此，您可以通过在 -fast 后附带预期的优化级别或代码生成选项来覆盖优化级别和代码生成选项。使用 -fast -xO4 对进行编译类似于使用 -xO2 -xO4 对进行编译。后者优先。

在 x86 上，-fast 选项包括 -xregs=frameptr。有关详细信息，特别是编译混合 C、Fortran 和 C++ 源代码时，请参见该选项的介绍。

请勿对依赖于 IEEE 标准异常处理的程序使用此选项；否则会产生不同的数值结果、程序提前终止或意外的 SIGFPE 信号。

要在运行的平台上查看 -fast 的实际扩展，请使用

```
% cc -fast -xdryrun
```

B.2.18 -fd

报告 K&R 样式的函数定义和声明。

B.2.19 -features=[*v*]

下表列出了可代替 *v* 使用的值。

表 B–7 -features 标志

值	含义
[no%]conststrings	启用或禁用将文本字符串放置在只读内存中。缺省值为 -features=conststrings，这会将文本字符串放在只读的数据段中。请注意，现在，使用此选项进行编译时，编译尝试写入文本字符串内存位置的程序会导致段故障。
extensions	允许零大小的结构/联合声明以及返回语句返回一个值的 void 函数起作用。
extinl	将外部内联函数生成为全局函数。这是缺省值，符合 1999 C 标准。使用 -features=no%extinl 编译新代码可获得与旧版的 C 和 C++ 编译器相同的 extern 内联函数处理方式。
no%extinl	将外部内联函数生成为静态函数。
%none	此选项被禁用。

旧的 C 和 C++ 对象（使用本发行版之前的 Solaris Studio 编译器创建的对象）可以与新的 C 和 C++ 对象链接，而不会更改旧对象的行为。要获得标准的符合规范的行为，您必须使用当前编译器重新编译旧代码。

如果不为 -features 指定设置，编译器将把它设置为 -features=extinl。

B.2.20 -flags

输出每个可用编译器选项的简短摘要。

B.2.21 -flteval[={ any|2}]

(x86) 使用此选项可以控制浮点表达式的求值方式。

表 B–8 -flteval 标志

标志	含义
2	浮点表达式求值为 long double。
any	根据构成表达式的变量和常量类型的组合来对浮点表达式进行求值。

如果未指定 -flteval，编译器会将其设置为 -flteval=any。如果指定了 -flteval 但未提供值，编译器会将其设置为 -flteval=2。

您不能将以下选项与 -flteval=2 一起指定：

- -fprecision
- -nofstore
- -xarch=amd64
- -xarch=sse2

另请参见第 302 页中的 “D.1.1 浮点计算器的精度”。

B.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。

乘加指令可以免除乘法和加法之间的中间舍入步骤。因此，如果使用 **-fma=fused** 编译，程序可能会生成不同的结果，但精度通常会增加而不是降低。

B.2.23 -fnonstd

(SPARC) 此选项是用于 -fns 和 -ftrap=common 的宏。

B.2.24 -fns[={no|yes}]

- (SPARC) 打开 SPARC 非标准浮点模式。

  缺省值为 -fns=no ，即 SPARC 标准浮点模式。-fns 与 -fns=yes 相同。

  可以选择使用 =yes 或 =no，此方法使您可以切换包含 -fns 的其他某个宏标志（如 -fast）后面的 -fns 标志。

在一些 SPARC 系统上，使用非标准浮点模式会禁用“渐进下溢”，导致将微小的结果刷新为零，而不是产生非正规数。它还导致次正规操作数被替换为零而无提示。在那些不支持硬件中的渐进下溢和次正规数的 SPARC 系统上，使用此选项将显著提高某些程序的性能。

在启用了非标准模式的情况下，浮点运算可能会产生不符合 IEEE 754 标准要求的结果。有关详细信息，请参见《数值计算指南》。

此选项仅对 SPARC 系统且仅在编译主程序时使用才有效。在 x86 系统上，忽略此选项。

- (*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。

  此选项不会影响不利用 SSE 或 SSE2 指令集的传统 x86 浮点运算。

B.2.25 -fPIC

与 **-KPIC** 等效

B.2.26 -fpic

与 **-Kpic** 等效

B.2.27 -fprecision=*p*

(*x86*) `-fprecision={single, double, extended}`

将浮点控制字中的舍入精度模式位分别初始化为 single（24 位）、double（53 位）或 extended（64 位）。缺省的浮点舍入精度模式为 extended。

注意，在 x86 中，浮点舍入精度模式的设置只影响精度，不影响指数、范围。

该选项仅在 x86 系统上且仅在编译主程序时才有效，但如果编译 64 位 (`-m64`) 或 SSE2 启动的 (`-xarch=sse2`) 处理器，忽略此选项。在 SPARC 系统上也忽略此选项。

B.2.28 -fround=*r*

设置程序初始化期间在运行时建立的 IEEE 754 舍入模式。

r 必须是以下值之一： `nearest`、 `tozero`、`negative`、`positive`。

缺省值为 `-fround=nearest`。

含义与 ieee_flags 子例程的含义相同。

如果 *r* 为 tozero、negative 或 positive，此标志将在程序开始执行时分别将舍入方向模式设置为舍入到零、舍入到负无穷大或舍入到正无穷大。当 *r* 为 nearest 或未使用 -fround 标志时，舍入方向模式将保留其初始值不变（缺省为舍入到最接近的值）。

只有编译主程序时该选项才有效。

B.2.29 -fsimple[= *n*]

允许优化器进行有关浮点运算的简化假定。

编译器缺省采用 -fsimple=0。指定 -fsimple 与指定 -fsimple=1 等效。

如果存在 *n*，它必须是 0、1 或 2。

表 B–9 -fsimple 标志

值	含义
-fsimple=0	允许无简化假定。保持严格的 IEEE 754 一致性。
-fsimple=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。 ■ 可以删除不生成可见结果（潜在的浮点异常除外）的计算。 ■ 使用无穷大或 NaN 作为操作数的计算无需将 NaN 传送给其结果；例如，x*0 可能替换为 0。 ■ 计算不依赖于零的符号。 如果使用 -fsimple=1，则**不**允许优化器进行完全优化，而不考虑舍入或异常。特别是，在运行时舍入模式包含常量的情况下，浮点计算不能由产生不同结果的计算替换。
-fsimple=2	包含 -fsimple=1 的所有功能，当 -xvector=simd 生效时，还允许使用 SIMD 指令计算约简。 编译器尝试主动浮点优化，这可能导致很多程序因舍入更改而产生不同数值结果。例如，-fsimple=2 允许优化器将给定循环中 x/y 的所有计算都替换为 x*z，其中保证在循环中至少对 x/y 进行一次求值，z=1/y，并且已知 y 和 z 的值在循环执行期间具有常量值。

即使 -fsimple=2，也不允许优化器在不产生任何内容的程序中引入浮点异常。

有关优化对精度的影响的详细说明，请参见由 Rajat Garg 和 Ilya Sharapov 合著的《*Techniques for Optimizing Applications: High Performance Computing*》。

B.2.30 -fsingle

（仅限 -Xt 和 -Xs 模式）使编译器按单精度而非双精度对 float 表达式求值。由于已按单精度对 float 表达式进行求值，因此如果在 -Xa 或 -Xc 模式下使用编译器，此选项将无效。

B.2.31 -fstore

(x86) 将表达式或函数赋值给一个变量时或将表达式强制转换为短浮点类型时，该命令可使编译器将浮点表达式或函数的值转换为赋值语句左侧的类型，而不是在寄存器中保留值。由于舍入和截尾，结果可能会与使用寄存器值生成的结果不同。这是缺省模式。

要关闭此选项，请使用 -nofstore 选项。

B.2.32 -ftrap=*t*[,*t*...]

设置启动时生效的 IEEE 捕获模式，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。

t 可以是下列值之一。

表 B–10 -ftrap 标志

标志	含义
[no%]division	[不] 在除以零时自陷。
[no%]inexact	[不] 在结果不精确时自陷。
[no%]invalid	[不] 在无效操作上自陷。
[no%]overflow	[不] 在溢出上自陷。
[no%]underflow	[不] 在下溢上自陷。
%all	在所有以上内容中自陷。
%none	不在以上任何内容中自陷。

表 B–10 -ftrap 标志 （续）

标志	含义
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。

如果不指定 -ftrap，则编译器假定 -ftrap=%none。

示例：–ftrap=%all,no%inexact 意味着设置所有陷阱，但 inexact 除外。

如果使用 -ftrap=*t* 编译一个例程，就还要使用相同的 -ftrap=*t* 选项来编译该程序的所有例程；否则就会获得意外的结果。

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。

B.2.33 -G

生成共享对象而非动态链接的可执行文件。此选项将传递给 ld(1)，并且无法与 -dn 选项一起使用。

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。

如果通过指定 -G 以及其他必须在编译时和链接时指定的编译器选项来创建共享对象，请确保在与生成的共享对象链接时也指定这些选项。

创建共享对象时，用 -xarch=v9 编译的所有目标文件还必须用显式 -xcode 值（如第 234 页中的 “B.2.86 -xcode[= *v*]”中所记录）进行编译。

B.2.34 -g

为使用 dbx(1) 和性能分析器 analyzer(1) 进行调试生成附加符号表信息。

如果指定 -g，并且优化级别为 -xO3 或更低，则编译器提供几乎进行全面优化的最佳效果符号信息。尾部调用优化和后端内联被禁用。

如果指定 -g，并且优化级别为 -xO4，则编译器提供进行全面优化的最佳效果符号信息。

使用 -g 选项进行编译，以使用性能分析器的全部功能。虽然某些性能分析功能不需要使用 -g，但必须使用 -g 进行编译，以便查看注释的源代码、部分函数级别信息以及编译器注释消息。有关更多信息，请参见 analyzer(1) 手册页和《性能分析器》手册。

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。

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，除非您在命令行上指定了 -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。

有关调试的更多信息，请参见《使用 dbx 调试程序》手册。

B.2.35 -H

输出到标准错误，每行一个错误，包含当前编译期间每个文件的路径名。这样输出的目的是显示包含在其他文件中的文件。

此处，程序 sample.c 包含文件 stdio.h 和 math.h；math.h 包含文件 floatingpoint.h，该文件本身还包含使用 sys/ieeefp.h 的函数：

```
% cc -H sample.c
    /usr/include/stdio.h
    /usr/include/math.h
        /usr/include/floatingpoint.h
            /usr/include/sys/ieeefp.h
```

B.2.36 -h *name*

为动态共享库分配一个名称，从而可以获得一个库的不同版本。通常，-h 后面的 *name* 应与 -o 选项后指定的文件名相同。-h 和 *name* 之间的空格是可选的。

链接程序会为该库分配指定的 *name*，并在库文件中将该名称记录为库的**内在**名称。如果 -h*name* 选项不存在，则库文件中不记录任何内在名称。

当运行时链接程序将库装入可执行文件时，它将内在名称从库文件复制到可执行文件和一组所需共享库文件中。每个可执行文件都有这样的列表。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。

B.2.37 -I[- | *dir*]

`-I` *dir* 将 *dir* 添加到在其中搜索 `#include` 文件（`/usr/include` 前是相对文件名）的目录列表中，也就是说，这些目录路径不以 `/`（斜线）开头。

以指定的顺序搜索多个 `-I` 选项的目录。

有关编译器搜索模式的更多信息，请参见第 55 页中的 “2.16.1 使用 -I- 选项更改搜索算法”。

B.2.38 -i

将选项传递给链接程序，以忽略任何 `LD_LIBRARY_PATH` 或 `LD_LIBRARY_PATH_64` 设置。

B.2.39 -include *filename*

此选项使编译器将 *filename* 视为作为 `#include` 预处理程序指令出现在主源文件的第一行。考虑源文件 `t.c`：

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

如果使用命令 **`cc -include t.h t.c`** 编译 `t.c`，则编译时好像源文件包含以下项：

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

编译器在其中搜索 *filename* 的第一个目录是当前工作目录而不是包含主源文件的目录，这就是显式包括某个文件时的情况。例如，下面的目录结构包含两个名称相同但位置不同的头文件：

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

如果您的工作目录是 `foo/bar`，并且您使用命令 **`cc ../t.c -include t.h`** 进行编译，则编译器会包括 `foo/bar` 中的 `t.h` 而不是 `foo/` 中的此文件，后者是源文件 `t.c` 内包含 `#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，则会在正常目录路径中搜索该文件。如果您指定多个 `-include` 选项，则文件的包括顺序与它们在命令行中的顺序相同。

B.2.40 -KPIC

(SPARC) 已废弃。您不该使用此选项。改用 `-xcode=pic32`。

有关更多信息，请参见第 234 页中的 “B.2.86 -xcode[= *v*]”。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

(x86) `-KPIC` 与 `-Kpic` 相同。

B.2.41 -Kpic

(SPARC) 已废弃。您不该使用此选项。改用 `-xcode=pic13`。有关更多信息，请参见第 234 页中的 “B.2.86 -xcode[= *v*]”。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

(x86) 生成要在共享库中使用的与位置无关的代码。允许最多引用 2**11 个唯一外部符号。

B.2.42 -keeptmp

保留编译期间创建的临时文件，而不自动将其删除。

B.2.43 -L*dir*

将 *dir* 添加到通过 `ld`(1) 在其中搜索库的目录列表中。此选项及其参数将传递给 `ld`(1)。

注 – 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。

B.2.44 -l*name*

与目标库 `lib`*name*`.so` 或 `lib`*name*`.a` 链接。库在命令行中的顺序很重要，因为符号是从左向右进行解析。

此选项必须位于 *sourcefile* 参数之后。

B.2.45 -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 编译的模块必须还使用 -m32 |-m64 进行链接。有关在编译时和链接时都必须指定的编译器选项的完整列表，请参见第 185 页中的 “A.1.2 编译时选项和链接时选项”。

在 x86/x64 平台上使用 -m64 编译具有大量静态数据的应用程序时，可能还需要 -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 Solaris 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。

B.2.46 -mc

从目标文件的 .comment 部分中删除重复字符串。使用 -mc 标志时，会调用 mcs -c。

B.2.47 -misalign

(SPARC) 已废弃。您不该使用此选项。应改用 -xmemalign=1i 选项。有关更多信息，请参见第 249 页中的 “B.2.116 -xmemalign=*ab*”。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

B.2.48 -misalign2

(SPARC) 已废弃。您不该使用此选项。应改用 -xmemalign=2i 选项。有关更多信息，请参见第 249 页中的 “B.2.116 -xmemalign=*ab*”。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

B.2.49 -mr[, *string*]

-mr 可从 .comment 部分中删除所有字符串。使用此标志时，会调用 mcs -d -a。

-mr,*string* 可从 .comment 部分中删除所有字符串，并在目标文件的该部分插入 *string*。如果 *string* 包含嵌入空白，则必须将其括入引号。空 *string* 将导致 .comment 部分为空。此选项将作为 -d -*astring* 传递给 mcs。

B.2.50 -mt[={yes |no}]

使用此选项，可以通过 Solaris 线程或 POSIX 线程 API 编译和链接多线程代码。-mt=yes 选项确保库以正确的顺序链接。

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=yes 选项进行编译。要在 Solaris 平台上使用 POSIX 线程，请包括 pthread.h 头文件并使用 -mt=yes 选项进行编译。

在 Linux 平台上，只有 POSIX 线程 API 可用。（Linux 平台上没有 libthread）。因此，Linux 平台上的 -mt=yes 添加 -lpthread 而不是 -lthread。要在 Linux 平台上使用 POSIX 线程，请使用 -mt 进行编译。

请注意，当使用 -G 进行编译时，-lthread 和 -lpthread 均不会自动包括在 -mt=yes 内。生成共享库时，您需要显式列出这些库。

-xopenmp 选项（用于使用 OpenMP 共享内存并行化 API）自动包含 -mt=yes。

如果使用 -mt=yes 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=yes 选项。如果使用 -mt=yes 编译和链接一个转换单元，则必须使用 -mt=yes 编译和链接该程序的所有单元。

-mt=yes 是编译器的缺省行为。如果不需要该行为，使用 -mt=no 编译。

选项 -mt 等效于 -mt=yes。

B.2.50.1 另请参见

-xnolib、Solaris《多线程编程指南》和《链接程序和库指南》

B.2.51 -native

此选项与 -xtarget=native 意义相同。

B.2.52 -nofstore

(x86) 将表达式或函数赋值给一个变量时或将表达式强制转换为短浮点类型时，该命令不将浮点表达式或函数的值转换为赋值语句左侧的类型，而在寄存器中保留该值。另请参见第 207 页中的 “B.2.31 -fstore”。

B.2.53 -O

使用缺省优化级别 -xO3。-O 宏现在扩展为 -xO3 而非 -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，-x03 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。

B.2.54 -o *filename*

将输出文件命名为 *filename*（与缺省文件名 a.out 相对）。*filename* 的名称不能与 *sourcefile* 相同，因为 cc 不覆盖源文件。此选项及其参数将传递给 ld(1)。

B.2.55 -P

运行 通过 C 预处理程序运行源文件。然后将输出放在带有 .i 后缀的文件中。与 -E 不同，此选项不在输出中包括预处理程序类型的行号信息。另请参见 -E 选项。

B.2.56 -p

已废弃，请参见第 262 页中的 “B.2.132 -xpg”。

B.2.57 -Q[y|n]

向或不向输出文件发出标识信息。-Qy 是缺省值。

如果使用 -Qy，则将关于每个调用的编译工具的标识信息添加到输出文件的 .comment 部分，使用 mcs 可访问这部分文件。此选项对软件管理十分有用。

-Qn 可抑制此信息。

B.2.58 -qp

与 -p 相同。

B.2.59 -R*dir*[:*dir*]

将用来指定库搜索目录的、冒号分隔的目录列表传递给运行时链接程序。如果此选项存在且非空，则它记录在输出目标文件中并传递给运行时链接程序。

如果同时指定了 LD_RUN_PATH 和 -R 选项，则 -R 选项优先。

B.2.60 -S

指示 cc 生成汇编源文件，而不汇编程序。

B.2.61 -s

从输出目标文件中删除所有符号调试信息。此选项不能与 -g 一同指定。

传递给 ld(1)。

B.2.62 -traceback[={ %none|common|*signals_list*}]

如果执行中出现严重错误，将发出栈追踪。

当程序生成某些信号时，-traceback 选项会导致可执行文件向 stderr 发出栈跟踪、转储内核并退出。如果多个线程都生成一个信号，则只为第一个生成栈跟踪。

要使用回溯，请在链接时将 -traceback 选项添加到编译器命令行中。编译时也接受该选项，除非生成可执行二进制文件，否则将忽略此选项。使用 -traceback 和 -G 创建共享库是个错误。

表 B–11 -traceback 选项

选项	含义
common	指定在出现下列任一常见信号时发出栈跟踪：sigill、sigfpe、sigbus、sigsegv 或 sigabrt。
signals_list	指定应生成栈跟踪的信号名称的逗号分隔列表，采用小写形式。可以捕捉以下信号（导致生成核心文件的信号）：sigquit、sigill、sigtrap、sigabrt、sigemt、sigfpe、sigbus、sigsegv、sigsys、sigxcpu、sigxfsz。 在上述任一信号前加上 no% 可以禁用信号缓存。 例如：**-traceback=sigsegv,sigfpe** 将在 sigsegv 或 sigfpe 出现时生成栈跟踪和核心转储。
%none 或 none	禁用回溯

如果不指定该选项，则缺省值为 -traceback=%none

只使用 -traceback 而不使用 = 信号表示 -traceback=common

注意：如果不需要核心转储，用户可以使用以下方法将 coredumpsize 限制设置为零：

```
% limit coredumpsize 0
```

-traceback 选项对运行时性能没有影响。

B.2.63 -U*name*

取消定义预处理程序符号 *name*。此选项可删除由 -D 在命令行上创建的预处理程序符号 *name* 的任何初始定义，包括那些由命令行驱动程序放置的定义。

-U 对源文件中的预处理程序指令没有任何影响。可以在命令行上指定多个 -U 选项。

如果在命令行中为 -*D* 和 -U 指定了相同的 name，则取消定义 *name*，而无论这些选项出现的顺序如何。在以下示例中，-U 取消定义 __sun：

```
cc -U__sun text.c
```

由于已取消定义 __sun，因此在 test.c 中采用以下形式的预处理程序语句将不会生效。

```
#ifdef(__sun)
```

有关预定义符号的列表，请参见第 197 页中的 “B.2.7 -D*name*[(*arg*[,*arg*])][= *expansion*]”。

B.2.64 -V

指示 cc 在编译器执行时输出每个组件的名称和版本 ID。

B.2.65 -v

指示编译器执行更严格的语义检查并启用其他与 lint 类似的检查。例如，如下所示代码：

```
#include <stdio.h>
main(void)
{
     printf("Hello World.\n");
}
```

编译和执行时不会出现问题。如果使用 -v，该代码仍可编译；但是，编译器会显示以下警告：

```
"hello.c", line 5: warning: function has no return statement:
 main
```

-v 不能给出 lint(1) 给出的所有警告。尝试通过 lint 运行以上示例。

B.2.66 -W*c* ,*arg*

将参数 *arg* 传递给指定的组件 *c*。有关组件的列表，请参见表 1–1。

每个参数与前一个参数之间仅以逗号分隔。所有 -W 参数均在常规命令行参数之后进行传递。在紧靠逗号之前使用转义符 \（反斜杠）可将逗号作为参数的一部分。所有 -W 参数均在常规命令行参数之后进行传递。

例如，-Wa,-o,objfile 按顺序将 -o 和 objfile 传递给汇编程序。此外，-Wl,-I,*name* 将导致链接阶段覆盖动态链接程序的缺省名称 /usr/lib/ld.so.1。

参数传递到工具的顺序可能会因其他指定的命令行选项而更改。

c 可以是以下项之一：

表 B–12 -W 标志

标志	含义
a	汇编程序：(fbe); (gas)
c	C 代码生成器：(cg) *(SPARC)*;
d	cc 驱动程序
h	中间代码翻译者 (ir2hf)(*x86*)
l	链接编辑器 (ld)
m	mcs
O（大写的 o）	过程间优化器
o（小写的 o）	后优化器
p	预处理程序 (cpp)
u	C 代码生成器 (ube) (*x86*)
0 (Zero)	编译器 (acomp) (ssbd, *SPARC*)
2	优化器：(iropt)

B.2.67 -w

抑制编译器警告消息。

此选项会覆盖 error_messages pragma。

B.2.68 -X[c|a| t|s]

-X（注意大写X）选项指定符合 ISO C 标准的各种级别。-xc99 的值影响 -X 选项所应用的 ISO C 标准的版本。-xc99 选项的缺省值为支持 1999 ISO/IEC C 标准的 -xc99=all。-xc99=none 支持 1990 ISO/IEC C 标准。有关对 1999 ISO/IEC 支持功能的讨论，请参见表 C–6。有关 ISO/IEC C 与 K&R C 的不同点的讨论，请参见第 345 页中的 “G.1 libfast.a 库 (SPARC)”。

缺省模式是 -Xa。

-Xc

（c =一致性）对使用非 ISO C 构造的程序发出错误和警告。此选项与 ISO C 严格一致，没有 K&R C 兼容性扩展。如果使用 -Xc 选项，预定义的宏 __STDC__ 的值为 1。

-Xa

这是缺省编译器模式。ISO C 以及 K&R C 兼容性扩展，具有 ISO C 要求的语义更改。如果 K&R C 和 ISO C 为同一构造指定不同语义，则编译器使用 ISO C 解释。如果 -Xa 选项与 -xtransition 选项配合使用，则编译器发出关于不同语义的警告。如果使用 -Xa 选项，预定义的宏 __STDC__ 的值为 -0。

-Xt

(t = transition) 该选项使用 ISO C 以及 K&R C 兼容性扩展，**不具有** ISO C 要求的语义更改。 如果 K&R C 和 ISO C 为同一构造指定不同语义，则编译器使用 ISO C 解释。如果将 -Xt 选项与 -xtransition 选项配合使用，则编译器发出关于不同语义的警告。使用 -Xt 选项时，预定义的宏 __STDC__ 的值为 0。

-Xs

(s = K&R C) 试图对 ISO C 和 K&R C 产生不同行为的所有语言构造发出警告。编译器语言包括与 K&R C 兼容的所有功能。此选项调用 cpp 以进行预处理。__STDC__ 在此模式下未定义。

B.2.69 -x386

(x86) 已废弃。您不该使用此选项。改用 -xchip=generic。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

B.2.70 -x486

(x86) 已废弃。您不该使用此选项。改用 `-xchip=generic`。有关废弃选项的完整列表，请参见第 193 页中的 “A.1.15 废弃的选项”。

B.2.71 -xaddr32[=yes|no]

（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为 32 位地址空间。

以这种方式编译的可执行文件会导致创建限定为 32 位地址空间的进程。

当指定了 **`-xaddr32=no`** 时，会产生通常的 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`**，则使用 **`-xaddr32=yes`**。

此选项仅适用于 **`-m64`** 编译，并且仅仅在支持 `SF1_SUNW_ADDR32` 软件功能的 Solaris 平台上适用。由于 Linux 内核不支持地址空间限制，因此该选项在 Linux 上不可用。

链接时，如果单个目标文件是使用 **`-xaddr32=yes`** 编译的，则假定整个输出文件是使用 **`-xaddr32=yes`** 编译的。

限定为 32 位地址空间的共享对象必须由在受限的 32 位模式地址空间内执行的进程装入。

有关详细信息，请参阅《链接程序和库指南》中的 `SF1_SUNW_ADDR32` 软件功能定义。

B.2.72 -xalias_level[= *l*]

编译器使用 `-xalias_level` 选项确定为了使用基于类型的别名分析执行优化可以进行何种假设。此选项使指定的别名级别对正在编译的转换单元生效。

如果不指定 `-xalias_level` 命令，编译器将假定 `-xalias_level=any`。如果指定不带值的 `-xalias_level`，则缺省值为 `-xalias_level=layout`。

`-xalias_level` 选项要求的优化级别为 `-xO3` 或更高。如果优化级别设置较低，则发出警告并忽略 `-xalias_level` 选项。

切记，如果使用 `-xalias_level` 选项，但无法坚持为任何别名级别描述的关于别名的所有假定和约束，则程序的行为未定义。

将 *l* 替换为下表中的某一术语。

表 B–13 别名歧义消除级别

标志	含义
any	编译器假定所有内存引用都可在此级别上互为别名。在级别 -xalias_level=any 上，不存在基于类型的别名分析。
basic	如果使用 -xalias_level=basic 选项，编译器将假定调用不同 C 基本类型的内存引用不互为别名。此外，编译器还假定对其他所有类型的引用互为别名，并可以使用任何 C 基本类型作为别名。编译器假定使用 char * 的引用可以使用任何其他类型的引用作为别名。 例如，在 -xalias_level=basic 级别上，编译器假定类型为 int * 的指针变量不会访问浮点对象。因此，编译器可安全执行优化，该优化假定类型为 float * 的指针不会使用 int * 类型指针引用的相同内存作为别名。
weak	如果使用 -xalias_level=weak 选项，编译器会假定任何结构指针都可指向任何结构类型。 任何结构或联合类型，只要它包含对编译的源代码的表达式中引用的任何类型的引用，或者包含对从编译的源代码外部引用的任何类型的引用，就必须在编译的源代码中的表达式之前进行声明。 您可以通过包含某个程序的所有头文件来满足此约束，这些头文件包含的类型引用了在所编译的源代码的表达式中引用的对象的类型。 在级别 -xalias_level=weak 上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用使用涉及任何其他类型的内存引用作为别名。
layout	如果使用 -xalias_level=layout 选项，编译器将假定所涉及类型与内存中类型具有相同顺序的内存引用可以互为别名。 编译器假定其类型在内存中看起来并不相同的两个引用并不互为别名。如果初始结构成员在内存中看起来相同，则编译器假定任何两个内存均通过不同的结构类型进行访问。然而，在此级别上，不应使用指向结构的指针来访问不同结构对象的某字段，该字段不属于这两个结构之间在内存中看起来相同的公共初始成员序列。这是因为编译器假定此类引用并不互为别名。 在 -xalias_level=layout 级别上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用可以使用涉及其他类型的内存引用作为别名。

表 B–13 别名歧义消除级别 （续）

标志	含义
strict	如果使用 -xalias_level=strict 选项，编译器将假定涉及结构或联合等类型并且在删除标记后相同的内存引用可以互为别名。反之，编译器假定涉及那些即使在删除标记后也不相同的类型的内存引用不互为别名。 然而，任何结构或联合类型，只要它包含对编译的源代码的表达式中引用的任何类型的引用，或者包含对从编译的源代码外部引用的任何类型的引用，就必须在编译的源代码中该表达式之前进行声明。 您可以通过包含某个程序的所有头文件来满足此约束，这些头文件包含的类型引用了在所编译的源代码的表达式中引用的对象的类型。在 -xalias_level=strict 级别上，编译器假定涉及不同 C 基本类型的内存引用并不互为别名。编译器假定使用 char * 的引用可以使用任何其他类型的引用作为别名。
std	如果使用 -xalias_level=std 选项，编译器将假定类型和标记必须相同才能作为别名，但是，使用 char * 的引用可以使用涉及其他任何类型的引用作为别名。此规则与 1999 ISO C 标准中对指针解除引用的限制相同。正确使用此规则的程序将非常易于移植，而且优化之后性能大大提高。
strong	如果使用 -xalias_level=strong 选项，则与 std 级别应用相同限制，但此外，编译器还假定类型 char * 的指针只用来访问类型为 char 的对象。此外，编译器还假定不存在内部指针。内部指针被定义为指向结构成员的指针。

B.2.73 - xannotate[=yes|no]

(Solaris) 指示编译器创建以后可由诸如 binopt(1) 之类的二进制修改工具转换的二进制文件。使用 -xannotate=no 选项可以阻止这些工具修改该二进制文件。-xannotate=yes 选项必须与 --xO1 或更高的优化级别一起使用时才有效。

此选项在 Linux 平台上不可用。

B.2.74 –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。

B.2.74.1 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 B–14 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。
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。可使用 -m64 指定 64 位编译，来取代 - 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。
sparcvis3	针对 SPARC-V 9 ISA 的 SPARC VIS 版本 3 进行编译。允许编译器使用 SPARC-V9 指令集、UltraSPARC 扩展（包括可视指令集 (Visual Instruction Set, VIS) 版本 1.0、UltraSPARC-III 扩展（包括可视指令集版本 2.0 以及混合乘加指令和可视指令集版本 3.0 中的指令。
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。
sparcima	针对 SPARC-V9 ISA 的 sparcima 版本进行编译。使编译器可以使用如下指令集内的指令：SPARC-V9 指令集、UltraSPARC 扩展（包括可视化指令集 (Visual Instruction Set, VIS) 版本 1.0）、UltraSPARC-III 扩展（包括可视化指令集 (Visual Instruction Set, VIS) 版本 2.0）、SPARC64 VI 扩展（用于浮点乘加）和 SPARC64 VII 扩展（用于整数乘加）。

表 B–14 用于 SPARC 平台的 -xarch 标志 （续）

标志	含义
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：

- 用 generic,sparc, sparcvis2, sparcvis3, sparcfmaf, sparcima 编译的对象二进制文件 (.o) 可以链接起来并一起执行，但只能在支持链接的所有指令集的处理器上运行。
- 对于任何特定选择，生成的可执行文件在传统体系结构中可能不运行或运行缓慢。而且，由于所有指令集中均未实现四精度 (long double) 浮点指令，因此编译器不在其生成的代码中使用这些指令。

B.2.74.2 用于 x86 的 -xarch 标志

下表列出了 x86 平台上的 -xarch 标志。

表 B–15 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。可使用 -m64 指定 64 位编译，来取代 - 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。

表 B–15 x86 上的 -xarch 标志 （续）

标志	含义
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	使用 SSSE3 指令集补充 pentium_pro、SSE、SSE2 和 SSE3 指令集。
sse4_1	使用 SSE4.1 指令集补充 pentium_pro、SSE、SSE2、SSE3 和 SSSE3 指令集。
sse4_2	使用 SSE4.2 指令集补充 pentium_pro、SSE、SSE2、SSE3、SSSE3 和 SSE4.1 指令集。
amdsse4a	使用 AMD SSE4a 指令集。

B.2.74.3 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。

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）。

因为内部数学库（例如，sin(x)）不同，所以 Solaris 和 Linux 之间的数值结果也会不同。

B.2.74.4 二进制兼容验证

从 Solaris 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，看其是否在相应的平台上运行。

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。

B.2.74.5 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=sparcvis -xchip=ultra2 -xcache=16/32/1:512/64/1。在以下命令中，-xarch=generic 覆盖了由 -xtarget=ultra2 的扩展设置的 -xarch=sparcvis。

```
example% cc -xtarget=ultra2 -xarch=generic foo.c
```

B.2.74.6 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。

B.2.75 -xautopar

注 – 此选项不能启动 OpenMP 并行化指令。

为多个处理程序打开自动并行化。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 或更高级别，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《Solaris Studio 12 Update 1： 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，而且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。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

B.2.76 -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。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。

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。

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。缺省值为 -xbinopt=off。

B.2.77 -xbuiltin[=(%all|%none)]

如果要改进调用标准库函数的代码的优化，请使用 -xbuiltin[=(%all|%none)] 命令。很多标准库函数（例如在 math.h 和 stdio.h 中定义的函数）通常由多个程序使用。此命令使编译器可在对性能有益时替换内函数或内联系统函数。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。

但是，这些替换会使 errno 的设置变得不可靠。如果您的程序依赖于 errno 的值，请不要使用此选项。另请参见第 51 页中的 “2.13 保留 errno 的值”。

如果不指定 -xbuiltin，则缺省值为 -xbuiltin=%none，该值表示不替换或内联标准库中的任何函数。如果指定 -xbuiltin，但未提供任何参数，则缺省值为 -xbuiltin%all，该值表示编译器在确定优化好处时替换内部函数或内联标准库函数。

如果使用 -fast 进行编译，则 -xbuiltin 设置为 %all。

注 – -xbuiltin 仅内联系统头文件中定义的全局函数，从不内联用户定义的静态函数。

B.2.78 -xCC

当指定 -xc99=none 和 -xCC 时，编译器将接受 C++ 样式注释。特别是，可使用 // 指示注释的开始。

B.2.79 -xc99[= *o*]

-xc99 选项可控制编译器对根据 C99 标准（ISO/IEC 9899:1999，编程语言 - C）实现的功能的识别。

o 可以是包含以下内容的以逗号分隔的列表：

表 B–16 -xc99 标志

标志	含义
[no]_lib	[不] 启用同时在 1990 和 1999 C 标准中出现的例程的 1999 C 标准例程库语义。
all	打开识别支持的 C99 语言功能，并启用同时在 1990 和 1999 C 标准中出现的例程的 1999 C 标准库语义。
none	关闭识别支持的 C99 语言功能，并且不启用同时在 1990 和 1999 C 标准中出现的例程的 1999 C 标准库语义。

如果未指定 -xc99，编译器缺省采用 -xc99=all,no_lib。如果指定了 -xc99，但没有指定任何值，该选项将设置为 -xc99=all。

注 – 虽然编译器支持级别缺省为 C99 标准的语言功能，但 /usr/include 中的 Solaris 8 和 Solaris 9 操作系统提供的标准头文件不符合 1999 ISO/IEC C 标准。如果遇到错误消息，请尝试指定 -xc99=none，从而获取这些头文件的 1990 ISO/IEC C 标准行为。

出现在 1999 和 1999 C 标准中的 1990 C 标准例程库语义不可用，因此无法在 Solaris 8 和 Solaris 9 软件中启用。在 Solaris 8 或 Solaris 9 软件上直接或间接指定 -xc99=lib 时，编译器会发出错误消息。

B.2.80 -xcache[= *c*]

定义可供优化器使用的缓存属性。此选项不保证使用每个特定的缓存属性。

注 – 尽管该选项可单独使用，但它是 -xtarget 选项扩展的一部分；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 [/t*i*]，该属性用来设置可以共享缓存的线程数。如果没有为 t 指定值，则缺省值为 1。

c 必须是以下值之一：

- generic

- native
- *s*1/ *l*1/*a* 1[/*t*1]
- *s*1/ *l*1/*a* 1[/*t*1]: *s*2/*l* 2/*a*2[/*t*2]
- *s*1/ *l*1/*a* 1[/*t*1]: *s*2/*l* 2/*a*2[/*t*2]: *s*3/*l* 3/*a*3[/*t*3]

s/*l* /*a*/*t* 属性定义如下：

si	级别为 *i* 的数据高速缓存的大小，以千字节为单位
li	级别为 *i* 的数据高速缓存的行大小，以字节为单位
ai	级别为 *i* 的数据高速缓存的关联性
ti	共享级别为 *i* 的缓存的硬件线程数

下表列出了 `-xcache` 值。

表 B–17 `-xcache` 标志

标志	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 1 级高速缓存属性。
s1/*l1* /*a1*[/*t*1] :*s2*/*l2* /*a2*[/*t* 2]	定义 1 级和 2 级高速缓存属性。
s1/*l1* /*a1*[/*t*1] :*s2*/*l2* /*a2*[/*t* 2]:*s3*/ *l3*/*a3*[/ *t*3]	定义 1 级、2 级和 3 级缓存属性。

示例： `-xcache=16/32/4:1024/32/1` 指定以下内容：

级别 1 缓存具有以下属性：	2 级缓存具有：
16 千字节	1024 千字节
32 字节行大小	32 字节行大小
4 方向关联	指示映射关联

B.2.81 –xcg[89|92]

(SPARC) 已废弃。您不该使用此选项。当前的 Solaris 操作系统不再支持 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。

B.2.82 -xchar[= *o*]

提供此选项仅仅是为了便于从将字符类型定义为无符号的系统中迁移代码。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。

o 可以是下列值之一：

表 B–18 `-xchar` 标志

标志	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。

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`。

`-xchar` 选项仅更改用 `-xchar` 编译的代码中类型 char 的值范围。该选项不更改任何系统例程或头文件中类型 char 的值范围。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` 不再表示无格式字符中可编码的值的范围。

如果使用 -xchar，则在将字符与预定义的系统宏进行比较时要特别小心，原因是宏中的值可能带符号。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，因此在将字符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 将类型 char 指定为带符号，并且系统库也按此指定。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，而修改您的代码使其不依赖于类型 char 是否带符号。类型 char 的符号因不同的编译器和操作系统而不同。

B.2.83 -xchar_byte_order[= *o*]

以指定的字节顺序放置多字符字符常量构成的字符，以生成一个整型常量。可将 *o* 替换成下列值之一：

- low：按由低到高的字节顺序放置多字符字符常量构成的字符。
- high：按由高到低的字节顺序放置多字符字符常量构成的字符。
- default：按编译模式第 218 页中的 “B.2.68 -X[c|a| t|s]”所确定的顺序放置多字符字符常量构成的字符。有关更多信息，请参见第 32 页中的 “2.1.2 字符常量”。

B.2.84 -xcheck[= *o*]

添加针对栈溢出的运行时检查并初始化局部变量。

可将 *o* 替换成下列值之一：

表 B–19 -xcheck 标志

标志	含义
%none	不执行任何 -xcheck 检查。
%all	执行全部 -xcheck 检查。
stkovf	启用栈溢出检查。-xcheck=stkovf 将添加针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的栈溢出运行时检查。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)。
no%stkovf	关闭栈溢出检查。
init_local	初始化局部变量。有关详细信息，请参见以下介绍。
no%init_local	不初始化局部变量。

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=%all。

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。

B.2.84.1 -xcheck=init_local 的初始化值

使用 -xcheck=init_local，编译器在没有初始化程序的情况下，将声明的局部变量初始化为下表中所示的预定义值：（注意这些值会发生更改，因此不应该依赖它们。）

基本类型

表 B–20 基本类型的 init_local 初始化

类型	初始化值
Char, __Bool	0x85
short	0x8001
int, long, enum (-m32)	0xff80002b
long (-m64)	0xfff00031ff800033
long long	0xfff00031ff800033
pointer	0x00000001 (-m32)
	0x0000000000000001 (-m64)
float, float _Imaginary	0xff800001
float _Complex	0xff80000fff800011
double	SPARC: 0xfff00003ff800005
	x86: 0xfff00005ff800003
double _Imaginary	0xfff00013ff800015
long double, long double _Imaginary	SPARC: 0xffff0007ff800009 / 0xfff0000bff80000d
	x86: 12 bytes (-m32): 0x80000009ff800005 / 0x0000ffff
	x86 - 16 bytes (-m64): 0x80000009ff800005 / 0x0000ffff00000000
double _Complex	0xfff00013ff800015 / 0xfff00017ff800019

表 B–20 基本类型的 init_local 初始化 （续）

类型	初始化值
long double _Complex	SPARC: 0xffff001bff80001d / 0xfff0001fff800021 / 0xffff0023ff800025 / 0xfff00027ff800029 x86 - 12 bytes (-m32): 0x7fffb01bff80001d / 0x00007fff / 0x7fffb023ff800025 / 0x00007fff x86 - 16 bytes (-m64): 0x00007fff00080000 / 0x1b1d1f2100000000 / 0x00007fff00080000 / 0x2927252300000000

注意，为结合计算的 goto 使用而声明的局部变量（即简单的 void * 指针），将根据上表中所述的指针说明进行初始化。

永远不会初始化以下局部变量类型：限定的 const、register、计算的 goto 的标签号、局部标签、可变长度数组 (VLA)

初始化结构、联合和数组

struct 中作为基本类型的域将根据上表中所述进行初始化，union 中第一个声明的 pointer 或 float 域也是如此。这样便最大程度地增加了未初始化引用生成可见错误的可能性。

数组元素也按上表所述进行初始化。

按如上所述对内嵌 struct、union、数组域进行初始化，但以下情况除外：struct 包含位域，union 没有 pointer 或 float 域，或者数组类型无法进行完整的初始化。将使用用于类型 double 的局部变量的值对它们进行初始化。不对可变长度组数进行初始化。

B.2.85 -xchip[= *c*]

指定供优化器使用的目标处理器。

c 必须是以下值之一：
generic、old、super、super2、micro、micro2、hyper、hyper2、powerup、ultra、ultra2、ultra2e、ultra2i、ultra3、ultra3cu、pentium、pentium_pro。

尽管该选项可单独使用，但它是 -xtarget 选项扩展的一部分；其**主要**用途是覆盖 -xtarget 选项提供的值。

此选项通过指定目标处理器来指定计时属性。其部分影响表现在以下方面：

- 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

下表列出了用于 SPARC 平台的 -xchip 值：

表 B–21 SPARC -xchip 标志

标志	含义
generic	使用计时属性，以便在大多数 SPARC 体系结构上获得良好性能。 这是缺省值，该值指示编译器使用最佳计时属性以便在多数处理器上获得良好性能，而不会使任何处理器性能明显下降。
native	设置在主机环境中最佳性能的参数。
old	使用 SuperSPARC 以前的处理器的计时属性。
sparc64vi	针对 SPARC64 VI 处理器进行优化。
sparc64vii	针对 SPARC64 VII 处理器进行优化
super	使用 SuperSPARC 处理器的计时属性。
super2	使用 SuperSPARC II 处理器的计时属性。
micro	使用 microSPARC 处理器的计时属性。
micro2	使用 microSPARC II 处理器的计时属性。
hyper	使用 hyperSPARC 处理器的计时属性。
hyper2	使用 hyperSPARC II 处理器的计时属性。
powerup	使用 Weitek PowerUp 处理器的计时属性。
ultra	使用 UltraSPARC 处理器的计时属性。
ultra2	使用 UltraSPARC II 处理器的计时属性。
ultra2e	使用 UltraSPARC IIe 处理器的计时属性。
ultra2i	使用 UltraSPARC IIi 处理器的计时属性。
ultra3	使用 UltraSPARC III 处理器的计时属性。
ultra3cu	使用 UltraSPARC III Cu 处理器的计时属性。
ultra3i	使用 UltraSPARC IIIi 处理器的计时属性。
ultra4	使用 UltraSPARC IV 处理器的计时属性。
ultra4plus	使用 UltraSPARC IVplus 处理器的计时属性。
ultraT1	使用 UltraSPARC T1 处理器的计时属性。
ultraT2	使用 UltraSPARC T2 处理器的计时属性。
ultraT2plus	使用 UltraSPARC T2+ 处理器的计时属性。

表 B–21 SPARC -xchip 标志 （续）

标志	含义
ultraT3	使用 UltraSPARC T3 处理器的计时属性。

下表列出了用于 x86 平台的 -xchip 值：

表 B–22 x86 -xchip 标志

标志	含义
generic	使用计时属性，以便在大多数 x86 体系结构上获得良好性能。 这是缺省值，该值指示编译器使用最佳计时属性以便在多数处理器上获得良好性能，而不会使任何处理器性能明显下降。
native	设置在主机环境中最佳性能的参数。
core2	针对 Intel Core2 处理器进行优化。
nehalem	针对 Intel Nehalem 处理器进行优化。
opteron	针对 AMD Opteron 处理器进行优化。
penryn	针对 Intel Penryn 处理器进行优化。
pentium	使用 x86 pentium 体系结构的计时属性。
pentium_pro	使用 x86 pentium_pro 体系结构的计时属性。
pentium3	使用 x86 Pentium 3 体系结构的计时属性。
pentium4	使用 x86 Pentium 4 体系结构的计时属性。
amdfam10	针对 AMD AMDFAM10 处理器进行优化。

B.2.86 -xcode[= *v*]

(SPARC) 指定代码地址空间。

注 – 强烈建议您通过指定 -xcode=pic13 或 -xcode=pic32 来生成共享对象。可以使用 -xarch=v9 -xcode=abs64 和 -xarch=v8 -xcode=abs32 生成可用的共享对象，但这样做效率很低。用 -xarch=v9、-xcode=abs32 或 -xarch=v9、-xcode=abs44 生成的共享对象无法工作。

---

v 必须是以下项之一：

表 B–23 -xcode 标志

值	含义
abs32	这是 32 位体系结构的缺省值。生成 32 位绝对地址。代码 + 数据 + bss 的大小被限制为 2**32 字节。
abs44	这是 64 位体系结构的缺省值。生成 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 位体系结构。
abs64	生成 64 位绝对地址。只适用于 64 位架构。
pic13	生成共享库（小模型）中使用的与位置无关的代码。与 -Kpic 等效。允许在 32 位体系结构中最多引用 2**11 个唯一的外部符号，而在 64 位体系结构中可以最多引用 2**10 个。
pic32	生成共享库（大模型）中使用的与位置无关的代码。与 -KPIC 等效。允许在 32 位体系结构中最多引用 2**30 个唯一的外部符号，而在 64 位体系结构中可以最多引用 2**29 个。

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，存在两项名义性能成本。

- 用 -xcode=pic13 或 -xcode=pic32 编译的例程会在入口点执行一些附加指令，以便将寄存器设置为指向用于访问共享库的全局或静态变量的表 (_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。如果编译包括 -xcode=pic32，则每个全局和静态内存引用中都会有两个附加指令。

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 .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）。

要确定使用 -xcode=pic13 还是 -xcode=pic32，请使用 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) 的大小并查找节标题 sh_name: .got。sh_size 值是 GOT 的大小。如果 GOT 小于 8,192 字节，请指定 -xcode=pic13，否则指定 -xcode=pic32。

通常，应根据以下准则来确定如何使用 -xcode：

- 如果是生成可执行文件，则不应该使用 – xcode=pic13 或 -xcode=pic32。
- 如果是生成仅用于链接到可执行文件的归档库，则不应该使用 -xcode=pic13 或 -xcode=pic32。
- 如果要生成共享库，请先使用 -xcode=pic13，一旦 GOT 大小超过 8,192 字节，再使用 -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。

B.2.87 -xcrossfile

已废弃，不使用。改用 -xipo。-xcrossfile 是 -xipo=1 的别名。

B.2.88 -xcsi

允许 C 编译器接受在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。这些语言环境包括： ja_JP.PCK。

处理此类语言环境所需的编译器转换阶段可能导致编译时间明显延长。仅当编译的源文件包含此类语言环境中某个语言环境的源代码字符时，才应该使用此选项。

除非指定 -xcsi，否则编译器不识别在不符合 ISO C 源字符代码要求的语言环境中编写的源代码。

B.2.89 -xdebugformat=[stabs|dwarf]

如果您维护会读取 dwarf 格式的调试信息的软件，请指定 -xdebugformat=dwarf。使用此选项，编译器会生成使用 dwarf 标准格式的调试信息，这是缺省设置。

表 B–24 -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 使用 dwarf 标准格式（缺省设置）生成调试信息。

如果未指定 -xdebugformat，编译器将假定 -xdebugformat=dwarf。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。

有关更多信息，另请参见 dumpstabs(1) 和 dwarfdump(1) 手册页。

B.2.90 -xdepend=[yes| no]

分析循环以了解迭代间数据依赖性并执行循环重构，包括循环交互、循环熔合以及标量替换。

对于所有优化级别 -xO3 以及更高级别， -xdepend 缺省为 -xdepend=on。指定 -xdepend 的显式设置会覆盖任何缺省设置。

指定不带参数的 -xdepend 等效于 -xdepend=yes。

依赖性分析在单处理器系统中可能很有用。但是，如果您在单处理器系统上使用 -xdepend，则不应同时指定 -xautopar，否则会针对多处理器系统执行 -xdepend 优化。

B.2.91 -xdryrun

此选项是用于 -### 的宏。

B.2.92 -xe

对源文件仅执行语法和语义检查，但不生成任何目标或可执行代码。

B.2.93 -xF[=*v*[,*v*...]]

由链接程序对函数和变量进行最优的重新排序。

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》手册中关于如何生成函数的映射文件（或“链接程序和库向导”中关于如何生成数据的映射文件）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。

B.2.93.1 值

v 可以是下列其中一个或多个值：

表 B–25 -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。

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。

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。

请参见 analyzer(1) 和 ld(1) 手册页。

B.2.94 -xhelp=*f*

显示联机帮助信息。

f 必须为 flags 或 readme。

-xhelp=flags 显示编译器选项汇总信息。

-xhelp=readme 显示 README 文件。

B.2.95 -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。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

-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)。

-xhwcprof 和 -g 的组合会增加编译器临时文件的存储需求，而且比单独指定 -xhwcprof 和 -g 所引起的增加总量还多。

下列命令可编译 example.c，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.c
```

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。

B.2.96 -xinline=*list*

用于 *-xinline* 的 list 的格式如下所示：[{%auto,*func_name* ,no%*func_name*}[,{%auto ,*func_name*,no%*func_name*}]...]

-xinline 尝试只内联可选列表中指定的函数。该列表可为空，或者由逗号分隔的 func_name、no%func_name 或 %auto 列表组成，其中 *func_name* 是函数名称。仅在 -xO3 或更高级别上，-xinline 才起作用。

表 B–26 -xinline 标志

标志	含义
%auto	指定编译器将尝试自动内联源文件中的所有函数。仅在 -xO4 或更高优化级别上，%auto 才起作用。在 -xO3 或更低优化级别上，%auto 被忽略而无提示。

表 B–26 -xinline 标志 （续）

标志	含义
func_name	指定编译器内联已命名的函数。
no%func_name	指定编译器不内联已命名的函数。

该列表值从左至右进行累积。因此对于 -xinline=%auto,no%foo 的规范，编译器试图内联除 foo 之外的所有函数。对于 -xinline=%bar,%myfunc,no%bar 的规范，编译器仅尝试内联 myfunc。

在优化级别为 -xO4 或更高级别上编译时，编译器通常尝试内联源文件中定义的对函数的所有引用。通过指定 -xinline 选项，您可以限定编译器试图内联的函数集。如果只指定 -xinline=，即不指定任何函数或 %auto，这表示不内联源文件中的任何例程。如果您指定了一系列 func_name 和 no%func_name，而未指定 %auto，则编译器只试图内联列表中指定的函数。如果在优化级别设置为 -xO4 或更高级别时用 -xinline 选项在值列表中指定 %auto，编译器将试图内联所有未被 no%func_name 显式排除的函数。

如果以下任何条件适用，则不内联函数。且不发出任何警告。

- 优化级别低于 -xO3。
- 无法找到例程。
- 优化器无法内联例程。
- 正编译的文件中不存在该例程的源代码（请参见 -xcrossfile）。

如果在命令行指定多个 -xinline 选项，它们将不会累积。命令行上的最后一个 -xinline 指定编译器试图内联的函数。

另请参见 -xldscope。

B.2.97 -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。

B.2.98 -xipo[= *a*]

用 *0*、1 或 2 替换 a。没有任何参数的 -xipo 等效于 -xipo=1。-xipo=0 是缺省设置，表示关闭 -xipo。在 -xipo=1 时，编译器会跨所有源文件执行内联。

在 -xipo=2 时，编译器执行过程间调用别名分析同时优化内存分配和布局，以提高缓存的性能。

编译器可通过调用过程间分析组件执行部分程序优化。它在链接步骤中跨所有目标文件执行优化，并且不限于编译命令的源文件。但是，使用 -xipo 执行的整个程序优化不包括汇编 (.s) 源文件。

编译时和链接时都必须指定 -xipo。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

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。在编译步骤中创建的目标文件具有在这些文件内编译的附加分析信息，这样就可以在链接步骤中执行跨文件优化。

在编译和链接大型多文件应用程序时，-xipo 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。

-xipo 是多阶段的，因此如果要在单独的步骤中进行编译和链接，需要为每一步指定 -xipo。

关于 -xipo 的其他重要信息：

- 它要求优化级别最低为 -xO4。
- 编译时未使用 -xipo 的对象可以自由地与使用 -xipo 编译的对象链接。

B.2.98.1 示例

在此例中，编译和链接在单独的步骤中进行：

```
cc -xipo -xO4 -o prog part1.c part2.c part3.c
```

优化器在三个源文件之间执行交叉文件内联。这是在最终链接步骤中完成的，因此源文件的编译不必全部在单个编译中进行，可以通过多个单独的编译来进行，且每个编译都要指定 -xipo。

在此例中，编译和链接在单独的步骤中进行：

```
cc -xipo -xO4 -c part1.c part2.c
cc -xipo -xO4 -c part3.c
cc -xipo -xO4 -o prog part1.o part2.o part3.o
```

即使用 -xipo 编译，仍存在库不参与交叉文件过程间分析的约束，如下例所示：

```
cc -xipo -xO4 one.c two.c three.c
ar -r mylib.a one.o two.o three.o
...
cc -xipo -xO4 -o myprog main.c four.c mylib.a
```

在此例中，过程间调用优化是在以下例程之间执行的：one.c、two.c 和 three.c 之间，main.c 和 four.c 之间，但不在 main.c 或 four.c 和 mylib.a 上的例程之间执行。（第一次编译可能产生未定义符号警告，但是由于它是编译与链接步骤，所以会执行过程间调用优化。）

B.2.98.2 何时不使用 -xipo=2 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：

- foo() 无法在运行时被在该目标文件集合之外定义的其他例程显式调用。
- 来自该目标文件集合中的任何例程的 foo() 调用将不受在该目标文件集合以外定义的不同版本的 foo() 的干预。

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() 和 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。

B.2.99 -xipo_archive=[*a*]

-xipo_archive 选项使编译器可在生成可执行文件之前，用通过 -xipo 编译且驻留在归档库 (.a) 中的目标文件来优化传递给链接程序的目标文件。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 B–27 -xipo_archive 标志

值	含义
writeback	生成可执行文件之前，编译器使用通过 -xipo 编译的目标文件（驻留在归档库 (.a) 中）来优化传递到链接程序的目标文件。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	生成可执行文件之前，编译器使用通过 -xipo 编译的目标文件（驻留在归档库 (.a) 中）来优化传递到链接程序的目标文件。 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。请注意，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。

B.2.100 -xjobs=*n*

指定 -xjobs 选项，以设置编译器为完成工作需要创建的进程数。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*，过程间优化器就将 *n* 作为其在编译不同文件时可调用的最大代码生成器实例数。

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）。

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。

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。

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.c t2.c t3.c
```

指定不带标志的 -xipo_archive 是非法的。

B.2.101 -xldscope={*v*}

指定 -xldscope 选项，以更改用于外部符号定义的缺省链接程序作用域。由于实现更隐蔽，因此更改缺省值可使共享库更快、更安全。

v 必须是下列值之一：

表 B–28 -xldscope 标志

标志	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。该符号的所有引用都绑定到在第一个动态模块中定义该符号的定义上。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。从所链接的动态模块内部对符号的所有引用都绑定到在模块内部定义的符号上。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。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。动态模块内部的所有引用都绑定到该模块内部的一个定义上。符号在模块外部是不可视的。

如果未指定 -xldscope，编译器将假定 -xldscope=global。如果指定不带参数的 -xldscope，编译器会发出错误信息。在到达最右边的实例之前，命令行上此选项的多个实例相互覆盖。

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。编译程序在以下情况下将内联函数：用 -xinline 指定函数的名称、在可以自动内联的 -xO4 或更高级别进行编译、使用内联说明符、使用内联 pragma 或者使用交叉文件优化。

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，则在调用 ABC_allocator 的库组件中不能进行该替换。最终程序将包括函数的不同版本。

生成库时，用 __hidden 或 __symbolic 说明符声明的库函数可以内联生成。假定这些库函数不被客户端覆盖。请参见第 32 页中的 “2.2 链接程序作用域说明符”。

用 __global 说明符声明的库函数不应内联声明，并且应该使用 -xinline 编译器选项来防止内联。

另请参见 -xinline、-xO、-xcrossfile、#pragma、inline。

B.2.102 -xlibmieee

强制为异常类中的数学例程返回 IEEE 754 样式的值。在此情况下，不输出异常消息，并且不应依赖 errno。

B.2.103 -xlibmil

内联一些库例程，以加快执行速度。此选项可为浮点选项和系统平台选择合适的汇编语言内联模板。

无论将函数的任何规范作为 -xlibmil 标记的一部分，-xlibmil 都会内联函数。

但是，这些替换会导致 errno 的设置变得不可靠。如果您的程序依赖于 errno 的值，请尽量不要使用此选项。另请参见第 51 页中的 “2.13 保留 errno 的值”。

B.2.104 -xlibmopt

使编译器可以使用优化数学例程的库。使用此选项时，必须通过指定 -fround=nearest 来使用缺省舍入模式。

数学例程库优化了性能，并且通常会生成运行速度更快的代码。结果可能与普通数学库产生的结果略有不同。如果有差别，通常是最后一位不同。

但是，这些替换会使 errno 的设置变得不可靠。如果您的程序依赖于 errno 的值，请不要使用此选项。另请参见第 51 页中的 “2.13 保留 errno 的值”。

该库选项在命令行上的顺序并不重要。

此选项由 -fast 选项设置。

另请参见：-fast -xnolibmopt

B.2.105 -xlic_lib=sunperf

Solaris Studio 提供的性能库中的链接。

B.2.106 -xlicinfo

编译器忽略此选项且不显示任何提示。

B.2.107 -xlinkopt[=*level*]

(SPARC) 指示编译器对可重定位的目标文件执行链接时优化。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。还可以在生成共享库（用 -G 编译）时使用 -xlinkopt。

级别设置执行的优化级别，必须为 0、1 或 2。优化级别为：

表 B–29 -xlinkopt 标志

标志	含义
0	禁用后优化器。（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
```

有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

请注意，仅当编译器链接时才使用级别参数。在以上示例中，即使二进制目标代码是用隐含级别 1 编译的，使用的后优化级别仍然是 2。

指定 -xlinkopt 时若不带级别参数，则表示 -xlinkopt=1。

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 264 页中的 “B.2.136 –xprofile=*p*”。

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。

B.2.108 -xloopinfo

显示哪些循环已并行化以及哪些循环未并行化。简要说明循环未并行化的原因。仅当指定了 -xautopar 时，-xloopinfo 选项才有效，否则，编译器将发出警告。

要达到更快的执行速度，则该选项需要多处理器系统。在单处理器系统中，生成的代码通常运行得较慢。

B.2.109 -xM

只对指定的 C 程序运行预处理程序，请求它生成 makefile 依赖性并将结果发送至标准输出（有关 make 文件和依赖性的详细信息，请参见 make(1)）。

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
```

如果您指定 -xM 和 -xMF，则编译器会将所有 makefile 依赖性信息追加到使用 -xMF 指定的文件中。

B.2.110 -xM1

生成 makefile 依赖性类似 -xM，但不包括 /usr/include 文件。例如：

```
more hello.c
#include<stdio.h>
main()
{
```

```
    (void)printf("hello\n");
}
cc– xM hello.c
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

使用 -xM1 编译不会报告头文件的依赖性：

```
cc– xM1 hello.c
hello.o: hello.c
```

-xM1 在 -Xs 模式下不可用。

如果您指定 -xM 和 -xMF，则编译器会将所有 makefile 依赖性信息附加至使用 -xMF 指定的文件中。

B.2.111 -xMD

像 **-xM** 一样生成 makefile 依赖性，但编译继续。**-xMD** 为从 -o 输出文件名（如果指定）或输入源文件名派生的 makefile 依赖性信息生成一个输出文件，替换（或添加）后缀为 **.d** 的文件名。如果指定 **-xMD** 和 **-xMF**，预处理程序会将所有 makefile 依赖性信息写入到使用 **-xMF** 指定的文件中。不允许使用 **-xMD -xMF** 或 **-xMD -o** *filename* 来编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆写该文件。

B.2.112 -xMF *filename*

使用此选项可为 makefile 依赖性输出指定一个文件。目前，没有方法可以在一个命令行上使用 **-xMF** 为多个输入文件指定单独的文件名。不允许使用 **-xMD -xMF** 或 **-xMMD -xMF** 编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆写该文件。

B.2.113 -xMMD

使用此选项可生成不包括系统头文件的 makefile 依赖性。此选项与 **-xM1** 的功能相同，但是编译继续。**-xMMD** 为从 **-o** 输出文件名（如果指定）或输入源文件名派生的 makefile 依赖性信息生成一个输出文件，替换（或添加）后缀为 **.d** 的文件名。如果您指定 **-xMF**，则编译器将改用您提供的文件名。不允许使用 **-xMMD -xMF** 或 **-xMMD -o** *filename* 来编译多个源文件，否则会生成错误。如果已存在依赖性文件，将覆写该文件。

B.2.114 -xMerge

将数据段合并为文本段。在此编译所生成的目标文件中初始化的数据是只读数据，并可（除非用 ld-N 链接）在进程间共享。

三个选项 **-xMerge -ztext -xprofile=collect** 不应同时使用。**-xMerge** 会强制将静态初始化的数据存储到只读存储器中，**-ztext** 禁止在只读存储器中进行依赖于位置的符号重定位，而 **-xprofile=collect** 会在可写存储器中生成静态初始化的、依赖于位置的符号重定位。

B.2.115 -xmaxopt[=*v*]

此命令可将 pragma opt 的级别限制为指定级别。*v* 是 off、1、2、3、4、5 中的一个值。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 设置的优化级别不能超过 -xmaxopt 值。

B.2.116 -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未对齐数据访问行为。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。下表列出了 -xmemalign 的对齐值和行为值。

表 B–30 -xmemalign 对齐和行为标志

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，请参见表 A–2。

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。

-xmemalign 选项允许您在这些未确定情况下指定编译器要假定的数据最大内存对齐。它还指定在运行时发生未对齐内存访问时要执行的错误行为。

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。

以下缺省值仅适用于未使用 -xmemalign 选项时：

- -xmemalgin=8i（适于所有 v8 体系结构）。
- -xmemalign=8s（适于所有 v9 体系结构）。

在出现 -xmemalign 选项但未给定任何值时，缺省值为：

- -xmemalign=1i（对于所有 -xarch 值）。

下表说明了如何使用 -xmemalign 来处理不同的对齐情况。

表 B–31 -xmemalign 示例

命令	情况
-xmemalign=1s	大量未对齐访问导致了自陷处理非常缓慢。
-xmemalign=8i	在发生错误的代码中存在偶然的、有目的的、未对齐访问。
-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。

B.2.117 -xmodel=[*a*]

(x86) -xmodel 选项使编译器可针对 Solaris x86 平台修改 64 位对象的格式，只能为此类对象的编译指定此选项。

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 B–32 -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。

B.2.118 -xnolib

缺省情况下不链接任何库；即不向 ld(1) 传递任何 -l 选项。通常， cc 驱动程序将 -lc 传递给 ld。

在您使用 -xnolib 时，必须自己传递所有 -l 选项。

B.2.119 -xnolibmil

不内联数学库例程。在 –fast 选项之后使用。例如：

```
% cc– fast– xnolibmil....
```

B.2.120 -xnolibmopt

通过关闭以前指定的所有 -xlibmopt 选项，禁止编译器使用优化的数学库。在 -fast 之后使用此选项（-fast 可以通过设置 -xlibmopt 允许使用优化数学库）：

```
% cc -fast -xnolibmopt ...
```

B.2.121 -xnorunpath

不将共享库的运行时搜索路径生成到可执行文件中。

通常 cc 不向链接程序传递任何 -R 路径。有一些选项会向链接程序传递 -R 路径，如 -xliclib=sunperf 和 -xopenmp。可使用 - xnorunpath 选项来加以阻止。

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。

B.2.122 -x0[1|2| 3|4|5]

优化目标代码；注意大写字母 O 后跟数字 1、2、3、4 或 5。一般说来，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，-x02 级别的性能可能比其他优化级别好，而 -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。

如果优化器内存不足，它将尝试通过在较低的优化级别中重试当前的过程来进行恢复，并以命令行选项中指定的原始级别继续后续过程。

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。

如果使用 -g 并且优化级别为 -x03 或更低，编译器将提供几乎进行全面优化的最佳效果符号信息。尾部调用优化和后端内联被禁用。

如果使用 -g 并且优化级别为 -x04 或更高，编译器将提供进行全面优化的最佳效果符号信息。

使用 -g 进行调试不会抑制 -x0*n*，但 -x0*n* 会在某些方面限制 -g。例如，优化选项会降低调试的效用，以致无法显示 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》第 1 章中的“调试优化的代码”。

如果同时指定了 -x0 和 -xmaxopt，那么用 -x0 设置的优化级别不得超过 -xmaxopt 值。

如果在 -x03 或 -x04 级别上优化非常大的程序（在同一程序中有数千行代码），优化器可能需要大量虚拟内存。在这种情况下，机器性能可能会降低。

B.2.122.1 SPARC 优化的说明

下表描述了它们在 SPARC 平台上如何操作。

表 B–33 SPARC 平台上的 -x0 标志

值	含义
-x01	执行基本局部优化（窥孔优化）。
-x02	执行基本局部和全局优化。其中包括感应变量排除、局部和全局常用子表达式排除、代数简化、副本传播、常量传播、循环不可变优化、寄存器分配、基本块合并、尾部循环排除、死代码排除、尾部调用排除和复杂表达式扩展。 -x02 级别不会将全局、外部或间接引用或定义分配给寄存器。它将这些引用和定义视为被声明为 volatile。一般说来，-x02 级别产生的代码最小。
-x03	类似于执行 -x02，但还会优化外部变量的引用或定义。还执行循环解开和软件流水线作业。该级别不跟踪指针赋值的结果。在编译设备驱动程序或从信号处理程序内部修改外部变量的程序时，可能需要使用 volatile 类型限定符来保护对象，使其免于优化。一般说来，-x03 级别会导致代码增大。
-x04	类似于执行 -x03，但是还自动内联包含在相同文件中的函数；这通常会提高执行速度。如果要控制内联哪些函数，请参见第 239 页中的 “B.2.96 -xinline=*list*”。 该级别跟踪指针赋值的效果，通常导致代码增大。
-x05	试图生成最高优化级别。使用编辑时间更长或减少执行时间的程度不是很高的优化算法。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 264 页中的 “B.2.136 –xprofile=*p*”。

B.2.122.2 x86 优化的说明

下表描述了优化级别在 x86 平台上的工作原理。

表 B–34 x86 平台上的 -x0 标志

值	含义
-x01	从内存、交叉跳跃（尾部合并）以及缺省优化的单个传递中预装入参数。
-x02	同时调度高级和低级指令，并执行改进的溢出分析、循环内存引用排除、寄存器生命周期分析、增强的寄存器分配以及全局通用子表达式的排除。
-x03	执行循环长度约简、感应变量排除以及在级别 2 完成的优化。
-x04	除执行 -x03 的优化之外，还自动内联包含在同一文件中的函数。自动内联通常会提高执行速度，但有时却会使速度变得更慢。通常该级别会增加代码的大小。

表 B–34 x86 平台上的 -xO 标志 （续）

值	含义
-xO5	生成最高级别优化。使用编辑时间更长或减少执行时间的程度不是很高的优化算法。其中包含为导出的函数生成局部的调用约定入口点、进一步优化溢出代码和增加分析，以改善指令调度。

有关调试的更多信息，请参见《Sun Studio：使用 dbx 调试程序》手册。有关优化的更多信息，请参见《Solaris Studio 12：性能分析器》手册。

另请参见 -xldscope 和 -xmaxopt。

B.2.123 -xopenmp[= *i*]

使用 -xopenmp 选项可通过 OpenMP 指令启用显式并行化。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。

下表列出了 *i* 值：

表 B–35 -xopenmp 标志

值	含义
parallel	启用 OpenMP pragma 的识别。-xopenmp=parallel 时的优化级别为 -x03。如有必要，编译器会将优化级别更改为 -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。 如果将优化级别显式设置为低于 -O3，如同在 cc -O2 -xopenmp=noopt 中一样，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	该标志为缺省设置；它不启用 OpenMP pragma 识别，不更改程序的优化级别且不预定义任何预处理程序标记。

如果指定 -xopenmp 但未给定值，编译器将假定 -xopenmp=parallel。如果不指定 -xopenmp，编译器将假定 -xopenmp=none。

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。

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。

如果您在生成任何 .so 时使用 -xopenmp，则必须在链接可执行文件时使用 -xopenmp，并且可执行文件的编译器版本不能比使用 -xopenmp 生成 .so 的编译器版本低。这在编译包含 OpenMP 指令的库时尤其重要。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

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。

有关特定于 OpenMP 的 C 实现的更多信息，请参见第 60 页中的 “3.2 OpenMP 并行化”。

有关 OpenMP 的信息，请参见《Solaris Studio 12 Update 1： OpenMP API 用户指南》。

B.2.124 -xP

编译器对源文件仅执行语法和语义检查，以输出所有 K&R C 函数的原型。此选项不生成任何目标代码或可执行代码。例如，对以下源文件指定 -xP，

```
f()
{
}

main(argc,argv)
int argc;
char *argv[];
{
}
```

产生以下输出：

```
int f(void);
int main(int, char **);
```

B.2.125 -xpagesize=*n*

为栈和堆设置首选页面大小。

下列值在 SPARC 上有效：4k、8K、64K、512K、2M、4M、32M、256M、2G、16G 或 default。

下列值在 x86 上有效：4K、2M。4M、1G 或 default。

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。必须指定适于目标平台的有效页面大小。

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。

可以使用 pmap(1) 或 meminfo(2) 来确定目标平台的页面大小。

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，请参见表 A–2。

如果指定 -xpagesize=default，Solaris 操作系统将设置页面大小。

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。

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。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。

B.2.126 -xpagesize_heap=*n*

在内存中为堆设置页面大小。

此选项与 -xpagesize 接受相同的值。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。

可以使用 pmap(1) 或 meminfo(2) 在目标平台中设置页面大小。

如果指定 -xpagesize_heap=default，Solaris 操作系统将设置页面大小。

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。

除非在编译和链接时使用，否则 -xpagesize_heap 选项不会生效。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

B.2.127 -xpagesize_stack=*n*

在内存中为栈设置页面大小。

此选项与 -xpagesize 接受相同的值。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。

如果指定 -xpagesize_stack=default，Solaris 操作系统将设置页面大小。

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。

除非在编译和链接时使用，否则 -xpagesize_stack 选项不会生效。有关在编译和链接时都必须指定的所有编译器选项的完整列表，请参见表 A–2。

B.2.128 -xpch=*v*

此编译器选项可激活预编译头文件功能。*v* 可以是 auto、autofirst、collect:*pch_filename* 或 use:*pch_filename*。通过将 -xpch（在第 257 页中的 “B.2.128 -xpch=*v*”中进行了详细介绍）和 -xpchstop（在第 261 页中的 “B.2.129 -xpchstop=[*file*|<include>]”中进行了详细介绍）选项与 #pragma hdrstop 指令（在第 42 页中的 “2.11.8 hdrstop”中进行了详细介绍）结合使用，可利用此功能。

使用 -xpch 选项可以创建预编译头文件并减少编译时间。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，这些 include 文件往往包含大量的源代码。预编译头文件的工作机理是，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编译其他有同样头文件顺序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。

另请参见：

- 第 261 页中的 “B.2.129 -xpchstop=[*file*|<include>]”。
- 第 42 页中的 “2.11.8 hdrstop”。

B.2.128.1 自动创建预编译头文件

可以让编译器自动生成预编译头文件。选择以下两种方法之一来实现此目的。一种方法是让编译器从在源文件找到的第一个 include 文件创建预编译头文件。另一种方法是让编译器从在源文件中找到的 include 文件集合中选择，选择范围从第一个 include 文件开始，直到已经定义好的确定哪个 include 文件是最后一个的点结束。使用以下标志之一可以确定编译器用于自动生成预编译头文件的方法：

表 B–36 -xpch 标志

标志	含义
-xpch=auto	预编译头文件的内容基于编译器在源文件中找到的最长的活前缀（有关如何识别活前缀的说明，请参见下文）。此标志生成的预编译头文件可能包含最多的头文件。
-xpch=autofirst	此标志生成的预编译头文件仅包含在源文件中找到的第一个头文件。

B.2.128.2 手动创建预编译头文件

如果决定手动创建预编译头文件，则必须首先使用 -xpch，并指定 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.c 创建名为 myheader.cpch 的预编译头文件：

```
cc -xpch=collect:myheader a.c
```

有效的预编译头文件通常具有后缀 .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.c，则预编译的头文件名为 foo.cpch。

B.2.128.3 编译器如何处理现有的预编译头文件

如果编译器无法使用 -xpch=auto 和 -xpch=autofirst 时的预编译头文件，则会生成新的预编译头文件。如果编译器无法使用 -xpch=use 时的预编译头文件，将发出一个警告，并且使用真实头文件来完成编译。

B.2.128.4 指示编译器使用特定的预编译头文件

您还可以指示编译器使用特定的预编译头文件。要实现此目的，请指定 -xpch=use:*pch_filename*。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 模式中的命令可类似于以下命令： cc -xpch=use:foo.cpch foo.c bar.c foobar.c.

如果以下条件都成立，则只能使用现有的预编译的头文件。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。由某一版本的编译器创建的预编译头文件不能被其他版本的编译器使用。
- 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。

B.2.128.5 活前缀

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）。标记是指关键字、名称或标点符号。被 #if 指令排除的注释和代码不能被编译器识别为标记。该初始标记序列称为活前缀。也就是说，活前缀是源文件中通用于所有源文件的最靠前的部分。创建预编译头文件并进而确定源文件中哪些头文件是预编译的时，编译器使用此活前缀作为整个操作的依据。

编译器在当前编译期间找到的活前缀必须与用于创建预编译头文件的活前缀匹配。也就是说，必须在使用相同预编译头文件的所有源文件中对活前缀给出一致的解释。

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。注释被忽略。

指定 -xpch=auto 或 -xpch=autofirst 时，编译器自动确定活前缀的终点，定义如下：

- 第一个声明/定义
- 第一个 #line 指令
- #pragma hdrstop 指令
- 在指定的 include 文件之后（如果您指定 -xpch=auto 和 -xpchstop）
- 第一个 include 文件（如果您指定 -xpch=autofirst）

注 – 预处理程序条件编译语句中的终点会生成一个警告，并禁止自动创建预编译头文件。另外，如果同时指定 #pragma hdrstop 和 -xpchstop 选项，编译器将使用两个停止点中较早的那一个来终止活前缀。

对于 -xpch=collect 或 -xpch=use，活前缀以 #pragma hdrstop 结尾。

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。

B.2.128.6 浏览头文件以查找问题

如何能够使头文件可以预编译？在不同的源文件中解释一致时，头文件可以预编译。具体来说，就是当它仅包含完全声明时。也就是说，任何一个文件中的声明都必须独自成为有效声明。不完全的类型声明，例如 `struct S;`，是有效声明。完全类型声明可以出现在某些其他文件中。请考虑这些头文件示例：

```
file a.h
struct S {
#include "x.h" /* not allowed */
};

file b.h
struct T; // ok, complete declaration
struct S {
   int i;
[end of file, continued in another file] /* not allowed*/
```

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上述约束的程序的编译结果。

- 头文件不得使用 `__DATE__` 和 `__TIME__`。
- 头文件不得包含 `#pragma hdrstop`。

如果头文件还包含变量和函数定义，则也是可预编译的。

B.2.128.7 预编译头文件高速缓存

编译器自动创建预编译头文件时，会将该文件写入 `SunWS_cache` 目录中。此目录始终位于创建目标文件的位置。该文件的更新受锁保护，这样可在 `dmake` 下正常工作。

如果需要强制编译器重新生成自动生成的预编译头文件，可以使用 `CCadmin` 工具清除预编译头文件高速缓存目录。有关更多信息，请参见 `CCadmin`(1) 手册页。

B.2.128.8 警告

- 不要在命令行上指定冲突的 `-xpch` 标志。例如，同时指定 `-xpch=collect` 和 `-xpch=auto` 或同时指定 `-xpch=autofirst` 和 `-xpchstop=<include>` 会产生错误。
- 如果指定 `-xpch=autofirst`，或指定不带 `-xpchstop` 的 `-xpch=auto`，则出现在第一个 include 文件之前或出现在使用用于 `-xpch=auto` 的 `-xpchstop` 指定的 include 文件之前的任何声明、定义或 `#line` 指令都会生成警告，并禁止自动生成预编译头文件。
- `-xpch=autofirst` 或 `-xpch=auto` 时的第一个 include 文件之前的 `#pragma hdrstop` 会禁止自动生成预编译头文件。

B.2.128.9 预编译头文件依赖性和 make 文件

指定 `-xpch=collect` 时，编译器会生成预编译头文件的依赖性信息。需要在 make 文件中创建适当的规则，以利用这些依赖性。考虑下面的 make 文件示例：

```
%.o : %.c shared.cpch
        $(CC) -xpch=use:shared -xpchstop=foo.h -c $<
default : a.out

foo.o + shared.cpch : foo.c
        $(CC) -xpch=collect:shared -xpchstop=foo.h foo.c -c

a.out : foo.o bar.o foobar.o
        $(CC) foo.o bar.o foobar.o

clean :
        rm -f *.o shared.cpch .make.state a.out
```

这些 make 规则以及编译器生成的依赖性，会在与 `-xpch=collect` 一起使用的任何源文件或属于预编译头文件的一部分的任何头文件发生更改时，强制重新创建手动创建的预编译头文件。这样可防止使用过期的预编译头文件。

您无需在 make 文件中为 `-xpch=auto` 或 `-xpch=autofirst` 创建任何其他 `make` 规则。

B.2.129 -xpchstop=[*file*|<include>]

使用 `-xpchstop=`*file* 选项可为预编译头文件指定活前缀的最后一个 include 文件。在命令行使用 `-xpchstop` 与将 `hdrstop` pragma 置于第一个包含指令之后等效，此包含指令在您使用 `cc` 命令指定的每个源文件中引用 *file*。

结合使用 `-xpchstop=<include>` 和 `-xpch-auto` 可以创建基于从其往上并包括 `<include>` 的头文件的预编译头文件。此标志覆盖缺省的 `-xpch=auto` 行为（使用整个活前缀中包含的所有头文件的行为）。

在以下示例中，`-xpchstop` 选项指定了预编译头文件的活前缀以 `projectheader.h` 的包含结束。因此，`privateheader.h` 不是活前缀的一部分。

```
example% cat a.c
     #include <stdio.h>
     #include <strings.h>
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% cc -xpch=collect:foo.cpch a.c -xpchstop=projectheader.h -c
```

另请参见 `-xpch`。

B.2.130 - xpec[={yes|no}]

生成可移植的可执行代码 (Portable Executable Code, PEC) 库。PEC 二进制文件可与自动调优系统 (Automatic Tuning System, ATS) 一起使用，该系统出于优化和故障排除的目的重新生成编译的 PEC 二进制文件（原始源代码不是必需的）。http://cooltools.sunsource.net/ 上提供了有关 ATS 的更多信息。

使用 -xpec 生成的二进制文件通常比未使用 -xpec 生成的文件大五倍到十倍。

如果不指定 -xpec，则编译器会将其设置为 -xpec=no。如果您指定 -xpec，但不提供标志，则编译器会将其设置为 -xpec=yes。

B.2.131 -xpentium

(x86) 为 Pentium 处理器生成代码。

B.2.132 -xpg

准备目标代码，以收集用 gprof(1) 进行文件配置所需的数据。此选项调用在正常终止情况下产生 gmon.out 文件的运行时记录机制。

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，配置文件中包含大概的用户 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）不进行分析。

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。

在 x86 系统中，-xpg 与 -xregs=frameptr 不兼容，这两个选项不应一起使用。还请注意，-fast 中包括 -xregs=frameptr。

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。

如果在编译时指定 -xpg，则还必须在链接时指定它。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 185 页中的 “A.1.2 编译时选项和链接时选项”。

B.2.133 -xprefetch[= *val*[, *val*]]

在支持预取的体系结构中启用预取指令。

显式预取只应在度量支持的特殊环境下使用。

val 必须是以下值之一：

表 B–37 -xprefetch 标志

标志	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。请参见第 263 页中的“B.2.133.1 预取等待时间比率”。
[no%]auto	[禁用] 启用预取指令的自动生成
[no%]explicit	[禁用] 启用显式预取宏
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。

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。

sun_prefetch.h 头文件提供了可用来指定显式预取指令的宏。预取的位置大约为可执行文件中对应于宏出现的位置。

B.2.133.1 预取等待时间比率

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。

该因子必须是形式为 *n.n.* 的正数。

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。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。要减小此值，请使用小于 1 的因子。

要使用 latx:factor 子选项，则以接近 1.0 的因子值开始并对应用程序进行性能测试。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。

B.2.134 -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`。

类似 `-xalias_level` 的选项可以影响计算候选间接预取的主动性，进而影响因更好的内存别名歧义消除信息而发生的自动插入间接预取的主动性。

B.2.135 -xprefetch_level=*l*

使用 `-xprefetch_level` 选项可以控制通过 `-xprefetch=auto` 确定的预取指令的自动插入的主动性。*l* 必须为 `1`、`2` 或 `3`。编译器更加主动，换句话说，引入了更多更高 `-xprefetch_level` 级别的预取。

`-xprefetch_level` 的适当值取决于应用程序可能具有的缓存缺失的数量。较高级别的 `-xprefetch_level` 值具有提高应用程序性能的潜能。

仅当使用 `-xprefetch=auto` 进行编译且优化级别为 3 或更高级别时此选项才有效，并可为支持预取（`v8plus`、`v8plusa`、 `v9`、`v9a`、`v9b`、`generic64`、 `native64`）的平台生成代码。

`-xprefetch_level=1` 启用预取指令的自动生成。`-xprefetch_level=2` 启用级别 1 之外的额外生成，`-xprefetch_level=3` 启用级别 2 之外的额外生成。

指定了 `-xprefetch=auto` 时，缺省值为 `-xprefetch_level=1`。

B.2.136 –xprofile=*p*

收集用于配置文件的数据或使用配置文件进行优化。

p 必须为 `collect`[:*profdir*]、`use`[:*profdir*] 或 `tcov`[:*profdir*]。

此选项可在执行期间收集并保存执行频率数据，然后在后续运行中可以使用该数据来改进性能。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。只有指定 `-xO2` 或更高优化级别时，此选项才有效。如果分别执行编译和链接，则链接步骤和编译步骤中必须都出现同一 `-xprofile` 选项。

`collect`[:*profdir*]
: 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。

  `-xMerge`, `-ztext`, 和 `-xprofile=collect` 不应同时使用。`-xMerge` 会强制将静态初始化的数据存储到只读存储器中，`-ztext` 禁止在只读存储器中进行依赖于位置的符号重定位，而 `-xprofile=collect` 会在可写存储器中生成静态初始化的、依赖于位置的符号重定位。

配置文件目录名 *profdir*（如果指定）是包含已进行文件配置的对象代码的程序或共享库在执行时用来存储配置文件数据的目录的路径名。如果 *profdir* 路径名不是绝对路径，在使用选项 `-xprofile=use:`*profdir* 编译程序时将相对于当前工作目录来解释该路径。如果未指定配置文件目录名，配置文件数据将存储在名为 *program*`.profile` 的目录中，其中 *program* 是已进行文件配置的主程序的基本名称。

示例[1]：在生成程序所在的同一目录中的 myprof.profile 目录中收集和使用配置文件数据：

```
demo: cc -xprofile=collect:myprof.profile -xO5 prog.c -o prog
demo: ./prog
demo: cc -xprofile=use:myprof.profile    -xO5 prog.c -o prog
```

示例[2]：在目录 `/bench/myprof.profile` 中收集配置文件数据，然后在优化级别 `-xO5` 的反馈编译中使用收集的配置文件数据：

```
demo: cc -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.c -o prog
...run prog from multiple locations..
demo: cc -xprofile=use:/bench/myprof.profile
\    -xO5 prog.c -o prog
```

可以设置环境变量 `SUN_PROFDATA` 和 `SUN_PROFDATA_DIR` 控制使用 `-xprofile=collect` 编译的程序存储配置文件数据的位置。如果设置了这两个环境变量，`-xprofile=collect` 数据将写入 `$SUN_PROFDATA_DIR/$SUN_PROFDATA`。

这些环境变量同样控制由 `tcov` 写入的配置文件数据文件的路径和名称，如 `tcov`(1) 手册页中所述。如果未设置这些环境变量，配置文件数据就写入当前目录中的目录 *profdir* `.profile`，其中 *profdir* 是可执行文件的名称或在 `-xprofile=collect:` *profdir* 标志中指定的名称。如果 *profdir* 已在 `.profile` 中结束，`-xprofile` 不会将 `.profile` 附加到 *profdir* 中。如果多次运行程序，那么执行频率数据会累积在 *profdir*`.profile` 目录中；也就是说，以前执行的输出不会丢失。

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使用 `-xprofile=collect` 进行链接。

`use[:`*profdir*`]`

通过从使用 `-xprofile=collect[:`*profdir*`]` 编译的代码中收集的执行频率数据优化在执行已进行文件配置的代码时执行的工作。*profdir* 是一个目录的路径名，该目录包含运行用 `-xprofile=collect[:`*profdir*`]` 编译的程序所收集的配置文件数据。

profdir 路径名是可选的。如果未指定 *profdir*，将使用可执行二进制文件的名称。如果未指定 `-o`，将使用 `a.out`。如果未指定 *profdir*，编译器将查找 *profdir*`.profile/feedback` 或 **`a.out.profile/feedback`**。例如：

```
demo: cc -xprofile=collect    -o myexe prog.c
demo: cc -xprofile=use:myexe -xO5 -o myexe    prog.c
```

程序是使用以前生成并保存在 feedback 文件中的执行频率数据优化的，此数据是先前执行用 `-xprofile=collect` 编译的程序时写入的。

除了 `-xprofile` 选项之外，源文件和其他编译器选项必须与用于编译（该编译过程创建了生成 feedback 文件的编译程序）的相应选项完全相同。编译器的相同版本必须同时用于 `collect` 生成和 `use` 生成。

如果用 `-xprofile=collect:`*profdir* 编译，则必须将相同的配置文件目录名 *profdir* 用在优化编译中：`-xprofile=use:` *profdir*。

另请参见 `-xprofile_ircache`，以了解有关加速 `collect` 阶段和 `use` 阶段之间的编译的说明。

`tcov[:`*profdir*`]` 使用 `tcov`(1) 校验基本块覆盖率分析的对象文件。

如果指定可选的 *profdir* 参数，编译器将在指定位置创建配置文件目录。该配置文件目录中存储的数据可通过 `tcov`(1) 或由编译器通过 `-xprofile=use:`*profdir* 来使用。如果省略可选的 *profdir* 路径名，执行已进行文件配置的程序时将创建配置文件目录。只能通过 `tcov`(1) 使用该配置文件目录中存储的数据。使用环境变量 `SUN_PROFDATA` 和 `SUN_PROFDATA_DIR` 可以控制配置文件目录的位置。

如果 *profdir* 指定的位置不是绝对路径名，则编译时相对于要将当前的目标文件写入到的目录来解释该位置。如果为任何对象文件指定了 *profdir*，则必须为同一程序中的所有对象文件指定同一位置。由 *profdir* 指定位置的目录必须在要执行已进行文件配置的程序的所有计算机中都可以访问。除非不再需要配置文件目录中的内容，否则不应删除该目录，因为除非重新编译，否则编译器存储在其中的数据将无法恢复。

示例 [1]：如果用 `-xprofile=tcov:/test/profdata` 编译一个或多个程序的对象文件，编译器会创建一个名为 `/test/profdata.profile` 的目录并将其用来存储描述已进行文件配置的对象文件的数据。该同一目录还可在执行时用来存储与已进行文件配置的对象文件关联的执行数据。

示例 [2]：如果名为 myprog 的程序用 -xprofile=tcov 编译并在目录 /home/joe 中执行，将在运行时创建目录 /home/joe/myprog.profile 并将其用来存储运行时配置文件数据。

B.2.137 -xprofile_ircache[=*path*]

(SPARC) 将 -xprofile_ircache[=*path*] 与 -xprofile=collect|use 一起使用，通过重用 collect 阶段保存的编译数据来缩短 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.c t2.c
example% a.out    // run collects feedback data
example% cc -xO5 -xprofile=use -xprofile_ircache t1.c t2.c
```

B.2.138 -xprofile_pathmap

(SPARC) 如果同时还指定 -xprofile=use 命令，请使用 *-xprofile_pathmap=collect_prefix*:use_prefix 选项。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。将 *-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。

B.2.139 -xreduction

在自动并行化期间打开约简识别。必须使用 -xautopar 指定 -xreduction，否则编译器将会发出警告。

当启用约简识别时，编译器并行化约简，例如 *dot* 产品、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。

B.2.140 -xregs=*r*[,*r*...]

为生成的代码指定寄存器用法。

r 是一个逗号分隔列表，它包含下面的一个或多个子选项：appl、float、frameptr。

用 no% 作为子选项的前缀会禁用该子选项。

请注意，-xregs 子选项仅限于特定的硬件平台。

示例： -xregs=appl,no%float

表 B–38 -xregs 子选项

值	含义
appl	(SPARC) 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。由于需要更少的装入和存储指令，因此使用这些寄存器可提高性能。但是，这样使用可能与某些用汇编代码编写的旧库程序冲突。
float	(SPARC) 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 B–38 -xregs 子选项 （续）

值	含义
frameptr	(x86) 允许编译器将帧指针寄存器（IA32 上的 %ebp、AMD64 上的 %rbp）通用寄存器。 缺省值为 -xregs=no%frameptr。 除非也用 -features=no%except 禁用了异常，否则 C++ 编译器将忽略 -xregs=frameptr。请注意，-xregs=frameptr 是 -fast 的一部分，但除非同时指定 -features=no%except，否则 C++ 编译器将忽略此设置。 通过 -xregs=framptr，编译器可以自由使用帧指针寄存器来改进程序性能。但是，这可能会限制调试器和性能测量工具的某些功能。栈跟踪、调试器和性能分析器不能报告用 -xregs=frameptr 编译的功能。 而且，对 Posix pthread_cancel() 的 C++ 调用将找不到清理处理程序。 如果直接调用或从 C 或 Fortran 函数间接调用的 C++ 函数会引发异常，不应该用 -xregs=frameptr 编译 C、Fortran 和 C++ 混合代码。如果使用 -fast 编译此类混合源代码，请在命令行中的 -fast 选项后添加 -xregs=no%frameptr。 由于 64 位平台上的可用寄存器更多，因此相对于 64 位代码相比，使用 -xregs=frameptr 编译更容易改进 32 位代码的性能。 如果同时指定了 -xpg，编译器会忽略 -xregs=frameptr 并发出警告。

SPARC 缺省值为 -xregs=appl,float。

x86 缺省值为 -xregs=no%frameptr。

在 x86 系统中，-xpg 与 -xregs=frameptr 不兼容，这两个选项不应一起使用。还请注意，-fast 中包括 -xregs=frameptr。

强烈推荐您用 -xregs=no%appl,float 编译那些用于与应用程序链接的共享库的代码。至少共享库应该显式说明它如何使用应用程序寄存器，以便与这些库链接的应用程序知道这些寄存器分配。

例如，在某种全局意义上使用寄存器（例如，使用寄存器指向一些关键数据结构）的应用程序，需要确切地知道其代码未使用 -xregs=no%appl 编译的某个库如何使用应用程序寄存器，以便安全地与该库链接。

B.2.141 -xrestrict[=*f*]

将赋值为指针的函数参数视为受限参数。*f* 为 %all、%none 或由下面的一个或多个函数名构成的逗号分隔列表：{%all|%none|*fn*[,*fn*...]。

如果使用该选项指定函数列表，则指定的函数中的指针参数将被视为限定的；如果指定 -xrestrict=%all，则整个 C 文件中的所有指针参数均被视为限定的。有关更多信息，请参阅第 75 页中的 “3.8.2 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.c
```

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.c
```

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=%all 等效。

B.2.142 -xs

允许在不使用目标文件的情况下由 dbx 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。

B.2.143 -xsafe=mem

(SPARC) 允许编译器假定没有内存保护违规发生。

该选项允许在 SPARC V9 体系结构中使用无故障装入指令。

注 – 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。

仅当与优化级别 -xO5 及以下 -xarch 值中的一个一起使用时，此选项才能有效：sparc、sparcvis、sparcvis2 或 sparcvis3（用于 -m32 和 -m64）。

B.2.144 -xsb

已废弃－不使用。不再支持源代码浏览器功能。

B.2.145 -xsbfast

已废弃－不使用。不再支持源代码浏览器功能。

B.2.146 -xsfpconst

将无后缀的浮点常量表示为单精度模式，而非缺省的双精度模式。与 -Xc 一起使用时无效。

B.2.147 -xspace

不执行可增加代码大小的循环优化或并行化。

示例：如果编译器增加代码大小，它不会解开循环或并行化循环。

B.2.148 -xstrconst

此选项已废弃，可能会在将来的发行版中删除。-xstrconst 是 -features=conststrings 的别名。

B.2.149 -xtarget=*t*

为指令集和优化指定目标系统。

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=sun4/15 与以下内容等效： -xarch=v8a -xchip=micro -xcache=2/16/1。

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。

表 B–39 -xtarget 值（所有平台）

标志	含义
native	在主机系统上获取最佳性能。 编译器生成能在主机系统上提供最佳性能的代码。它决定了运行编译器的计算机的可用架构、芯片和缓存属性。
native64	在主机系统上获取 64 位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。

表 B–39 -xtarget 值（所有平台） （续）

标志	含义
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	获取指定系统的最佳性能。 从以下代表您所面向的实际系统的列表中选择系统名称：

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。

B.2.149.1 -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。如果为 -xtarget 指定 native64 或 generic64 以外的标志，则还必须如下所示指定 -m64 选项：-xtarget=ultra ... -m64，否则编译器将使用 32 位内存模型。

表 B–40 SPARC 上的 -xtarget 扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
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

表 B–40 SPARC 上的 -xtarget 扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4:2048/64/4/2:32768/64/4
ultraT1	sparc	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
ultraT3	sparcvis3	ultraT3	8/16/4:6144/64/24
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10

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 227 页中的 “B.2.80 -xcache[= *c*]”。

B.2.149.2 -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。如果为 -xtarget 指定 native64 或 generic64 以外的标志，则还必须如下所示指定 -m64 选项：-xtarget=opteron ... -m64，否则编译器将使用 32 位内存模型。

表 B–41 x86 上的 -xtarget 扩展

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
opteron	sse2a	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	32/64/8:256/64/8:8192/64/1
penryn	sse4_1	penryn	2/64/8:6144/64/24

表 B–41 x86 上的 -xtarget 扩展 （续）

-xtarget=	-xarch	-xchip	-xcache
woodcrest	ssse3	core2	32/64/8:4096/64/16
barcelona	amdsse4a	amdfam10	64/64/2:512/64/16

B.2.150 -xtemp=*dir*

将 cc 使用的临时文件的目录设置为 *dir*。在此选项字符串中不允许有空格。如果不指定此选项，临时文件将保存到 /tmp。-xtemp 优先于 TMPDIR 环境变量。

B.2.151 -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 33 页中的 “2.3 线程局部存储说明符”。

o 必须为下列值之一：

表 B–42 -xthreadvar 标志

标志	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。

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 -xthreadvar，但未指定任何值，则该选项设置为 -xthreadvar=dynamic。

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。

在 Solaris 软件的不同版本上使用线程变量需要在命令行中使用不同选项。

- 在 Solaris 8 软件中，对于使用 __thread 的对象，必须使用 -mt 进行编译，且必须使用 -mt -L/usr/lib/lwp -R/usr/lib/lwp 进行链接。
- 在 Solaris 9 软件中，使用 __thread 的对象必须使用 -mt 来编译和链接。

另请参见：-xcode、-KPIC、-Kpic

B.2.152 -xtime

报告每个编译组件所用的时间和资源。

B.2.153 -xtransition

针对 K&R C 与 Solaris ISO C 之间的差异发出警告。

-xtransition 选项与 -Xa 和 -Xt 选项一起使用时将发出警告。通过适当编码，可以消除关于不同行为的所有警告消息。除非使用 -xtransition 选项，否则不再出现以下警告：

- \a is ISO C “alert” character
- \x is ISO C hex escape
- bad octal digit
- base type is really *type tag*: *name*
- comment is replaced by “##”
- comment does not concatenate tokens
- declaration introduces new type in ISO C: *type tag*
- macro replacement within a character constant
- macro replacement within a string literal
- no macro replacement within a character constant
- no macro replacement within a string literal
- operand treated as unsigned
- trigraph sequence replaced
- ISO C treats constant as unsigned: *operator*
- semantics of *operator* change in ISO C; use explicit cast

B.2.154 -xtrigraphs

-xtrigraphs 选项确定编译器是否识别 ISO C 标准定义的三字符序列。

缺省情况下，编译器假定 -xtrigraphs=yes 并识别整个编译单元的所有三字符序列。

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。-xtrigraphs=no 选项可关闭整个编译单元内的所有三字母识别。

请考虑以下名为 trigraphs_demo.c 的源文件示例。

```
#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.c
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.c
example% a.out
(??) in a string appears as (??)
```

B.2.155 -xunroll=*n*

建议优化器解开循环 *n* 次。*n* 是正整数。当 *n* 等于 1 时，它是一个命令，此时编译器不解开任何循环。当 *n* 大于 1 时，-xunroll=*n* 只建议编译器解开循环 *n* 次。

B.2.156 -xustr={ascii_utf16_ushort| no}

如果您需要支持使用 ISO10646 UTF-16 串文字的国际化应用程序，请使用此选项。换句话说，如果代码中包含您希望在目标文件中由编译器转换成 UTF-16 字符串的串文字，请使用该选项。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。该选项使编译器可以将 U"*ASCII_string*" 串文字识别成无符号短整型数组。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。

通过指定 -xustr=no，可以关闭编译器识别 U"*ASCII_string*" 串文字。该选项在命令行上最右侧的实例覆盖了先前的所有实例。

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。

指定 -xustr=ascii_utf16_ushort 时未指定 U"*ASCII_string*" 串文字不是错误。

不是所有文件都必须使用该选项编译。

下面的示例显示了带有 U 前缀的带引号文本字符串。还显示了指定 -xustr 的命令行。

```
example% cat file.c
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.c -c
```

8 位字符文字可以带有 U 前缀，以形成一个 unsigned short 类型的 16 位 UTF-16 字符。示例：

```
const unsigned short x = U'x';
const unsigned short y = U'\x79';
```

B.2.157 -xvector[= *a*]

启用对向量库函数的调用的自动生成和/或 SIMD（Single Instruction Multiple Data，单个指令多个数据）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 B–43 -xvector 标志

值	含义
[no%]lib	如果循环内的数学库调用可转换为对等效向量数学例程的单个调用，则 [不] 允许编译器进行此类转换。此类转换可提高那些循环计数较大的循环的性能。（仅限 Solaris）
[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。在 x86 中，缺省情况下以优化级别 3 和更高级别使用流扩展。可以使用子选项 no%simd 将其禁用。 仅当目标体系结构中存在流扩展（即目标 ISA 至少为 SSE2）时，编译器才会使用 SIMD。例如，可在现代平台中指定 -xtarget=woodcrest、-xarch=generic64、-xarch=sse2、-xarch=sse3 或 -fast 来使用它。如果目标 ISA 没有流扩展，子选项将无效。
yes	此选项已废弃，改为指定 -xvector=lib。
no	此选项已废弃，改为指定 -xvector=none。

在 x86 平台上的缺省值为 -xvector=simd，在 SPARC 平台上的缺省值为 -xvector=%none。如果指定不带子选项的 -xvector，编译器将采用 -xvector=simd、lib (x86)、-xvector=lib（SPARC、Solaris）和 -xvector=simd (Linux)。

-xvector 选项需要 **-x03** 或更高的优化级别。如果优化级别未指定或低于 **-x03**，编译将不会继续，同时会发出消息。

B.2.158 -xvis

(SPARC) 在使用 VIS[tm] 指令集软件开发者工具包 (VIS[tm] instruction-set Software Developers Kit, VSDK) 中定义的汇编语言模板时，请使用 -xvis=[yes| no] 命令。缺省值为 -xvis=no。指定 -xvis 与指定 -xvis=yes 等效。

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。

有关 VSDK 的更多信息，请访问 http://www.sun.com/processors/vis/。

B.2.159 -xvpara

发出有关可能存在的并行编程相关问题的警告，这些问题可能导致在使用 OpenMP 时出现错误的结果。与 -xopenmp 和 OpenMP API 指令一起使用。

编译器在检测到下列情形时会发出警告。

- 循环是使用 MP 指令并行化的，而这些指令中的不同循环迭代之间存在数据依赖性
- OpenMP 数据共享属性子句存在问题。例如，声明在 OpenMP 并行区域中的访问可能导致数据争用的变量 "shared"，或者声明其在并行区域中的值在并行区域之后使用的变量 "private"。

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 -vpara any.c
```

注 – Solaris 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**》。

B.2.160 -Y*c* , *dir*

指定新目录 *dir* 作为组件 *c* 的位置。*c* 可以包含任何字符，这些字符表示 -W 选项下列出的组件。

如果已指定组件的位置，则工具的新路径名称为 *dir*/*tool*。如果对任何一项应用了多个 -Y 选项，则保留最后一个选项。

B.2.161 -YA, *dir*

指定用来搜索所有编译器组件的目录 *dir*。如果 *dir* 中找不到组件，搜索将转至安装编译器的目录。

B.2.162 `-YI,` *dir*

更改搜索 `include` 文件的缺省目录。

B.2.163 `-YP,` *dir*

更改用于查找库文件的缺省目录。

B.2.164 `-YS,` *dir*

更改启动目标文件的缺省目录。

B.2.165 `-Zll`

(SPARC) 为 `lock_lint` 创建程序数据库，但不生成可执行代码。有关更多信息，请参阅 `lock_lint`(1) 手册页。

B.3 传递给链接程序的选项

`cc` 可以识别 `-a`、`-e`、`-r`、`-t`、`-u` 和 `-z` 并将这些选项及其参数传递给 `ld`。`cc` 会将所有无法识别的选项都传递给 `ld`，并显示警告。在 Solaris 平台上，**`-i`** 选项及其参数也会传递到链接程序。

◆ ◆ ◆ 附录 C

实现定义的 ISO/IEC C99 行为

ISO/IEC 9899:1999 编程语言 C 标准指定以 C 语言编写的程序的形式并加以解释。但是，此标准留下许多实现定义的问题，即因编译器而异的问题。本章将详细介绍这些方面的内容。这些章节的编号将作为本附录中标题的一部分提供，以便与 ISO/IEC 9899:1999 标准本身进行比较：

- 每节的标题均使用与 ISO 标准相同的章节文本和 *letter.number* 标识符。
- 每节都提供了 ISO 标准的要求（前面加一个项目符号），该标准描述了实现应定义的内容。然后在这一要求的后面附上实现的说明。

C.1 实现定义的行为 (J.3)

合乎标准的实现需要记录它在该子子句中所列每个区域内的行为选择。下面列出了实现定义的项：

C.1.1 转换 (J.3.1)

- 如何标识诊断（3.10，5.1.1.3）。

  错误和警告消息具有以下格式：

  filename , *line number*: *message*

  其中 *filename* 是错误或警告所在文件的名称，

  line number 是错误或警告所在行的编号，*message* 是诊断消息。
- 空白字符的每个非空序列（换行除外）在转换阶段 3 中保留还是替换为一个空格字符 (5.1.1.2)。

  包含制表符 (\t)、换页 (\f) 或纵向输入 (\v) 的非空字符序列将替换为一个空格字符。

C.1.2 环境 (J.3.2)

- 在转换阶段 1 中设置物理源文件多字节字符与源字符集之间的映射 (5.1.1.2)。

  对于 ASCII 部分，一个字符中有八位；对于特定于语言环境的扩展部分，一个字符中的位数是八位的倍数，具体取决于语言环境。
- 在独立式环境中，程序启动时所调用函数的名称和类型 (5.1.2.1)。

  该实现为主机环境。
- 在独立式环境中终止程序的影响 (5.1.2.1)。

  该实现处于主机环境中。
- 定义 `main` 函数可能采用的备选方法 (5.1.2.2.1)。

  除了标准中定义的方法之外，不存在定义 `main` 的备选方法。
- 为字符串指定的值通过 `argv` 参数指向主函数 (5.1.2.2.1)。

  `argv` 是指向命令行参数的指针数组，其中 `argv[0]` 代表程序名称（如果可用）。
- 交互式设备的要素 (5.1.2.3)。

  交互式设备是系统库调用 `isatty()` 为其返回非零值的设备。
- 信号集、信号语义和信号缺省处理 (7.14)。

  下表显示了 `signal` 函数识别的每个信号的语义：

表 C–1 `signal` 函数信号的语义

信号编号	缺省事件	信号的语义
`SIGHUP 1`	退出	挂起
`SIGINT 2`	退出	中断（破坏）
`SIGQUIT 3`	信息转储	退出 (ASCII FS)
`SIGILL 4`	信息转储	非法指令（找到时不重置）
`SIGTRAP 5`	信息转储	跟踪陷阱（捕获时不重置）
`SIGIOT 6`	信息转储	IOT 指令
`SIGABRT 6`	信息转储	由中止使用
`SIGEMT 7`	信息转储	EMT 指令
`SIGFPE 8`	信息转储	浮点异常
`SIGKILL 9`	退出	中止（找不到，也无法忽略）
`SIGBUS 10`	信息转储	总线错误
`SIGSEGV 11`	信息转储	段违规

表 C–1 signal 函数信号的语义 （续）

信号编号	缺省事件	信号的语义
SIGSYS 12	信息转储	系统调用参数错误
SIGPIPE 13	退出	写在管道上，但无读取者
SIGALRM 14	退出	报警时钟
SIGTERM 15	退出	来自中止的软件终止信号
SIGUSR1 16	退出	用户定义的信号 1
SIGUSR2 17	退出	用户定义的信号 2
SIGCLD 18	忽略	子项状态更改
SIGCHLD 18	忽略	子进程状态更改别名 (POSIX)
SIGPWR 19	忽略	电源故障，重新启动
SIGWINCH 20	忽略	窗口大小更改
SIGURG 21	忽略	紧急套接字条件
SIGPOLL 22	退出	发生了可轮询事件
SIGIO 22	Sigpoll	可能有套接字 I/O
SIGSTOP 23	停止	停止（找不到，也无法忽略）
SIGTSTP 24	停止	来自 tty 的用户停止请求
SIGCONT 25	忽略	停止的进程已继续
SIGTTIN 26	停止	已尝试后台 tty 读
SIGTTOU 27	停止	已尝试后台 tty 写
SIGVTALRM 28	退出	虚拟计时器已过期
SIGPROF 29	退出	文件配置计时器已过期
SIGXCPU 30	信息转储	已超出 cpu 限制
SIGXFSZ 31	信息转储	已超出文件大小限制
SIGWAITING 32	忽略	线程代码不再使用保留的信号
SIGLWP 33	忽略	线程代码不再使用保留的信号
SIGFREEZE 34	忽略	检查点暂停
SIGTHAW 35	忽略	检查点恢复
SIGCANCEL 36	忽略	线程库使用的抵消信号
SIGLOST 37	忽略	资源丢失（记录锁定丢失）

表 C–1 signal 函数信号的语义 （续）

信号编号	缺省事件	信号的语义
SIGXRES 38	忽略	超出资源控制（参见 setrctl(2)）
SIGJVM1 39	忽略	保留供 Java 虚拟机 1 使用
SIGJVM2 40	忽略	保留供 Java 虚拟机 2 使用

- 除 SIGFPE、SIGILL 和 SIGSEGV 之外的信号值与计算异常相对应 (7.14.1.1)。

  有关 SIGILL、SIGFPE、SIGSEGV、SIGTRAP、SIGBUS 和 SIGEMT，请参见表 C–1。
- 与 signal(sig, SIG_IGN) 等效的信号；在程序启动时执行 (7.14.1.1)。

  有关 SIGILL、SIGFPE、SIGSEGV、SIGTRAP、SIGBUS 和 SIGEMT，请参见表 C–1。
- 环境名称集以及用来改变 getenv 函数所用环境列表的方法 (7.20.4.5)。

  手册页 environ(5) 中列出了这些环境名称。
- 系统函数执行字符串的方式 (7.20.4.6)。

  摘自 system(3C) 手册页：

  system() 函数会导致 *string* 作为输入提供给 shell，如同在终端将 *string* 作为命令键入一样。调用程序将等待，直到 shell 完成，然后以 waitpid(2) 指定的格式返回 shell 的退出状态。

  如果 *string* 为空指针，则 system() 会检查 shell 是否存在以及是否处于可执行状态。如果 shell 可用，system() 将返回非零值；否则将返回 0。

C.1.3 标识符 (J.3.3)

- 可能出现在标识符中的附加多字节字符以及它们与通用字符名的对应关系 (6.4.2)。

  无
- 标识符中有效初始字符的数目 (5.2.4.1, 6.4.2)。

  1023

C.1.4 字符 (J.3.4)

- 字节中的位数 (3.6)。

  一个字节中有 8 位。
- 执行字符集成员的值 (5.2.1)。

  源代码字符与执行字符之间映射相同。
- 为每个标准字母转义序列生成的执行字符集成员的唯一值 (5.2.2)。

表 C–2 标准字母换码序列唯一值

换码序列	唯一值
\a（报警）	7
\b（退格）	8
\f（换页）	12
\n（换行）	10
\r（回车）	13
\t（水平制表符）	9
\v（垂直制表符）	11

- 将除基本执行字符集成员之外的所有字符存储到其中的 char 对象的值 (6.2.5)。

  它是与分配给 char 对象的字符相关联的低 8 位的数值。
- 是带符号的 char 还是无符号的 char 与“无格式”char 具有相同的范围、表示形式和行为 (6.2.5, 6.3.1.1)。

  将带符号 char 视为“无格式”char。
- 源代码字符集成员（用字符常量和文本字符串表示）到执行字符集成员的映射 (6.4.4.4, 5.1.1.2)。

  源代码字符与执行字符之间映射相同。
- 包含多个字符或包含未映射到单字节执行字符的字符或转义序列的整型字符常量的值 (6.4.4.4)。

  一个多字符常量，它不是具有从每个字符的数值派生的值的转义序列。
- 包含多个多字节字符或不以扩展执行字符集表示的多字节字符或转义序列的宽字符常量的值 (6.4.4.4)。

  一个多字符宽字符常量，它不是具有从每个字符的数值派生的值的转义序列。
- 用于将映射到扩展执行字符集成员的单个多字节字符组成的宽字符常量转换为对应的宽字符代码的当前语言环境 (6.4.4.4)。

  由 LC_ALL、LC_CTYPE 或 LANG 环境变量中指定的有效语言环境。
- 用于将宽文本字符串转换为对应的宽字符代码的当前语言环境 (6.4.5)。

  由 LC_ALL、LC_CTYPE 或 LANG 环境变量指定的有效语言环境。
- 包含不以执行字符集表示的多字节字符或转义序列的文本字符串的值 (6.4.5)。

  多字节字符的每个字节均形成文本字符串的一个字符，并具有等效于多字节字符中该字节数值的值。

C.1.5 整数 (J.3.5)

- 实现中存在的任何扩展整型 (6.2.5)。

  无

- 是否使用符号与大小、2 的补码或 1 的补码来表示带符号整型，以及异常值是陷阱表示还是普通值 (6.2.6.2)。

  带符号整型表示为 2 的补码。异常值是普通值。

- 与具有相同精度的另一个扩展整型有关的任意扩展整型的级别 (6.3.1.1)。

  不适用于此实现。

- 当无法以整型的对象表示该值时，将整数转换为带符号整型的结果或由此引发的信号 (6.3.1.3)。

  当整数转换为较短的带符号整型数时，将低阶位从较长的整数复制到较短的带符号整型数中。结果可能为负数。

  当无符号整型数转换为同等长度的带符号整型数时，将低阶位从无符号整型数复制到带符号整型数中。结果可能为负数。

- 对带符号整型数执行某些按位操作的结果 (6.5)。

  对带符号类型应用按位操作的结果是操作数的按位操作，包括符号位。因此，当且仅当两个操作数中每个对应的位均已置位时，结果中的每个位才置位。

C.1.6 浮点 (J.3.6)

- 浮点运算和返回浮点结果的 `<math.h>` 和 `<complex.h>` 中库函数的准确度 (5.2.4.2.2)。

  浮点运算的准确度与 `FLT_EVAL_METHOD` 的设置一致。`<math.h>` 和 `<complex.h>` 中库函数的准确度已在 `libm`(3LIB) 手册页中指定。

- 以 `FLT_ROUNDS` 的非标准值为特征的舍入行为 (5.2.4.2.2)。

  不适用于此实现。

- 以 `FLT_EVAL_METHOD` 的非标准负值为特征的计算方法 (5.2.4.2.2)。

  不适用于此实现。

- 当整数转换为不能精确表示原始值的浮点数时的舍入方向 (6.3.1.4)。

  它将接受主要的舍入方向模式。

- 当浮点数转换为较窄的浮点数时的舍入方向 (6.3.1.5)。

  它将接受主要的舍入方向模式。

- 如何为某些浮点常量选择最接近的可表示值或与最接近的可表示值接近的较大或较小的可表示值 (6.4.4.2)。

  浮点常量始终舍入为最接近的可表示值。

- 当 `FP_CONTRACT` pragma 未禁止浮点表达式时，是否以及如何规定浮点表达式 (6.5)。

不适用于此实现。

- FENV_ACCESS pragma 的缺省状态 (7.6.1)。

  对于 -fsimple=0，其缺省值为 ON。否则，对于 -fsimple 的所有其他值，FENV_ACCESS 的缺省值为 OFF。

- 其他浮点异常、舍入模式、环境、分类及其宏名称 (7.6, 7.12)。

  不适用于此实现。

- FP_CONTRACT pragma 的缺省状态 (7.12.2)。

  对于 -fsimple=0，其缺省值为 OFF。否则，对于 -fsimple 的所有其他值，FP_CONTRACT 的缺省值都为 ON。

- 当实际舍入的结果与符合 IEC 60559 要求的实现中的数学结果不相等时，是否产生“不精确”的浮点异常 (F.9)。

  结果是不能确定的。

- 当结果很小，但在符合 IEC 60559 要求的实现中很精确时，是否产生下溢（及“不精确”）的浮点异常 (F.9)。

  当禁用对下溢的捕获时（缺省设置），在这种情况下硬件不会产生下溢或不精确。

C.1.7 数组和指针 (J.3.7)

- 将指针转换为整数或 -Xarch=v9 的结果，或者将整数或 -Xarch=v9 转换为指针的结果 (6.3.2.3)。

  在转换指针和整数时，位模式不会更改。当结果无法以整数或指针类型表示时除外，此时结果未定义。

- 减去指向同一个数组元素的两个指针所得结果的大小 (6.5.6)。

  stddef.h 中定义的 int。对于 -Xarch=v9 为 long。

C.1.8 提示 (J.3.8)

- 使用寄存器存储类说明符建议的程度是有效的 (6.7.1)。

  有效寄存器声明数取决于每个函数内使用和定义的模式，并受可供分配的寄存器数的限制。不要求编译器和优化器接受寄存器声明。

- 使用 inline 函数说明符建议的程度是有效的 (6.7.4)。

  只有在使用优化功能时，并且只有在优化器确定进行内联有益时，inline 关键字才能有效对代码进行内联。有关优化选项的列表，请参见第 183 页中的 “A.1.1 优化和性能选项”。

C.1.9 结构、联合、枚举和位字段 (J.3.9)

- “无格式”int 位字段作为带符号 int 位字段处理还是作为无符号 int 位字段处理 (6.7.2, 6.7.2.1)。

  作为无符号 int 进行处理。
- 除 _Bool、带符号 int 和无符号 int 之外其他允许的位字段类型 (6.7.2.1)。

  可将位字段声明为任何整型。
- 位字段是否可以跨存储单元边界 (6.7.2.1)。

  位字段不可以跨存储单元边界。
- 单元中位字段的分配顺序 (6.7.2.1)。

  在存储单元中从高阶到低阶分配位字段。
- 非位字段结构成员的对齐 (6.7.2.1)。除非一个实现编写的二进制数据由另一个实现来读取，否则不会出现任何问题。

表 C–3 结构成员的填充和对齐

类型	对齐边界	字节对齐
`char` and `_Bool`	字节	1
`short`	半字	2
`int`	字	4
`long -m32`	字	4
`long -m64`	双字	8
`float`	字	4
`double -m64`	双字	8
`double` (SPARC) `-m32`	双字	8
`double` (x86) `-m32`	双字	4
`long double` (SPARC) `-m32`	双字	8
`long double` (x86) `-m32`	字	4
`long double -m64`	四倍长字	16
`pointer -m32`	字	4
`pointer -m64`	四倍长字	8
`long long -m64`	双字	8
`long long` (x86) `-m32`	字	4

表 C–3 结构成员的填充和对齐 （续）

类型	对齐边界	字节对齐
`long long` (SPARC) `-m32`	双字	8
`_Complex float`	字	4
`_Complex double -m64`	双字	8
`_Complex double` (SPARC) `-m32`	双字	8
`_Complex double` (x86) `-m32`	双字	4
`_Complex long double -m64`	四倍长字	16
`_Complex long double` (SPARC) `-m32`	四倍长字	8
`_Complex long double` (x86) `-m32`	四倍长字	4
`_Imaginary float`	字	4
`_Imaginary double -m64`	双字	8
`_Imaginary double` (x86) `-m32`	双字	4
`_Imaginary` (SPARC) `-m32`	双字	8
`_Imaginary long double` (SPARC) `-m32`	双字	8
`_Imaginary long double -m64`	四倍长字	16
`_Imaginary long double` (x86) `-m32`	字	4

- 与每种枚举类型兼容的整型 (6.7.2.2)。
  这是 int。

C.1.10 限定符 (J.3.10)

- 访问具有 volatile 限定类型的对象的要素 (6.7.3)。
  对象名称的每个引用构成对该对象的访问。

C.1.11 预处理指令 (J.3.11)

- 如何将这两种头名称形式表示的序列映射到头文件或外部源文件名称 (6.4.7)。
  源文件字符映射至其相应的 ASCII 值。

- 控制条件包含的常量表达式中字符常量的值是否与执行字符集内相同字符常量的值匹配 (6.10.1)。

  预处理指令中的字符常量与其在任何其他表达式中的数值相同。
- 控制条件包含的常量表达式中单字符字符常量的值是否可以有负值 (6.10.1)。

  此上下文中的字符常量可以有负值。
- 搜索空格以查找包括的以 < > 分隔的头文件，并确定如何使用空格来指定其他头文件 (6.10.2)。

  头文件的位置取决于在命令行上指定的选项，以及 `#include` 指令所在的文件。有关更多信息，请参见第 54 页中的 “2.16 如何指定 include 文件”。
- 如何在指定源文件中搜索包括的以 " " 分隔的头文件 (6.10.2)。

  头文件的位置取决于在命令行上指定的选项，以及 `#include` 指令所在的文件。有关更多信息，请参见第 54 页中的 “2.16 如何指定 include 文件”。
- 将 `#include` 指令中预处理标记（可能由宏扩展生成）合并为头文件名称所使用的方法 (6.10.2)。

  构成头文件名称的所有标记（包括空白）都被视为搜索头文件时使用的文件路径，如第 54 页中的 “2.16 如何指定 include 文件”中所述。
- `#include` 处理的嵌套限制 (6.10.2)。

  编译器不施加任何限制。
- # 操作符是否在字符常量或文本字符串中位于通用字符名称开头的 \ 字符之前插入 \ 字符 (6.10.3.2)。

  不可以。
- 每个已识别的非 STDC `#pragma` 指令的行为 (6.10.6)。

  有关每个已识别的非 STDC `#pragma` 指令的行为说明，请参见第 40 页中的 “2.11 Pragma”。
- 当转换的日期和时间不可用时，`__DATE__` 和 `__TIME__` 的各自定义 (6.10.8)。

  环境中总是提供这些宏。

C.1.12 库函数 (J.3.12)

- 独立式程序可用的任何库工具，子句 4 要求的最低设置除外 (5.1.2.1)。

  该实现处于主机环境中。
- 断言宏输出的诊断格式 (7.2.1.1)。

  诊断的结构如下：

  Assertion failed: *statement*. 文件 *filename*、*line number*、*function name*

  statement 是使断言失败的语句。*filename* 是 `__FILE__` 的值。*line number* 是 `__LINE__` 的值。*function name* 是 `__func__` 的值。

- 由 fegetexceptflag 函数存储的浮点状态标志的表示 (7.6.2.2)。

  fegetexceptflag 存储在状态标志中的每个异常均可扩展至具有值的整数常量表达式，这样，所有常量组合的按位包括 OR 会得到不同的值。
- 除了“上溢”或“下溢”浮点异常之外，feraiseexcept 函数是否会产生“不精确”浮点异常 (7.6.2.3)。

  不，不会产生“不精确”异常。
- 除 "C" 和 "" 之外可以作为第二个参数传递给 setlocale 函数的其他字符串 (7.11.1.1)。

  故意留空。
- 当 FLT_EVAL_METHOD 宏的值小于零或大于二时针对 float_t 和 double_t 定义的类型 (7.12)。
  - 对于 SPARC，类型如下：

    ```
    typedef float float_t;

    typedef double double_t;
    ```
  - 对于 x86，类型如下：

    ```
    typedef long double float_t;

    typedef long double double_t;
    ```

  数学函数的域错误，与此国际标准的要求不同 (7.12.1)。

  如果输入参数为 0、+/-Inf 或 NaN，ilogb()、ilogbf() 和 ilogbl() 会生成无效的异常。
- 发生域错误时数学函数返回的值 (7.12.1)。

  在完整的 C99 模式 (-xc99=%all,lib) 下发生域错误时返回的值，将遵循 ISO/IEC 9899:1999 编程语言 C 附录 F 中的规定。
- 发生下溢范围错误时数学函数返回的值，当整型表达式 math_errhandling & MATH_ERRNO 为非零时， 为非零时，是否产生“下溢”浮点异常。是否将 errno 设置为宏 ERANGE 的值，以及当整形表达式 math_errhandling & MATH_ERREXCEPT 为非零时是否产生“下溢”浮点异常。(7.12.1)。

  对于下溢范围错误：如果该值可表示为非正规数，则返回非正规数；否则根据情况返回 +-0。

  至于当整型表达式 math_errhandling & MATH_ERRNO 为非零时是否将 errno 设置为宏 ERANGE 的值，由于在我们的实现中 (math_errhandling & MATH_ERRNO) == 0，因此不会应用这一部分。

  当整型表达式 math_errhandling & MATH_ERREXCEPT 为非零时是否会产生“下溢”浮点异常 (7.12.1)，如果在发生浮点下溢的同时伴随着准确度的降低，则会产生异常。
- 当 fmod 函数还有一个参数为零时，发生域错误还是返回零 (7.12.10.1)。

  发生域错误。

- 在减小商时 remquo 函数使用的以 2 为基数的模数对数 (7.12.10.3)。

  31。
- 在调用信号处理程序之前是否执行 signal(sig, SIG_DFL); 的等效函数，如果不执行，则阻塞执行的信号 (7.14.1.1)。

  在调用信号处理程序之前执行 signal(sig, SIG_DFL); 的等效函数。
- 宏 NULL 扩展到的空指针常量 (7.17)。

  NULL 扩展到 0。
- 文本流的最后一行是否需要一个终止换行符 (7.19.2)。

  最后一行不需要以换行符结束。
- 写出到换行符前面的文本流中的空格字符在读入时是否出现 (7.19.2)。

  读该流时，所有字符均出现。
- 可附加至写入二进制流的数据的空字符数 (7.19.2)。

  空字符不附加至二进制流。
- 附加模式流的文件位置指示符最初位于文件开头还是结尾 (7.19.3)。

  文件位置指示符最初位于文件结尾。
- 对文本流的写操作是否导致关联的文件在该点之后被截断 (7.19.3)。

  对文本流的写操作不会导致文件在该点之后被截断，除非硬件设备强制这种情况发生。
- 文件缓冲的特征 (7.19.3)。

  除标准错误流 (stderr) 之外，输出流在输出至文件时缺省情况下缓冲，在输出至终端时采用行缓冲。标准错误输出流 (stderr) 在缺省情况下不缓冲。

  缓冲的输出流保存多个字符，然后将这些字符作为块进行写入。未缓冲的输出流将信息排队，以便立即在目标文件或终端上写入。行缓冲的输出将输出的每行排队，直至行完成（请求换行符）时为止。
- 零长度文件是否确实存在 (7.19.3)。

  由于零长度文件有目录项，因此它确实存在。
- 书写有效文件名的规则 (7.19.3)。

  有效文件名的长度可以为 1 到 1,023 个字符，并且可以使用除字符 null 和 /（斜杠）之外的所有字符。
- 同一文件是否可以同时打开多次 (7.19.3)。

  同一文件可以多次打开。
- 文件中多字节字符所用编码的性质和选项 (7.19.3)。

  对于每个文件来说，多字节字符所用的编码均相同。
- `remove` 函数对打开的文件的作用 (7.19.4.1)。

  在执行关闭文件的最后一个调用时删除文件。程序不能打开已删除的文件。

- 在调用 rename 函数之前存在一个具有新名称的文件时的作用 (7.19.4.2)。

  如果该文件存在，则将其删除，并且新文件改写先前存在的文件。
- 程序异常终止后是否删除打开的临时文件 (7.19.4.3)。

  如果在文件创建与解除链接这段时期内终止进程，则可能会留下永久文件。请参见 freopen(3C) 手册页。
- 允许模式进行哪些更改（如果有），并且在什么情况下允许更改 (7.19.5.4)。

  允许模式进行以下更改，具体取决于流下面的文件描述符的访问模式：
  - 指定 + 后，文件描述符模式必须为 O_RDWR。
  - 指定 r 后，文件描述符模式必须为 O_RDONLY 或 O_RDWR。
  - 指定 a 或 w 后，文件描述符模式必须为 O_WRONLY 或 O_RDWR。

    请参见 freopen (3C) 手册页。

  用于输出无穷大或 NaN 的样式，以及为 NaN 输出的任何 n-char 或 n-wchar 序列的含义 (7.19.6.1, 7.24.2.1)。

  [-]Inf, [-]NaN。具有 F 转换说明符时，则为 [-]INF, [-]NAN。
- 在 fprintf 或 fwprintf 函数中 %p 转换的输出 (7.19.6.1, 7.24.2.1)。

  %p 的输出与 %x 的相同。
- 在 fscanf() 或 fwscanf() 函数中 %[转换的扫描列表中，当 - 字符既不是第一个字符也不是最后一个字符，又不是 ^ 字符为第一个字符时的第二个字符，此时对 - 字符的解释 (7.19.6.2, 7.24.2.1)。

  如果 - 存在于扫描列表中，并且既不是第一个字符，也不是第二个字符（当第一个字符为 ^ 时），又不是最后一个字符，则表示符合字符的范围。

  请参见 fscanf(3C) 手册页。
- fscanf() 或 fwscanf() 函数中 %p 转换匹配的序列集合以及相应输入项的解释 (7.19.6.2, 7.24.2.2)。

  匹配与相应 printf(3C) 函数的 %p 转换所生成的序列集合相同的序列集合。相应的参数必须是 void 指针的指针。如果输入项是在执行同一个程序时较早转换的值，则得到的指针将等于该值；否则 %p 转换的行为不确定。

  请参见 fscanf(3C) 手册页。
- 发生故障时 fgetpos、fsetpos 或 ftell 函数将宏 errno 设置为的值 (7.19.9.1, 7.19.9.3, 7.19.9.4)。
  - EBADF 流下面的文件描述符无效。请参见 fgetpos(3C) 手册页。
  - ESPIPE 流下面的文件描述符与管道、FIFO 或套接字相关联。请参见 fgetpos(3C) 手册页。
  - EOVERFLOW 在类型为 fpos_t 的对象中无法正确地表示文件位置的当前值。请参见 fgetpos(3C) 手册页。
  - EBADF 流下面的文件描述符无效。请参见 fsetpos(3C) 手册页。

- ESPIPE 流下面的文件描述符与管道、FIFO 或套接字相关联。请参见 fsetpos(3C) 手册页。
- EBADF 流下面的文件描述符不是打开的文件描述符。请参见 ftell(3C) 手册页。
- ESPIPE 流下面的文件描述符与管道、FIFO 或套接字相关联。请参见 ftell(3C) 手册页。
- EOVERFLOW 在类型为 long 的对象中无法正确地表示当前文件偏移。请参见 ftell(3C) 手册页。

在表示由 strtod()、strtof()、strtold()、wcstod()、wcstof() 或 wcstold() 函数转换的 NaN 的字符串中，任何 n-char 或 n-wchar 序列的含义 (7.20.1.3, 7.24.4.1.1)。

未给 n-char 序列赋予特殊意义。

- 发生下溢时，strtod、strtof、strtold、wcstod、wcstof 或 wcstold 函数是否将 errno 设置为 ERANGE (7.20.1.3, 7.24.4.1.1)。

  是，发生下溢时将 errno 设置为 ERANGE。
- 当请求的大小为零时，calloc、malloc 和 realloc 函数返回空指针还是返回指向已分配对象的指针 (7.20.3)。

  将返回空指针或可传递给 free() 的唯一指针。

  请参见 malloc(3C) 手册页。
- 调用 abort 或 _Exit 函数时，是刷新具有未写入缓冲数据的开放流、关闭开放流还是删除临时文件 (7.20.4.1, 7.20.4.4)。

  异常终止处理至少包括 fclose(3C) 对所有开放流的影响。请参见 abort(3C) 手册页。

  关闭开放流，并且不刷新开放流。请参见 _Exit(2) 手册页。
- abort、exit 或 _Exit 函数返回给主机环境的终止状态 (7.20.4.1, 7.20.4.3, 7.20.4.4)。

终止操作可用于 wait(3C) 或 waitpid(3C) 的状态是由 SIGABRT 信号终止的进程状态。请参见 abort(3C)、exit(1) 和 _Exit(2) 手册页。

由 exit 或 _Exit 返回的终止状态，具体取决于正在进行的调用进程的父进程。

如果调用进程的父进程正在执行 wait(3C)、wait3(3C)、waitid(2) 或 waitpid(3C)，并且既未设置 SA_NOCLDWAIT 标志，也未将 SIGCHLD 设置为 SIG_IGN，则它会被告知调用进程的终止，并且它可使用状态的低阶八位（即，位 0377）。如果父进程未处于等待状态，则当父进程随后执行 wait()、wait3()、waitid() 或 waitpid() 时可以使用子进程的状态。

- system 函数在其参数不是空指针时返回的值 (7.20.4.6)。

  采用 waitpid(3C) 指定格式的 shell 的退出状态。
- 本地时区和夏令时 (7.23.1)。

  本地时区由环境变量 TZ 设置。

- 可以用 clock_t 和 time_t 表示的时间的范围和精度 (7.23)。

  clock_t 和 time_t 的精度是一百万分之一秒。x86 和 sparc v8 上的范围为 -2147483647-1 到 4294967295 百万分之一秒。而 SPARC v9 上的范围为 -9223372036854775807LL-1 到 18446744073709551615。

- clock 函数的年代 (7.23.2.1)。

  时钟的年代以时钟周期（以程序的起始执行时间为起点）表示。

- "C" 语言环境中 strftime 和 wcsftime 函数的 %Z 说明符的替换字符串 (7.23.3.5, 7.24.5.1)。

  时区名称或缩写，如果未确定任何时区，则不使用任何字符。

- trigonometric、hyperbolic、以 e 为基数的指数、以 e 为基数的对数、错误和对数伽玛函数是否或何时在符合 IEC 60559 要求的实现中产生“不精确”的浮点异常 (F.9)。

  当不能准确地表示结果时，通常会产生不精确的异常。即使能准确地表示结果，也可能产生不精确的异常。

- <math.h> 中的函数是否在符合 IEC 60559 要求的实现中接受舍入方向模式 (F.9)。

  不尝试对 <math.h> 中的所有函数强制使用缺省舍入方向模式。

C.1.13 体系结构 (J.3.13)

- 为头文件 <float.h>、<limits.h> 和 <stdint.h> 中指定的宏分配的值或表达式 (5.2.4.2, 7.18.2, 7.18.3)。
  - 下面是可用于 <float.h> 中所指定宏的值或表达式：

```
#define CHAR_BIT 8 /* max # of bits in a “char” */
#define SCHAR_MIN (-128) /* min value of a “signed char” */
#define SCHAR_MAX 127 /* max value of a “signed char” */
#define CHAR_MIN SCHAR_MIN /* min value of a “char” */
#define CHAR_MAX SCHAR_MAX /* max value of a “char” */
#define MB_LEN_MAX 5
#define SHRT_MIN (-32768) /* min value of a “short int” */
#define SHRT_MAX 32767 /* max value of a “short int” */
#define USHRT_MAX 65535 /* max value of “unsigned short int” */
#define INT_MIN (-2147483647-1) /* min value of an “int” */
#define INT_MAX 2147483647 /* max value of an “int” */
#define UINT_MAX 4294967295U /* max value of an “unsigned int” */
#define LONG_MIN (-2147483647L-1L)
#define LONG_MAX 2147483647L /* max value of a “long int” */
#define ULONG_MAX 4294967295UL /* max value of “unsigned long int” */
#define LLONG_MIN (-9223372036854775807LL-1LL)
#define LLONG_MAX 9223372036854775807LL
#define ULLONG_MAX 18446744073709551615ULL

#define FLT_RADIX 2
#define FLT_MANT_DIG 24
#define DBL_MANT_DIG 53
#define LDBL_MANT_DIG 64
```

```
#if defined(__sparc)
#define DECIMAL_DIG 36
#elif defined(__i386)
#define DECIMAL_DIG 21
#endif
#define FLT_DIG 6
#define DBL_DIG 15
#if defined(__sparc)
#define LDBL_DIG 33
#elif defined(__i386)
#define LDBL_DIG 18
#endif

#define FLT_MIN_EXP (-125)
#define DBL_MIN_EXP (-1021)
#define LDBL_MIN_EXP (-16381)

#define FLT_MIN_10_EXP (-37)
#define DBL_MIN_10_EXP (-307)
#define LDBL_MIN_10_EXP (-4931)

#define FLT_MAX_EXP (+128)
#define DBL_MAX_EXP (+1024)
#define LDBL_MAX_EXP (+16384)

#define FLT_EPSILON 1.192092896E-07F
#define DBL_EPSILON 2.2204460492503131E-16

#if defined(__sparc)
#define LDBL_EPSILON 1.92592994438723585305597794258492731 9E-34L
#elif defined(__i386)
#define LDBL_EPSILON 1.08420217248550443400 75E-19L
#endif

#define FLT_MIN 1.175494351E-38F
#define DBL_MIN 2.2250738585072014E-308

#if defined(__sparc)
#define LDBL_MIN 3.3621031431120935062626778173217526 03E-4932L
#elif defined(__i386)
#define LDBL_MIN 3.362103143112093506262 7E-4932L
#endif
```

下面是可用于 <limits.h> 中所指定宏的值或表达式：

```
#define INT8_MAX (127)
#define INT16_MAX (32767)
#define INT32_MAX (2147483647)
#define INT64_MAX (9223372036854775807LL)

#define INT8_MIN (-128)
#define INT16_MIN (-32767-1)
#define INT32_MIN (-2147483647-1)
#define INT64_MIN (-9223372036854775807LL-1)

#define UINT8_MAX (255U)
#define UINT16_MAX (65535U)
#define UINT32_MAX (4294967295U)
#define UINT64_MAX (18446744073709551615ULL)
```

```
#define INT_LEAST8_MIN INT8_MIN
#define INT_LEAST16_MIN INT16_MIN
#define INT_LEAST32_MIN INT32_MIN
#define INT_LEAST64_MIN INT64_MIN

#define INT_LEAST8_MAX INT8_MAX
#define INT_LEAST16_MAX INT16_MAX
#define INT_LEAST32_MAX INT32_MAX
#define INT_LEAST64_MAX INT64_MAX

#define UINT_LEAST8_MAX UINT8_MAX
#define UINT_LEAST16_MAX UINT16_MAX
#define UINT_LEAST32_MAX UINT32_MAX
#define UINT_LEAST64_MAX UINT64_MAX
```

- 下面是可用于 <stdint.h> 中所指定宏的值或表达式：

```
#define INT_FAST8_MIN INT8_MIN
#define INT_FAST16_MIN INT16_MIN
#define INT_FAST32_MIN INT32_MIN
#define INT_FAST64_MIN INT64_MIN

#define INT_FAST8_MAX INT8_MAX
#define INT_FAST16_MAX INT16_MAX
#define INT_FAST32_MAX INT32_MAX
#define INT_FAST64_MAX INT64_MAX

#define UINT_FAST8_MAX UINT8_MAX
#define UINT_FAST16_MAX UINT16_MAX
#define UINT_FAST32_MAX UINT32_MAX
#define UINT_FAST64_MAX UINT64_MAX
```

- 在任何对象（如果此国际标准中未明确指定）中字节的数值、顺序和编码 (6.2.6.1)。

本章中其他位置的内容定义了 1999 C 标准中未明确指定的对象的实现定义数值、顺序和编码。

- sizeof 操作符结果的值 (6.5.3.4)。

  下表列出了 sizeof 的结果。

表 C–4 sizeof 操作符所得到的结果（以字节计）

类型	大小（以字节计）
char 和 _Bool	1
short	2
int	4
long	4
long -m64	8
long long	8

表 C–4 sizeof 操作符所得到的结果（以字节计） （续）

类型	大小（以字节计）
float	4
double	8
long double (SPARC)	16
long double (x86) -m32	12
long double (x86) -m64	16
pointer	4
pointer -m64	8
_Complex float	8
_Complex double	16
_Complex long double (SPARC)	32
_Complex long double (x86) -m32	24
_Complex long double (x86) -m64	32
_Imaginary float	4
_Imaginary double	8
_Imaginary long double (SPARC)	16
_Imaginary long double (x86) -m32	12
_Imaginary long double (x86) -m64	16

C.1.14 语言环境特定的行为 (J.4)

主机环境的以下特征与特定的语言环境有关，并且要求通过该实现将其编制成文档：

- 基本字符集之外的源代码字符集和执行字符集的附加成员 (5.2.1)。
  特定于语言环境（C 语言环境中无扩展）。
- 基本字符集之外的执行字符集中附加多字节字符的存在、含义和表示 (5.2.1.2)。
  缺省或 C 语言环境的执行字符集中不存在多字节字符。
- 用于多字节字符编码的移位状态 (5.2.1.2)。
  无移位状态。
- 连续输出字符的写入方向 (5.2.2)。
  输出总是从左到右进行。
- 小数点字符 (7.1.1)。

特定于语言环境（在 C 语言环境中为 "."）。

- 输出字符集（7.4，7.25.2）。

  特定于语言环境（在 C 语言环境中为 "."）。

- 控制字符集（7.4，7.25.2）。

  控制字符集由水平制表符、垂直制表符、换页、报警、退格、回车和换行符组成。

- 由
  `isalpha`、`isblank`、`islower`、`ispunct`、`isspace`、`isupper`、`iswalpha`、`iswblank`、`iswlower`
  或 `iswupper` 函数测试的字符集（7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11）。

  有关 `isalpha`() 和 `iswalpha`() 的说明以及有关上述相关宏的信息，请参见 `isalpha`(3C) 和 `iswalpha`(3C) 手册页。请注意，通过更改语言环境，可以修改其行为。

- 本地环境 (7.11.1.1)。

  如 `setlocale`((3C) 手册页所述，本地环境由 `LANG` 和 `LC_*` 环境变量指定。但是，如果未设置这些环境变量，则将本地环境设置为 C 语言环境。

- 数值转换函数接受的附加主题序列 (7.20.1, 7.24.4.1)。

  基数字符是在程序的语言环境（种类为 `LC_NUMERIC`）中定义的，并且可能被定义为除句点 (.) 之外的其他值。

- 执行字符集的整理序列 (7.21.4.3, 7.24.4.4.2)。

  特定于语言环境（C 语言环境中的 ASCII 整理）。

- `strerror` 函数设置的错误消息字符串的内容 (7.21.6.2)。

  如果应用程序与 `-lintl` 相链接，则此函数返回的消息以 `LC_MESSAGES` 语言环境种类指定的本地语言显示。否则以 C 语言环境中的设置显示。

- 时间和日期的格式（7.23.3.5，7.24.5.1）。

  特定于语言环境。下面几个表显示 C 语言环境中的格式。

  指定的月份名称如下：

表 C–5 月份名称

1 月	5 月	9 月
2 月	6 月	10 月
3 月	7 月	11 月
4 月	8 月	12 月

指定的周日期名称如下：

表 C–6 周日期及缩写

日期	缩写
星期日 星期四	日 四
星期一 星期五	一 五
星期二 星期六	二 六
星期三	三

时间的格式为：

`%H:%M:%S`

日期的格式为：

`%m/%d/ -Xc` 模式。

上午和下午的格式为：上午 下午

- `towctrans` 函数支持的字符映射 (7.25.1)。

  程序的语言环境（种类为 `LC_CTYPE`）中有关字符映射的信息定义了已编码字符集的规则，该规则可能规定了除 `tolower` 和 `toupper` 之外的字符映射。有关可用语言环境及其定义的详细信息，请参阅《*Solaris Internationalization Guide For Developers*》 。
- `iswctype` 函数支持的字符分类 (7.25.1)。

  有关可用语言环境以及任何非标准保留字符分类的详细信息，请参见《*Solaris Internationalization Guide For Developers*》

◆ ◆ ◆ 附 录 D

支持的 C99 功能

此附录列出了 C 编程语言标准 ISO/IEC 9899:1999 支持的功能。

`-xc99` 标志可控制编译器对实现功能的识别。有关 `-xc99` 语法的更多信息，请参见第 227 页中的 “B.2.79 -xc99[= *o*]”。

注 – 虽然编译器在缺省情况下支持下面列出的 C99 功能，但是 `/usr/include` 中由 Solaris 软件提供的标准头文件仍不符合 1999 ISO/IEC C 标准。如果遇到错误消息，请尝试使用 `-xc99=none` 获取这些头文件的 1990 ISO/IEC C 标准行为。

D.1 讨论和示例

本附录提供了有关下列某些支持的功能的讨论和示例。

- 子条款 5.2.4.2.2 浮点类型 `<float.h>` 的特性
- 子条款 6.2.5 _Bool
- 子条款 6.2.5 _Complex 类型
- 子条款 6.3.2.1 不限制仅在左值进行数组到指针的转化
- 子条款 6.4.1 关键字
- 子条款 6.4.2.2 预定义标识符
- 6.4.3 通用字符名
- 子条款 6.4.4.2 十六进制浮点文字
- 子条款 6.4.9 注释
- 子条款 6.5.2.2 函数调用
- 子条款 6.5.2.5 复合文字
- 子条款 6.7.2 函数说明符
- 子条款 6.7.2.1 结构和联合说明符
- 子条款 6.7.3 类型限定符
- 子条款 6.7.4 函数说明符
- 子条款 6.7.5.2 数组声明符
- 子条款 6.7.8 初始化

- 子条款 6.8.2 复合语句
- 子条款 6.8.5 迭代语句
- 子条款 6.10.3 宏替换
- 子条款 6.10.6 STDC pragma
- 子条款 6.10.8 __STDC_IEC_559 和 __STDC_IEC_559_COMPLEX 宏
- 子条款 6.10.9 Pragma 操作符

D.1.1 浮点计算器的精度

5.2.4.2.2 浮点类型 <float.h> 的特性

使用浮点操作数进行运算的值以及同时符合常见算术转换和浮点常量的值经过计算所得格式的范围和精度可能大于该类型所要求的范围和精度。计算格式的使用以实现定义的 FLT_EVAL_METHOD 值为特征：

表 D–1 FLT_EVAL_METHOD 值

值	含义
-1	不能确定的。
0	编译器仅根据该类型的范围和精度计算所有运算和常量的结果。
1	编译器根据双精度的范围和精度计算浮点和双精度类型的运算和常量的结果。根据长双精度的范围和精度计算长双精度运算和常量的结果。
2	编译器仅根据长双精度的范围和精度计算所有运算和常量的结果。

当您将 float.h 包括在 SPARC 体系结构中时，缺省情况下 FLT_EVAL_METHOD 将扩展到 0，并根据其类型计算所有浮点表达式的结果。

当您将 float.h 包括在 x86 体系结构中时，缺省情况下 FLT_EVAL_METHOD 将扩展到 -1（当 -xarch=sse2 或 -xarch=amd64 时除外），并根据其类型计算所有浮点常量表达式的结果，同时将所有其他浮点表达式按长双精度类型进行计算。

当您指定 -flteval=2 并包括 float.h 时，FLT_EVAL_METHOD 将扩展到 2，并将所有浮点表达式按长双精度类型进行计算。有关更多信息，请参见第 203 页中的 “B.2.21 -flteval[={ any|2}]”。

当您在 x86 上指定 -xarch=sse2（或任何比 SSE2 处理器系列更新的版本，例如 sse3、ssse3、sse4_1、sse4_2 等）或 -m64 并包括 float.h时，FLT_EVAL_METHOD 将扩展到 0，并将所有浮点表达式按它们的类型进行计算。

即使浮点表达式按双精度类型来计算结果，-Xt 选项也不影响 FLT_EVAL_METHOD 的扩展。有关更多信息，请参见第 218 页中的 “B.2.68 -X[c|a| t|s]”。

使用 -fsingle 选项将导致浮点表达式以单精度来计算结果。有关更多信息，请参见第 207 页中的 “B.2.30 -fsingle”。

当您在 x86 体系结构上指定 `-fprecision` 以及 `-xarch=sse2`（或 SSE2 处理器系列的任何更高版本，例如 `sse3`、`ssse3`、`sse4_1`、`sse4_2` 等）或 `-m64`，并包括 `float.h` 时，`FLT_EVAL_METHOD` 将扩展到 `-1`。

D.1.2 C99 关键字

6.4.1 关键字

C99 标准引入了以下新的关键字。在 `-xc99=none` 的情况下编译时，如果使用这些关键字作为标识符，编译器将会发出警告。在不设置 `-xc99=none` 的情况下，编译器会根据上下文针对使用这些关键字作为标识符发出警告或错误消息。

- `inline`
- `_Imaginary`
- `_Complex`
- `_Bool`
- `restrict`

D.1.2.1 使用 `restrict` 关键字

通过 `restrict` 限定指针访问的对象要求对该对象的所有访问都直接或间接使用该特定 `restrict` 限定指针的值。通过任何其他方式访问该对象可能导致不确定的行为。`restrict` 限定符的既定用途是允许编译器做出提升优化的假定。

有关如何有效使用 `restrict` 限定符的示例和说明，请参见第 75 页中的 “3.8.2 限定指针”。

D.1.3 `__func__` 支持

6.4.2.2 预定义的标识符。

编译器支持预定义标识符 `__func__`。__func__ 定义为字符数组，它包含 __func__ 所在的当前函数的名称。

D.1.4 通用字符名 (UCN)

6.4.3 通用字符名

UCN 允许在 C 源码中使用任何字符，而不仅仅是英文字符。UCN 的格式是：

- \u 4_hex_digits_value
- \U 8_hex_digits_value

UCN 不能指定 0024 ($)、0040 (@) 或 0060 (?) 以外小于 00A0 的值，也不能指定 D800 到 DFFF（包含）范围内的值。

UCN 可用于标识符、字符常量和文本字符串中，以指定 C 基本字符集中不存在的字符。

UCN \unnnnnnnn 指定了其八位短标识符（根据 ISO/IEC 10646 的规定）为 nnnnnnnn 的字符。同样，通用字符名nnnn 指定了其四位短标识符为 nnnn（其八位短标识符为 0000nnnn）的字符。

D.1.5 使用 // 注释代码

6.4.9 注释

字符 // 引入包含直到（但不包括）新换行符的所有多字节字符的注释，除非 // 字符出现在字符常量、文本字符串或注释中。

D.1.6 禁止隐式 int 和隐式函数声明

6.5.2.2 函数调用

与 1990 C 标准不同，1999 C 标准不再允许隐式声明。C 编译器的以前版本仅在设置了 `-v`（详细）的情况下发出有关隐式定义的警告消息。只要标识符隐式定义为 `int` 或函数，系统便会对隐式定义发出这些消息及其他新警告。

该编译器的几乎所有用户均可能注意到这种变化，原因是它会导致大量警告消息。常见原因包括未能包含用于声明所使用函数的相应系统头文件，如需要包含 `<stdio.h>` 的 `printf`。可以使用 `-xc99=none` 恢复无提示地接受隐式声明的 1990 C 标准行为。

C 编译器对隐性函数声明生成警告：

```
example% cat test.c
void main()
{
  printf("Hello, world!\n");
}
example% cc test.c
"test.c", line 3: warning: implicit function declaration: printf
example%
```

D.1.7 使用隐式 int 的声明

6.7.2 类型说明符：

每个声明中的声明说明符中应至少指定一个类型说明符。另请参见第 304 页中的 “D.1.6 禁止隐式 int 和隐式函数声明”。

现在，C 编译器会对任何隐式 `int` 声明都发出警告，如以下示例所示：

```
example% more test.c
volatile i;
const foo()
{
  return i;
}
example% cc test.c
  "test.c", line 1: warning: no explicit type given
  "test.c", line 3: warning: no explicit type given
example%
```

D.1.8 灵活的数组成员

6.7.2.1 结构和联合说明符

也被称为 "struct hack"。允许结构的最后一个成员是长度为零的数组，如 `int foo[] ;`。这种结构一般用作访问 malloc 内存的头文件。

例如，在结构 `struct s { int n; double d[]; } S;` 中，数组 `d` 是不完整数组类型。对于 `S` 的该成员，C 编译器不对任何内存偏移进行计数。换句话说，`sizeof(struct s)` 与 `S.n` 的偏移相同。

可以像使用任何普通数组成员一样使用 `d`。`S.d[10] = 0;`。

如果没有 C 编译器对不完整数组类型的支持，您将按以下称为 `DynamicDouble` 的示例所示定义和声明结构：

```
typedef struct { int n; double d[1]; ) DynamicDouble;
```

请注意，数组 `d` 不是不完整数组类型，并且已使用一个成员进行声明。

下一步，声明指针 `dd` 并分配内存，如下所示：

```
DynamicDouble *dd = malloc(sizeof(DynamicDouble)+(actual_size-1)*sizeof(double));
```

然后，将偏移的大小存储在 `S.n` 中，如下所示：

```
dd->n = actual_size;
```

由于编译器支持不完全数组类型，因此您无需使用一个成员声明数组即可达到同样的效果。

```
typedef struct { int n; double d[]; } DynamicDouble;
```

如以前一样声明指针 `dd` 并分配内存，只是不必再从 actual_size 中减去 1：

```
DynamicDouble *dd = malloc (sizeof(DynamicDouble) + (actual_size)*sizeof(double));
```

如以前一样将偏移存储在 S.n 中，因此：

```
dd->n = actual_size;
```

D.1.9 幂等限定符

6.7.3 类型限定符：

如果同一限定符在同一说明符限定符列表中出现多次（无论直接出现还是通过一个或多个 typedef），行为与该类型限定符仅出现一次时相同。

在 C90 中，以下代码会导致错误：

```
%example cat test.c

const const int a;

int main(void) {
  return(0);
}

%example cc -xc99=none test.c
"test.c", line 1: invalid type combination
```

但是，对于 C99，C 编译器接受多个限定符。

```
%example cc -xc99 test.c
%example
```

D.1.10 `inline` 函数

6.7.4 函数说明符

完全支持 1999 C ISO 标准定义的内联函数。

请注意，根据 C 标准，内联仅仅是对 C 编译器的建议。C 编辑器可以选择不内联任何内容，而编译对实际函数的调用。

仅当在 **-x03** 或更高优化级别进行编译并且优化器的启发式算法判断这样有利时，Solaris Studio C 编译器才会内联 C 函数调用。C 编译器不提供强制内联函数的方法。

静态内联函数很简单。在引用处内联使用内联函数说明符定义的函数，或者对实际函数进行调用。编译器可以选择在每个引用处执行的操作。编辑器确定在 **-x03** 及更高级别进行内联是否有益。如果内联无益（或在低于 **-x03** 的优化级别进行内联），将编译对实际函数的引用，并将函数定义编译为目标代码。请注意，如果程序使用函数的地址，将在目标代码中编译实际的函数，而不进行内联。

外部内联函数较为复杂。有两种类型的外部内联函数：**内联定义**和**外部内联函数**。

内联定义是使用关键字 `inline` 定义的函数，不包含关键字 `static` 或 `extern`，并且显示在源文件（或包含的文件）中的所有原型也包含关键字 `inline`，而不包含关键字 `static` 或 `extern`。对于内联定义，编译器不得创建函数的全局定义。这意味着对非内

联的内联定义的任何引用都将是对在其他位置定义的全局函数的引用。换句话说，编译此翻译单元（源文件）所产生的目标文件将不会包含内联定义的全局符号。对非内联的函数的任何引用都将是对由其他某个目标文件或库在链接时提供的外部（全局）符号的引用。

外部内联函数由具有 extern 存储类说明符（即函数定义和/或原型）的文件范围声明来声明。对于外部内联函数，编译器将在生成的目标文件中提供此函数的全局定义。编译器可以选择内联对该函数的任何引用（该函数在提供其定义的翻译单元（源文件）中可见），也可以选择调用全局函数。

未定义依赖于函数调用实际是否内联的任何程序的行为。

另外需要注意的是，具有外部链接的内联函数不得声明或引用翻译单元任意位置处的静态变量。

D.1.10.1 Solaris Studio C 编译器针对内联函数的 gcc 兼容性

要从 Solaris Studio C 编译器中获取与大多数程序中外部内联函数的 gcc 实现兼容的行为，请使用 **`-features=no%extinl`** 标志。指定了此标志时，Solaris Studio C 编译器会将此函数视为已声明为静态内联函数。

当获取此函数的地址时，会发生此行为不兼容的情况。对于 gcc，这将是全局函数的地址；对于 Sun 编译器，将使用局部静态定义地址。

D.1.11 `Static` 及数组声明符中允许的其他类型限定符

6.7.5.2 数组声明符：

现在，关键字 `static` 可以出现在函数声明符中参数的数组声明符中，表示编译器至少可以假定许多元素将传递到所声明的函数中。使优化器能够作出以其他方式无法确定的假定。

C 编译器会将数组参数调整为指针，因此 `void foo(int a[])` 与 `void foo(int *a)` 相同。

如果您指定 `void foo(int * restrict a);` 等类型限定符，则 C 编译器使用实质上与声明受限指针相同的数组语法 `void foo(int a[restrict]);` 表示它。

C 编译器还使用 `static` 限定符保留关于数组大小的信息。例如，如果您指定 `void foo(int a[10])`，则编译器仍将其表示为 `void foo(int *a)`。按以下所示使用 `static` 限定符：`void foo(int a[static 10])`，让编译器知道指针 a 不是 NULL，并且使用它可访问至少包含十个元素的整数数组。

D.1.12 可变长度数组 (VLA)：

6.7.5.2 数组声明符

在栈中分配 VLA 时，仿佛调用了 alloca 函数。无论其作用域如何，其生存期与通过调用 alloca 在栈中分配数据时相同；直到函数返回时为止。如果在其中分配 VLA 的函数返回时释放栈，则释放分配的空间。

尚未对可变长度数组强制施加所有约束。约束违规导致不确定的结果。

```
#include <stdio.h>
void foo(int);

int main(void) {
  foo(4);
  return(0);
}

void foo (int n) {
  int i;
  int a[n];
  for (i = 0; i < n; i++)
    a[i] = n-i;
  for (i = n-1; i >= 0; i--)
    printf("a[%d] = %d\n", i, a[i]);
}

example% cc test.c
example% a.out
a[3] = 1
a[2] = 2
a[1] = 3
a[0] = 4
```

D.1.13 指定的初始化函数

6.7.8 初始化

指定的初始化函数为初始化稀疏数组提供了一种机制，这在数字编程的实践中很常见。

指定的初始化函数可以对稀疏结构进行初始化，这在系统编程中很常见，并且可以通过任何成员对联合进行初始化，而不管其是否为第一个成员。

请看以下这些示例。此处的第一个示例显示了如何使用指定的初始化函数来对数组进行初始化：

```
 enum { first, second, third };
          const char *nm[] = {
          [third] = "third member",
          [first] = "first member",
          [second] = "second member",
          };
```

下面的示例证明了如何使用指定的初始化函数来对结构对象的字段进行初始化：

```
division_t result = { .quot = 2, .rem = -1 };
```

下面的示例显示了如何使用指定的初始化函数对复杂的结构进行初始化（否则这些结构可能会被误解）：

```
 struct { int z[3], count; } w[] = { [0].z = {1}, [1].z[0] = 2 };
```

通过使用单个指示符可以从两端创建数组：

```
int z[MAX] = {1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0};
```

如果 MAX 大于 10，则数组将在中间位置包含取值为零的元素；如果 MAX 小于 10，则前五个初始化函数提供的某些值将被后五个初始化函数的值覆盖。

联合的任何成员均可进行初始化：

```
union { int i; float f;} data = { .f = 3.2 };
```

D.1.14 混合声明和代码

6.8.2 复合语句

现在，C 编译器接受关于可执行代码的混合类型声明，如以下示例所示：

```
#include <stdio.h>

int main(void){
  int num1 = 3;
  printf("%d\n", num1);

  int num2 = 10;
  printf("%d\n", num2);
  return(0);
}
```

D.1.15 for 循环语句中的声明

6.8.5 迭代语句

C 编译器接受作为 for 循环语句中第一个表达式的类型声明：

```
for (int i=0; i<10; i++){ //loop body };
```

for 循环的初始化语句中声明的任何变量的作用域是整个循环（包括控制和迭代表达式）。

D.1.16 具有可变数目的参数的宏

6.10.3 宏替换

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
```

如果宏定义中的 *identifier_list* 以省略号结尾，则意味着调用中的参数比宏定义中的参数（不包括省略号）多。否则，宏定义中参数的数目（包括由预处理标记组成的参数）与调用中参数的数目匹配。对于在其参数中使用省略号表示法的 `#define` 预处理指令，在其替换列表中使用标识符 `__VA_ARGS__`。以下示例说明可变参数列表宏工具。

```
#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));
```

D.1.17 _Pragma

6.10.9 Pragma 操作符

`_Pragma` (*string-literal*) 形式的一元操作符表达式处理如下：

- 如果文本字符串具有 L 前缀，则删除该前缀。
- 删除前导和结尾双引号。
- 用双引号替换每个换码序列 `'`。
- 用单个反斜杠替换每个换码序列 `\\`。

预处理标记的结果序列作为 pragma 指令中的预处理程序标记进行处理。

删除一元操作符表达式中的最初四个预处理标记。

与 `#pragma` 比较，`_Pragma` 的优势在于：`_Pragma` 可以用于宏定义。

`_Pragma("string")` 与 #pragma 字符串行为完全相同。考虑以下示例。首先列出示例的源代码，然后在预处理程序使其通过预处理之后，再列出示例的源代码。

```
example% cat test.c

#include <omp.h>
#include <stdio.h>

#define Pragma(x) _Pragma(#x)
```

```
#define OMP(directive) Pragma(omp directive)

void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  OMP(parallel)
  {
  printf("Hello!\n");
  }
}

example% cc test.c -P -xopenmp -x03
example% cat test.i
```

下面是预处理程序完成后的源代码。

```
void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  # pragma omp parallel
  {
    printf("Hellow!\n");
  }
}

example% cc test.c -xopenmp -->
example% ./a.out
Hello!
Hello!
example%
```

◆ ◆ ◆ 附录 E

实现定义的 ISO/IEC C90 行为

ISO/IEC 9899:1999 编程语言 C 标准指定以 C 语言编写的程序的形式并加以解释。但是，此标准留下许多实现定义的问题，即因编译器而异的问题。本章将详细介绍这些方面的内容。它们很容易与 ISO/IEC 9899:1990 标准本身比较：

- 每一项均使用 ISO 标准中的相同节文本。
- 每一项的前面均有 ISO 标准中相应的节号。

E.1 与 ISO 标准比较的实现

E.1.1 转换 (G.3.1)

圆括号中的编号与 ISO/IEC 9899:1990 标准中的节号对应。

E.1.1.1 (5.1.1.3) Identification of diagnostics（识别诊断）：

错误消息具有以下格式：

filename, `line` *line number: message*

警告消息具有以下格式：

filename, `line` *line number: warning message*

其中：

- *filename* 是错误或警告所在文件的名称
- *line number* 是错误或警告所在行的编号
- *message* 是诊断消息

E.1.2 环境 (G.3.2)

E.1.2.1 (5.1.2.2.1) Semantics of arguments to `main`（ main 的参数的语义 ）：

```
int main (int argc, char *argv[])
{
....
}
```

`argc` 是调用程序时所用的命令行参数的数量。在任何 shell 扩展之后，`argc` 总是至少等于 `1`，即程序名称。

`argv` 是指向命令行参数的指针数组。

E.1.2.2 (5.1.2.3) What constitutes an interactive device（ 交互式设备的要素 ）：

交互式设备是系统库调用 `isatty()` 返回非零值的设备。

E.1.3 标识符 (G.3.3)

E.1.3.1 (6.1.2) The number of significant initial characters (beyond 31) in an identifier without external linkage（ 不带外部链接的标识符中的有效初始字符（ 第 31 个字符之后 ）的数目 ）：

前 1,023 个字符是有效字符。标识符区分大小写。

(6.1.2) The number of significant initial characters (beyond 6) in an identifier with external linkage（ 带外部链接的标识符中的有效初始字符（ 第 6 个字符之后 ）的数目 ）：

前 1,023 个字符是有效字符。标识符区分大小写。

E.1.4 字符 (G.3.4)

E.1.4.1 (5.2.1) The members of the source and execution character sets, except as explicitly specified in the Standard（ 除非标准中明确指定，否则为源代码字符集和执行字符集的成员 ）：

这两个字符集都与 ASCII 字符集相同，并有特定于语言环境的扩展。

E.1.4.2 (5.2.1.2) The shift states used for the encoding of multibyte characters（用于多字节字符编码的移位状态）：

无移位状态。

E.1.4.3 (5.2.4.2.1) The number of bits in a character in the execution character set（执行字符集中字符的位数）：

对于 ASCII 部分，一个字符中有 8 位；对于特定于语言环境的扩展部分，一个字符中的位数是 8 的倍数，具体取决于语言环境。

E.1.4.4 (6.1.3.4) The mapping of members of the source character set (in character and string literals) to members of the execution character set（源代码字符集成员（用字符和文本字符串表示）至执行字符集成员的映射）：

源代码字符与执行字符之间映射相同。

E.1.4.5 (6.1.3.4) The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant（包含一个字符的整型字符常量的值或不以基本执行字符集或宽字符常量的扩展字符集表示的换码序列的值）：

它是最右边字符的数值。例如，`'\q'` 等于 `'q'`。如果这样的换码序列出现，则会发出警告。

E.1.4.6 (3.1.3.4) The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character（包含多个字符的整型字符常量的值或包含多个多字节字符的宽字节常量的值）：

一个多字符常量，它不是具有从每个字符的数值派生的值的转义序列。

E.1.4.7 (6.1.3.4) The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant（用于针对宽字符常量将多字节字符转换为相应宽字符的当前语言环境）：

`LC_ALL`、`LC_CTYPE` 或 `LANG` 环境变量指定的有效语言环境。

E.1.4.8 (6.2.1.1) Whether a plain `char` has the same range of values as `signed char` or `unsigned char`（无格式 **char** 的值范围是与 **signed char** 相同还是与 **unsigned char** 相同）：

A `char` 被视为 `signed char`。

E.1.5 整数 (G.3.5)

E.1.5.1 (6.1.2.5) The representations and sets of values of the various types of integers（各种类型的整数的表示和值集）：

表 E–1 整数的表示和值集

整数	位	最小值	最大值
`char`	8	-128	127
`signed char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`	32	-2147483648	2147483647
`signed int`	32	-2147483648	2147483647
`unsigned int`	32	0	4294967295
`long -m32`	32	-2147483648	2147483647
`long -m64`	64	-9223372036854775808	9223372036854775807
`signed long -m32`	32	-2147483648	2147483647
`signed long -m64`	64	-9223372036854775808	9223372036854775807
`unsigned long -m32`	32	0	4294967295
`unsigned long -m64`	64	0	18446744073709551615
`long long`	64	-9223372036854775808	9223372036854775807
`signed long long`[1]	64	-9223372036854775808	9223372036854775807
`unsigned long long`[1]	64	0	18446744073709551615

[1] 在 `-Xc` 模式下无效

E.1.5.2 (6.2.1.2) The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented（值无法表示的情况下，整数转换为较短的带符号整型数的结果，或者无符号整型数转换为同等长度的带符号整型数的结果）：

整数转换为较短的 `signed` 整数时，低阶位从较长的整数复制到较短的 `signed` 整数中。结果可能为负数。

无符号整数转换为同等长度的 `signed` 整数时，低阶位从 `unsigned` 整数复制到 `signed` 整数。结果可能为负数。

E.1.5.3 (6.3) The results of bitwise operations on signed integers（带符号整型数的按位操作的结果）：

对 `signed` 类型应用按位操作的结果是操作数的按位操作，包括 `sign` 位。因此，当且仅当两个操作数中每个对应的位均已置位时，结果中的每个位才置位。

E.1.5.4 (6.3.5) The sign of the remainder on integer division（整数除法的余数的符号）：

结果的符号与被除数相同，因此，-23/4 的余数是 -3。

E.1.5.5 (6.3.7) The result of a right shift of a negative-valued signed integral type（负值带符号整型的右移的结果）：

右移的结果为 `signed` 右移。

E.1.6 浮点 (G.3.6)

E.1.6.1 (6.1.2.5) The representations and sets of values of the various types of floating-point numbers（各种类型的浮点数的表示形式和值集）：

表 E–2 `float` 值

float	
位	32
最小值	1.17549435E-38
最大值	3.40282347E+38
Epsilon	1.19209290E-07

表 E–3 `double` 值

double	
位	64
最小值	2.2250738585072014E-308
最大值	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16

表 E–4 `long double` 值

long double	
位	128 *(SPARC)* 80 *(x86)*
最小值	3.3621031431120935062626778173217526 03E-4932 (SPARC) 3.362103143112093506262 7E-4932 *(x86)*
最大值	1.1897314953572317650857593266280070 16E+4932 *(SPARC)* 1.189731495357231765021 3E4932 *(x86)*
Epsilon	1.9259299443872358530559779425849273 19E-34 *(SPARC)* 1.0842021724855044340075E-19 *(x86)*

E.1.6.2 (6.2.1.3) The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value（当整数转换为不能精确地表示原始值的浮点数时截断的方向）：

数舍入为可以表示的最近的值。

E.1.6.3 (6.2.1.4) The direction of truncation or rounding when a floating point number is converted to a narrower floating-point number（当浮点数转换为较窄的浮点数时截断或舍入的方向）：

数舍入为可以表示的最近的值。

E.1.7 数组和指针 (G.3.7)

E.1.7.1 （6.3.3.4，7.1.1）The type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` operator, `size_t`（存放数组的最大大小所需的整型；即 sizeof 操作符的类型 size_t）：

`stddef.h` 中定义的 `unsigned int`（对于 `-m32`）。

`unsigned long`（对于 `-m64`）

E.1.7.2 (6.3.4) The result of casting a pointer to an integer, or vice versa（将指针强制转换为整数的结果，或将整数强制转换为指针的结果）：

对于指针以及类型为 `int`、`long`、`unsigned int` 和 `unsigned long` 的值，位模式不变。

E.1.7.3 （6.3.6、7.1.1）The type of integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t`（存放指向同一数组中成员的两个指针之差所需的整型 ptrdiff_t）：

如 `stddef.h` 中定义的 `int`（对于 `-m32`）。

`long`（对于 `-m64`）

E.1.8 寄存器 (G.3.8)

E.1.8.1 (6.5.1) The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier（可通过使用 register 存储类说明符实际放入寄存器中的对象的范围）：

有效寄存器声明数取决于每个函数内使用和定义的模式，并受可供分配的寄存器数的限制。不要求编译器和优化器接受寄存器声明。

E.1.9 结构、联合、枚举和位字段 (G.3.9)

E.1.9.1 (6.3.2.3) A member of a union object is accessed using a member of a different type（使用不同类型的成员访问的联合对象的成员）：

访问存储在联合成员中的位模式，并根据访问成员时所用的成员类型解释值。

E.1.9.2 (6.5.2.1) The padding and alignment of members of structures（结构成员的填充和对齐）。

表 E–5 结构成员的填充和对齐

类型	对齐边界	字节对齐
`char` 和 `_Bool`	字节	1
`short`	半字	2
`int`	字	4
`long -m32`	字	4
`long -m64`	双字	8
`long long -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`long long -m64`	双字	8
`float`	字	4
`double -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`double -m64`	双字	8
`long double -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`long double -m64`	四倍长字	16
`pointer -m32`	字	4
`pointer -m64`	四倍长字	8
`float _Complex`	字	4
`double _Complex -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`double _Complex -m64`	双字	8
`long double _Complex -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`long double _Complex -m64`	四倍长字	16
`float _Imaginary`	字	4

表 E–5 结构成员的填充和对齐 （续）

类型	对齐边界	字节对齐
`double _Imaginary -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`double _Imaginary -m64`	双字	8
`long double _Imaginary -m32`	双字 (SPARC) 字 (x86)	8 (SPARC) 4 (x86)
`long double _Imaginary -m64`	双字	16

内部填充结构成员，以便各个元素在适当的边界上对齐。

结构的对齐与其更严格对齐的成员相同。例如，只包含 `char` 数据的 `struct` 无对齐限制，而包含用 `-m64` 编译的 `double` 数据的 `struct` 将按 8 字节边界对齐。

E.1.9.3 (6.5.2.1) Whether a plain `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field（无格式 int 位字段是视为 signed int 位字段还是视为 unsigned int 位字段处理）：

视为 `unsigned int`。

E.1.9.4 (6.5.2.1) The order of allocation of bit-fields within an `int`（在 int 中位字段的分配顺序）：

在存储单元中从高阶到低阶分配位字段。

E.1.9.5 (6.5.2.1) Whether a bit-field can straddle a storageunit boundary（位字段是否可以跨存储单元边界）：

位字段不可以跨存储单元边界。

E.1.9.6 (6.5.2.2) The integer type chosen to represent the values of an enumeration type（选择用来表示枚举类型的值的整型）：

这是 `int`。

E.1.10 限定符 (G.3.10)

E.1.10.1 (6.5.5.3) What constitutes an access to an object that has volatile-qualified type（什么构成对为 volatile 限定类型的对象的访问）：

对象名称的每个引用构成对该对象的访问。

E.1.11 声明符 (G.3.11)

E.1.11.1 (6.5.4) The maximum number of declarators that may modify an arithmetic, structure, or union type（可修改算术、结构或联合类型的声明数最大值）：

编译器不施加任何限制。

E.1.12 语句 (G.3.12)

E.1.12.1 (6.6.4.2) The maximum number of `case` values in a `switch` statement（一个 switch 语句中 case 值的最大数目）：

编译器不施加任何限制。

E.1.13 预处理指令 (G.3.13)

E.1.13.1 (6.8.1) Whether the value of a singlecharacter character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set（控制条件包含的常量表达式中字符常量的值是否与执行字符集内的相同字符常量的值匹配）：

预处理指令中的字符常量与其在任何其他表达式中的数值相同。

E.1.13.2 (6.8.1) Whether such a character constant may have a negative value（这样的字符常量是否可以具有负值）：

此上下文中的字符常量可以有负值。

E.1.13.3 (6.8.2) The method for locating includable source files（定位可包含的源文件的方法）：

对于由 < > 限定其名称的文件，先在 `-I` 选项指定的目录中查找，然后在标准目录中查找。标准目录为 `/usr/include`，除非使用 `-YI` 选项指定另一个缺省位置。

对于由引号限定其名称的文件，先在包含 `#include` 的源文件的目录中查找，然后在 `-I` 选项指定的目录中查找，最后在标准目录中查找。

如果 < > 或双引号中的文件名以 `/` 字符开头，则文件名解释为根目录开头的路径名。查找该文件时只能在根目录中。

E.1.13.4 (6.8.2) The support of quoted names for includable source files（对可包含的源文件的带引号名称的支持）：

支持 `include` 指令中的带引号文件名。

E.1.13.5 (6.8.2) The mapping of source file character sequences（源文件字符序列的映射）：

源文件字符映射至其相应的 ASCII 值。

E.1.13.6 (6.8.6) The behavior on each recognized `#pragma` directive（每个识别的 #pragma 指令的行为）：

支持以下 pragma。有关更多信息，请参见第 40 页中的 “2.11 Pragma”。

- `align` *integer (variable[, variable])*
- `c99 ("implicit" | "no%implicit")`
- `does_not_read_global_data` *(funcname [, funcname])*
- `does_not_return` *(funcname[, funcname])*
- `does_not_write_global_data` *(funcname[, funcname])*
- `error_messages` *(on|off|default, tag1[tag2... tagn])*
- `fini` *(f1[, f2..., fn])*
- `hdrstop`
- `ident` *string*
- `init` *(f1[, f2..., fn])*
- `inline` *(funcname[, funcname])*
- `int_to_unsigned` *(funcname)*
- `MP serial_loop`
- `MP serial_loop_nested`
- `MP taskloop`
- `no_inline` *(funcname[, funcname])*
- `no_warn_missing_parameter_info`
- `nomemorydepend`

- `no_side_effect` *(funcname[, funcname])*
- `opt_level` *(funcname[, funcname])*
- `pack`*(n)*
- `pipeloop`*(n)*
- `rarely_called` *(funcname[, funcname])*
- `redefine_extname` *old_extname new_extname*
- `returns_new_memory` *(funcname[, funcname])*
- `unknown_control_flow` *(name[, name])*
- `unroll` *(unroll_factor)*
- `warn_missing_parameter_info`
- `weak` *symbol1 [= symbol2]*

E.1.13.7 (6.8.8) The definitions for `__DATE__` and `__TIME__` when, respectively, the date and time of translation are not available（转换的日期和时间分别不可用时 `__DATE__` 和 `__TIME__` 的定义）：

环境中总是提供这些宏。

E.1.14 库函数 (G.3.14)

E.1.14.1 (7.1.6) The null pointer constant to which the macro `NULL` expands（宏 NULL 扩展为的空指针常量）：

`NULL` 等于 0。

E.1.14.2 (7.2) The diagnostic printed by and the termination behavior of the `assert` function（assert 函数输出的诊断及该函数的终止行为）：

诊断为：

`Assertion failed:` *statement* `. file` *filename*`, line` *number*

其中：

- *statement* 是使断言失败的语句
- *filename* 是失败所在文件的名称
- *line number* 是失败所在行的编号

E.1.14.3 (7.3.1) The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions（isalnum、isalpha、iscntrl、islower、isprint 和 supper 函数测试的字符集）：

表 E–6 `isalpha`、`islower` 等测试的字符集

`isalnum`	ASCII 字符 A-Z、a-z 和 0-9
`isalpha`	ASCII 字符 A-Z 和 a-z，以及特定于语言环境的单字节字母
`iscntrl`	值为 0-31 和 127 的 ASCII 字符
`islower`	ASCII 字符 a-z
`isprint`	特定于语言环境的单字节可输出字符
`isupper`	ASCII 字符 A-Z

E.1.14.4 (7.5.1) The values returned by the mathematics functions on domain errors（发生域错误时数学函数返回的值）：

表 E–7 发生域错误时返回的值

错误	数学函数	编译器模式	
		`-Xs`、`-Xt`	`-Xa`、`-Xc`
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0

表 E–7 发生域错误时返回的值 （续）

错误	数学函数	编译器模式	
		-Xs、-Xt	-Xa、-Xc
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

E.1.14.5 (7.5.1) Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors（在出现下溢范围错误时，数学函数是否将整数表达式 errno 设置为宏 ERANGE 的值）：

检测到下溢时，除 scalbn 之外的数学函数将 errno 设置为 ERANGE。

E.1.14.6 (7.5.6.4) Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero（fmod 函数的第二个参数为零时，发生域错误还是返回零）：

在此情况下，它返回第一个参数并发生域错误。

E.1.14.7 (7.7.1.1) The set of signals for the signal function（用于 signal 函数的信号集）：

下表列出了 signal 函数可识别的每个信号的语义：

表 E–8 signal 信号的语义

信号	不可以。	缺省	事件
SIGHUP	1	退出	挂起
SIGINT	2	退出	中断
SIGQUIT	3	信息转储	退出
SIGILL	4	信息转储	非法指令（找到时不重置）
SIGTRAP	5	信息转储	跟踪陷阱（捕获时不重置）

表 E–8 signal 信号的语义 （续）

信号	不可以。	缺省	事件
SIGIOT	6	信息转储	IOT 指令
SIGABRT	6	信息转储	由中止使用
SIGEMT	7	信息转储	EMT 指令
SIGFPE	8	信息转储	浮点异常
SIGKILL	9	退出	中止（找不到，也无法忽略）
SIGBUS	10	信息转储	总线错误
SIGSEGV	11	信息转储	段违规
SIGSYS	12	信息转储	系统调用参数错误
SIGPIPE	13	退出	写在管道上，但无读取者
SIGALRM	14	退出	报警时钟
SIGTERM	15	退出	来自中止的软件终止信号
SIGUSR1	16	退出	用户定义的信号 1
SIGUSR2	17	退出	用户定义的信号 2
SIGCLD	18	忽略	子项状态更改
SIGCHLD	18	忽略	子项状态更改别名
SIGPWR	19	忽略	电源故障，重新启动
SIGWINCH	20	忽略	窗口大小更改
SIGURG	21	忽略	紧急套接字条件
SIGPOLL	22	退出	发生了可轮询事件
SIGIO	22	退出	可能有套接字 I/O
SIGSTOP	23	停止	停止（找不到，也无法忽略）
SIGTSTP	24	停止	来自 tty 的用户停止请求
SIGCONT	25	忽略	停止的进程已继续
SIGTTIN	26	停止	已尝试后台 tty 读
SIGTTOU	27	停止	已尝试后台 tty 写
SIGVTALRM	28	退出	虚拟计时器已过期
SIGPROF	29	退出	文件配置计时器已过期

表 E–8 signal 信号的语义 （续）

信号	不可以。	缺省	事件
SIGXCPU	30	信息转储	已超出 cpu 限制
SIGXFSZ	31	信息转储	已超出文件大小限制
SIGWAITINGT	32	忽略	进程的 lwp 受阻

E.1.14.8 (7.7.1.1) The default handling and the handling at program startup for each `signal` recognized by the signal function（针对信号函数识别的每个 signal 的缺省处理和程序启动时的处理）：

参见以上内容。

E.1.14.9 (7.7.1.1) If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed（如果在调用信号处理程序之前未执行 signal(sig, SIG_DFL) 的等效函数，则阻塞执行的信号）：

总是执行 `signal(sig,SIG_DFL)` 的等效函数。

E.1.14.10 (7.7.1.1) Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the signal function（为信号函数指定的处理程序接收到 SIGILL 信号时，是否重置缺省处理）：

接收到 `SIGILL` 时不重置缺省处理。

E.1.14.11 (7.9.2) Whether the last line of a text stream requires a terminating new-line character（文本流的最后一行是否需要终止换行符）：

最后一行不需要以换行符结束。

E.1.14.12 (7.9.2) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in（写出到换行符紧前面的文本流中的空格字符在读入时是否出现）：

读该流时，所有字符均出现。

E.1.14.13 (7.9.2) The number of null characters that may be appended to data written to a binary stream（可附加至写入二进制流的数据的空字符数）：

空字符不附加至二进制流。

E.1.14.14 (7.9.3) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file（附加模式流的文件位置指示器最初位于文件开头还是结尾）：

文件位置指示符最初位于文件结尾。

E.1.14.15 (7.9.3) Whether a write on a text stream causes the associated file to be truncated beyond that point（文本流中的写是否导致联合文件在该点之后被截断）：

对文本流的写操作不会导致文件在该点之后被截断，除非硬件设备强制这种情况发生。

E.1.14.16 (7.9.3) The characteristics of file buffering（文件缓冲的特性）：

除标准错误流 (`stderr`) 之外，输出流在输出至文件时采用缺省缓冲，在输出至终端时采用行缓冲。标准错误输出流 (`stderr`) 在缺省情况下不缓冲。

缓冲的输出流保存多个字符，然后将这些字符作为块进行写入。未缓冲的输出流将信息排队，以便立即在目标文件或终端上写入。行缓冲的输出将输出的每行排队，直至行完成（请求换行符）时为止。

E.1.14.17 (7.9.3) Whether a zero-length file actually exists（零长度文件是否实际存在）：

由于零长度文件有目录项，因此它确实存在。

E.1.14.18 (7.9.3) The rules for composing valid file names（书写有效文件名的规则）：

有效文件名的长度可以为 1 到 1,023 个字符，可以使用除字符 `null` 和 `/`（斜杠）之外的所有字符。

E.1.14.19 (7.9.3) Whether the same file can be open multiple times（同一文件是否可以多次打开）：

同一文件可以多次打开。

E.1.14.20 (7.9.4.1) The effect of the `remove` function on an open file（remove 函数对打开的文件的影响）：

在执行关闭文件的最后一个调用时删除文件。程序不能打开已删除的文件。

E.1.14.21 (7.9.4.2) The effect if a file with the new name exists prior to a call to the `rename` function（在调用 rename 函数之前已存在具有新名称的文件时的影响）：

如果该文件存在，则将其删除，并且新文件改写先前存在的文件。

E.1.14.22 (7.9.6.1) The output for `%p` conversion in the `fprintf` function（fprintf 函数中 %p 转换的输出）：

`%p` 的输出与 `%x` 的相同。

E.1.14.23 (7.9.6.2) The input for `%p` conversion in the `fscanf` function（fscanf 函数中 %p 转换的输入）：

`%p` 的输入与 `%x` 的相同。

E.1.14.24 (7.9.6.2) The interpretation of a - character that is neither the first nor the last character in the scan list for `%[` conversion in the `fscanf` function（对既不是 fscanf 函数中 %[转换的扫描列表中的第一个字符也不是最后一个字符的 - 字符的解释）：

- 字符表示包括边界的范围，因此 `[0-9]` 与 `[0123456789]` 等效。

E.1.15 语言环境特定的行为 (G.4)

E.1.15.1 (7.12.1) The local time zone and Daylight Savings Time（本地时区和夏令时）：

本地时区由环境变量 `TZ` 设置。

E.1.15.2 (7.12.2.1) The era for the `clock` function（clock 函数的年代）

时钟的年代以时钟周期（以程序的起始执行时间为起点）表示。

宿主环境的下列特性特定于语言环境：

E.1.15.3 (5.2.1) The content of the execution character set, in addition to the required members（执行字符集的内容，以及必需的成员）：

特定于语言环境（C 语言环境中无扩展）。

E.1.15.4 (5.2.2) The direction of printing（输出方向）：

输出总是从左到右进行。

E.1.15.5 (7.1.1) The decimal-point character（小数点字符）：

特定于语言环境（在 C 语言环境中为 "."）。

E.1.15.6 (7.3) The implementation-defined aspects of character testing and case mapping functions（字符测试和条件映射函数的实现定义方面）：

与 4.3.1 相同。

E.1.15.7 (7.11.4.4) The collation sequence of the execution character set（执行字符集的整理序列）：

特定于语言环境（C 语言环境中的 ASCII 整理）。

E.1.15.8 (7.12.3.5) The formats for time and date（时间和日期的格式）：

特定于语言环境。下面几个表显示 C 语言环境中的格式。月份名称为：

表 E–9 月份名称

1 月	5 月	9 月
2 月	6 月	10 月
3 月	7 月	11 月
4 月	8 月	12 月

周日期的名称为：

表 E–10 周日期及缩写

日期		缩写	
星期日	星期四	日	四
星期一	星期五	一	五
星期二	星期六	二	六
星期三		三	

时间的格式为：

```
%H:%M:%S
```

日期的格式为：

```
%m/%d/%y
```

上午和下午的格式为： AM PM

◆ ◆ ◆ 附 录 F

ISO C 数据表示法

本附录描述 ISO C 如何表示存储器中的数据以及向函数传递参数的机制。其目的是为想要编写或使用非 C 语言模块并将这些模块与 C 代码连接的程序员提供指导。

F.1 存储分配

下表显示了数据类型及其表示方法。

注 – 栈中分配的存储空间（带有内部、自动或链接的标识符）应限于 2G 字节或更少。

表 F–1 数据类型的存储分配（**大小以字节为单位**）

C 类型	LP64 (-m64) 大小	LP64 对齐	ILP32 (-m32) 大小	ILP 32 对齐
整数				
_Bool char signed char unsigned char	1	1	1	1
short signed short unsigned short	2	2	2	2
int signed int unsigned int enum	4	4	4	4

表 F–1 数据类型的存储分配（大小以字节为单位） （续）

C 类型	LP64 (-m64) 大小	LP64 对齐	ILP32 (-m32) 大小	ILP 32 对齐
long signed long unsigned long	8	8	4	4
long long signed long long unsigned long long	8	8	8	4 (x86) / 8 (SPARC)
指针				
any-type * any-type (*) ()	8	8	4	4
浮点				
float	4	4	4	4
double	8	8	8	4 (x86) / 8 (SPARC)
long double	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 8 (SPARC)
复合				
float _Complex	8	4	8	4
double _Complex	16	8	16	4 (x86) / 8 (SPARC)
long double _Complex	32	16	24 (x86) / 32 (SPARC)	4 (x86) / 16 (SPARC)
虚数				
float _Imaginary	4	4	4	4
double _Imaginary	8	8	8	4 (x86) / 8 (SPARC)
long double _Imaginary	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 16 (SPARC)

F.2 数据表示法

任何给定数据元素的位编号取决于使用的体系结构：SPARC 工作站机器将位 0 用作最低有效位，将字节 0 用作最高有效字节。本节中的表描述各种表示法。

F.2.1 整数表示法

ISO C 中使用的整型有 `short`、`int`、`long` 和 `long long`：

表 F–2 `short` 的表示法

位	内容
8- 15	字节 0 (SPARC) 字节 1 *(x86)*
0- 7	字节 1 (SPARC) 字节 0 (x86)

表 F–3 `int` 的表示法

位	内容
24- 31	字节 0 (SPARC) 字节 3 (x86)
16- 23	字节 1 (SPARC) 字节 2 (x86)
8- 15	字节 2 (SPARC) 字节 1 (x86)
0- 7	字节 3 (SPARC) 字节 0 (x86)

表 F–4 `long` 的表示形式以及使用 `-m32` 进行编译

位	内容
24- 31	字节 0 (SPARC) 字节 3 (x86)
16- 23	字节 1 (SPARC) 字节 2 (x86)
8- 15	字节 2 (SPARC) 字节 1 (x86)
0- 7	字节 3 (SPARC) 字节 0 (x86)

表 F–5 long (-m64) 和 long long（-m32 和 -m64）

位	内容
56- 63	字节 0 (SPARC) 字节 7 (x86)
48- 55	字节 1 (SPARC) 字节 6 (x86)
40- 47	字节 2 (SPARC) 字节 5 (x86)
32- 39	字节 3 (SPARC) 字节 4 (x86)
24- 31	字节 4 (SPARC) 字节 3 (x86)
16- 23	字节 5 (SPARC) 字节 2 (x86)
8- 15	字节 6 (SPARC) 字节 1 (x86)
0- 7	字节 7 (SPARC) 字节 0 (x86)

F.2.2 浮点表示法

float、double 和 long double 数据元素按照 ISO IEEE 754-1985 标准来表示。表示为：

$(-1)^{s} * 2^{(e - bias)} * [j.f]$

其中：

- *s* = sign
- *e* = 偏置指数
- *j* 为前导位，由 *e* 的值确定。在 long double *(x86)* 情况下，前导位是显式的；在所有其他情况下，它是隐式的。
- *f* = 尾数
- *u* 表示位可以是 1 或 0（在下列表格中使用）。

对于 IEEE Single 和 Double，*j* 总是隐式的。偏置指数为 0 时，*j* 为 0，只要 *f* 不为 0，生成的数字就不太正常。偏置指数大于 0 时，只要该数字是有限的，*j* 就为 1。

对于 Intel 80 位 Extended，*j* 总是显式的。

下表显示各个位的位置。

表 F–6 `float` 表示法

位	名称
31	符号
23- 30	偏置指数
0- 22	尾数部分

表 F–7 `double` 表示法

位	名称
63	符号
52- 62	偏置指数
0- 51	尾数部分

表 F–8 `long double` 表示法 (SPARC)

位	名称
127	符号
112- 126	偏置指数
0- 111	尾数部分

表 F–9 `long double` 表示法 (x86)

位	名称
80- 95	不使用
79	符号
64- 78	偏置指数
63	前导位
0- 62	尾数部分

有关详细信息，请参阅《数值计算指南》。

F.2.3 异常值

`float` 和 `double` 数被认为包含一个“隐藏的”或隐含的位，从而比不包含该位时的精度高一位。对于 `long double`，前导位为隐式 (SPARC) 或显式 (x86)；该位对于正规数为 1，对于非正规数为 0。

表 F–10 `float` 表示法

正规数 (0<e<255)：	$(-1)^s 2^{(e-127)} 1.f$
非正规数 (e=0, f!=0):	$(-1)^s 2^{(-126)} 0.f$
零 (e=0, f=0):	$(-1)^s 0.0$
信号 NaN	s=u，e=255(max)；f=.0uuu-uu；至少一个位必须为非零
静态 NaN	s=u，e=255(max)；f=.1uuu-uu
无穷	s=u，e=255(max)；f=.0000-00（全为零）

表 F–11 `double` 表示法

正规数 (0<e<2047)：	$(-1)^s 2^{(e-1023)} 1.f$
非正规数 (e=0, f!=0):	$(-1)^s 2^{(-1022)} 0.f$
零 (e=0, f=0):	$(-1)^s 0.0$
信号 NaN	s=u，e=2047(max)；f=.0uuu-uu；至少一个位必须为非零
静态 NaN	s=u，e=2047(max)；f=.1uuu-uu
无穷	s=u，e=2047(max)；f=.0000-00（全为零）

表 F–12 `long double` 表示法

正规数 (0<e<32767)：	$(-1)^s 2^{(e-16383)} 1.f$
非正规数 (e=0, f!=0):	$(-1)^s 2^{(-16382)} 0.f$
零 (e=0, f=0):	$(-1)^s 0.0$
信号 NaN	s=u，e=32767(max)；f=.0uuu-uu；至少一个位必须为非零
静态 NaN	s=u，e=32767(max)；f=.1uuu-uu
无穷	s=u，e=32767(max)；f=.0000-00（全为零）

F.2.4 选定的数的十六进制表示

下表显示十六进制表示。

表 F–13 选定数的十六进制表示法 (SPARC)

值	float	double	long double
+0	00000000	0000000000000000	00000000000000000000000000000000
-0	80000000	8000000000000000	80000000000000000000000000000000
+1.0	3F800000	3FF0000000000000	3FFF0000000000000000000000000000
-1.0	BF800000	BFF0000000000000	BFFF0000000000000000000000000000
+2.0	40000000	4000000000000000	40000000000000000000000000000000
+3.0	40400000	4008000000000000	40080000000000000000000000000000
正无穷	7F800000	7FF0000000000000	7FFF0000000000000000000000000000
负无穷	FF800000	FFF0000000000000	FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF

表 F–14 选定数的十六进制表示法 *(x86)*

值	float	double	long double
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000
+1.0	3F800000	000000003FF00000	3FFF8000000000000000
-1.0	BF800000	00000000BFF00000	BFFF8000000000000000
+2.0	40000000	0000000040000000	40008000000000000000
+3.0	40400000	0000000040080000	4000C000000000000000
正无穷	7F800000	000000007FF00000	7FFF8000000000000000
负无穷	FF800000	00000000FFF00000	FFFF8000000000000000
NaN	7FBFFFFF	FFFFFFFF7FF7FFFF	7FFFBFFFFFFFFFFFFFFF

有关详细信息，请参阅《数值计算指南》。

F.2.5 指针表示

C 中的一个指针占 4 个字节。C 中的一个指针在 SPARC v9 体系结构中占 8 个字节。`NULL` 值指针等于零。

F.2.6 数组存储

数组及其元素按特定的存储顺序存储。元素实际上按存储元素的线性序存储。

C 数组按以行优先的方式存储；多维数组中的最后一个下标变化最快。

字符串数据类型是 char 元素的数组。文本字符串或宽文本字符串（串联后）中允许的字符数最大值为 4,294,967,295。

有关栈中存储分配大小限制的信息，请参见第 333 页中的 “F.1 存储分配”。

表 F–15 数组类型和存储

类型	-m32 元素的最大数目	-m64 元素的最大数目
char	4,294,967,295	2,305,843,009,213,693,951
short	2,147,483,647	1,152,921,504,606,846,975
int	1,073,741,823	576,460,752,303,423,487
long	1,073,741,823	288,230,376,151,711,743
float	1,073,741,823	576,460,752,303,423,487
double	536,870,911	288,230,376,151,711,743
long double	268,435,451	144,115,188,075,855,871
long long	536,870,911	288,230,376,151,711,743

静态数据和全局数组可以容纳更多元素。

F.2.7 异常值的算术运算

本节介绍了对异常和普通浮点值的组合应用基本算术运算所得的结果。以下信息假定不执行陷阱或任何其他异常操作。

下表解释缩写：

表 F–16 缩写用法

缩写	含义
Num	非正规数或正规数
Inf	无穷（正或负）
NaN	不是数
Uno	无序

下表描述对不同类型的操作数的组合执行算术运算所得值的类型。

表 F–17 加法和减法结果

	右操作数：0	右操作数：Num	右操作数：Inf	右操作数：NaN
左操作数：0	0	Num	Inf	NaN
左操作数：Num	Num	请参见[1]	Inf	NaN
左操作数：Inf	Inf	Inf	请参见[1]	NaN
左操作数：NaN	NaN	NaN	NaN	NaN

[1] 结果太大（溢出）时，Num + Num 可能为 Inf 而不是 Num。无穷值具有相反的 `sign` 时，Inf + Inf = NaN。

表 F–18 乘法结果

	右操作数：0	右操作数：Num	右操作数：Inf	右操作数：NaN
左操作数：0	0	0	NaN	NaN
左操作数：Num	0	Num	Inf	NaN
左操作数：Inf	NaN	Inf	Inf	NaN
左操作数：NaN	NaN	NaN	NaN	NaN

表 F–19 除法结果

	右操作数：0	右操作数：Num	右操作数：Inf	右操作数：NaN
左操作数：0	NaN	0	0	NaN
左操作数：Num	Inf	Num	0	NaN
左操作数：Inf	Inf	Inf	NaN	NaN
左操作数：NaN	NaN	NaN	NaN	NaN

表 F–20 比较结果

	右操作数：0	右操作数：+Num	右操作数：+Inf	右操作数：+NaN
左操作数：0	=	<	<	Uno
左操作数：+Num	>	比较的结果	<	Uno
左操作数：+Inf	>	>	=	Uno
左操作数：+NaN	Uno	Uno	Uno	Uno

注 – NaN 与 NaN 比较结果为无序，从而导致不相等。+0 与 - 0 的比较结果是相等。

F.3 参数传递机制

本节描述在 ISO C 中如何传递参数。

- 传递给 C 函数的所有参数均通过值进行传递。
- 实际参数按函数声明中声明参数的反向顺序传递。
- 本身为表达式的实际参数在函数引用之前求值。然后表达式的结果置入寄存器或推入栈。

F.3.1 32 位 SPARC

函数在寄存器 `%o0` 中返回 `integer` 结果，在寄存器 `%f0` 中返回 `float` 结果，在寄存器 `%f0` 和 `%f1` 中返回 `double` 结果。

`long long` 整数以较高词序在 `%oN` 寄存器中进行传递，以较低词序在 `%o(N+1)` 寄存器中进行传递。寄存器中的结果在 `%o0` 和 `%o1` 中返回，排序相似。

除 `doubles` 和 long double 外，所有参数都作为四字节值来传递。`double` 作为八字节值传递。前六个四字节值（`double` 计为 8）在寄存器 `%o0` 至 `%o5` 中传递。其余值传递到栈中。结构的传递方式是复制结构并将指针传递到副本。`long double` 的传递方式与结构的传递方式相同。

此处描述的寄存器是可被调用程序识别的寄存器。

F.3.2 64 位 SPARC

所有整型参数均作为 8 字节值传递。

浮点参数尽可能在浮点寄存器中传递。

F.3.3 x86/x64

遵循 Intel 386 psABI 和 AMD64 psABI。

函数在以下寄存器中返回结果 :

表 F–21 x86 函数返回类型所使用的寄存器

寄存器	返回的类型
int	%eax
long long	%edx 和 %eax
float、double 和 long double	%st(0)
float _Complex	实部为 %eax，虚部为 %edx
double _Complex 和 long double _Complex	与包含相应浮点类型的两个元素的结构相同。

有关详细信息，请参阅 http://www.x86-64.org/documentation/abi.pdf 上的 AMD64 psABI

除了 struct、union、long long、double 和 long double 之外的所有参数都作为四字节值传递；long long 作为八字节值传递，double 作为八字节值传递，long double 作为 12 字节值传递。

struct 和 union 被复制到栈中。大小向上舍入为 4 字节的倍数。返回 struct 和 union 的函数被传递一个隐藏的首参数，并指向返回的 struct 或 union 的存储位置。

从一个函数返回时，需要调用程序从栈中弹出参数，但 struct 和 union 返回的附加参数除外，它由调用的函数弹出。

◆ ◆ ◆ 附录 G

性能调节

本附录介绍了 C 程序的性能调节。另请参见《Solaris Studio 性能分析器》手册。

G.1 libfast.a 库 (SPARC)

libfast.a 是一个特定于 SPARC 的 32 位版标准 C 库，为单线程、单可执行文件应用程序提供优化的内存分配。由于它是可选库，因此它可以使用尽管对大多数应用程序可以提高性能，但是可能不适合标准 C 库的算法和数据表示。

使用文件配置确定以下清单中的例程对于您应用程序的性能是否重要，然后根据该清单确定 libfast.a 是否有益于性能：

- 如果内存分配的性能很重要，并且分配的块大小普遍接近 2 的幂，请**务必**使用 libfast.a。这些重要例程是：malloc()、free()、realloc()。
- 如果块移动或块填充例程的性能很重要，请**务必**使用 libfast.a。这些重要例程是：bcopy()、bzero()、memcpy()、memmove() 和 memset()。
- 如果应用程序为多线程应用程序，请**勿**使用 libfast.a。

链接应用程序时，将选项 -lfast 添加到链接时使用的 cc 命令中。在标准 C 库中，cc 命令会先于相应的其他命令链接 libfast.a 中的例程。

◆ ◆ ◆ 附 录 H

K&R Solaris Studio C 与 Solaris Studio ISO C 之间的差异

本附录介绍早期 K&R Solaris Studio C 与 Solaris Studio ISO C 之间的差异。

有关更多信息，请参见第 27 页中的 “1.5 标准一致性”。

H.1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容

主题	Solaris Studio C (K&R)	Solaris Studio ISO C
`main()` 的 `envp` 参数	允许 `envp` 作为 `main()` 的第三个参数。	允许第三个参数；但是，该用法并不严格符合 ISO C 标准。
关键字	将标识符 `const`、`volatile` 和 `signed` 视为普通标识符。	`const`、`volatile` 和 `signed` 是关键字。
块内部的 `extern` 和 `static` 函数声明	将这些函数声明提升为文件作用域。	ISO 标准不保证块作用域函数声明提升为文件作用域。
标识符	允许标识符中使用美元符号 (`$`)。	不允许使用 `$`。
`long float` 类型	接受 `long float` 声明，并将这些声明视为 `double`。	不接受这些声明。

多字符字符常量	`int mc = 'abcd';`
	产生：
	`abcd`
		`int mc = 'abcd';`
		产生：
		`dcba`

整型常量	在八进制换码序列中接受 8 或 9。	在八进制换码序列中不接受 8 或 9。

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 （续）

主题	Solaris Studio C (K&R)	Solaris Studio ISO C
赋值操作符	将以下操作符对视为两个标记，因此允许它们之间出现空白： `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	将它们视为单个标记，因此不允许它们之间出现空白。
表达式的无符号保留语义	支持无符号保留，即 `unsigned char/short` 转换为 `unsigned int`。	支持值保留，即 `unsigned char/short` 转换为 `int`。
单/双精度计算	将浮点表达式的操作数提升为 `double`。 声明以返回 `float` 的函数总是将其返回值提升为 `double`。	允许在单精度计算中执行对 `float` 的操作。 允许这些函数具有 `float` 返回类型。
`struct/union` 成员的名称空间	允许使用成员选择操作符（'.'、'->'）的 `struct`、`union` 和算术类型处理其他 `struct` 或 `union` 的成员。	要求各个唯一 `struct/union` 具有自己的唯一名称空间。
作为 `lvalue` 的强制类型转换	支持将整数类型和指针类型强制转换为 `lvalue`。例如： `(char *)ip = &char;`	不支持此功能。

隐含的 `int` 声明	支持不带显式类型说明符的声明。`num;` 等声明被视为隐含的 `int`。例如：
	`num; /*num` **隐含为** `int*/`
	`int num2; /*` **显式声明** `num2*/`
	`/*` **声明一个** `int */`
		不支持 `num;` 声明（不带显式类型说明符 `int`），生成语法错误。

空声明	允许空声明，如： `int;`	除标记之外，禁止空声明。

类型定义中的类型说明符	允许 `typedef` 声明中出现 `unsigned`、`short`、`long` 等类型说明符。例如：
	`typedef short small;`
	`unsigned small x;`
		不允许类型说明符修改 `typedef` 声明。

位字段中允许的类型	允许所有整型的位字段，包括未命名位字段。 ABI 要求支持未命名位字段及其他整型。	只支持类型为 `int`、`unsigned int` 和 `signed int` 的位字段。未定义其他类型。

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 （续）

主题	Solaris Studio C (K&R)	Solaris Studio ISO C

不完全声明中标记的处理	忽略不完全类型声明。在以下示例中，f1 是指外部 struct：
	`struct x { . . . } s1;`
	`{struct x; struct y {struct x f1; } s2; struct x`
	`{ . . . };}`
		在符合 ISO 的实现中，不完整的 struct 或 union 类型说明符将隐藏具有相同标记的封闭声明。

struct/union/enum 声明中的不匹配	允许嵌套的 struct/union 声明中标记的 struct/enum/union 类型出现不匹配。在以下示例中，第二个声明被视为 struct：
	`struct x {. . . }s1;`
	`{union x s2;. . .}`
		将内部声明视为新声明，隐藏外部标记。

表达式中的标签	将标签视为 `(void *) lvalue`。	不允许在表达式中使用标签。
switch 条件类型	通过将 float 和 double 转换为 int，允许使用这两种类型。	对于 switch 条件类型，只对整型（int、char 和枚举类型）求值。
条件包含指令的语法	预处理程序忽略 #else 或 #endif 指令后面的结尾标记。	禁止这样的构造。
标记粘贴和 ## 预处理程序操作符	不识别 ## 操作符。通过在粘贴的两个标记之间放置注释来完成标记粘贴： `#define PASTE(A,B) A/*任意注释*/B`	将 ## 定义为执行标记粘贴的预处理程序操作符，如以下示例中所示： `#define PASTE(A,B) A##B` 而且，Solaris Studio ISO C 处理器无法识别 Solaris Studio C 方法。相反，它将两个标记之间的注释视为空白。

预处理程序重新扫描	预处理程序递归替换：
	`#define F(X) X(arg)`
	`F(F)`
	产生
	`arg(arg)`
		在重新扫描过程中，如果在替换列表中找到宏，则不替换宏：
		`#define F(X)X(arg)F(F)`
		产生：
		`F(arg)`

形式参数列表中的 typedef 名称	可以使用 typedef 名称作为函数声明中的形式参数名称。“隐藏”typedef 声明。	禁止使用声明为 typedef 名称的标识符作为形式参数。

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 （续）

主题	Solaris Studio C (K&R)	Solaris Studio ISO C

特定于实现的聚集初始化	对括号中部分省略的初始化函数进行分析和处理时，使用自下而上的算法：
	`struct{ int a[3]; int b; }\ w[ ]={{1},2};`
	产生
	`sizeof(w)=16`
	`w[0].a=1,0,0`
	`w[0].b=2`
		使用自上而下的分析算法。例如：
		`struct{int a[3];int b;}\`
		`w[]={{1},2};`
		产生
		`sizeof(w)=32w[0].a=1,0,0w[0]. =0w[1].a=2,0,0w[1].b=0`

跨 include 文件的注释	允许在 #include 文件中开始的注释由包含第一个文件的文件终止。	在编译的转换阶段中注释被替换为空白字符，然后再处理 #include 指令。

字符常量中的形式参数替换	在字符常量与替换列表宏匹配时，替换字符常量中的字符：
	`#define charize(c)'c'`
	`charize(Z)`
	产生：
	'Z'
		不替换字符：
		`#define charize(c) 'c'charize(Z)`
		产生：
		'c'

字符串常量中的形式参数替换	字符串常量包含形式参数时，预处理程序替换形式参数：
	`#define stringize(str) 'str'`
	`stringize(foo)`
	产生：
	"foo"
		应使用 # 预处理程序操作符：
		`#define stringize(str) 'str'`
		`stringize(foo)`
		产生：
		"str"

内置到编译器“前端”的预处理程序	编译器调用 cpp(1)，然后调用编译系统的所有其他组件，具体取决于指定的选项。	ISO C 转换阶段 1-4 涉及预处理程序指令的处理，直接内置到 acomp 中，因此除了在 -Xs 模式下之外，编译时不直接调用 cpp。
使用反斜杠的行串联	不识别此上下文中的反斜杠字符。	要求换行符紧跟在反斜杠字符后面并拼接在一起。
文本字符串中的三字母	不支持此 ISO C 功能。	
asm 关键字	asm 是关键字。	将 asm 视为普通标识符。

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 （续）

主题	Solaris Studio C (K&R)	Solaris Studio ISO C

标识符的链接	不将未初始化的 static 声明视为暂定声明。因此，第二个声明将生成“重新声明”错误，如下所示：
	`static int i = 1;`
	`static int i;`
		将未初始化的 static 声明视为暂定声明。

名称空间	只分为三类：struct/union/enum 标记、struct/union/enum 的成员及其他。	识别四种不同的名称空间：标签名称、标记（跟在关键字 struct、union 或 enum 后面的名称）、struct/union/enum 的成员以及普通标识符。
long double 类型	不支持。	允许 long double 类型声明。
浮点常量	不支持浮点后缀 f、l、F 和 L。	
无后缀整型常量可以具有不同的类型	不支持整型常量后缀 u 和 U。	
宽字符常量	不接受宽字符常量的 ISO C 语法，如下所示： `wchar_t wc = L'x';`	支持此语法。
'\a' 和 '\x'	将它们视为字符 'a' 和 'x'。	将 '\a' 和 '\x' 视为特殊转义序列。
文本字符串的串联	不支持相邻文本字符串的 ISO C 串联。	
宽字符文本字符串语法	不支持此示例中所示的 ISO C 宽字符文本字符串语法： `wchar_t *ws = L"hello";`	支持此语法。
指针：void * 与 char *	支持 ISO C void * 功能。	
一元加运算符	不支持此 ISO C 功能。	
函数原型－省略号	不支持。	ISO C 定义使用省略号 "..." 表示变量参数列表。
类型定义	禁止另一个具有相同类型名称的声明在内部块中重新声明 typedef。	允许另一个具有相同类型名称的声明在内部块中重新声明 typedef。
extern 变量的初始化	不支持显式声明为 extern 的变量的初始化。	将显式声明为 extern 的变量的初始化视为定义。
聚集的初始化	不支持联合或自动结构的 ISO C 初始化。	
原型	不支持此 ISO C 功能。	

表 H–1 K&R Solaris Studio C 与 Solaris Studio ISO C 不兼容 （续）

主题	Solaris Studio C (K&R)	Solaris Studio ISO C
预处理指令的语法	仅识别第一列中具有 # 的指令。	ISO C 允许在 # 指令前使用前导空白字符。
# 预处理程序操作符	不支持 ISO C # 预处理程序操作符。	
`#error` 指令	不支持此 ISO C 功能。	
预处理程序指令	支持 `unknown_control_flow` 和 `makes_regs_inconsistent` 两个 pragma 以及 `#ident` 指令。预处理程序发现无法识别的 pragma 时发出警告。	不指定预处理程序对无法识别的 pragma 的行为。

预定义宏名称	未定义以下由 ISO C 定义的宏名称：
	`__STDC__`
	`__DATE__`
	`__TIME__`
	`__LINE__`	

H.2 关键字

下面几个表列出 ISO C 标准、Solaris Studio ISO C 编译器以及 Solaris Studio C 编译器的关键字。

第一个表列出 ISO C 标准定义的关键字。

表 H–2 ISO C 标准关键字

`_Bool`[1]	`_Complex`[1]	`_Imaginary`[1]	`auto`
`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`
`else`	`enum`	`extern`	`float`
`for`	`goto`	`if`	`inline`[1]
`int`	`long`	`register`	`restrict`
`return`	`short`	`signed`	`sizeof`
`static`	`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`	`volatile`
`while`			

[1] 只能使用 -xc99=all 定义

C 编译器还定义一个附加关键字 asm。但是，在 -Xc 模式下不支持 asm。

下面列出 Solaris Studio C 中的关键字。

表 H–3 Solaris Studio C (K&R) 关键字

asm	auto	break	case
char	continue	default	do
double	else	enum	extern
float	for	fortran	goto
if	int	long	register
return	short	sizeof	static
struct	switch	typedef	union
unsigned	void	while	

索引

数字和符号

-#，87, 196
-###，88, 196
// 注释指示符
  使用 -xCC，226
  在 C99 中，304

A

-A，196
-a，88
abort 函数，294
acomp（C 编译器），30
-Aname 的预断言，196
any 级别别名歧义消除，220
ascftime 函数，91
__asm 关键字，52
#assert，38–39, 196
ATS：自动调优系统，261–262

B

-B，197
-b，88
basic 级别别名歧义消除，220
binopt，221

C

C 编程工具，30
C 编译器
  编译程序，195, 196
  编译模式和依赖性，51
  更改搜索库的缺省目录，196
  传递给链接程序的选项，279
  组件，30
-C，88, 197
-c，88, 197
C99
  // 注释指示符，304
  FLT_EVAL_METHOD，302
  for 循环中的类型声明，309
  __func__ 支持，303
  inline 函数说明符，306
  _Pragma，310
  Studio 编译器实现，281
  关键字列表，303
  混合声明和代码，309
  可变长度数组，307
  类型说明符要求，304
  灵活的数组成员，305
  幂等限定符，306
  数组声明符，307
  隐式函数声明，304
C99 中的可变长度数组，307
C99 中的幂等限定符，306
calloc 函数，294
case 语句，322
cc 编译器命令行选项, -fd，203
cc 编译器选项
  -X
    与 FLT_EVAL_METHOD 的交互，302

cc 命令行选项，196–279
-#，189, 196
-###，189, 196
-A，187, 196
-B，191, 197
-C，188, 197
-c，189, 197
-D，188
-d，191, 198, 208
与 -G 交互，208
-E，188, 198
-errfmt，190, 198
-erroff，190, 199
-errshort，190, 199
-errtags，190, 200
-errwarn，190, 200
-fast，183, 185, 201
-fd，188
-features，189, 203
-flags，203
-flteval，186, 192, 203
与 FLT_EVAL_METHOD 的交互，302
-fnonstd，186
-fns，186, 204
作为 -fast 扩展选项的一部分，202
-fprecision，186, 192, 205
与 -flteval 的交互，204
与 FLT_EVAL_METHOD 交互，303
-fround，186, 205
与 -xlibmopt 的交互，245
-fsimple，186, 206
作为 -fast 扩展选项的一部分，202
-fsingle，186, 207
与 FLT_EVAL_METHOD 的交互，302
作为 -fast 扩展选项的一部分，202
-fstore，186, 192, 207
-ftrap，186, 207
-G，191, 208
-g，190, 208
-H，188, 209
-h，191, 209
-I，188, 210
-i，191, 210
-include，188, 210

cc 命令行选项（续）
-keeptmp，189, 211
-KPIC，211
-Kpic，211
-L，191, 211
-l，191, 211
-mc，191, 212
-mr，191, 213
-mt，213
-mt，185, 187
-native，213
-nofstore，187, 192, 214
与 -flteval 的交互，204
作为 -fast 扩展选项的一部分，202
-O，214
-o，189, 214
-P，188, 214
-p，183, 185
-Q，191, 214
-qp，214
-R，191, 215
-S，189, 215
-s，190, 215
-traceback，215–216
-U，188, 216
-V，189, 216
-v，190, 216
-W，189, 217
-w，190, 218
-X，188, 189, 218
-x ibmil，184
-xaddr32，219
-xalias_level，183, 219
示例，116–124, 124
说明，111
作为 -fast 扩展选项的一部分，202
--xannotate，221
-xannotate，183
-xarch，185, 192
与 -flteval 的交互，204
与FLT_EVAL_METHOD 交互，302
-xautopar，185, 187, 225
-xbinopt，183, 226
-xbuiltin，183, 226

cc 命令行选项, -xbuiltin（续）
作为 -fast 扩展选项的一部分，202
-xc99，188, 189, 227
在数学转换中，37
-xcache，192
-xCC，188, 226
-xchar，188, 189, 229
-xchar_byte_order，186, 230
-xcheck，187, 190, 230
-xchip，192, 232
-xcode，191, 234
-xcsi，188, 236
-xdebugformat，190, 236
-xdepend，184, 186, 187, 237
-xdryrun，237
-xe，190, 237
-xF，184, 237
-xhelp，189, 238
-xhwcprof，184, 185, 238
-xinline，184, 239
-xipo，184, 185, 241
-xipo_archive，184, 242
-xjobs，184, 189, 243
-xldscope，32, 191, 244
-xlibmieee，187, 245
-xlibmil，245
作为 -fast 扩展选项的一部分，202
-xlibmopt，184, 245
作为 -fast 扩展选项的一部分，202
-xlic_lib，184
-xlicinfo，193
-xlinkopt，184, 185, 246
与 -G 交互，246
-xloopinfo，187, 247
-xM，188, 247
-xM1，188, 247
-xmaxopt，184, 249
与 -xO 的交互，249
-xMD，248
-xmemalign，185, 186, 249
作为 -fast 扩展选项的一部分，202
-xMerge，191, 248
-xMF，248
-xMMD，188, 248

cc 命令行选项（续）
-xmodel，192, 250
-xnolib，191, 251
-xnolibmil，184, 191, 251
-xnolibmopt，184, 251
与 -xlibmopt 的交互，245
-xO，184, 252
与 -xmaxopt 的交互，252
-xopenmp，185, 186, 187, 254
-xP，188, 255
-xp c，261–262
-xpagesize，184, 185, 191, 255
-xpagesize_heap，184, 186, 191, 256
-xpagesize_stack，184, 185, 191, 256
-xpch，184, 189, 257
-xpchstop，184, 189, 261
--xpec，184
-xpentium，184, 192, 262
-xpg，186, 188, 262
-xprefetch，184, 262
-xprefetch_auto_type，185, 263
-xprefetch_level，184, 264
-xprofile，185, 186, 264–267
-xprofile_ircache，185, 267
-xprofile_pathmap，185, 267
-xreduction，187, 268
-xregs，192, 268
-xrestrict，185, 269
-xs，191, 270
-xsafe，185, 270
-xsfpconst，187, 271
-xspace，185, 271
-xstrconst，192, 271
-xtarget，192, 271
-xtemp，189, 274
-xtime，189, 275
-xtransition，190, 275
对三字母发出警告，134
-xtrigraphs，188, 275
-xunroll，185, 276
-xustr，188, 276
-xvector，186, 187, 277
-xvis，191, 277
-xvpara，187, 190, 278

cc 命令行选项（续）
-Y，189, 278
-YA，189, 278
-YI，190, 279
-YP，190, 196, 279
-YS，190, 279
-Zll，187, 279
cftime 函数，91
cg（代码生成器），30
char, 有符号性，229
clock 函数，295, 330
const，137–140
const，151
cpp（C 预处理程序），30
creat 函数，91
cscope，167, 181
编辑源文件，168, 174, 175, 180, 181
环境变量，176, 177
环境设置，168, 181
命令行使用，168, 169, 175, 176
搜索源文件，167, 168, 169, 174
用法示例，168, 175, 177, 180
cscope 使用的 TERM 环境变量，168

D

-d，198
__DATE__，290, 324
dbx 工具
符号表信息，208
禁用自动读取，270
#define，197
-dirout，88
dwarf 调试器数据格式，236

E

-E，198
EDITOR，168, 180
elfdump，236
er_src 实用程序，226
ERANGE，326
ERANGE 宏，291
-err，88
-errchk，88
-errfmt，89, 198
-errhdr，90
errno
C98 实现，326
-fast 的影响，201
-xbuiltin 的影响，226
-xlibmieee 的影响，245
-xlibmil 的影响，245
-xlibmopt 的影响，245
保留值，51
初始化函数的影响，43
头文件，143, 144
完成函数的影响，42
在下溢时将值设置为 ERANGE，291, 293, 294
-erroff，90, 199
#error，40
-errsecurity，91
-errshort，199
-errtags，92, 200
-errwarn，92, 200
exec 函数，92
_Exit 函数，294

F

-F，93
-fast，201
fbe（汇编程序），30
fclose 函数，294
-fd，93, 203
-features，203
fegetexceptflag 函数，291
feraiseexcept 函数，291
fgetc 函数，92
fgetpos 函数，293
-flags，203
-flagsrc，93
float.h
定义的宏，295
在 C90 中，302
FLT_EVAL_METHOD
C99 中的计算格式，302

FLT_EVAL_METHOD（续）
对 float_t 和 double_t 的影响，291
对库函数准确度的影响，286
非标准负值，286
-flteval，203
fmod 函数，291
-fns，204
fopen 函数，91
-fprecision，205
fprintf 函数，293, 330
free 函数，294
-fround，205
fscanf 函数，293, 330
fsetpos 函数，293
-fsimple，206
-fsingle，207
-fstore，207
ftell 函数，293
-ftrap，207
__func__，303
function，290–295
ilogbf，291
ilogbl，291
prototypes，129
using varying argument lists，130
fwprintf 函数，293
fwscanf 函数，293

G

-G，208
-g，208
getc 函数，92
getenv 函数，284
gets 函数，91
getutxent 函数，163
__global，33

H

-H，209
-h，93, 209
__hidden，33

I

-I，93, 210
-i，210
ilogb 函数，291
ilogbf 函数，291
ilogbl function，291
#include, 添加头文件，54
-include，210
ipo（C 编译器），30
ir2hf（C 编译器），30
iropt（代码优化器），30
isalnum 函数，325
isalpha 函数，299, 325
isatty 函数，282
iscntrl 函数，325
islower 函数，325
ISO C vs. K&R C，218
ISO/IEC 9899\
1999 C 编程语言，301
1999 编程语言 C，27
ISO/IEC 9899-1990 标准，31
isprint 函数，325
isupper 函数，325
iswalpha 函数，299
iswctype 函数，300

J

ja_JP.PCK 语言环境，236

K

K&R C vs. ISO C，218
-k，93
-keeptmp，211

L

-L，93, 211
-l，93, 211
LANG 环境变量
在 C90 中，315

LANG 环境变量（续）
在 C99 中，285, 299
layout 级别别名歧义消除，220
LC_ALL 环境变量
在 C90 中，315
在 C99 中，285
LC_CTYPE 环境变量
在 C90 中，315
在 C99 中，285
-ld_open，221
ld（C 编译器），30
libfast.a，345
limits.h, 定义的宏，296
lint
lint command-line options
-errfmt，89
lint 命令行选项
-#，87
-###，88
-a，88
-b，88
-C，88
-c，88
-dirout，88
-err=warn，88
-errchk，88
-errhdr，90
-erroff，90
-errsecurity，91
-errtags，92
-errwarn，92
-F，93
-fd，93
-flagsrc，93
-h，93
-I，93
-k，93
-L，93
-l，93
-m，94
-n，96
-Ncheck，94
-Nlevel，95
-o，96
lint, lint 命令行选项（续）
-p，96
-R，96
-s，96
-u，96
-V，96
-v，96
-W，97
-x，98
-Xalias_level，97
-Xc99，97
-XCC，97
-Xkeeptmp，98
-Xtemp，98
-Xtime，98
-Xtransition，98
-Xustr，98
-y，99
lint 如何检查代码，86
messages
formats of，101
过滤器，109, 110
基本模式
调用，86
引入的，85
简介，85–110
可疑的构造，107, 108
可移植性检查，106, 107
库，108, 109
头文件, 查找，87
消息
格式，100
禁止，99
消息 ID（标记）, 标识，92, 99
一致性检查，105
预定义，39
增强模式
调用，86
引入的，85
诊断，105, 108
指令，102, 105
lint 的过滤器，110
lint 的过滤器 的过滤器，109
lint 基本模式，85

lint 增强模式，85
lint 执行的可移植性检查，106, 107
llib-l*x*.ln 库，108
long double, 在 ISO C 中传递，342
long int，37
long long，36–37, 37
  表示法，336
  返回，342
  后缀，31
  算术提升，37
  值保留，32
  传递，342, 343

M

-m，94
main function，282
makefile 依赖性，247
malloc 函数，294
mbarrier.h，83–84
-mc，212
mcs（C 编译器），30
MP C，59, 82
-mr，213

N

-n，96
-native，213
-Ncheck，94
-Nlevel，95
-nofstore，214
NULL, 值，324
NULL 宏，292

O

-O，214
-o，96, 214
OMP_DYNAMIC 环境变量，53
OMP_NESTED 环境变量，53
OMP_NUM_THREADS，60
OMP_NUM_THREADS 环境变量，53
OMP_SCHEDULE 环境变量，53
OpenMP
  sunw_mp_register，60
  -xopenmp 命令，254
  如何编译，60

P

-P，214
-p，96
PARALLEL，60
  环境变量，53
PEC：可移植的可执行代码，261–262
Pentium，273
POSIX 线程，213
postopt（C 编译器），30
pragma，40–51, 112–114
_Pragma，310
pragma
  #pragma alias，113
  #pragma alias_level，112–114
  #pragma c99，40–41
  #pragma does_not_read_global_data，41
  #pragma does_not_return，41
  #pragma does_not_write_global_data，41–42
  #pragma error_messages，42
  #pragma fini，42
  #pragma hdrstop，42–43
  #pragma ident，43
  #pragma init，43
  #pragma inline，44
  #pragma int_to_unsigned，44
  #pragma may_not_point_to，114
  #pragma may_point_to，113
  #pragma MP serial_loop，44, 77
  #pragma MP serial_loop_nested，45, 77
  #pragma MP taskloop，40, 45, 77–82
  #pragma no_inline，44
  #pragma no_side_effect，45–46, 46
  #pragma noalias，113, 114
  #pragma nomemorydepend，45
  #pragma opt，46
  #pragma pack，46–47

pragma（续）
#pragma pipeloop，47
#pragma rarely_called，47
#pragma redefine_extname，48
#pragma returns_new_memory，49
#pragma unknown_control_flow，49
#pragma unroll，49–50
#pragma warn_missing_parameter_info，50
#pragma weak，50–51
printf 函数，293

Q

-Q，214
-qp，214

R

-R，96, 215
realloc 函数，294
remove 函数，292, 329
rename 函数，293, 330
_Restrict，52
restrict 关键字
由 -Xs 识别，76
在并行化代码中使用，62
在并行化代码中作为类型限定符，76
作为支持的 C99 功能的一部分，303

S

-S，215
-s，96, 215
scanf 函数，91
setlocale(3C)，144, 146
setlocale 函数，291
signal 函数，282
signed，316
sizeof 函数，162
Solaris 线程，213
ssbd（C 编译器），30
stabs 调试器数据格式，236
STACKSIZE 的从属线程缺省设置，61
STACKSIZE 环境变量，61
stat 函数，92
std 级别别名歧义消除，221
__STDC__ 在 -Xc 下的值，218
stdint.h, 定义的宏，297
strerror 函数，299
strftime 函数，295
strict 级别别名歧义消除，221
strlcpy 函数，91
strong 级别别名歧义消除，221
strtod 函数，294
strtof 函数，294
strtold 函数，294
sun_prefetch.h，263
SUN_PROFDATA, 定义，53
SUN_PROFDATA_DIR, 定义，53
sunw_mp_register_warn() 函数，60
SUNW_MP_THR_IDLE 环境变量，54, 61
SUNW_MP_WARN 环境变量，61
__symbolic，33
system 函数，284, 294

T

tcov, 和 -xprofile，266
__thread，33
__TIME__，290, 324
/tmp，54
TMPDIR 环境变量，54
towctrans 函数，300
TZ，330

U

-U，216
-u，96
ube（C 编译器），30
unsigned，316
unsigned long long，36

V

-V，96, 216
-v，96, 216
varargs(5)，127
VIS 软件开发者工具包，277
volatile, explanation of keyword and usage，138
volatile
　定义和示例，139, 140
　关键字和用法说明，137–140
　兼容的声明，151
　在 C90 中，322
VPATH 环境变量，168

W

-W，97, 217
-w，218
wait 函数，294
wait3 函数，294
waitid 函数，294
waitpid 函数，294
#warning，40
wcsftime 函数，295
wcstod 函数，294
wcstof 函数，294
wcstold 函数，294
weak 级别别名歧义消除，220

X

-X，218
-x，98
-Xalias_level，97
-xalias_level，219
-xarch=*isa*, 编译器选项，221
-xautopar，225
-xbinopt，226
-xbinopt 和，226
-xbuiltin，226
-Xc99，97
-xc99，227
-XCC，97
-xCC，226
-xchar，229
-xchar_byte_order，230
-xcheck，230
-xchip，232
-xcode，234
-xcsi，236
-xdebugformat，236
-xdepend，237
-xdryrun，237
-xe，237
-xF，237
-xhelp，238
-xhwcprof，238
-xinline，239
-xipo，241
-xipo_archive，242
-xjobs，243
-Xkeeptmp，98
-xldscope，244
-xlibmieee，245
-xlibmil，245
-xlibmopt，245
-xlinkopt，246
-xloopinfo，247
-xM，247
-xM1，247
-xmaxopt，249
-xmemalign，249
-xMerge，248
-xMF，248
-xMMD，248
-xmodel，250
-xnolib，251
-xnolibmil，251
-xnolibmopt，251
-xO，252
-xopenmp，254
-xP，255
-xpagesize，255
-xpagesize_heap，256
-xpagesize_stack，256
-xpch，257
-xpchstop，261
-xpec，261–262

-xpentium，262
-xpg，262
-xprefetch，262
-xprefetch_auto_type，263
-xprefetch_level，264
-xprofile，264–267
-xprofile_ircache，267
-xprofile_pathmap，267
-xreduction，268
-xregs，268
-xrestrict，269
-xs，270
-xsafe，270
-xsfpconst，271
-xspace，271
-xstrconst，271
-xtarget，271
-xtemp，274
-Xtemp，98
-xthreadvar，274
-xthreadvar，编译器选项，274
-xtime，275
-Xtime，98
-xtransition，275
-Xtransition，98
-xtrigraphs，275
-xunroll，276
-xustr，276
-Xustr，98
-xvector，277
-xvis，277
-xvpara，278

Y

-Y，278
-y，99
-YA，278
-YI，279
-YP，279
-YS，279

Z

-Zll，279

按

按单精度对 float 表达式，207

绑

绑定，静态与动态，197

包

包含类型声明的 for 循环，309

保

保留名称，142，144
　供扩展使用，143
　供实现使用，142
　选择准则，143
保留字符的有符号性，229

本

本地时区，330

编

编辑，源文件，**请参见**cscope

变

变量，线程局部存储说明符，33
变量的线程局部存储，33
变量声明说明符，32

标
标记，134, 137
标准符合性，27, 31

表
表达式, 分组和求值，146, 148
表示, 整数，316–317
表示形式, 浮点，317–318

别
别名歧义消除，111, 124

并
并行化，59, 82
  另请参见OpenMP
  环境变量，60, 62
  使用 `-xreduction` 打开约简识别，268
  使用 `-xvpara` 检查正确并行化的循环，278
  用 `-xautopar` 为多个处理程序打开，225
  用 `-xloopinfo` 查找并行化循环，247
  用 `-xopenmp` 指定 OpenMP pragma，254
  用 `-Zll` 创建程序数据库，279

不
不停止, 浮点运算，34
不完全类型，149–150, 150

常
常量
  特定于 Solaris Studio ISO C，31–32, 32
  提升整型，133

错
错误消息，313
  将前缀 "error\
    " 添加到，198
  控制类型不匹配的长度，199
  在 `lint` 中抑制，90

代
代码生成器，30
代码优化
  通过使用 `-fast`，201
  用 `-xO`，252
  优化器，30

带
带符号整型数的按位操作，317

调
调试器数据格式，236
调试信息, 删除，215

动
动态链接，198

独
独立式环境，57–58

堆
堆, 设置页面大小，255

多
多处理，59, 82
  -xjobs，243
多媒体类型, 处理，277
多线程，213
多字节字符和宽字符，140–141, 141

二
二进制文件优化，226

浮
浮点，317–318
  表示形式，317–318
  不停止，34
  渐进下溢，34
  截断，318
  值，317–318

符
符号调试信息, 删除，215
符号声明说明符，32

覆
覆盖率分析 (tcov)，266

功
功能, 声明说明符，32

共
共享库, 命名，209

关
关键字，52
  用于 C99 的列表，303

国
国际化，140, 141, 144, 146

过
过程间分析过程，241
过时选项, 列表，193–194

函
函数
  abort，294
  ascftime，91
  calloc，294
  cftime，91
  clock，295, 330
  creat，91
  exec，92
  _Exit，294
  fclose，294
  fegetexceptflag，291
  feraiseexcept，291
  fgetc，92
  fgetpos，293
  fmod，291, 326
  fopen，91
  fprintf，293, 330
  free，294
  fscanf，293, 330
  fsetpos，293
  ftell，293
  fwprintf，293
  fwscanf，293
  getc，92
  getenv，284
  gets，91
  getutxent，163

函数（续）
ilogb，291
isalnum，325
isalpha，299, 325
isatty，282
iscntrl，325
islower，325
isprint，325
isupper，325
iswalpha，299
iswctype，300
main，282
malloc，294
printf，293
realloc，294
remove，292, 329
rename，293, 330
scanf，91
setlocale，291
signal，282
sizeof，162
stat，92
strerror，299
strftime，295
strlcpy，91
strtod，294
strtof，294
strtold，294
sunw_mp_register，60
system，284, 294
towctrans，300
wait，294
wait3，294
waitid，294
waitpid，294
wcsftime，295
wcstod，294
wcstof，294
wcstold，294
使用可变参数列表，129
隐式声明，304
原型，105, 126
原型，lint 检查，109
重新排序，237

宏
宏
__DATE__，290, 324
ERANGE，291
FLT_EVAL_METHOD，291, 302
NULL，292
__TIME__，290, 324
在 float.h 中指定，295
在 limits.h 中指定，296
在 stdint.h 中指定，297
宏扩展，135

环
环境变量
cscope 使用的 EDITOR，168, 180
cscope 使用的 TERM，168
cscope 使用的 VPATH，168
LANG
在 C90 中，315
在 C99 中，285, 299
LC_ALL
在 C90 中，315
在 C99 中，285
LC_CTYPE
在 C90 中，315
在 C99 中，285
OMP_DYNAMIC，53
OMP_NESTED，53
OMP_NUM_THREADS，53, 60
OMP_SCHEDULE，53
PARALLEL，53, 60
STACKSIZE，61
SUN_PROFDATA，53
SUN_PROFDATA_DIR，53
SUNW_MP_THR_IDLE，54, 61
SUNW_MP_WARN，61
TMPDIR，54
TZ，330

缓
缓冲，329

缓存, 供优化器使用，227

换
换行符, 终止，328

回
回溯，215–216

汇
汇编程序，30
汇编语言模板，277

活
活前缀，259

基
基于类型的别名歧义消除，111, 124

计
计算转移 (computed goto) 语句，34

兼
兼容性选项，195, 218

渐
渐进下溢，34

交
交互式设备，314

结
结构
  对齐，320–321
  填充，320–321
结构对齐，320–321
结构填充，320–321

警
警告消息，313

静
静态链接，198

可
可移植性, 代码，106, 107

空
空格字符，328
空字符不附加至数据，328

库
库
  cc搜索的缺省目录，196
  libfast.a，345
  lint，108, 109
  llib-l*x*.ln，108
  sun_prefetch.h，263
  共享或非共享，197
  内在名称，209
  生成共享库，236

库（续）
指定动态或静态链接，197
重命名共享，209
库绑定，197

酷
酷类工具 URL，261–262

宽
宽文本字符串，141
宽字符，140, 141
宽字符常量，141

类
类型
const 和 volatile 限定符，137–140, 140
for 循环中的声明，309
不完全，149–150, 150
存储分配，333
兼容和复合，151, 152
声明和代码，309
声明中的说明符要求，304

链
链接, 静态与动态，198
链接程序
从编译器接收的选项，279
抑制链接，197
指定动态或静态链接，198
链接时选项, 列表，185
链接时优化，246

临
临时文件，54

零
零长度文件，329

流
流，328

模
模式, 编译器，218

目
目标文件
使用 er_src 实用程序读取编译器注解，226
为每个源文件生成目标文件，197
抑制删除，197
用 ld 链接，197
目标文件中的编译器注解, 使用 er_src 实用程序读取，226

内
内存边界内部函数，83–84
内联，245
内联扩展模板，245, 251

缺
缺省
编译器行为，218
处理和 SIGILL，328
语言环境，315

日
日期和时间格式，331–332

三
三字符序列，134

删
删除符号调试信息，215

舍
舍入行为，34

声
声明符，322
声明说明符
  `__global`，33
  `__hidden`，33
  `__symbolic`，33
  `__thread`，33

省
省略号，127, 129, 152

实
实现定义的行为，313–332, 332

时
时间和日期格式，331–332

使
使用 C 编程的工具，30

适
适用于 C 的编程工具，30

手
手册页, 访问，28

输
输出，36, 330

属
属性，39

数
数据类型
  `long long`，36
  `unsigned long long`，36
数据重新排序，237
数学函数, 域错误，325–326
数组
  C99 中不完整数组类型，305
  C99 中的声明符，307

搜
搜索, 源文件, **请参见**`cscope`
搜索库的缺省目录，196

算
算术转换，36, 37

提
提升，131, 133

提升（续）
缺省参数，127
位字段，132
整型常量，133
值保留，131

头

头文件
#include 指令的格式，54
C90 中的 float.h，302
标准头文件列表，142
标准位置，54, 55
如何包含，55
如何包括，54
使用 lint，87

位

位, 在执行字符集中，315
位字段
赋予常量时的可移植性，106
视为带符号或无符号，321
受转换为 ISO C 的影响，152
提升，132

文

文本
段和文本字符串，271
流，328
文本段中的文本字符串，271
文档, 访问，21–22
文档索引，21
文件, 临时，54
文件配置, -xprofile，264

线

线程, **请参见**并行化

限

限定符，322

消

消息, 错误，313
消息 ID（标记），199, 200

小

小数点字符，331

信

信号，326–328, 328

行

行为, 实现定义，313–332, 332

性

性能
用 -fast 进行优化，201
用 -xO 优化，252
针对 *SPARC* 进行优化，345

选

选项, lint，99
选项, 命令行
lint，87–99
按字母顺序列出的引用，196–279
选项，命令行
另请参见cc 命令行选项
按功能分组，183–194

循
循环，237

页
页面大小, 为栈或堆设置，255

易
易读文档，21–22

用
用于 C99 的 `inline` 函数说明符，306
用于类型的存储分配，333

优
优化
- `-fast` 和，201
- `pragma` **选项**和，46
- `-xipo` 和，241
- `-xO` 和，252
- 用 `-xmaxopt`，249
- 优化器，30
- 在链接时，246
- 针对 *SPARC*，345

由
由 `lint` 进行一致性检查，105

右
右移，317

语
语言环境，144, 146
- `ja_JP.PCK`，236
- 缺省，315
- 使用不符合，236
- 行为，330–332

域
域错误, 数学函数，325–326

预
预编译头文件，257
预处理，134, 137
- 标记粘贴，136
- 如何保留注释，197
- 预定义的名称，51
- 指令，51, 54, 55, 197, 322–324
- 字符串化，136

预取，262

源
源代码中的汇编，52
源文件
- K&R C 的兼容性，195
- 编辑
  - **请参见**`cscope`
- 定位，323
- 使用 `lint` 进行检查，110
- 搜索
  - **请参见**`cscope`

在
在文本流中写，329
在源代码中使用汇编，52

栈
栈
  内存分配最大值，333
  设置页面大小，255
栈中的内存分配，333
栈中内存分配的限制，333

诊
诊断, 格式，313

整
整个程序优化，241
整数，316–317, 317
整型常量, 提升，133

值
值
  浮点，317–318
  整数，316–317

指
指令, **请参见**pragma

重
重命名共享库，209
重新排序函数和数据，237

主
主要, 参数的语义，314

注
注释
  防止预处理程序删除，197
  通过发出 -xCC 使用 //，226
  在 C99 中使用 //，304

转
转换，36, 37
  整数，317

传
传递, 名称和版本，216

字
字符
  单字符字符常量，322
  多字节, 移位状态，315
  集, 整理序列，331
  集的测试，325
  集的源代码和执行，314
  集中的位，315
  空格，328
  小数点，331
  映射集，314–316
字符的有符号性，229

自
自述文件，27